engenharia de software magazine

Métodos Ágeis
Conceitos introdutórios sobre planejamento ágil de projetos

Projeto
Desenvolvimento de Software Centrado em Arquitetura

DevMedia group Ano I - Edição 08

ISSN 1983127-7

MATURIDADE + EVOLUÇÃO

- Avaliação Independente de Garantia da Qualidade
- Maturidade em Gerenciamento de Projetos: Uma Visão Analítica

Projeto
Entenda os conceitos básicos da orientação a objetos e da UML

Projeto
Saiba o que é confiabilidade e por que esta característica é importante em projetos de software

Validação, Verificação & Teste
Entenda o que são testes de mutação e como executá-los utilizando o plug-in MuClipse

Validação, Verificação & Teste
Veja como realizar testes automatizados de software em Web Services

Aulas desta edição:

- Controle de versões com Subclipse
- Análise de Pontos de Função
- Persistência de Objetos com o SGBDOO Jade
- Introdução à Construção de Diagrama de Classes da UML – Parte 8
- Introdução à Construção de Diagrama de Classes da UML – Parte 9
- Introdução à Construção de Diagrama de Classes da UML – Parte 10
- Introdução à Construção de Diagrama de Classes da UML – Parte 11

engenharia de software magazine

Ano 1 - 8ª Edição 2008 - Impresso no Brasil

Corpo Editorial

Colaboradores
Rodrigo Oliveira Spínola
rodrigo@sqlmagazine.com.br

Marco Antônio Pereira Araújo
Eduardo Oliveira Spínola

Editor de Arte
Vinicius O. Andrade
viniciusoandrade@gmail.com

Diagramação
Ampel Produções Editoriais

Revisão
Gregory Monteiro
gregory@clubedelphi.net

Na Web
www.devmedia.com.br/esmag

Apoio

Parceiros:

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 Engenharia de Software Magazine.

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

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

EDITORIAL

Esta edição da Engenharia de Software Magazine traz como tema de capa Maturidade + Evolução. Com isso, buscamos discutir alguns assuntos que podemos ter em mente ao realizar atividades de melhoria de processo. Para abordar este tema, destacamos dois artigos:

Maturidade em Gerenciamento de Projetos: Uma Visão Analítica

"Nos dias atuais, a execução de projetos está se tornando comum e remete ao desafio de gerenciar projetos com eficiência. O uso efetivo de tecnologias durante esta atividade pode determinar o sucesso de qualquer negócio e satisfazer as expectativas dos clientes. Contudo, em busca do sucesso, o uso adequado destas tecnologias é crucial. É possível observar uma crescente busca por melhoria de processos relacionados, principalmente, à gerência de projetos e à engenharia de software em várias organizações de Tecnologia da Informação (TI). Dentro deste contexto, este artigo tem o objetivo de apresentar os fatores que levam ao sucesso de projetos, conceitos básicos da área de maturidade em gerenciamento de projetos e uma visão comparativa de alguns dos modelos que estão em evidência."

Avaliação Independente de Garantia da Qualidade

"A Garantia da Qualidade visa avaliar a aderência das atividades executadas e dos produtos de trabalho gerados a padrões, processos, procedimentos e requisitos estabelecidos e aplicáveis, fornecendo uma visão objetiva e independente, tanto para atividades de processo quanto de produto, em relação a desvios e pontos de melhoria, de forma a assegurar que a qualidade planejada não será comprometida. Neste artigo apresenta-se o processo de Garantia da Qualidade com base no MPS.BR nível F, destacando a importância e a necessidade em se realizar uma avaliação independente de Garantia da Qualidade e fatores críticos de sucesso desta avaliação."

Além destas duas matérias, esta edição traz mais seis artigos:

- Planejamento Ágil de Projetos;
- Desenvolvimento de Software Centrado em Arquitetura;
- Conceitos de orientação a objetos e UML;
- Confiabilidade de Software: Determinante da Qualidade de Sistemas de Software;
- Testes de Mutação com o plug-in MuClipse;
- Testes Automatizados de Software em Web Services.

Desejamos uma ótima leitura!

Equipe Editorial Engenharia de Software Magazine

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

Caro Leitor,
Para esta edição, temos um conjunto de 7 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:

01 – Planejamento
Título: Análise de Pontos de Função
Autor: Marco Antônio Pereira Araújo
Mini-Resumo: Medições em software são boas ferramentas para a redução de riscos nas estimativas para construção de sistemas. Esta vídeo-aula apresenta a técnica de Análise de Pontos de Função com o objetivo de dimensionar o tamanho de aplicações a serem construídas ou mantidas, auxiliando no processo de estimativa de desenvolvimento.

02 – Projeto
Título: Controle de versões com Subclipse
Autor: Marco Antônio Pereira Araújo
Mini-Resumo: A utilização de sistemas de controle de versão é uma boa prática no gerenciamento de projetos de software, principalmente em seu processo de manutenção. Esta vídeo-aula apresenta a ferramenta SubVersion (SVN) para gerenciamento de versões através do plug-in Subclipse, para a IDE Eclipse.

03 – Projeto
Título: Persistência de Objetos com o SGBDOO Jade
Autor: Marco Antônio Pereira Araújo
Mini-Resumo: A persistência é sempre um fator de discussão quando se trata de desenvolvimento orientado a objetos. Dentre as diversas alternativas, sistemas de gerenciamento de bancos de dados orientados a objetos podem ser uma alternativa mais natural para este processo. Esta vídeo-aula apresenta o Jade, um SGBDOO para persistência de objetos, com o objetivo de demonstrar este tipo de persistência.

04 – Projeto
Título: Introdução à Construção de Diagrama de Classes da UML – Parte 8
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Esta vídeo aula dá continuidade à elaboração do diagrama de classes para o sistema de locadora de veículos. Em particular, são elaboradas as classes identificadas no diagrama de classes.

05 – Projeto
Título: Introdução à Construção de Diagrama de Classes da UML – Parte 9
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Esta vídeo aula dá continuidade à elaboração do diagrama de classes para o sistema de locadora de veículos. Em particular, os atributos identificados são atribuídos às suas respectivas classes e acrescentados ao diagrama de classes.

06 – Projeto
Título: Introdução à Construção de Diagrama de Classes da UML – Parte 10
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Esta vídeo aula dá continuidade à elaboração do diagrama de classes para o sistema de locadora de veículos. Em particular, são identificadas as operações que deverão estar contempladas no diagrama de classes.

07 – Projeto
Título: Introdução à Construção de Diagrama de Classes da UML – Parte 11
Autor: Rodrigo Oliveira Spínola
Mini-Resumo: Esta vídeo aula dá continuidade à elaboração do diagrama de classes para o sistema de locadora de veículos. Em particular, as operações identificadas na aula anterior são atribuídas às suas respectivas classes e são acrescentadas ao diagrama de classes.

Atendimento ao Leitor

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

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

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

ÍNDICE

Você não está mais sozinho.

Planejamento Ágil de Projetos

Dairton Bassi
dbassi@gmail.com
Mestre em Engenharia de Software com foco em Métodos Ágeis pelo IME-USP. Bacharel em Ciência da Computação pelo IME-USP. Membro-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.

De que se trata o artigo:
Neste artigo veremos os principais problemas associados ao planejamento e à gestão de projetos de software. Em seguida, mostraremos como os planos são tratados por Metodologias Ágeis de desenvolvimento de software.

Para que serve:
O planejamento e o desenvolvimento de um software sob o paradigma ágil oferecem mais flexibilidade do que os modelos tradicionais para a condução de projetos de software que possuem incerteza ou instabilidade.

Em que situação o tema útil:
Além de ser um modelo que permite a criação de planos flexíveis, as técnicas ágeis evitam desperdícios de tempo e esforços, pois focam em alcançar rapidamente meios de validar as características do produto em desenvolvimento e de avaliar a evolução do projeto.

Nos últimos anos, o uso de métodos ágeis tem chamado a atenção da indústria de software em todo o mundo. Essas metodologias têm ganhado destaque e gerado muita discussão na comunidade de Engenharia de Software. Graças a isso, muitas empresas estão se interessando em entender como os métodos ágeis funcionam e se eles poderão ajudá-las a resolver seus problemas com desenvolvimento de software ou mesmo melhorar a sua produtividade.

Esta é a primeira parte de uma série sobre planejamento e gestão ágil de projetos. Ao longo da série repensaremos alguns dos conceitos largamente usados pela indústria de software e apresentaremos soluções baseadas em métodos ágeis que visam a melhorar o desempenho das equipes e simplificar o gerenciamento do projeto.

Neste artigo, veremos os impactos causados por estratégias e processos inadequados para o desenvolvimento de software e como as metodologias ágeis tratam o planejamento e os requisitos de um projeto de software.

Planos, Processos e Fracassos

Tradicionalmente, durante o planejamento de desenvolvimento de software, são definidos prazos, custos, recursos

e o escopo do projeto. Esses dados são usados como base por outras áreas da empresa para, por exemplo, organizar e encadear ações comerciais e de marketing. Quando o desenvolvimento do software não cumpre o plano inicial, uma cadeia de eventos pode ser prejudicada e isso se torna motivo de estresse e pressão sobre a equipe de desenvolvimento. O clima tenso instaura-se e começa a busca por culpados. A posição mais cômoda é culpar a equipe de desenvolvimento pela falha no cumprimento do planejado. Outra possibilidade é assumir que o plano não era adequado, portanto inviável de ser cumprido. Certamente existem os dois casos, mas para entender mais sobre esse problema, vamos olhar para alguns números da indústria de software.

Historicamente, inúmeros estudos apontam problemas nos resultados de projetos de software. Em 1992, Albert Lederer e Jayesh Prasad relataram que 66,7% dos projetos de software ultrapassam significativamente os seus custos [1]. Em 1994, o Standish Group analisou 3682 projetos e percebeu que 83,2% foram cancelados ou entregues excedendo o prazo ou o custo [2]. Em 2002, Jim Jonhson apontou que 64% das funcionalidades implementadas são raramente ou nunca utilizadas [3]. Em 2006, o PMI Brasil divulgou um relatório no qual o não-cumprimento dos prazos foi um dos principais problemas no gerenciamento de projetos. No relatório de 2007, problemas com os prazos foram verificados em 82% das empresas analisadas e, problemas para cumprir o custo estão presentes em 66% delas. Além desses, os outros principais problemas apontados foram referentes à comunicação e a mudanças no escopo do projeto [4].

Esses estudos mostram a baixa exatidão entre o planejado e o realizado. Isso indica o quão imatura ainda é a indústria de software e quão limitada é a sua capacidade de fazer previsões. A enorme falta de realismo nas estimativas, o baixo senso de prioridade, os gastos muito além do previsto e o pouco alinhamento com os interesses de negócio somam evidências que indicam que as estratégias de planejamento e os processos de desenvolvimento são incompatíveis com a realidade dos projetos.

Pensando nisso, faz sentido rever a forma de planejar os projetos de desenvolvimento de software. A opção que apresentamos é a abordagem usada por métodos ágeis. A seguir, vamos discutir a maneira usada pela indústria de software para fazer planejamentos. Depois, vamos apontar os cenários onde métodos ágeis podem ser usados, e então, apresentar o planejamento sob a perspectiva ágil.

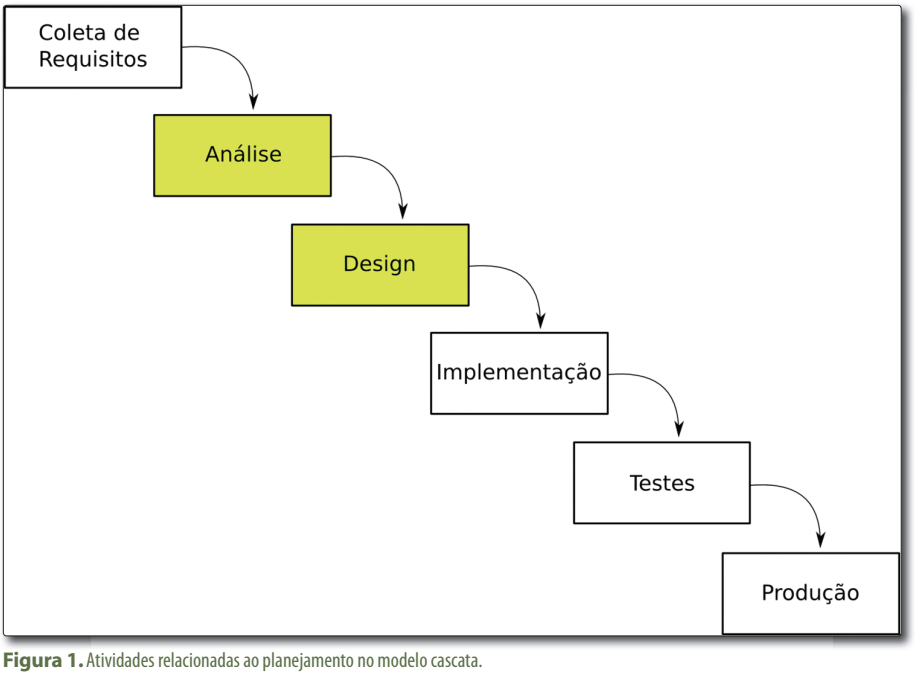


Figura 1. Atividades relacionadas ao planejamento no modelo cascata.

Os Problemas Tradicionais

Durante algumas décadas, a supremacia do modelo Cascata proposto por Winston Royce instituiu uma clara separação entre planejamento e execução [5]. Na segunda metade da década de 80, Berry Boehm propôs o modelo Espiral [6] com duas grandes inovações para a indústria de software: 1) o desenvolvimento iterativo e, 2) a inclusão de uma etapa de análise. Essas inovações permitiram que avaliações pudessem ser feitas periodicamente e que o software e o planejamento fossem criados em ciclos. Nas **Figuras 1** e **2** vemos em amarelo as etapas onde estão concentradas as atividades relacionadas ao planejamento.

Muitas empresas ainda trabalham com análises e estimativas concentradas no início do projeto, a exemplo do modelo cascata, usando o resultado desse trabalho durante todo o projeto, sem considerar a possibilidade de rever as estimativas durante a implementação. Outras empresas usam um modelo que mistura o cascata com o espiral. Estas realizam uma fase de análise e planos no início do projeto e depois praticam desenvolvimento iterativo em busca das metas estabelecidas no início.

Criar um plano completo e detalhado no início do projeto implica fazer previsões e tomar decisões. Quanto mais cedo as decisões são tomadas, mais premissas precisam ser assumidas, o que aumenta a chance da decisão ser inadequada. Como as análises e previsões são feitas antes do desenvolvimento, elas se baseiam em alguns dados, na experiência dos profissionais envolvidos e em premissas, que associam incerteza à decisão e, conseqüentemente, contribuem para aumentar as chances do plano falhar. Vimos muitos dados que indicam as altíssimas chances de planos criados desta forma estarem equivocados. Portanto, ter um plano simplesmente por tê-lo é, no mínimo, desperdício de esforço, tempo e recursos.

A grande questão é que, por definição, os planos são feitos no início do projeto e, justamente por isso, são chamados de planos. Porém, este é o momento em que menos se sabe a respeito do produto e sobre a sua criação. Por outro lado, no fim do projeto, é possível olhar para trás e perceber que muito foi aprendido. Pode-se entender quais foram as melhores escolhas e perceber que as dificuldades do início foram completamente diferentes das do final. Todo este conhecimento acaba não sendo usado a favor do projeto.

Figura 2. Atividades relacionadas ao planejamento no modelo espiral.

Se o plano puder considerar o conhecimento que a equipe adquiriu durante a implementação, ele poderá ter estimativas mais realistas e poderá ordenar suas prioridades de acordo com as necessidades do momento.

A proposta do planejamento ágil é aproveitar ao máximo o conhecimento adquirido durante o projeto. Para isso, esta proposta utiliza um modelo que segmenta o planejamento e prevê revisões constantemente. Dessa forma, o plano pode ser sempre baseado em informações atualizadas.

Usar informações atualizadas é de fato preciso, pois as necessidades do mercado evoluem cada vez mais rápido. Este cenário dificulta a criação de planos completos e detalhados, e os meses que separam a idealização de um produto da sua concepção criam espaço para que os requisitos sejam mais bem definidos e compreendidos.

Como nem todas as idéias e soluções vêm de uma vez é interessante aproveitar aquelas que surgem no decorrer do projeto. Métodos ágeis de desenvolvimento permitem que mudanças e novos requisitos sejam incorporados mesmo depois de iniciada a fase de implementação. Ao mesmo tempo, métodos ágeis impõem um limite que impede instabilidade excessiva no projeto.

Primeiro, o que é agilidade?

A associação de agilidade com velocidade é bastante comum. No contexto do desenvolvimento de software, agilidade é muito mais do que isso. Dizer que desenvolvimento ágil é aquele que entrega rápido é uma interpretação muito simplista e incorreta, pois as definições da metodologia ágil estão tão fortemente entrelaçadas que não faz sentido pensar só em velocidade.

O modelo ágil vai além das questões técnicas. Mais do que o gerenciamento técnico do projeto, ele inclui o relacionamento entre as pessoas. Desenvolver software com agilidade inclui considerar também as prioridades de negócio e os fatores humanos para chegar rapidamente a soluções de qualidade que atendam as necessidades do cliente.

Para atingir esse resultado, há um conjunto de métodos ágeis de desenvolvimento de software que seguem os valores e os princípios do Manifesto Ágil [7]. Dentre esses métodos, a Programação Extrema e o Scrum são os mais difundidos, no entanto, existem outros. Alguns exemplos são os métodos Crystal, Lean Software Development, Feature Driven Development (FDD), DSDM e o Adaptative Software Development [8].

Quem precisa de agilidade?

Se os projetos com os quais você lida têm o escopo fechado e muito bem definido, não possuem requisitos em evolução e não precisam que clientes ou usuários forneçam feedback durante o desenvolvimento para definir ou aprimorar as funcionalidades, então, talvez você não precise de um método ágil. Caso contrário, métodos ágeis poderão ajudá-lo a melhorar seus resultados.

Tente se lembrar... quantos projetos você conseguiu terminar sem que nenhum requisito mudasse? Quando mudou, qual foi o impacto de todas essas mudanças? Pense também quanto tempo foi gasto com identificação de todos os requisitos, análise e planejamento antes da primeira versão ficar pronta. Agora, seja crítico e questione-se sobre o uso desse tempo. Será que ele foi empregado da melhor maneira possível?

Se o tempo usado para levantar todos os requisitos mais o tempo gasto com as mudanças não previstas fosse usado para produzir uma solução simples, a sua empresa ou equipe teria alguma vantagem? Se você disse não, o que aconteceria se um concorrente seu usasse essa estratégia e oferecesse para seus clientes uma solução simples, mas que atendesse as necessidades deles, vários meses antes de você. E enquanto você desenvolve o seu produto completo, ele aprimorasse o dele e ampliasse a aceitação no mercado?

Para não criar mais softwares que são lembrados por seus números negativos, é importante escolher a estratégia olhando para o mercado e atento aos concorrentes. Com a velocidade com que as tecnologias evoluem e a grande competitividade do mercado, faz-se necessário que as idéias sejam revistas e aprimoradas. Neste cenário, onde demandas surgem e mudam com freqüência, as empresas que quiserem se manter em destaque precisam desenvolver a capacidade de adaptação. Para conseguir isso, o primeiro passo é ter ciência de que mudanças acontecerão, pois elas são inerentes a projetos de software, principalmente os que envolvem inovação. Aceitar este fato é o caminho para acabar com as lamentações toda vez que um cliente solicita uma mudança de requisitos. Tratá-las como problemas faz com que se tornem problemas. Se as mudanças

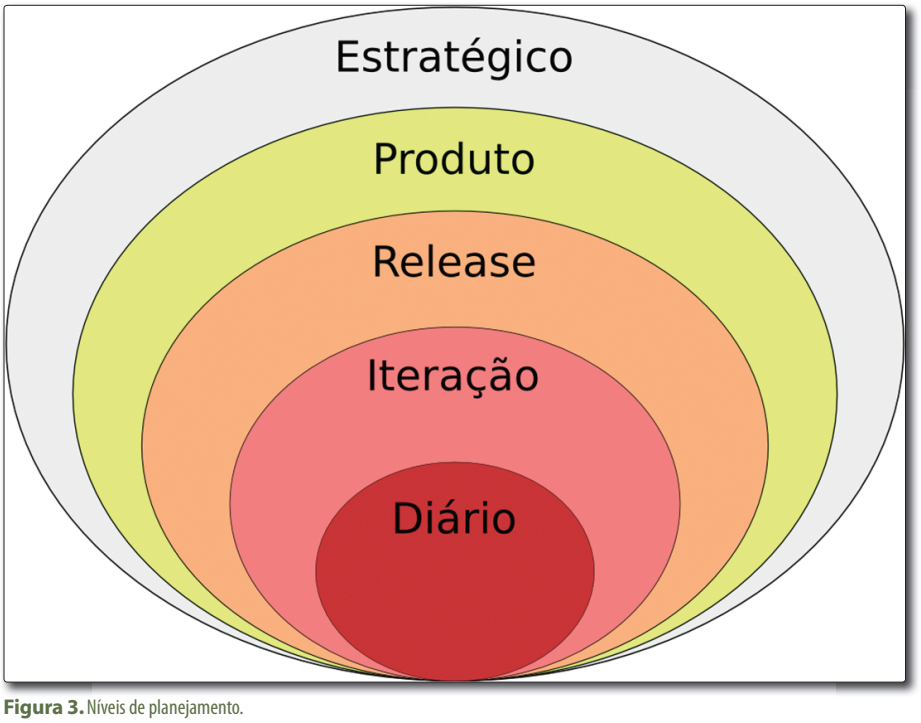


Figura 3. Níveis de planejamento.

estiverem previstas e houver regras para absorvê-las, elas poderão acontecer naturalmente sem prejudicar o projeto.

Planos e Projetos Ágeis

Quando falamos em metodologias ágeis, temos que considerar que o software é desenvolvido por pessoas e não por um processo, que as prioridades devem atender as necessidades de negócio e ter ciência de que existe incerteza nas estimativas. Iterativamente, a arquitetura, a modelagem e cada método do software são concebidos e aprimorados. O mesmo acontece com o planejamento e com as estimativas.

Modelos ágeis consideram que mudanças poderão acontecer ao invés de tentar prever e controlar todas as variáveis. Essa estratégia evita gastar energia demais tentando fazer um plano perfeito e minucioso que cerca todas as possibilidades [8]. Portanto, pensar em planejamento ágil significa começar com a visão dos objetivos de alto nível e seguir em frente sem saber ou tentar prever todos os detalhes no início. Para chegar aos detalhes, o processo de planejamento é segmentado em vários níveis distribuídos pelo projeto, assim, à medida que a implementação acontece, pequenas etapas de planejamento chegarão ao nível necessário de detalhes. A **Figura 3** apresenta os níveis de planejamento [9].

O papel e os objetivos de cada nível de planejamento são os seguintes:

- O **planejamento estratégico** geralmente fica fora do escopo de desenvolvimento de software. Nele são definidas as metas da empresa, posicionamento de mercado e os tipos de produtos necessários para atingir esses objetivos. Todas as áreas da empresa devem trabalhar unidas para que ela alcance os seus objetivos estratégicos;
- O **planejamento do produto** define as principais características da solução, avalia a viabilidade técnica, define a equipe, seleciona as tecnologias e produz uma estimativa de alto nível do tamanho das funcionalidades. A partir disso, chega-se à primeira estimativa de tamanho do projeto;
- O **planejamento da *release*** foca na definição de uma versão que possa ser entregue aos usuários. Neste nível, o principal objetivo é identificar grupos mínimos de funcionalidades que agreguem o máximo de valor ao produto e o tornem útil o suficiente para ser usado por usuários finais. Para isso, a equipe técnica e a equipe de negócios trabalham juntas. A equipe de negócios identifica necessidades dos clientes e usuários. A partir delas define as prioridades e esclarece dúvidas sobre o comportamento ideal do software. A equipe técnica atualiza suas estimativas de tamanho e dificuldade de cada funcionalidade com base no conhecimento que adquiriu nas *releases* anteriores. Essas estimativas ajudam a equipe de negócios a tomar decisões sobre as prioridades do projeto;
- O **planejamento da iteração** preocupa-se em criar um subgrupo de funcionalidades da *release* para ser implementado no período de poucas semanas. Os planejamentos das iterações servem como pontos de controle do plano da *release*. Neste nível, a equipe quebra as funcionalidades selecionadas em tarefas para implementação;
- O **planejamento do dia** acontece com uma reunião de poucos minutos da qual toda a equipe participa. Cada membro atualiza os outros sobre as funcionalidades que concluiu, sobre seus avanços e sobre o que pretende fazer naquele dia de trabalho. Se problemas ou grandes dificuldades estiverem acontecendo, esta é a oportunidade de apresentá-los para que possam ser tratados rapidamente. Além de servir como micro-planejamento para os desenvolvedores, essa reunião é importante para sincronizar a equipe e manter um nível elevado de comunicação.

Na **Figura 4**, várias iterações de cada tipo de planejamento podem ser visualizadas. Neste modelo, duas variáveis estão inver-

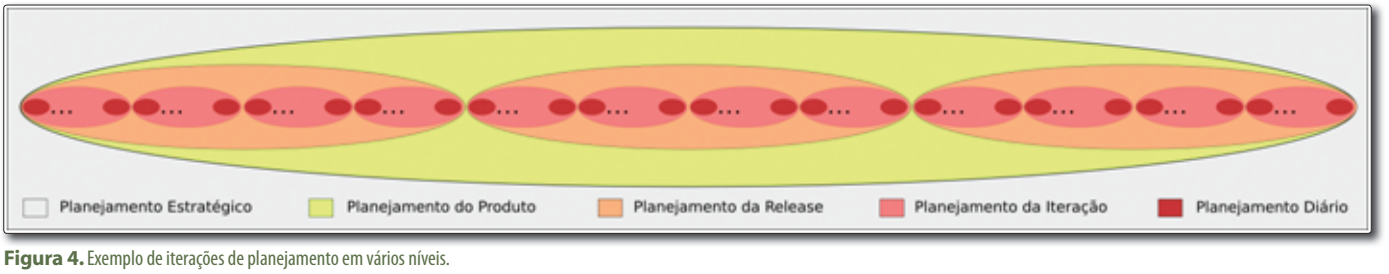


Figura 4. Exemplo de iterações de planejamento em vários níveis.

samente relacionadas: a extensão do plano e o seu nível de detalhe. O tamanho das elipses representa a extensão e a coloração avermelhada, a quantidade de detalhes que o plano contém. Planos de longo prazo concentram-se nos grandes objetivos e contêm pouco detalhamento técnico e operacional. Os detalhes são delegados para os níveis inferiores do planejamento. Os planos de curto prazo, por sua vez, chegam ao nível das tarefas de desenvolvimento e detalhes de implementação.

A separação em níveis permite distribuir melhor as tarefas de planejamento entre todos os envolvidos com o projeto. Assim, ao invés do gerente criar um cronograma de implementação para seus desenvolvedores, os próprios desenvolvedores podem planejar a implementação de uma iteração considerando as questões técnicas que só eles conhecem. Enquanto isso, o gerente pode concentrar a sua atenção em pontos mais amplos do projeto.

Com o planejamento em vários níveis a flexibilidade é outro grande benefício, pois é possível acomodar mudanças ou incluir novas funcionalidades com facilidade. A equipe de desenvolvimento pode analisar e estimar novas requisições para que a equipe de negócios defina as suas prioridades. Novas requisições de alta prioridade, conforme o seu impacto no projeto, podem entrar no plano da próxima iteração ou *release* ou, podem ser trocadas por funcionalidades menos importantes da *release* atual.

Conclusões

Métodos ágeis podem trazer ganhos expressivos de produtividade e qualidade, porém, não é trivial chegar até eles. Usar um método ágil exige uma mudança de paradigma, e, portanto, de valores que a equipe e a empresa precisam estar dispostas a fazer.

Planejar em níveis faz parte da mudança de paradigma, por isso requer experiência para chegar rapidamente a resultados expressivos. Uma forma fácil de começar a direcionar o seu esquema de trabalho para um modelo ágil é reduzir o tamanho dos ciclos de desenvolvimento. Ciclos completos de poucas semanas permitem que os planos sejam revistos freqüentemente.

No próximo artigo desta série vamos focar em **estimativas**. Mostraremos técnicas ágeis para estimar e explicaremos as diferenças entre estimativas de tamanho e estimativas de duração, além da importância de mantê-las separadas ●

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

[1] Albert L. Lederer and Jayesh Prasad. Nine management guidelines for better cost estimating. Commun. ACM, 35(2):51–59, 1992.

[2] Standish Group. The CHAOS Report, 1994. http://www.standishgroup.com/sample_research/chaos_1994_1.php.

[3] 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.

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

[5] W. W. Royce. Managing the development of large software systems: concepts and techniques. In ICSE '87: Proceedings of the 9th international conference on Software Engineering, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[6] Barry Boehm. A spiral model of software development and enhancement. ACM SIGSOFT Software Engineering Notes, 11(4):14–24, 1986.

[7] Manifesto for Agile Software Development, 2001, http://www.agilemanifesto.org.

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

[9] Mike Cohn. Agile Estimating and Planning. Prentice Hall, 2006.

Desenvolvimento de Software Centrado em Arquitetura

Vinicius Lourenço de Sousa
vinicius.lourenco.sousa@gmail.com
Atua no ramo de desenvolvimento de software há mais de 10 anos, é autor de diversos artigos publicados pelas revistas ClubeDelphi e SQL Magazine. É Graduado em Tecnologia da Informação pela ABEU Faculdades Integradas e Pós-Graduado em Análise, Projeto e Gerência de Sistemas pela PUC-RJ, IBM Certified: Especialista Rational Unified Process e instrutor de UML, Análise OO e Java. Atualmente trabalha na CPM Braxis como Especialista nas áreas de arquitetura, especificação de requisitos, levantamento e modelagem de processo de negócio e projetista de software em soluções com componentes de negócio, SOA e BPM. Mantém seu blog pessoal em http://viniciuslourenco.wordpress.com.

De que se trata o artigo:

A importância de desenvolver uma arquitetura de software robusta e que funcione como base em reuso em larga escala, e suas visões arquiteturais para demonstrar a compreensão e a organização do software para os envolvidos do projeto.

Para que serve:

Fornecer um meio de tornar partes do software reutilizável, flexíveis a ponto de responder as mudanças do software rapidamente e encapsular as dependências do sistema. Também serve como forma de comunicação entre os envolvidos do projeto, sejam eles gerentes, desenvolvedores ou clientes.

Em que situação o tema é útil:

Útil para o desenvolvimento de software quando está se pensando em reuso, flexibilidade, escalabilidade e quando queremos comunicar como está a organização do software em termos de camadas, classes, componentes e etc.

Atualmente, o cenário de desenvolvimento de software tem se mostrado em uma tentativa fervorosa de aproximar a parte tecnológica da parte de negócio. Isto se deve à grande demanda dos sistemas estarem cada vez mais integrados e serem flexíveis às mudanças no negócio das empresas em geral. Cada vez mais ouvimos falar sobre sistemas construídos sob a ótica de disponibilizar serviços corporativos e controlar o processo de negócio. Estou falando de SOA (Arquitetura Orientada a Serviços) e BPM (Gerenciamento de Processo de Negócio). Estas duas siglas se tornaram obrigatórias no vocabulário de equipes de desenvolvimento, gerentes de projetos e analistas de áreas de negócio dos mais variados tipos de segmentos de mercado.

Com o uso de SOA e BPM, o desenvolvimento de software não só alcançou um novo nível como também teve que adotar novas estratégias para a construção do software. Dessa forma, foi possível obter o máximo de reaproveitamento de sistemas legados para serem integrados com os novos sistemas e de reuso de regras de negócios através dos componentes e serviços de negócios. Enfim, os sistemas estão

ficando cada vez mais aderentes ao negócio e respondendo de forma mais rápida às mudanças exigidas pelo mercado.

Entretanto, para se alcançar um nível de desenvolvimento desse tipo, com reuso ao extremo e bastante flexibilidade, é preciso que a equipe de desenvolvimento crie uma arquitetura adequada que possa exatamente servir como pilar para o reuso e flexibilidade às mudanças, ou seja, o desenvolvimento deve ser centrado em arquitetura. Neste artigo apresentaremos as características do desenvolvimento de software centrado em arquitetura, como suas práticas, benefícios e visões arquiteturais de uma maneira ampla, sem aprofundar muito na parte técnica.

O que é Desenvolvimento Centrado em Arquitetura?

Um desenvolvimento de software centrado em arquitetura não diz respeito apenas ao código fonte da aplicação ou qual linguagem de programação será utilizada. Podemos dizer que a arquitetura é o "universo" em que o ciclo de vida do software vive, ou seja, em um desenvolvimento centrado em arquitetura teremos o código que é o artefato mais primário, a divisão entre as camadas arquiteturais, a divisão entre os componentes de negócio, regras de integração entre as camadas, componentes e entre os sistemas (novos e legados). É importante ter em mente que a arquitetura de um sistema não serve apenas para a codificação, mas também para o direcionamento de todo o desenvolvimento do software. Tanto que existem tipos de arquitetura e perfis de arquitetos diferenciados para cada foco e momento do projeto, como apresentado abaixo.

Arquitetura de Sistemas

Responsável pela parte tecnológica. É a representação de um sistema na qual existe um mapeamento de funcionalidades sobre componentes de software e hardware, um mapeamento sobre arquitetura de hardware e software, e interações humanas com esses componentes. Nesta arquitetura, o perfil *Arquiteto de Sistemas* além de conhecer profundamente a linguagem de programação, tem como foco a integração entre os frameworks que o sistema utiliza, sejam eles já existentes no mercado ou criado pela própria equipe de desenvolvimento, e a utilização de qualquer solução tecnológica, como por exemplo: EJB, JPA, JAAS, Struts, Spring, Corba e etc.

Arquitetura de Software

Responsável pela integração entre a parte de negócio e a parte tecnológica. É a representação das estruturas do sistema, na qual consistem componentes de software, as propriedades visíveis externamente desses componentes e o relacionamento entre eles. O perfil *Arquiteto de Software* tem como foco entender o modelo de negócio e transformá-lo em projeto de software, isto é, pensar em integração das camadas, integração entre sistemas, reuso, componentização, procurando minimizar os riscos tecnológicos. Em muitos casos, o *Arquiteto de Software* e o *Arquiteto de Sistemas* podem ser a mesma pessoa.

Arquitetura de Negócio

Totalmente voltada para a parte de negócio da empresa. O perfil *Arquiteto de negócio* tem como foco as seguintes atividades:

- Avaliar a situação da organização onde os sistemas serão entregues;
- Entender os requisitos dos usuários, suas estratégias e seus objetivos de negócio;
- Facilitar a modelagem do processo de negócio da organização;
- Entender o lado técnico do conjunto de soluções.

Em muitos casos o *Arquiteto de Negócio* e o *Analista de Negócio* (ler **Nota 1**) podem ser a mesma pessoa.

Benefícios do Desenvolvimento Centrado em Arquitetura

Abaixo veremos alguns benefícios que temos com o uso dessa abordagem:

• É uma efetiva base para reuso em larga escala

Com uma arquitetura bem robusta, estável e com interfaces bem definidas, podemos ter uma base corporativa onde todos os projetos podem utilizar dessa arquitetura, ou seja, as complexidades de TI estarão todas transparentes para os clientes dessa arquitetura.

• Melhorar a extensibilidade do sistema

Um sistema centrado em arquitetura tem como grande característica a capacidade de adicionar novos componentes e com isso crescer de forma fácil e rápida sem perder a coesão e o baixo acoplamento, mantendo a complexidade do sistema controlada.

• Encapsula as dependências do sistema

Encapsulando suas dependências, o sistema torna-se muito menos acoplado, mais coeso e com isso com maior grau de reuso. Em resumo, o sistema terá uma compreensão bem melhor e em caso de manutenção também será de extrema facilidade as alterações sem os impactos que poderiam ocorrer com tudo acoplado.

Arquitetura em Camadas

Uma das grandes características de uma boa arquitetura no desenvolvimento de software é a divisão da aplicação em camadas. A divisão em camadas pode proporcionar ao software um alto grau de escalabilidade, flexibilidade, alta coesão, baixo acoplamento e com isso teremos reuso dos componentes do software. Na arquitetura das aplicações existem dois tipos de camadas:

Camada Física (Tier)

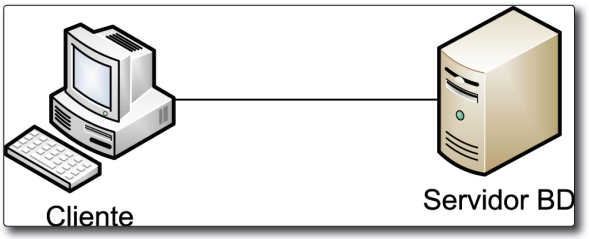
Este tipo de camada é representado pela parte física ou estrutural dos componentes de infra-estrutura da arquitetura que vão desde o hardware, o sistema operacional, os serviços, até os servidores, isto é, são os nós de processamento distribuídos no parque de TI da corporação. Ao longo dos anos foram utilizadas diversas arquiteturas da camada física, onde o intuito era criar uma infra-estrutura capaz de suportar toda a complexidade (regras de negócio, componentes distribuídos, plataformas heterogêneas e etc.) que os sistemas tinham. Veja abaixo os tipos de arquiteturas com camadas físicas existentes:

2 Camadas

Arquitetura de 2 Camadas (também conhecida como Cliente/Servidor) foi muito utilizada, onde tínhamos o sistema sendo executado no lado cliente com o executável da aplicação e regras de negócio e no lado servidor ficava o banco de dados que

Nota 1. Analista de negócio

O Analista de Negócio lidera e coordena o esforço de modelagem de negócio da empresa do cliente. Ele deve ser um facilitador e ter uma excelente habilidade de comunicação, conhecimento do domínio do negócio e estar preparado para avaliar a situação da organização alvo onde o sistema será entregue; entender os clientes e seus requisitos, suas estratégias e seus objetivos; e executar a análise de custo/benefício para cada mudança sugerida na organização alvo.



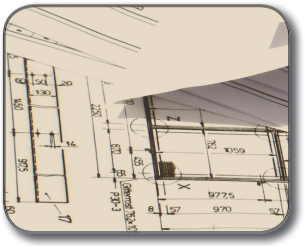


Figura 1. Arquitetura 2 camadas.

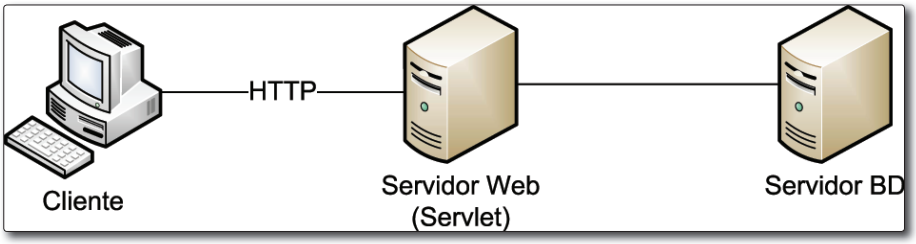


Figura 2. Arquitetura 3 camadas de um sistema Web.

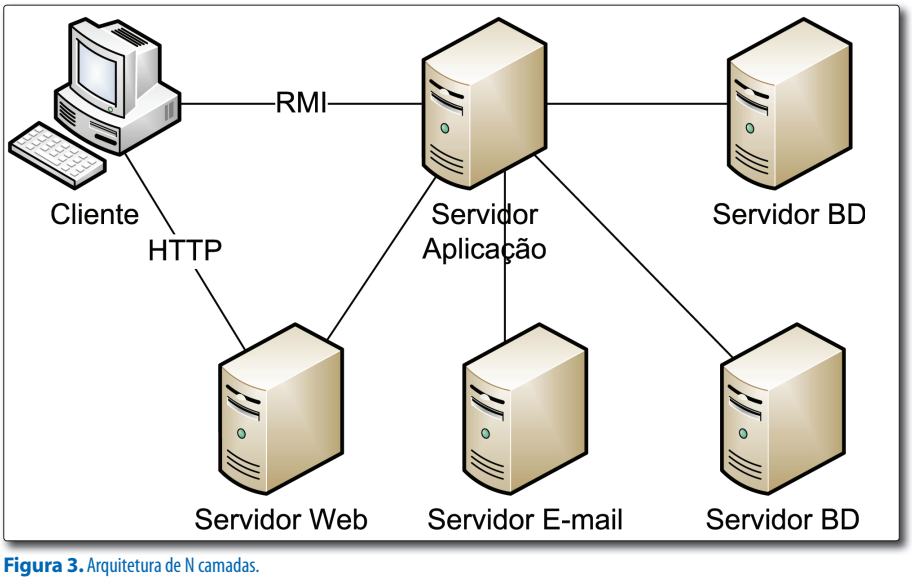


Figura 3. Arquitetura de N camadas.

também podia conter regras de negócio através de stored procedures, triggers e functions. Veja a **Figura 1**.

3 Camadas

Arquitetura utilizada para resolver problemas da arquitetura de 2 camadas como, por exemplo, a de regras de negócio duplicadas. Entre o lado cliente e o lado servidor (banco de dados) entrou um servidor de aplicação onde ficam todas as regras de negócio compartilhadas para os clientes. Esse tipo de arquitetura hoje em dia é muito utilizado em sistemas Web, onde a camada intermediária pode ser representada por um container Web contendo um servlet no caso de sistemas em Java. Veja a **Figura 2**.

N Camadas

Arquitetura utilizada para aplicações que exigem um alto grau de distribuição de seus componentes internos. Nesta arquitetura temos como nó de processamento o cliente, o servidor de aplicação onde ficam as regras de negócio, um servidor Web contendo as páginas Web, um servidor de banco de dados e outros servidores e/ou serviços como, por exemplo, serviços de e-mail, firewall, acesso ao mainframe e componentes distribuídos por toda a rede corporativa. Arquitetura de N camadas é amplamente utilizada hoje em dia, pois nas grandes corporações sempre existe integração entre sistemas/componentes, servidores de bancos replicados e vários serviços que estão inseridos no parque de TI da corporação e que são utilizados pelas aplicações. Veja a **Figura 3**.

Camada Lógica (Layer)

Este tipo de camada é conhecido como um padrão arquitetural (Padrão Layer) que serve para organizar e separar as responsabilidades (separação de conceitos) dos componentes internos de uma aplicação. É muito importante dividir as aplicações entre camadas lógicas, e estabelecer regras de interações entre as camadas. Se o código não está claramente separado entre camadas, logo se tornará muito difícil gerenciar mudanças no mesmo. Por existir dentro da aplicação, as camadas lógicas também existem dentro das camadas físicas e podem estar distribuídas por cada nó de processamento. Veja na **Figura 4** os tipos de camadas lógicas existentes.

Na **Figura 4** vemos quatro tipos de camadas lógicas que são as camadas usadas no Desenvolvimento Dirigido ao Domínio (DDD) criado por *Eric Evans* no livro *Projeto Dirigido ao Domínio*. Essas camadas devem ser independentes o máximo possível, pois como se trata de camadas lógicas, cada uma pode estar em máquinas diferentes, isto é, em camadas físicas diferentes. Em algumas literaturas as camadas lógicas podem variar de nome e até mesmo de quantidade, como apresentado no livro *Padrões de Arquitetura de Aplicações Corporativas* (*Martin Fowler*), onde a camada de aplicação não existe.

Não entrarei em detalhes do por que das diferenças entre os autores, mas recomendo a leitura desses dois livros que são excelentes fontes de estudo.

Apresentação

Esta camada é responsável por conter as interfaces dos usuários e as regras inerentes a qualquer componente visual, regras de validação de input de dados e de formatação de dados. Em uma aplicação que disponibilize serviços Web (Web Services), esses serviços também pertencerão à camada de Apresentação, pois eles são considerados interfaces entre sistemas. A

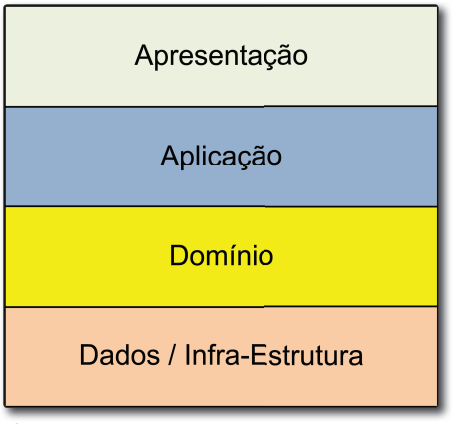


Figura 4. Camadas lógicas arquiteturais.

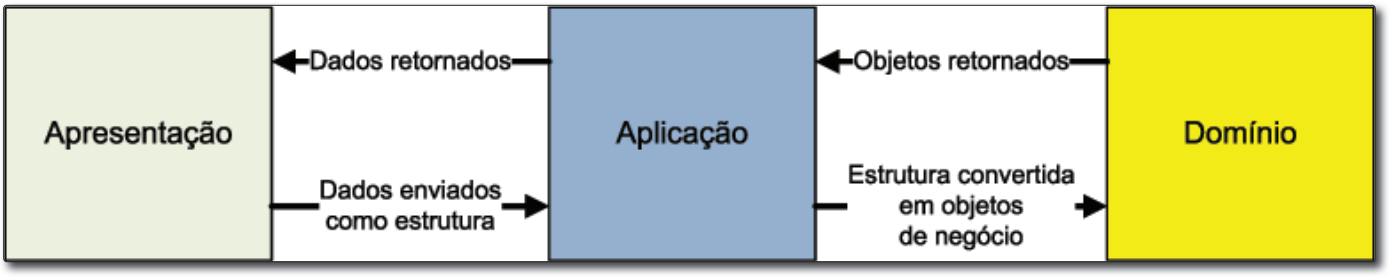


Figura 5. Transformação de estrutura de dados em objetos na camada de Aplicação.

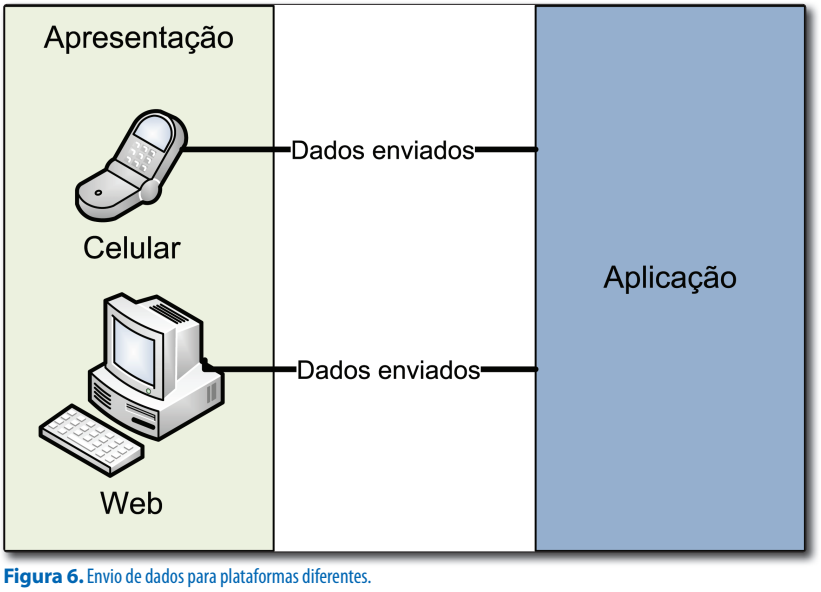


Figura 6. Envio de dados para plataformas diferentes.

camada de Apresentação envia e recebe dados da camada de Aplicação.

Aplicação

Esta camada é responsável por disponibilizar os dados para a camada de Apresentação e enviar os dados para a camada de Domínio. A camada de Aplicação também pode fazer alguns tratamentos nos dados caso seja necessário. Podemos citar dois exemplos importantes da camada de Aplicação:

1. Efetuar a transformação dos dados recebidos da camada de Apresentação que em aplicações distribuídas vem na forma de estruturas "desorganizadas" (os chamados Transfer Objects) para os objetos de domínio que representam o negócio da aplicação. Veja a **Figura 5**.
2. Em aplicações onde os dados devem ser disponibilizados em plataformas heterogêneas ao mesmo tempo, como na Web e no Celular. Sabemos que nos dispositivos móveis existe uma limitação de dados que podem ser trafegados, portanto, a camada de Aplicação tem como objetivo filtrar a quantidade de dados enviados por causa do limite de tráfego que a plataforma suporta. Veja a **Figura 6**.

Domínio

Esta camada é responsável por conter informações sobre o domínio do negócio do sistema, o estado dos objetos de negócio e suas regras, ou seja, o coração do negócio do software. Nesta camada é onde criamos nossos componentes de negócios através de decomposição funcional para encapsular todas as regras de negócio.

Acesso a Dados (Infra-Estrutura)

Esta camada é responsável por conter toda a lógica para acesso aos dados, sejam eles via banco de dados, arquivos textos, XML. É através desta camada que controlamos transações de banco de dados, executamos os SQL's. Esta camada também é responsável pelas integrações com outros sistemas, componentes e serviços, por exemplo, acessar uma DLL, um mainframe, um EJB ou Web Service de outra área da corporação ou de um parceiro do cliente ou até mesmo um servidor de e-mail ou de FTP.

Visões Arquiteturais

Quando desenvolvemos software pensando em arquitetura devemos sempre estar atentos para as visões que nossa arquitetura oferece para que o sistema possa ser compreendido com maior exatidão.

Uma visão arquitetural é uma janela para o sistema em uma perspectiva particular que serve como comunicação entre os envolvidos no sistema e é expressa em texto e diagramas UML. As visões arquiteturais são muito utilizadas no processo unificado de desenvolvimento de software, mais especificamente pelo Rational Unified

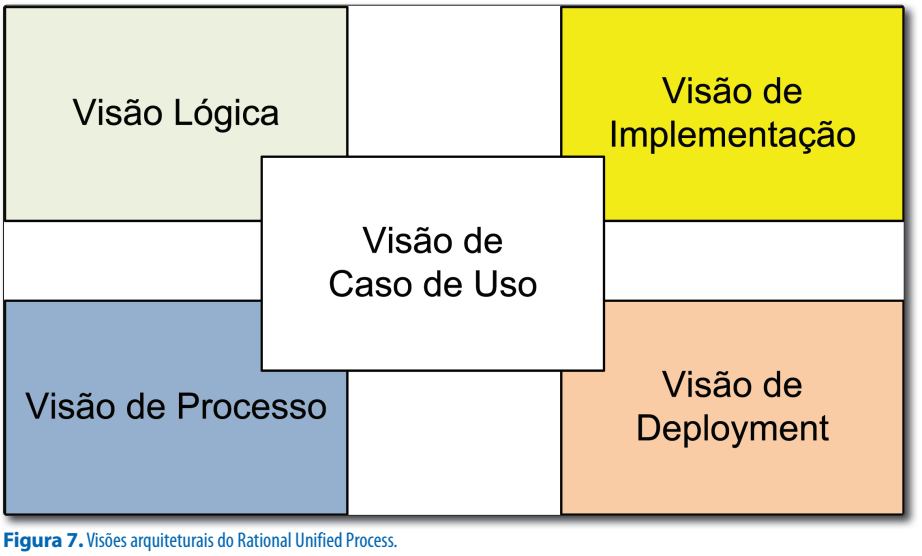


Figura 7. Visões arquiteturais do Rational Unified Process.

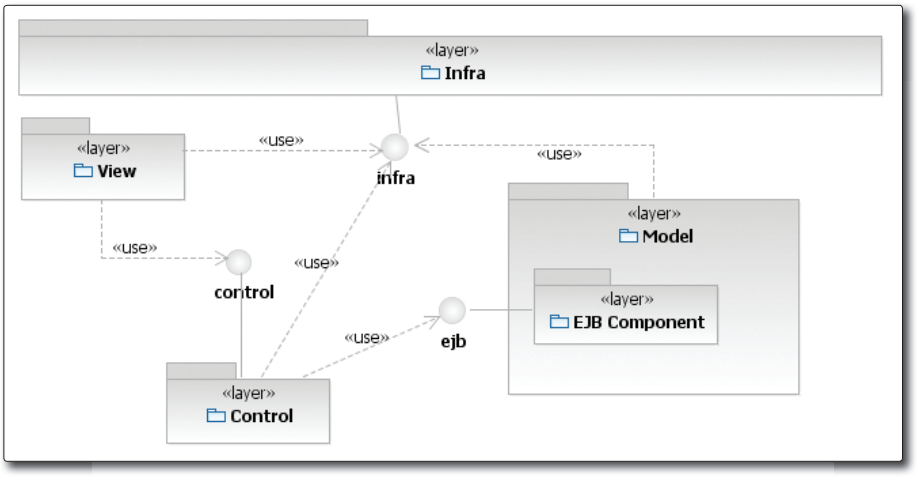


Figura 8. Visão Lógica organizada em camadas, subsistemas e interfaces.

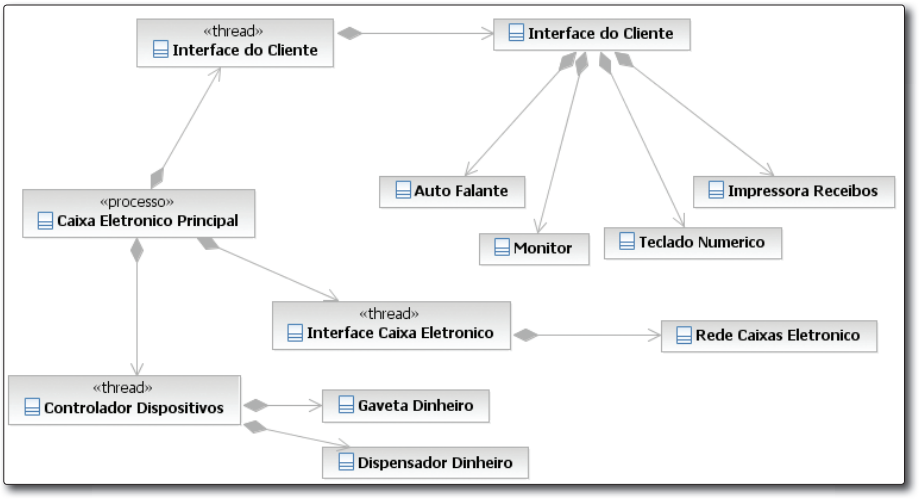


Figura 9. Visão de Processo com a organização de um sistema bancário.

Process. No RUP, as visões arquiteturais são conhecidas por *Modelo de Visão Arquitetural 4+1*, sendo todas elas dirigidas pela visão de caso de uso. As visões arquiteturais ficam documentadas no artefato *Documento de Arquitetura de Software*. Veja a **Figura 7**.

Como podemos perceber na **Figura 7**, as visões arquiteturais Lógica, de Processo, de Implementação e Deployment são dirigidas pela visão de caso de uso que contem toda a solução sistêmica do software (veja os artigos Desenvolvimento de Software Dirigido por Caso de Uso nas edições 2, 3 e 4 da Engenharia de Software Magazine). Porém, existem outras visões que não são muito utilizadas na maioria dos projetos, que são a visão de dados, visão de segurança e a visão de desenvolvimento. Veja abaixo a descrição de cada visão.

Visão Lógica

É responsável pela organização conceitual do software em termos de camadas, subsistemas, pacotes, frameworks, classes e interfaces mais importantes. Mostra a realização dos cenários dos casos de uso que ilustram aspectos chaves do sistema (**Figura 8**).

Na **Figura 8** podemos ver o exemplo de um sistema web feito na tecnologia Java EE organizado em camadas lógicas representado pelos pacotes da UML. Essa arquitetura é uma arquitetura padrão onde o sistema está divido nas camadas View (contendo as páginas web), Control (contendo os actions no caso do struts), Model (contendo as classes de negócio, classes de persistência e etc.), EJB Component que é uma subcamada da camada Model que representa o uso de EJB no sistema e, por fim, a camada Infra, onde ficam as classes genéricas, frameworks utilizados e etc. Como essa é uma organização conceitual não nos preocupamos nos detalhes técnicos como nome dos pacotes, nome de classes e etc. e sim na responsabilidade de cada camada e suas integrações.

Visão de Processo

É responsável por mapear os processos e as threads, suas responsabilidades, colaborações e a alocação dos elementos da visão lógica (camadas, subsistemas, classes, etc.) neles (**Figura 9**).

Visão de Deployment

É responsável pelo deployment (instalação) físico dos processos e componentes para os nós de processamentos e a configuração física da rede entre os nós. É nesta visão que mapeamos como a arquitetura estará na camada física (tier) e como cada nó de processamento estará configurado (**Figura 10**).

Na **Figura 10** temos um exemplo de configuração física de deployment de um sistema feito em Java EE. As caixinhas representam os nós de processamento e os componentes que serão executados. Temos como exemplo um nó de processamento chamado *Application Server* que é o servidor de aplicação para sistemas em Java. Esse nó possui dois atributos que são o nome (IBM Websphere 5.1) e o sistema operacional onde ele está sendo executado (Windows Server 2003). Esses atributos são informações importantes, pois através deles podemos conhecer as configurações do software tanto para instalação quanto para manutenção. O mesmo vale para os nós de processamento *SGBD*, *FTP Server* e *Mainframe*. Dentro do nó de processamento *Application Server* existem outros nós que representam camadas do servidor de aplicação ou arquivos do sistema.

O servidor de aplicação é dividido em Web Container que é onde ficam as páginas HTML, JSP ou aplicativos clientes como Web Services e JOB Schedules, e em EJB Container que é onde ficam os EJB's do sistema com todas as interfaces clientes e suas implementações. Vemos que no Web Container existe o atributo que é o nome do software que será executado (Quartz). O Quartz irá executar um EJB através da sua interface cliente (localizado no arquivo *sistema-ejbCliente.jar*) para a bean do EJB (localizado no arquivo *sistema-ejb.jar*) que é a implementação da interface cliente do EJB. Neste diagrama também é importante dizermos o protocolo de comunicação entre os nós de processamento, por exemplo, o nó *Application Server* se comunica com o nó *SGBD* através do JDBC e com o nó *Mainframe* através do CICS, que é um software para execução de serviços (COBOL) feitos na plataforma alta. Outro recurso interessante é que podemos dizer

o que está trafegando nesta comunicação entre os nós de processamento, como por exemplo, entre o *FTP Server* e o *Application Server* está trafegando pela rede um arquivo compactado chamado *arquivo.zip*.

Visão de Dados

É responsável pela visão abrangente dos fluxos dos dados, *schemas* dos dados persistentes, mecanismos de mapeamento dos objetos de negócio para dados de persistência (usualmente em bancos relacionais), stored procedures e triggers.

Visão de Segurança

É responsável pela visão abrangente dos schemas de segurança e os pontos dentro da arquitetura na qual a segurança é aplicada, tais como autenticação HTTP, autenticação de banco de dados, autenticação de usuários e outros.

Visão de Implementação

É responsável pelo modelo de implementação que diferente dos outros modelos que possuem diagramas e/ou textos, é o próprio código fonte e seus executáveis. Neste modelo todo o sistema é organizado para identificarmos sua estrutura, por exemplo: pacotes java, componentes, arquivos JAR e EAR e etc. (**Figura 11**).

Na **Figura 11** vemos os componentes que são acionados quando um Caixa Eletrônico, que também é um componente, é utilizado. Temos o componente que valida se o cartão do banco é válido, temos o leitor de cartões que irá ler os dados de agência, conta e dados pessoais do cliente a partir do próprio cartão e temos o componente que é o dispensador de dinheiro para quando o cliente for efetuar um saque.

Visão de Desenvolvimento

É responsável por resumir informações aos desenvolvedores para que eles possam conhecer as configurações do ambiente de desenvolvimento. Por exemplo, como todos os arquivos estão organizados em termos de diretórios e por quê? Como efetuar um build e executar os testes automatizados e qual o controle de versão utilizado.

Visão de Caso de Uso

Além de dirigir quase todas as demais visões, essa visão é responsável por mapear todos os casos de uso arquiteturalmente significantes e seus requisitos não funcionais.

Conclusão

Neste artigo foi apresentada a grande importância da arquitetura no desenvolvimento de software e como ela pode contribuir nas organizações em termos de formar uma grande base de reuso corporativo para os vários projetos de software.

Vimos também que através das camadas arquiteturais podemos dividir o software tanto fisicamente quanto logicamente por questões de desempenho, organização, reuso e etc. Vimos que a arquitetura nos oferece várias visões que servem para demonstrarmos uma determinada parte do software em termos de configuração, instalação, organização e componentização como uma forma de comunicação entre os usuários, equipe de desenvolvimento, equipe de infra-estrutura e demais envolvidos no projeto para entendermos o software sem ambigüidades. O desenvolvimento de software centrado em arquitetura tem muito mais a ser discutido, porém isso ficará para outros artigos ●

Links

RUP: Rational Unified Process 7.0
http://www-306.ibm.com/software/awdtools/rup/

Bibliografia

Padrões de Arquitetura de Aplicações Corporativas – Martin Fowler

Domain Driven Design – Eric Evans

Utilizando UML e Padrões – Larman, Craig

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

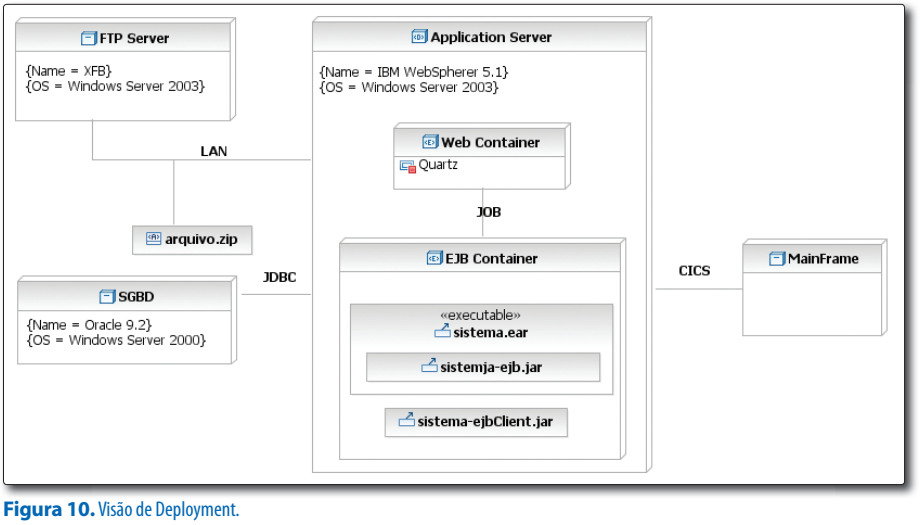


Figura 10. Visão de Deployment.

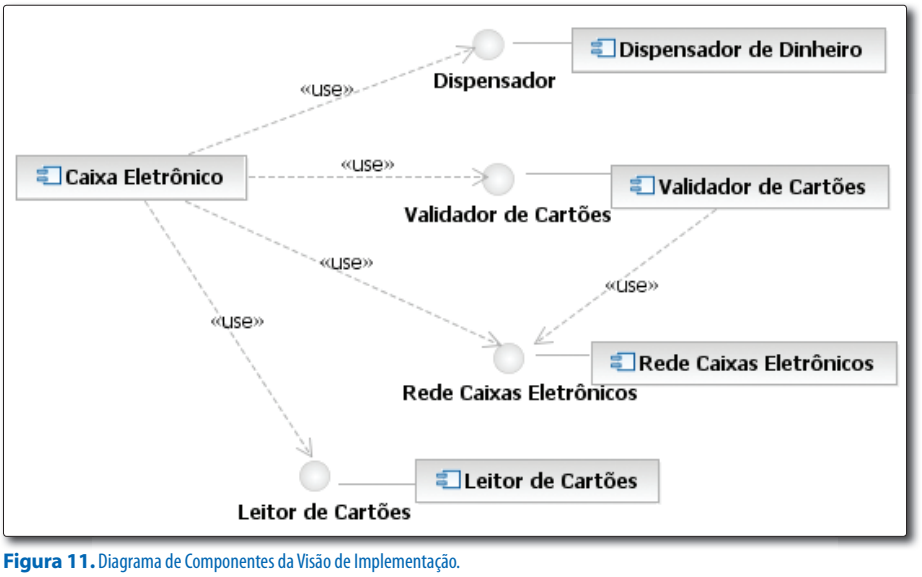


Figura 11. Diagrama de Componentes da Visão de Implementação.

Conceitos de orientação a objetos e UML

Entendendo o paradigma atual de desenvolvimento de sistemas

De que se trata o artigo:

Este artigo aborda a evolução do desenvolvimento de sistemas chegando aos dias de hoje com o paradigma da orientação a objetos. Em seguida são apresentados os conceitos da orientação a objetos, sua aplicabilidade em diversas fases do desenvolvimento de sistemas, e conclui com a apresentação da UML como modelo utilizado para desenvolvimento de sistemas OO.

Para que serve:

Fornecer aos desenvolvedores ou estudantes da área de sistemas a base necessária ao contexto de desenvolvimento atual – o paradigma de orientação a objetos.

Em que situação o tema é útil:

Atualmente há uma disseminação de sistemas desenvolvidos sob o paradigma orientado a objetos, sem que alguns desenvolvedores tenham uma completa visão da importância de toda a base de conceitos OO e da modelagem em UML.

Este artigo inicia mostrando como evoluímos até o paradigma atual da orientação a objetos. Em seguida, conceituaremos a base da orientação a objetos, com sua demonstração por meio de exemplos. Será apresentada uma visão geral da aplicabilidade em diversas fases do desenvolvimento de sistemas: levantamento, análise, projeto de banco de dados e implementação. E por fim, evidenciaremos a proposta da UML como linguagem de modelagem, com a apresentação dos diagramas da versão atual.

O começo de tudo

A história da computação teve início na necessidade do homem em conseguir realizar cálculos. O caminho foi longo, iniciado com o ábaco, muitos anos antes da era cristã. A primeira máquina de calcular que apenas somava e subtraía vem surgir apenas em 1642, desenvolvida por Blaise Pascal. Em 1694, Gottfried Von Leibniz constrói a primeira calculadora que podia executar as quatro operações básicas, e em 1822, o matemático inglês Charles Babbage estabelecia os princípios do funcionamento dos computadores eletrônicos no projeto

Ana Cristina Melo

informatica@anacristinamelo.com.br

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

de sua máquina diferencial, capaz de realizar os cálculos necessários para elaborar uma tabela de logaritmos. A partir daí, outras invenções abriram caminhos para o que temos hoje. O marco inicial se dá com o primeiro computador eletrônico, o ENIAC (*Eletrical Numerical Integrator and Calculator*), surgido em 1945, e pesando cerca de 30 toneladas. Até hoje os computadores ainda utilizam a arquitetura proposta por Von Neumann. Em 1951, surgia o primeiro computador fabricado comercialmente: o UNIVAC I, usado no censo americano por 12 anos seguidos.

A partir da década de 40, descobre-se a importância da computação, e essa passa a fazer parte da nossa história. Contudo, numa primeira fase ninguém pensava em software. Os esforços estavam voltados à evolução do hardware, buscando-se reduzir os problemas das primeiras máquinas. Assim, da primeira geração de computadores à válvula, passamos para a segunda geração, utilizando transistores.

A primeira linguagem de programação surgida foi a linguagem de máquina, na década de 50 — o *Assembly*. Nesse momento, a preocupação era restrita aos comandos, nem se pensava em análise, muito menos em modelagem de requisitos. A partir de então, surgem as linguagens de alto nível, como *Fortran*, *Algol* e *Cobol*.

Um rápido aumento na complexidade das demandas por software e a falta de técnicas para definição de novos sistemas culminaram em diversos problemas, entre eles: estouro de orçamento e prazo, softwares de baixa qualidade, requisitos não atendidos e código de manutenção difícil. Estava definida a **crise de software**. A solução para contornar a crise veio com o conceito da **Engenharia de Software**, em 1968. Objetivava-se trazer os princípios da Engenharia, com todo o seu planejamento e modelagem, para se resolver os problemas da área ainda imatura. No mesmo ano de 68, Dijkstra escreve sobre a programação estruturada; tinha início o marco do primeiro paradigma de desenvolvimento de sistemas.

Mas enquanto novas linguagens surgiam —Pascal, C —, ainda não se tinha a definição de uma metodologia de desenvolvimento de sistemas, até 1978, quando Tom DeMarco escreve seu livro sobre análise estruturada. Contudo, a dificuldade de manutenção dos diversos modelos gerados na fase de análise e projeto, fez com que se percebesse, nas duas décadas seguintes, que esse paradigma ainda não era a solução para a velha crise de software.

Naturalmente, logo se pensa que a orientação a objetos surgiu após a análise estruturada, como uma evolução dessa metodologia, buscando-se as soluções necessárias para um projeto confiável e de manutenção fácil. É correto imaginar que o paradigma da orientação a objetos tornou-se a solução para acompanhar a demanda cada vez maior e freqüente do mercado, mas o erro está no posicionamento histórico do conceito de orientação a objetos, pois este surgiu muito antes do conceito da programação estruturada.

Em 1962, Ole-Johan Dahl e Kristen Nygaard criaram uma linguagem chamada *Simula*, baseada na linguagem *Algol 60*. O diferencial residia em seu objetivo: permitir o projeto de simulações. Surgia a primeira linguagem orientada a objetos, apresentando os conceitos de classe e herança. Essa foi a semente que inspirou o desenvolvimento de uma nova linguagem, a primeira totalmente orientada a objetos —o *SmallTalk*. Nela, não existem tipos primitivos, tudo é representado em forma de objeto: números, caracteres etc. Disponibilizada ao público no início dos anos 80, *SmallTalk* solidificou para a comunidade os conceitos de classe, objeto, atributo, método, encapsulamento, herança e mensagem. A partir daí, novas linguagens surgiram, como o *C++* (versão OO da linguagem C), *Object Pascal* (versão OO do Pascal), *Eiffel* e *Java* (criado a partir do C++).

Conceitos fundamentais de orientação a objetos

Os conceitos da orientação a objetos surgiram da necessidade em se enfatizar unidades discretas, e obter a reutilização de código, mantendo-se a qualidade do software. O núcleo do pensamento OO predomina num foco sobre os dados, em vez dos processos, compondo módulos auto-suficientes — os objetos —, encerrando em sua estrutura todo o conhecimento dos dados e dos processos para manipulação desses dados.

O que se obtém de principal na modelagem orientada a objetos é a possibilidade de se abstrair diretamente os conceitos do mundo real, sem subterfúgios para se chegar à solução computacional.

Os conceitos fundamentais de orientação a objetos são o contrato que estabelece toda e qualquer implementação que se diga OO. Sendo assim, se um desses conceitos não for atendido, não podemos afirmar que determinada tecnologia possa ser nomeada como orientada a objetos. Isso aconteceu com a linguagem *Visual Basic*, que antes da sua versão .net não implementava todos os conceitos de orientação a objetos.

Vejamos então esses conceitos:

Objeto: um objeto é qualquer coisa existente no mundo real, em formato concreto ou abstrato, ou seja, que exista fisicamente ou apenas conceitualmente, e o qual se pode caracterizar e identificar comportamentos.

Podemos afirmar que um objeto é uma caixa-preta que recebe e envia mensagens, ou seja, num sistema orientado a objetos, os objetos trocam informações por meio de mensagens.

Num sistema orientado a objetos não modelamos apenas objetos de negócio. Muitas vezes, de acordo com a arquitetura utilizada, modelamos objetos computacionais, visuais ou não.

Exemplo: ao levantarmos os requisitos para informatizar uma concessionária, encontraremos o *objeto automovel* (físico), da mesma forma que podemos modelar o *objeto venda* (conceitual).

Atributo: as características associadas aos objetos são chamadas de atributos. Para os objetos de negócio, é comum usarmos o conceito de atributo. Para os objetos visuais, utilizamos o conceito de propriedade.

Exemplo: atributos da classe Cargo: *descricao* e *salario*. Atributos da classe Automovel: *modelo, cor, numeroPortas, ano, placa etc*.

Operação x Método: o comportamento dos objetos é representado pelas operações. Contudo, a operação para um objeto representa apenas a definição do serviço que ele oferece a outras estruturas. Quando tratamos da implementação dessa operação, ou seja, da sua representação em código, estamos nos referindo ao seu **método**. Os métodos de uma classe manipulam somente as estruturas de dados daquela classe, ou seja, para se ter acesso aos dados de outra classe, isso deve ser feito por meio de mensagens.

Exemplo: operações da classe Cargo: *cadastrar() e reajustarSalario(percentual: float)*.

Ao modelarmos uma classe precisamos sempre considerar o contexto. Se não fosse isso, bastaria um famoso metodologista publicar as soluções para todas as classes de negócio.

Desta forma, não teremos necessariamente os mesmos atributos e operações para a classe aluno, modelada num sistema acadêmico de uma escola de nível médio, e a mesma classe aluno, modelada num sistema acadêmico de uma Universidade. Não é garantia termos os mesmos atributos e operações nem se considerarmos dois sistemas acadêmicos modelados para distintas Universidades.

Exemplo: Vamos tomar por base a *classe Automovel*. Se estivermos no contexto de uma concessionária, teremos operações como: *cadastrar*, *alterarProprietario* etc. Em contrapartida, se estivermos no contexto de um simulador para auto-escola, seu comportamento deve reproduzir o objeto real, com operações como: *ligar*, *aumentar marcha*, *reduzir marcha*, *acelerar* etc.

Estado: são os valores assumidos pelos atributos de um objeto.

Exemplo: em um determinado *objeto cargo*, o estado do seu atributo *salario* é o valor R$ 5000,00.

Mensagem: é a solicitação que um objeto faz a outro, invocando a execução de um determinado serviço. Por exemplo, para que um objeto possa calcular a folha de pagamento, ele precisa saber o salário de cada funcionário. Assim, ele passa uma mensagem ao *objeto cargo*, solicitando a execução do serviço *obterSalario*, que nada mais é do que uma operação do *objeto cargo*.

Encapsulamento: o conceito de encapsulamento nos remete ao fato de que a utilização de um sistema orientado a objetos não deve depender de sua implementação interna, e sim de sua interface. Isso garante que os atributos e os métodos estarão protegidos, só podendo ser acessados pela interface disponibilizada pelo objeto, ou seja, sua lista de serviços. Essa proteção garante que os usuários de uma classe não sejam influenciados por quaisquer alterações feitas em seu conteúdo.

Na prática, imagine uma *classe Produto* que possua a operação *obterPrecoVenda: float*. Essa operação é pública, tornando-se um serviço disponibilizado a outras classes. Em qualquer rotina que se queira mostrar o preço de venda de um produto, basta instanciar um *objeto Produto*, e passar uma mensagem para executar o serviço, ou seja, chamar a execução da operação *obterPrecoVenda*.

Suponha que as rotinas A, B, C e D usem esse serviço e que até ontem esse valor de venda fosse obtido apenas com um cálculo simples:

```
precoVenda = precoCusto * (1 + lucro/100)
```

Esse cálculo está na implementação de uma operação, ou seja, é o seu método, e fica encapsulado (escondido).

Suponha agora que hoje foi colocada uma nova versão da *classe Produto*, na qual esse cálculo passa a ser:

```
precoVenda = precoCusto * (1 + lucro/100)
precoVenda = precoVenda * (1 -
descontoMes/100)
```

As rotinas A, B, C e D receberão uma exceção em virtude da regra de cálculo ter sido alterada? Não, eu respondo. É transparente para essas rotinas (e precisa ser assim) que houve alteração na forma de cálculo do preço de venda. Se estivermos diante de um componente (por exemplo, uma *dll*), absolutamente nada será preciso fazer com essas rotinas. Se estivermos com as rotinas dentro do mesmo pacote que a classe, basta recompilar tudo.

Herança: ao refinarmos a modelagem de um sistema é comum encontrarmos características redundantes entre objetos. Essa redundância pode ser evitada pela separação dos atributos e operações numa classe comum, identificada como superclasse. Essa classe comum (superclasse) passa a ser a generalização de outras classes que encerram em si apenas os atributos e operações específicos a cada uma.

Exemplo: Suponha que temos duas classes: *Cliente* e *Funcionario*. São atributos dessas classes: *Cliente* (*cpf*, *nome*, *dataNascimento*, *endereco*, *dataPrimeiraCompra*) e *Funcionario* (*cpf*, *nome*, *dataNascimento*, *endereco*, *dataAdmissao*, *funcao*). Imagine que existem operações que validam o CPF, retornam a idade de um cliente ou funcionário, ou formatam o endereço. E que todas essas funções apareçam em duplicidade nas classes Cliente e Funcionário. Numa situação dessas, o que se espera na orientação a objetos é que essa redundância seja eliminada com a herança. Assim, cria-se uma superclasse *Pessoa*, e a ela são atribuídos todos os atributos e operações comuns à *Cliente* e *Funcionario*. Sobram nas subclasses (*Cliente* e *Funcionario*) somente os atributos e operações específicos a cada um. Na prática, ao se usar uma subclasse, tem-se acesso a todos os elementos das classes ascendentes, como se tivessem sido criadas na própria classe filha. Veja na **Tabela 1** o resultado desse processo.

Quando uma subclasse possui mais do que uma superclasse é dito que temos uma **herança múltipla**.

A herança não precisa se limitar aos objetos de negócio. Pelo contrário, um dos maiores ganhos que nós temos com a orientação a objetos é a possibilidade de se estender todo esse conceito de utilização para todos os componentes do nosso sistema.

Exemplo: podemos criar uma classe para um relatório, com o logotipo da empresa, dados de identificação do cabeçalho e do rodapé, e a partir dessa classe, herdar todos os outros relatórios do sistema, particularizando em cada um, as características específicas. Imagine o esforço necessário para se trocar o logo e incluir o nome do usuário logado em todos os 1000 relatórios de um sistema: calculo uns dois minutos.

Polimorfismo: uma operação pode ter implementações diferentes em diversos pontos da hierarquia de classes. Isso significa que ela terá a mesma assinatura (mesmo nome, lista de argumentos e retorno), porém implementações diferentes (métodos). Isso é o polimorfismo (*poli* = muitas; *morphos* = forma).

Exemplo: Há uma operação *calcularSalario()* que pertence à *classe Funcionario*. Herdamos de *Funcionario* a *classe Professor*, o que resulta automaticamente na herança de *calcularSalario()*. Contudo o cálculo do salário de um professor não é o mesmo

Classes	Atributos
Antes da herança	
Cliente	cpf nome dataNascimento endereco dataPrimeiraCompra
Funcionario	cpf nome dataNascimento endereco dataAdmissao função
Após a herança	
Pessoa	cpf nome dataNascimento endereco
Cliente	dataPrimeiraCompra
Funcionário	dataAdmissao funcao

Tabela 1. Estrutura de classes antes e após remodelar com herança.

que o cálculo do salário de um funcionário, o que nos leva a ter a mesma operação, porém com métodos diferentes.

Camadas lógicas x camadas físicas

Há uma certa confusão quando se fala em camadas. A garantia de encapsulamento e autonomia do objeto está pautada no conceito das camadas lógicas.

As camadas lógicas (*layers*) são implementadas independentes da arquitetura do sistema, ou seja, de suas camadas físicas (*tiers*). Dessa forma, as três camadas continuarão a existir mesmo se estiverem implementadas num sistema cliente/servidor (*two-tier*) ou numa arquitetura web (*three-tier*). Podemos ter as três camadas lógicas implementadas até mesmo em um único computador, *stand-alone*, com base de dados local, mas ainda assim teremos três camadas lógicas distintas. A divisão das camadas lógicas é mostrada na **Tabela 2.**

A **camada de apresentação** normalmente é representada pela interface gráfica, podendo ser representada por qualquer outra forma de entrada de dados. Ao receber uma solicitação do usuário, essa camada se responsabiliza em convertê-lo em ações reconhecíveis pela próxima camada, a de negócio. Ao receber as respostas da camada de negócio, a camada de apresentação cuida em exibi-las para o usuário. Pode se apresentar de forma múltipla.

Exemplo: a rotina da camada de negócio que processa um contra-cheque pode ter saída em duas camadas de apresentação: em uma interface gráfica, para utilização interna, e em uma interface web, para acesso via internet.

A **camada de negócio** tem por objetivo receber as solicitações da camada de apresentação, e realizar à camada de persistência as solicitações que se façam necessárias, para processar e devolver o pedido original. Essa camada fica responsável por todo processamento, cálculo e validação inerente aos requisitos da aplicação.

A **camada de persistência** tem por objetivo receber as solicitações da camada de negócio, e para realizá-las, prepara e executa as *queries* necessárias para acesso ao banco de dados. Somente essa camada deve conhecer as fontes de dados e a estrutura das tabelas envolvidas com ela. Isso garante a não propagação de acesso aos dados, o que muitas vezes não só torna a manutenção difícil, como gera erros exponenciais, quando da abrangência errada de uma alteração.

Camada	Responsabilidade
Apresentação (*interface*)	Apresentação dos dados e interfaceamento entre o usuário e o sistema
Negócio (domínio)	Lógica do sistema, implementação das regras de negócio
Persistência	Acesso ao banco de dados

Tabela 2. Responsabilidades das camadas lógicas de um sistema orientado a objetos

Na **Figura 1** exemplificamos como as informações transitam entre as camadas. Suponha um usuário que via interface gráfica queira consultar o salário líquido de um funcionário cujo código é 100. Ele faz essa solicitação por meio da interface (*camada de apresentação*). Esta instancia um objeto Funcionário (*objFunc*) e faz a chamada do método *busca*, passando como parâmetro o código do funcionário (100). A solicitação é dirigida à *camada de negócio*, que nada pode fazer enquanto não obtiver os dados do banco de dados. Assim, ela repassa o objeto, ainda vazio, à *camada de persistência*, solicitando que seja feita a busca da persistência do referido objeto. A *camada de persistência*, por sua vez, a única que conhece a estrutura do banco, prepara uma *query* de consulta, para recuperar do BD os dados necessários. De posse dessas informações, cabe a essa camada associar o conteúdo retornado do BD aos atributos do objeto (*objFunc*) e, em seguida, devolvê-lo à camada chamadora. Contudo, para se chegar ao valor do salário líquido é preciso um cálculo, cuja regra pertence apenas à *camada de negócio*. Sendo assim, ao retornar à segunda camada, esta de posse do valor do salário bruto, já pode proceder ao cálculo que devolverá o valor do salário líquido. O objeto, agora completo, é então devolvido à *camada de apresentação*, onde seu conteúdo já pode ser exibido ao usuário.

Aplicabilidade da orientação a objetos

Ao se falar em orientação a objetos, pouco se imagina que a mesma tenha aplicabilidade em diversas fases do desenvolvimento de sistemas: levantamento de requisitos, análise, implementação e projeto de banco de dados.

Durante o **levantamento de requisitos** fomos beneficiados com o caso de uso. Você pode pensar imediatamente: "Mas o caso de uso é orientado a objetos?" E a resposta é direta: "Não!" A verdade é que ele não tem nenhuma referência à orientação a objetos, a não ser ter sido criada por Ivar Jacobson — um dos três autores da UML —, em sua metodologia orientada a objetos (*OOSE – Object Oriented System Engineering*). Seu formato e subdivisão em seções não possuem nenhuma similaridade com os conceitos de classe, herança etc, contudo sua importância faz coro com as vantagens do paradigma OO.

A Engenharia de Requisitos se preocupa desde a identificação do requisito até sua modelagem e especificação. Nas fases de

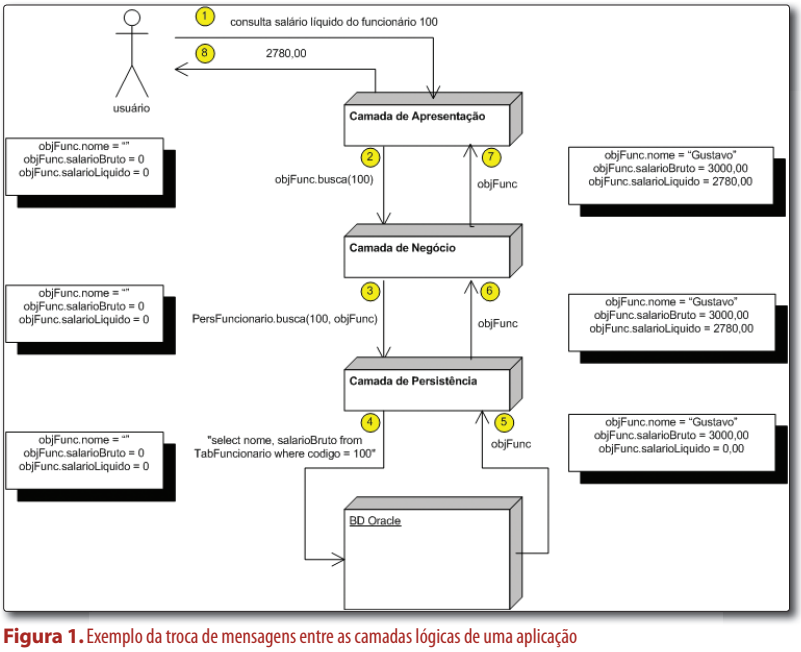


Figura 1. Exemplo da troca de mensagens entre as camadas lógicas de uma aplicação

modelagem e especificação é que o caso de uso vem a ser uma poderosa ferramenta, pois nos permite a comunicação com o usuário, sem que tenhamos a superficialidade de um DFD, ou o excesso de técnica de um pseudo-código.

Dentro de um sistema orientado a objetos, o caso de uso se torna a base para todo o processo de desenvolvimento, visto que a partir do mesmo é gerada a primeira versão do modelo de classes, a identificação das trocas de mensagens entre objetos, por meio da modelagem de um diagrama de seqüências, e a geração dos casos de teste que irão garantir a qualidade do produto final.

Numa visão de mais alto nível do sistema, temos o diagrama de casos de uso. Numa visão de mais baixo nível, com os detalhes necessários para que todos os requisitos sejam documentados e modelados, temos os cenários dos casos de uso.

A modelagem de um sistema na fase de **análise** tornou-se um processo mais transparente com a utilização da UML. Diagramas de classe passam a concentrar todas as informações necessárias sobre os dados (antes modelados com diagramas de entidade-relacionamento) e sobre as funcionalidades (antes modelados com diagramas de fluxo de dados). Não só conseguimos que essa fase seja mais transparente, como conseguimos que a transição da fase de análise para a fase de projeto seja feita de forma automática. A partir de um modelo de classes, não só geramos um script de banco de dados, como conseguimos gerar automaticamente a estrutura das classes na linguagem de programação escolhida. Da mesma forma, a partir de uma estrutura de banco de dados e de um código já implementado, podemos automaticamente atualizar nosso modelo, pelo processo de engenharia reversa, garantindo que toda a documentação esteja atualizada.

Para que essa transparência e praticidade aconteçam é imprescindível que um sistema modelado em UML tenha a sua implementação numa linguagem orientada a objetos. O mesmo já não é uma verdade absoluta quando se fala da modelagem de casos de uso. Hoje é comum encontrarmos equipes de desenvolvimento que vêm utilizando casos de uso para a modelagem e especificação dos requisitos, porém tendo as fases de análise e projeto voltadas para a metodologia de análise essencial.

Atualmente um padrão de mercado é a aplicabilidade da orientação a objetos em todas as fases do desenvolvimento de sistemas, exceto no projeto de banco de dados, que continua sendo feito sobre bancos relacionais. Essa passagem da modelagem OO para o projeto de bancos relacionais é conhecida como **modelagem objeto-relacional.** Contudo, o ideal (que esperamos para um futuro próximo) seria o projeto de banco de dados acompanhar a modelagem OO, ou seja, termos a nossa disposição um banco de objetos.

Na realidade, o banco, ou alguns bancos, objetos-relacionais já estão disponíveis no mercado, como o Oracle, DB2, Postgress, Caché etc. Tudo começou com o surgimento dos bancos de objetos, padrão definido pela OMG (*Object Management Group*) que cuidava de apresentar um modelo de dados orientado a objeto. Contudo os bancos BDOO criados sobre este padrão trouxeram a facilidade de se implementar modelos complexos, mas deixaram a desejar quanto ao desempenho. Numa corrida para não perder o filão do mercado, os fabricantes de bancos relacionais se reuniram, buscando um padrão alternativo, que veio a se chamar objeto-relacional ou relacional-estendido. Assim nasceu a versão 3 do SQL. Com isso, algumas versões recentes dos bancos BDR, atualizadas sobre o padrão 3, já oferecem ao mercado a possibilidade de modelar suas bases de dados no modelo OO.

O que falta então? Acredito sinceramente que patrocínio. Patrocínio das empresas de mercado em apostar nessa nova forma de persistir seus dados, pressionando assim os fabricantes a investirem mais no suporte de seus produtos como bancos objeto-relacionais.

Veja na **Figura 2** a diferença que temos na implementação da *classe Funcionario* em uma tabela relacional e em uma tabela de objetos.

Espera-se que toda a navegação através dos relacionamentos entre as classes de um banco objeto-relacional seja feita por meio da notação de ponto (*dot notation*). A notação de ponto é a forma existente para acessar as propriedades (atributos) ou métodos de um objeto. Desta forma, não é necessário que a *query* estabeleça explicitamente o relacionamento entre os objetos, visto que este relacionamento já foi previamente definido na definição da classe.

O nascimento da UML

Ao falarmos no começo do desenvolvimento de sistemas, vimos que a análise estruturada só surgiu após a programação estruturada. O mesmo ocorreu com a orientação a objetos (talvez esteja aí a origem de todos os problemas computacionais). No início da década de 90 havia mais de 50 métodos disputando o mercado para se tornar "o" método principal para a orientação a objetos. Contudo, a maior parte desses métodos cometia um grave pecado: ser uma extensão dos métodos estruturados. Os maiores prejudicados eram os usuários que não conseguiam encontrar uma solução única e devidamente discutida.

Nessa época, mesmo com contribuições valiosas de outros metodologistas, três metodologias começaram a dominar o mercado: *OMT – Object Modeling Technique* de James Rumbaugh, *método Booch* de Grady Booch e *OOSE – Object-Oriented Software Engineering* de Ivar Jacobson. Booch, ciente dos problemas em se ter

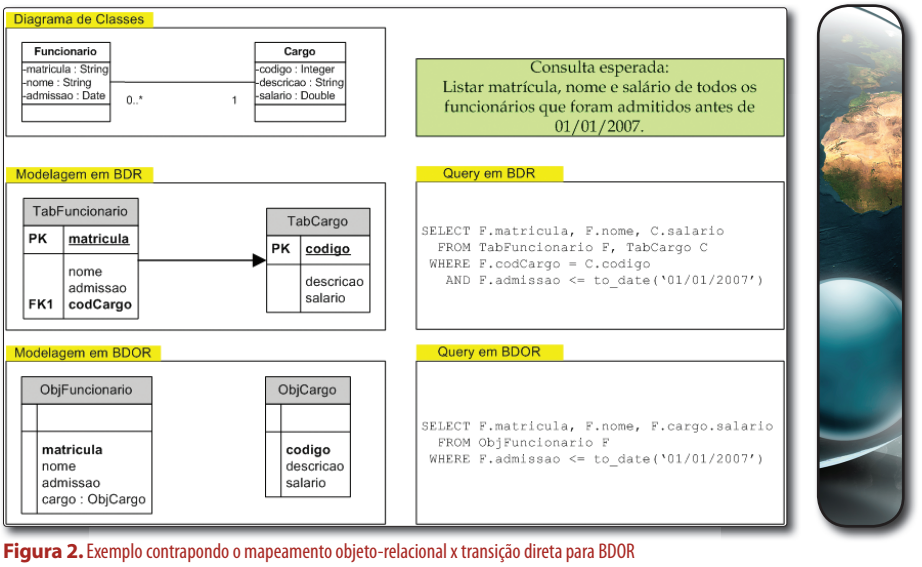


Figura 2. Exemplo contrapondo o mapeamento objeto-relacional x transição direta para BDOR

aquela diversidade de métodos, propôs ao mercado a união das metodologias, o que foi rechaçado pela maioria. Pouco depois, James Rumbaugh abandonou a General Electric e se juntou à Booch na Rational Software, produzindo o método unificado, na sua versão 0.8. Jacobson, ao perceber a similaridade do seu método com o método unificado, logo se uniu a eles. O que nasceu ainda como um método, teve a mudança de perspectiva, passando a ser uma linguagem de modelagem, desacoplando o processo de desenvolvimento. Nascia a UML – *Unified Modeling Language* na sua versão 0.9.

Em 1996, a UML já era vista pelas organizações como uma ótima estratégia para seus negócios. A OMG (*Object Management Group*) emitiu uma RFP (*Request for Proposals*), que objetivava receber propostas de padronização para uma metodologia de desenvolvimento orientado a objetos. Respostas foram recebidas da comunidade de engenharia de software e de grandes empresas (Digital, HP, IBM, Microsoft, Oracle e Unisys, entre outras) fortalecendo a proposta da UML. Em janeiro de 1997, a Rational lançou a versão 1.0 da UML como proposta para padronização na OMG. Entre janeiro e julho novas contribuições foram recebidas, e em 14 de novembro de 1997, a UML 1.1 era adotada como padrão pelo OMG.

A manutenção da UML passou a ser responsabilidade da RTF (*Revision Task Forces*), pertencente à OMG, sob a direção de Cris Kobryn, centralizando os comentários e pedidos de revisão da comunidade. Novas versões foram publicadas a partir daí, conforme a **Tabela 3**.

Diagramas da UML

O que há de mais relevante na UML é o fato de que a linguagem de modelagem nos oferece diversas ferramentas, ficando sob nossa responsabilidade a forma e a ordem como elas serão utilizadas. Além disso, a UML possui mecanismos de extensibilidade, que permitem a adequação da linguagem aos diversos sistemas existentes no mercado, sem que se perca o entendimento comum ao se usar um mesmo modelo.

A versão atual da UML contempla 13 diagramas, divididos em duas categorias:

Diagramas estruturais (estáticos):

- **Diagrama de classes** (*Class Diagram*) – apresenta classes conectadas por relacionamentos. Usado para exibir entidades do mundo real, além de elementos de análise e projeto.
- **Diagrama de objetos** (*Object Diagram*) – apresenta objetos e valores de dados. Corresponde a uma instância do diagrama de classes, mostrando o estado de um sistema em um determinado ponto do tempo.
- **Diagrama de componentes** (*Component Diagram*) – mostra as dependências entre componentes de software, apresentando suas interfaces.
- **Diagrama de estrutura composta** (*Composite Structure Diagram*) – usado para mostrar a composição de uma estrutura. Útil em estruturas compostas de estruturas complexas ou em projetos baseados em componentes
- **Diagrama de pacotes** (*Package Diagram*) – usado para organizar elementos de modelo e mostrar dependências entre eles
- **Diagrama de implantação** (*Deployment Diagram*) – mostra a arquitetura do sistema em tempo de execução, as plataformas de hardware, artefatos de software e ambientes de software (como sistemas operacionais e máquinas virtuais)

Diagramas comportamentais (dinâmicos):

- **Diagrama de casos de uso** (*Use Case Diagram*) – mostra os casos de uso, atores e seus relacionamentos que expressam a funcionalidade de um sistema.
- **Diagrama de atividades** (*Activity Diagram*) – representa a execução de ações ou atividades e os fluxos que são disparados pela conclusão de outras ações ou atividades.
- **Diagrama de máquina de estados** (*Statechart Diagram*) – representa as ações ocorridas em resposta ao recebimento de eventos.
- **Diagramas de interação:**
  - **Diagrama de seqüências** (*Sequence Diagram*) – mostra as interações que correspondem a um conjunto de mensagens trocadas entre objetos e a ordem que essas mensagens acontecem.
  - **Diagrama de comunicação** (*Communication Diagram*) – é o antigo diagrama de colaboração, que mostra objetos, seus inter-relacionamentos e o fluxo de mensagens entre eles
  - **Diagrama temporal** (*Timing Diagram*) – mostra a mudança de estado de um objeto numa passagem de tempo, em resposta a eventos externos.
  - **Diagrama de visão geral de interação** (*Interaction-Overview Diagram*) – 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.

Versão da UML	Publicação	Tipo de revisão
1.1	Novembro de 1997	Estrutural
1.2	Julho de 1998	Somente conteúdo
1.3	Março de 2000	Somente conteúdo
1.4	Maio de 2001	Estrutural
1.5	Março de 2003	Somente conteúdo
2.0	Julho de 2005	Estrutural
2.1.1	Agosto de 2007	Somente conteúdo
2.1.2	Novembro de 2007	Somente conteúdo

Tabela 3. Histórico das versões da UML

Podemos afirmar que é possível se completar a modelagem de um sistema de pequeno ou médio porte, com sucesso, com apenas três diagramas (casos de uso, classes e seqüências), tendo o suporte, dependendo do contexto, de três outros diagramas (objetos, atividades e máquina de estados).

O paradigma da orientação a objetos veio definitivamente ocupar um espaço que há muito se necessitava no mercado de desenvolvimento. Cabe aos desenvolvedores entenderem a importância de se respeitar todos os seus conceitos, para que se obtenha o melhor do que ele nos propõe.

Links

UML Site
www.uml.org

Artigo "What is Object-Oriented Software?", escrito por Terry Montlick
http://www.softwaredesign.com/objects.html

ODMG Site
www.odmg.org

Confiabilidade de Software

Determinante da Qualidade de Sistemas 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, tem mais de 30 artigos publicados em eventos nacionais e internacionais, colunista para Ciência e Tecnologia pela Revista Espaço Acadêmico com mais de 60 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.

De que se trata o artigo:
Apresenta definição conceitual da confiabilidade de software, destacando-a como principal atributo da qualidade. Discute a necessidade de considerá-la durante fases de desenvolvimento e operacional de um sistema de software.

Para que serve:
Informar a necessidade de se considerar a confiabilidade de software no desenvolvimento de produtos de software devido ao fato dela ser uma característica perceptível pelos usuários.

Em que situação o tema é útil:
Trata-se de uma prática de engenharia de software considerar a confiabilidade de software durante e após o desenvolvimento de um sistema de software. Permite ao engenheiro de software uma avaliação preliminar do produto.

Software é um produto que permeia nosso cotidiano e tem sido companheiro meu, seu e de quase todas as pessoas em uma gama enorme de aplicações. No dia-a-dia, podemos encontrar software embutido em forno microondas, nos jogos de computador ou celular bem como no controle de aeronaves e sistemas de telecomunicações, só para citar alguns exemplos.

Há, contudo, um atributo de qualidade associado ao software de suma importância para qualquer produto: confiabilidade. Isto vem da real necessidade de utilizar um produto ou sistema de software sem receio de qualquer falha operacional. Este artigo trata dos fundamentos e importância da confiabilidade de software.

Necessidade de Sistemas Confiáveis

Há aproximadamente cinco décadas atrás, o software constituía uma insignificante parte dos sistemas existentes e havia pouca preocupação com sua qualidade. Esse cenário começou a mudar com inserção do software como um elemento cada vez maior nos sistemas computacionais (ou seja, aqueles sistemas que têm software como componente). Antigamente, a maioria das funcionalidades implementadas nos sistemas era composta de compo-

nentes de hardware que exercia o controle sobre a operação dos sistemas.

Um exemplo de poucas décadas atrás era o sistema telefônico. Antigamente, as centrais telefônicas tinham seu controle e operação feitos à base de relés (isto é, um dispositivo de comutação) que possibilitava a comutação entre linhas telefônicas. Os sistemas mais antigos tinham a telefonista (ser humano) como uma profissional encarregada de realizar a comutação entre dois assinantes. Hoje em dia, essa funcionalidade é feita por um subsistema de software que identifica o assinante que origina a ligação telefônica, processa o número discado por ele e, por fim, realiza a comutação com o assinante chamado.

Agora, se considerarmos a história da indústria do software (ver seção Links), encontra-se o uso do software numa ampla variedade de aplicações tais como sistemas de manufatura, software científico, software embarcado, robótica e aplicações Web, dentre tantas. Como resultado disso, as funcionalidades e controle operacional dos sistemas computacionais se tornaram mais dependentes do software.

E, concomitante a este fato, houve crescimento no uso de computadores, principalmente, os de uso pessoal como o PC, além da descentralização dos grandes sistemas. Esses fatores, juntamente com a redução gradativa dos custos dos computadores, contribuíram para aumentar o uso do software numa ampla variedade de aplicações. O resultado de tudo isso foi o crescimento do software em termos de tamanho e complexidade.

Os projetos de sistemas computacionais mais antigos eram compostos de pequena parcela de software. Já os componentes de hardware, que eram a maior parte dos sistemas, eram analisados e testados quase exaustivamente, o que permitia a produção rápida de grandes quantidades de componentes e implicava em raros erros de projetos e falhas nos sistemas. Note que a facilidade de modificar o software, comparativamente ao hardware, foi e tem servido como motivador para seu uso.

A intensificação do uso do software numa larga variedade de aplicações resultou no seu crescimento em tamanho e complexidade. Como conseqüência, isto tornou proibitivo analisá-lo e testá-lo exaustivamente, além de impactar no custo de manutenção.

Fundamentos da Confiabilidade de Software

Ter o software como elemento essencial dos sistemas coloca sobre ele a necessidade de assegurar qualidade e, mais especificamente, sua capacidade de não apresentar falhas quando em uso. Associado com o software, há um atributo determinante da qualidade de qualquer produto: confiabilidade. A **confiabilidade de software** é definida como a probabilidade do software operar sem ocorrência de falhas durante um período especificado de tempo em um determinado ambiente.

Num contexto mais amplo, a qualidade de software é uma propriedade multidimensional que inclui, além da confiabilidade, outros fatores de satisfação do cliente como funcionalidade, usabilidade, desempenho, habilidade de prestação de serviço, manutenibilidade e documentação. Todavia, a confiabilidade é, comumente, aceita como um fator chave da qualidade de software, uma vez que a confiabilidade pode quantificar as falhas de software (que são percebidas pelo usuário).

No entanto, existem alguns termos que são utilizados no cotidiano das pessoas, mas que do ponto de vista da engenharia de software, têm significados específicos que precisam ser definidos para que haja entendimento e uso correto dos mesmos.

Inicialmente, há o termo **falha** que denomina um evento observado pelo usuário (ou algum mecanismo de detecção de falhas). Assim, durante a execução de um sistema de software, se um dado de saída está incorreto, então isto resulta ser uma falha. Em outras palavras, uma **falha** acontece quando o usuário percebe que um determinado programa deixa de prover o serviço ou funcionalidade por ele esperada. Portanto, uma falha é uma conseqüência de uma determinada falta que foi deixada no software.

Por outro lado, diz-se que um sistema possui uma **falta** se, para algum dado de entrada, o comportamento do sistema está incorreto, isto é, seu comportamento é diferente daquele descrito na especificação do software. Portanto, uma falta é uma causa identificada da falha de software ou erro interno do sistema, o qual é comumente denominado de **bug**.

Agora, quando a distinção entre falta (causa da falha) e falha (o efeito observável) não é critica, utiliza-se o termo genérico **defeito** para se referir à falta ou falha.

Vale ressaltar que uma falha poderia ser causada, não apenas por uma falta, mas também por um erro humano ou falha de hardware. Observe ainda que nem toda falta pode causar uma falha de software. Isso pode acontecer apenas se o componente (parte do software) que implementa uma determinada funcionalidade for utilizado durante a operação do sistema. Se o componente não é utilizado num cenário de uso do sistema, a falha não se manifesta e a falta permanece (oculta) no software já que nenhuma falha foi observada. Todavia, se o componente que possui uma falta é utilizado, ele irá apresentar um comportamento incorreto, resultando numa falha (observável).

Já o termo erro pode se originar em duas situações:

- *Erro humano (como causador da falha)* – que pode resultar numa falha observável ao usuário. Isto ocorre quando um erro humano motiva a falha, ou seja, ele pode desenvolver um produto contendo uma ou mais faltas como, por exemplo, especificando um sistema de forma inconsistente ou incompleta, bem como implementando-o de modo incorreto.
- *Discrepância de valores* – entre o valor observado ou computado e o valor (ou condição) contido na especificação.

É importante observar que ter o software como elemento essencial dos sistemas coloca sobre ele a necessidade de assegurar qualidade e, mais especificamente, sua capacidade de não apresentar falhas quando em uso. A confiabilidade de software se torna uma propriedade determinante para o produto porque ela é observável pelos usuários.

Outro atributo importante dos sistemas de software é a disponibilidade, que compreende a probabilidade de, em qualquer instante de tempo, um sistema funcionar satisfatoriamente num determinado ambiente. Em outras palavras, é a probabilidade de um sistema estar disponível quando necessário. A disponibilidade pode ser determinada pela relação:

Disponibilidade = ((MTTF) / (MTTF + MTTR)) x 100%

onde:

- MTTF (*Mean Time to Failure*) é o tempo médio até a ocorrência de falha.
- MTTR (*Mean Time to Repair*) é o tempo médio de reparo.

Esses dois atributos de qualidade, confiabilidade e disponibilidade, compreendem a base do que é denominado de Engenharia de Confiabilidade de Software (ECS), a qual é definida como o estudo quantitativo do comportamento operacional de sistemas de software com base nos requisitos de usuários relativo à confiabilidade.

ECS compreende técnicas de engenharia para desenvolver e prover manutenção a sistemas de software nos quais a confiabilidade pode ser avaliada quantitativamente. Objetivando adequadamente estimar e prever a confiabilidade de sistemas de software, dados das falhas de software necessitam ser medidos durante as fases de desenvolvimento e operacional.

A Engenharia de Confiabilidade de Software (ECS) inclui:

- Medição de confiabilidade de software, a qual inclui estimativa e previsão com o uso de modelos de confiabilidade de software.
- Atributos e métricas de projeto de produtos, processo de desenvolvimento, arquitetura de sistema, ambiente operacional do software e suas implicações sobre a confiabilidade.
- Aplicação deste conhecimento na especificação e projeto da arquitetura de software do sistema, desenvolvimento, testes, uso e manutenção.

Para tanto, torna-se necessária a coleta de dados de falhas. Esses dados são coletados para se fazer a medição da confiabilidade de software. Essa coleta compreende:

- *Contagem de falhas* – esse tipo de dado faz o rastreamento da quantidade de falhas detectadas por unidade de tempo;
- *Tempo médio entre falhas* – esse tipo de dado faz o rastreamento dos intervalos entre falhas consecutivas.

Além disso, ECS requer a medição de confiabilidade de software que envolve duas atividades: estimação e previsão de confiabilidade. A atividade de estimação determina a confiabilidade de software atual aplicando técnicas estatísticas de inferência aos dados de falhas obtidos durante o teste do sistema ou durante a operação do sistema. Esta é uma medida que considera a confiabilidade obtida desde um instante passado até o instante atual. Seu principal objetivo é avaliar a confiabilidade atual e determinar se o modelo de confiabilidade está bem calibrado.

Já a atividade de previsão determina a confiabilidade de software futura baseada em métricas de software e medidas disponíveis. Dependendo do estágio de desenvolvimento, a previsão pode envolver diferentes técnicas como:

- *Quando os dados de falhas estão disponíveis* – Por exemplo, o software encontra-se em testes ou no estágio operacional. Neste caso, as técnicas de estimação podem ser usadas para parametrizar e verificar os modelos de confiabilidade de software, os quais realizam previsão de confiabilidade de software futura.
- *Quando os dados de falhas não estão disponíveis* – Por exemplo, quando o software ainda encontra-se na fase de projeto ou implementação. Neste caso, as métricas obtidas do processo de desenvolvimento de software e as características do produto resultante podem ser utilizadas para determinar a confiabilidade de software durante a fase de testes.

Confiabilidade de Software no Desenvolvimento de Sistemas

O ciclo de vida de um produto como, por exemplo, um sistema de software consiste de duas grandes fases: desenvolvimento e operação. A fase de desenvolvimento compreende engenharia de sistemas, especificação de requisitos de software, projeto de software, codificação e testes. A fase operacional refere-se à fase na qual o sistema de software está em uso, oferecendo funcionalidades aos usuários.

Tradicionalmente, a maioria dos esforços da engenharia de software em produzir software confiável tem sido concentrada na fase de desenvolvimento. A razão para isto recai no fato da natureza inerente do software de não sofrer qualquer desgaste, isto é, por ser um produto intangível. Diferentemente do hardware que sofre desgaste, o software não é susceptível a esse tipo de problema. Cabe destacar que, se a produção de um sistema de software resultasse num software perfeito, ele permaneceria assim para sempre.

Todavia, resultados de esforços em produzir software livre de falhas tem sido um tanto desapontadores. Além disso, até mesmo os processos atuais de desenvolvimento de software ainda não asseguram software livre de faltas.

Agora, se considerarmos sistemas de grande porte como, por exemplo, os sistemas de telecomunicações, as técnicas baseadas unicamente no desenvolvimento que visam produzir software livre de faltas não são consideradas adequadas. Isto ocorre porque para tais sistemas, testes exaustivos têm custos proibitivos e, portanto, não são executados de modo completo a fim de assegurar correta funcionalidade para todos os possíveis cenários de uso do sistema.

Dentro do contexto destacado acima, níveis mais elevados da confiabilidade de software desses sistemas exigem técnicas que possam ser aplicadas durante a fase operacional, o qual é apresentado a seguir.

Melhoria da Confiabilidade de Software na Fase Operacional

Diversas técnicas têm sido propostas para melhorar a confiabilidade de software de um sistema no estágio operacional. Geralmente, elas podem ser classificadas como técnicas baseadas em software ou hardware. *Watchdog timer* é o exemplo mais proeminente de abordagem baseada em hardware para melhoria da confiabilidade de software.

Por outro lado, pode-se identificar e destacar *N-version programming (Programação N-versões)*, *Recovery Blocks (Blocos reparadores)*, *Software Audits (Auditores de software)* e *Software Supervision (Supervisão de software)* como os principais meios de obter níveis mais elevados de confiabilida-

Características vs. Técnica	Capacidade de supressão de falhas	Granulosidade da aplicação
Programação N-versões	Total	Grande
Blocos reparadores	Total	Pequena
Auditores de software	Parcial	Pequena
Supervisão de software	Parcial	Grande

Tabela 1. Técnicas de melhoria da confiabilidade de software na fase operacional.

de de software durante a fase operacional do software.

O quadro apresentado na **Tabela 1** caracteriza essas quatro técnicas de melhoria da confiabilidade de software na fase operacional, categorizando-as em função da capacidade de supressão de falhas e da granulosidade da aplicação.

N-version programming (NVP ou Programação N-versões) tenta compensar os erros introduzidos durante a fase de desenvolvimento. A suposição fundamental é que a probabilidade de duas ou mais equipes de desenvolvimento produzirem o mesmo erro em uma parte do software é muito pequena.

Esta técnica consiste em desenvolver, separadamente, N projetos e implementações de um software a partir de uma especificação de requisitos. Para essa técnica ser efetiva, cada uma das versões deveria ser produzida por uma equipe separada, com pouca ou nenhuma interação entre elas. Todas as versões do software são executadas de modo concorrente gerando saídas em função de um conjunto de entradas. Em tal situação, um algoritmo de votação decide qual saída de uma das implementações deveria ser utilizada como saída do sistema. É comum utilizar uma estratégia de selecionar aquela versão que tem a maioria na votação.

Embora *N-version programming* tenha mostrado ser efetiva quanto à redução de falhas de software, ela se depara com algumas dificuldades. Resultados experimentais mostram que uma cobertura completa dos erros, em geral, não pode ser garantida por esta abordagem. Isto ocorre porque diferentes implementações estarão cobrindo diferentes erros, mas não sua totalidade como discutido acima. Embora a diversidade de versões contribua para cobrir uma quantidade maior de erros e melhorar a confiabilidade do software, os custos que advém dessa abordagem, devido à necessidade de manter N equipes de desenvolvimento e fazer manutenção em N versões do programa, tornam sua aplicabilidade um tanto limitada.

Recovery Blocks (RB ou Blocos reparadores) é similar a *N-version programming*, uma vez que várias versões são utilizadas. Contudo, ela difere no fato que versões adicionais são usadas apenas se a atual é suspeita de estar produzindo resultados incorretos. Considere a necessidade de um algoritmo para calcular a raiz quadrada de um número real. Em tal situação, um teste de aceitação adequado pode ser obtido, comparando-se o quadrado do resultado com o valor original da entrada. Se há mais de um algoritmo para implementar essa funcionalidade, uma forma de verificar seu uso seria baseada na eficiência. Se o mais eficiente falhar, então o software iria recorrer a um algoritmo imediatamente menos eficiente.

Adicionalmente, a eficiência desta técnica depende do teste de aceitação, que pode ser difícil de ser desenvolvido ou até mesmo envolver um algoritmo complexo, também sujeito a erros. Vale ressaltar que o problema econômico de desenvolver e manter as diversas versões de software são similares aos mencionados para *N-version programming*. Isto acontece devido ao fato de ser necessário produzir várias versões de uma mesma funcionalidade com o objetivo de assegurar que o software irá funcionar de modo correto durante um período de tempo maior e, portanto, provendo uma maior confiabilidade.

Uma outra abordagem é *Software Audits* (SA ou Auditores de software) na qual erros de software são detectados e, possivelmente, corrigidos através de programas *auditores*. Estes programas auditores consistem de software adicional os quais têm acesso às estruturas de dados do programa principal. Tipicamente, programas auditores executam em um nível mais baixo de prioridade comparado ao programa principal e, apenas periodicamente checam erros nas estruturas de dados. Três tipos de erros que podem ser detectados por programas auditores compreendem:

- *Erros de comparação direta* – detectar esse tipo de erro exige que uma cópia dos dados seja mantida. Por exemplo, em estruturas estáticas, uma cópia pode ser mantida na memória.
- *Comparação por erros de associação* – comparação por associação requer que informação redundante seja mantida entre diferentes estruturas de dados ou campos de uma estrutura de dados. Assim, a detecção de inconsistência se torna possível baseada nesta informação redundante.
- *Erros de comparação de formatos* – a comparação de formatos assegura que dos dados das estruturas de dados sejam (praticamente) corretos. Verificação de limite de dados é o exemplo mais intuitivo de comparação de formatos.

É importante observar que as capacidades de retração de falhas dos programas auditores são limitadas. Na maioria dos casos, após a detecção de um erro, o programa auditor é apenas capaz de "resetar" o software para um estado de segurança ao invés de retratar o erro, ou seja, de levar o software para o último estado válido. Embora isto limite a aplicabilidade de programas auditores, eles têm sido usados em software de sistemas de telecomunicações.

A técnica que temos investigado é denominada de *Supervisão de Software* e compreende um sistema supervisionado (SS) por um supervisor. SS recebe os sinais de entrada do ambiente e responde gerando sinais de saída. O supervisor monitora os sinais de entrada e saída e, baseado na especificação do *SS*, determina se eles constituem comportamentos válidos. Em outras palavras, a função do supervisor é comparar as entradas e saídas para o sistema supervisionado com a especificação deste e, caso seja produzida uma saída que não está em conformidade com a especificação, então o supervisor irá detectar e relatar uma falha. Perceba que qualquer comportamento inválido (ou seja, não contido na especificação) do *SS* detectado pelo supervisor é relatado como falha.

A vantagem desta técnica é que diversas versões idênticas de software não precisam ser produzidas. Adicionalmente, eleva-se a disponibilidade operacional do sistema, pois quando qualquer falha é detectada pelo supervisor, ele prontamente relatará o ocorrido e permitirá que manutenção seja imediatamente realizada.

A técnica de supervisão de software tem sido empregada em sistemas de telecomunicações e, principalmente, em centrais telefônicas. O objetivo é ter um supervisor de software monitorando o processamento de ligações telefônicas feitas pelo software de controle da central e, quando o sistema supervisionado (ou seja, a central telefônica) fornece uma saída inválida, dado que a saída não está em conformidade com a especificação, então o supervisor observa o comportamento incorreto e o detecta com uma falha. Isto é registrado em relatório. Se várias falhas são detectadas, seu relato contínuo e imediato faz com que operadores possam tomar a atitude corretiva o mais cedo possível antes que haja uma degradação na qualidade do serviço oferecido pelo sistema.

Se também considerarmos o perfil operacional, isto é, o conjunto de operações que um software pode executar juntamente com a probabilidade na qual elas ocorrem do sistema de software (supervisionado), que retrata como ele é utilizado no estágio operacional, será possível especificar categorias de entradas para o sistema e a probabilidade de suas ocorrências. Isto permitirá uma avaliação mais adequada do novo padrão de uso do sistema de software (supervisionado) no estágio operacional.

Comentários Finais

Software permeia nosso cotidiano e é provavelmente uma das entidades quase onipresentes nos sistemas atuais e, portanto, essencial para atividades das pessoas e das empresas. Este artigo apresentou um dos atributos de qualidade essencial aos sistemas de software: a confiabilidade de software. Sua necessidade e importância foram destacadas e um conjunto de quatro técnicas discutidas. Considerar a confiabilidade como fator importante do desenvolvimento de software é obrigatório, e também considerá-la na fase operacional é essencial para sistemas de grande porte. O próximo artigo apresentará exemplos baseados nos conceitos introduzidos neste 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

Links

História da Indústria de Software
www.softwarehistory.org

Software Reliability Engineering: A Roadmap
http://csse.usc.edu/classes/cs589_2007/Reliability.pdf

The 19th IEEE International Symposium on Software Reliability Engineering (ISSRE)
http://www.csc2.ncsu.edu/conferences/issre/2008/

Software Metrics and Reliability
http://satc.gsfc.nasa.gov/support/ISSRE_NOV98/software_metrics_and_reliability.html

Software Quality
http://en.wikipedia.org/wiki/Software_quality

Reliability Engineering
http://en.wikipedia.org/wiki/Reliable_system_design

Maturidade em Gerenciamento de Projetos

Uma Visão Analítica

Luciana Leal
lql@cin.ufpe.br
Doutoranda e Mestre em Ciência da Computação pela Universidade Federal de Pernambuco. Graduada em Ciência da Computação pela Universidade Federal da Paraíba e tecnóloga em Telemática pelo Centro Federal de Educação Tecnológica da Paraíba.

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

Hermano Perrelli
hermano@cin.ufpe.br
Atualmente é professor Adjunto e Vice-Diretor do Centro de Informática da Universidade Federal de Pernambuco. Consultor e instrutor da *Qualiti Software Processes*. Certificado PMP - Project Management Professional pelo PMI. PhD In *Computing Science pela University of Glasgow*, mestre em Informática pela Universidade Federal de Pernambuco e graduado em Engenharia Eletrônica pela Universidade Federal de Pernambuco.

De que se trata o artigo:
Apresentação de uma visão geral sobre Maturidade em Gerenciamento de Projetos.

Para que serve:
Fornecer uma visão geral sobre maturidade às organizações que desejam melhorar o gerenciamento de projetos.

Em que situação o tema é útil:
Maturidade em gerenciamento de projetos é um assunto que vem sendo bastante explorado. Assim, este artigo esclarece alguns pontos a respeito do tema no sentido de auxiliar organizações na escolha de um modelo que avalie seu nível de maturidade e apresentar um conhecimento inicial para profissionais na área de gerenciamento e interessados

Nos dias atuais, a execução de projetos está se tornando comum e remete ao desafio de gerenciar projetos com eficiência. O uso efetivo de tecnologias durante esta atividade pode determinar o sucesso de qualquer negócio e satisfazer as expectativas dos clientes. Contudo, em busca do sucesso, o uso adequado destas tecnologias é crucial. É possível observar uma crescente busca por melhoria de processos relacionados, principalmente, à gerência de projetos e à engenharia de software em várias organizações de Tecnologia da Informação (TI).

Segundo diversos autores, o conceito de sucesso encontra-se atrelado ao de maturidade, podendo-se observar que os fatores determinantes para o sucesso de um projeto estão ligados ao grau de maturidade da organização em que ele é desenvolvido.

Dentro deste contexto, este artigo tem o objetivo de apresentar os fatores que levam ao sucesso de projetos, conceitos básicos da área de maturidade em gerenciamento de projetos e uma visão comparativa de alguns dos modelos que estão em evidência.

Fatores que Levam ao Sucesso de um Projeto

Diversos autores definem sucesso e fracasso para melhor entender o desempe-

nho de projetos. Uma definição simplista diz que o sucesso ocorre quando se atingem as metas originais de custo, prazo e qualidade. Mas é fato que existem outros fatores que o influenciam, por exemplo, a percepção que os *stakeholders* têm do desempenho do projeto.

Neste contexto, podemos visualizar uma sutil distinção entre sucesso do projeto e sucesso da gestão de projetos. O sucesso do projeto está relacionado à aceitação do produto por parte do mercado e o sucesso da gestão de projetos se caracteriza por aspectos clássicos: custo, prazos, qualidade e satisfação do cliente.

Existe uma definição de sucesso que considera na sua concepção aspectos externos e internos, onde os primeiros estão mais próximos do gerente e equipe, os últimos estão mais ligados ao comportamento dos clientes. Esses aspectos internos são: custo, prazo e qualidade e os aspectos externos: uso, satisfação e eficácia. Com base nesta definição, é possível dizer que o sucesso da gestão de projetos tende a influenciar positivamente o sucesso do produto.

Outra linha de pensamento acredita que não existe esta distinção de sucesso, mas que ele possui características multidimensionais e que cada uma destas dimensões pode variar com o tempo, sendo algumas delas: eficiência do projeto, impacto no consumidor, cumprimento de metas do projeto, satisfação com a qualidade técnica do produto, e sucesso do negócio, entre outras. Mas apesar dos fatores apresentados, convém lembrar que, para que o gerente e sua equipe possam tratar adequadamente os fatores condicionantes de sucesso, é necessário que haja um consenso sobre os critérios de sucesso utilizados no projeto e que estes critérios estejam bem definidos.

Algumas empresas definem sucesso em termos de fatores críticos (CSFs – *Critical Success Factors*) e de indicadores-chave de desempenho (KPIs – *Key Performance Indicators*) [Kerzner 2000]. Os CSFs atendem às necessidades dos clientes e medem o resultado final do projeto. Dentre os mais utilizados podem ser citados: cumprimento do cronograma, atendimento do orçamento, concretização da qualidade, conveniência e oportunidade da assinatura do contrato, cumprimento do processo de controle de mudança e aditivos do contrato. A qualidade do processo interno utilizado para alcançar os resultados finais dos projetos é medida através dos indicadores internos, os KPIs. Estes indicadores podem ser revisitados periodicamente ao longo do ciclo de vida do projeto, e incluem utilização da metodologia de gestão de projetos, estabelecimento dos processos de controle, uso de indicadores interinos, qualidade de recursos aplicados *versus* planejados e envolvimento do cliente.

Prado e Archibald (2007), de acordo com pesquisa realizada em 2006, indicam alguns fatores determinantes do sucesso de projetos de TI: (i) Complexidade dos projetos (dificuldades intrínsecas da carteira de projetos); (ii) Motivação da equipe; (iii) Nível de competência técnica da equipe para as necessidades da carteira de projetos; (iv) Cenário de clientes/concorrência/pressão dos negócios/fatores externos; (v) Nível de maturidade em gerenciamento de projetos do setor. Ainda dentro da mesma pesquisa, os autores afirmam que, para uma amostra suficientemente ampla de empresas, a soma dos quatro primeiros fatores (fatores de contingência) tem a mesma contribuição média apesar de apresentarem uma forte dispersão. E concluem que, caso o terceiro fator esteja incluído no quinto, é possível supor uma correlação positiva entre sucesso e maturidade.

A Relação entre Sucesso e Maturidade

O grande interesse sobre maturidade nos tempos atuais pode ser explicado pela estreita relação existente entre maturidade e sucesso. As pesquisas realizadas no assunto apresentam o nível de maturidade de uma organização como um dos fatores determinantes para se chegar ao sucesso em projetos.

Prado e Archibald (2007) apontam como positiva a relação entre os dois conceitos. O *Project Manager Competency Development*, do PMI, associa sucesso à maturidade, a fatores contingenciais e a competência do gerente de projetos. Kerzner (2000) também fala da relação sucesso-maturidade, além de outros autores.

Os resultados positivos encontrados nestas pesquisas aumentam o esforço das organizações a fim de evoluir dentro de níveis de maturidade, motivadas pelo desejo de aumento do sucesso na execução de projetos. As empresas estão cada vez mais conscientes tanto da importância do gerenciamento de projetos, para concretizar suas estratégias, como de que o amadurecimento pode levar à excelência [Prado 2008].

Maturidade em Gerenciamento de Projetos

O estudo da maturidade em gerenciamento, assim como o estudo do próprio gerenciamento de projetos, é assunto que vem sendo discutido há pouco tempo, mas tem ocupado lugar de destaque.

Maturidade pode ser definida como o desenvolvimento de sistemas e processos que são, por natureza, repetitivos e garantem uma alta probabilidade de que cada um deles seja um sucesso [Kerzner 2000]. Contudo, processos e sistemas repetitivos não são por si só, garantia de sucesso, apenas aumentam sua probabilidade. Assim, podemos observar que a maturidade em gestão de projetos está ligada a quão hábil uma organização está no exercício de gerenciar seus projetos [Prado 2008].

Kerzner (2000) classifica em cinco as fases do ciclo de vida para a maturidade em gestão de projetos: Embrionária, Aceitação pela gerência executiva, Aceitação pelos gerentes da área, Crescimento e Maturidade. O autor ainda afirma que é possível observar que as empresas que almejam desenvolver-se e chegar a níveis de maturidade maiores já passaram ou ainda estão passando por algumas destas fases.

Atualmente existem vários modelos de maturidade que possuem forte fundamentação em conceitos de gerenciamento da qualidade, como, por exemplo, os modelos baseados em cinco níveis incrementais de maturidade ou ainda as práticas para o melhoramento contínuo dos processos de gerenciamento.

Modelos de Maturidade

As organizações a cada dia têm aumentado a sua preocupação com os projetos que desenvolvem. É notório o investimento em ferramentas, técnicas, treinamento e capacitação em gerenciamento de projetos, que tem aumentado de forma considerável.

Modelos de maturidade são mecanismos capazes de quantificar numericamente a maturidade. Estes modelos auxiliam a elaboração de processos, indicam melhores práticas e fazem com que as organizações se desenvolvam de forma constante.

A partir da década de 90, surgiram diversos modelos para avaliar a maturidade das organizações em gerenciamento de projetos, quase todos inspirados no modelo de maturidade em desenvolvimento de software SW-CMM, que é voltado principalmente para aspectos técnicos do processo de desenvolvimento [Prado 2008].

O modelo de maturidade CMMI (*Capability Maturity Model Integration*) [SEI 2001] é organizado em áreas de processo, áreas de conhecimento e níveis de maturidade. Segundo o CMMI, atingir um nível de maturidade significa implementar todas as atividades deste nível e mais todas as atividades do nível ou níveis anteriores a ele.

Muitos dos modelos de maturidade para gerenciamento de projetos apresentam a estrutura de cinco níveis proposta pelo CMM/CMMI, contudo algumas diferenças são observadas no conteúdo de cada nível. Dentre os modelos existentes podem ser destacados Prado-MMGP [Prado 2008], P3M3 [OGC 2008], OPM3 [PMI 2003] e KPMMM [Kerzner 2001], que serão apresentados a seguir.

Modelos Prado-MMGP

Os Modelos de Maturidade em Gerenciamento de Projetos, também conhecidos como modelos Prado-MMGP, foram lançados entre 2002 e 2004 por Darci Prado. Estes modelos dizem respeito à (i) avaliação da maturidade nos setores de uma organização e (ii) a avaliação da organização como um todo. O Modelo Setorial foi o primeiro a ser desenvolvido e tem o intuito de realizar a avaliação isolada dos departamentos da organização. O Modelo Corporativo procura avaliar de forma global o gerenciamento de projetos de uma organização.

Os modelos desenvolvidos por Prado apresentam como sua principal vantagem a simplicidade, em meio a tantos modelos de maturidade complexos. Além disso, os modelos são pequenos e podem ser utilizados no desenvolvimento de um plano de crescimento, que almeja níveis maiores na escala da maturidade gerencial.

O Modelo Setorial possui cinco níveis de maturidade e seis dimensões. Os níveis dizem respeito ao grau de maturidade aplicado a cada setor ou organização. As dimensões são condizentes com a melhoria dentro de cada nível. Assim, cada nível pode conter as seis dimensões da maturidade que, dependendo do nível, podem apresentar variações de intensidade.

Os níveis de maturidade do modelo Prado-MMGP são: (1) Inicial, (2) Conhecido, (3) Padronizado, (4) Gerenciado e (5) Otimizado. E as dimensões são as seguintes: Competências Técnicas, Uso Prático de Metodologia, Informatização, Estrutura Organizacional, Alinhamento com os Negócios da Organização, Competências Comportamentais e Contextuais. Convém lembrar que, segundo os modelos Prado-MMGP, uma organização pode ocupar parcialmente diversos níveis de maturidade e que as dimensões do modelo são desenvolvidas e melhoradas a cada nível alcançado.

Uma organização no Nível Inicial tem como característica o gerenciamento de projetos executado de forma intuitiva e a inexistência de uma metodologia. Não se faz planejamento ou controle do projeto e o nível de conhecimento em gerenciamento de projetos não é uniforme entre os principais envolvidos.

No Nível Conhecido, os investimentos em treinamento são regulares e uma tentativa sutil de padronização de procedimentos indo em direção a uma metodologia única. A partir dos treinamentos realizados, procura-se utilizar uma linguagem comum. Este nível também se caracteriza por iniciativas isoladas de se efetuar planejamento e controle.

O Nível Padronizado caracteriza-se pela padronização de procedimentos, difundida e utilizada sob a liderança de um Escritório de Gerenciamento de Projetos (EGP) e metodologia única praticada por todos. Este nível procura efetuar alinhamento com as estratégias organizacionais.

No quarto nível da escala de maturidade consolidam-se as ações iniciadas no nível 3 em relação à metodologia, informatização, estrutura organizacional e alinhamento estratégico. Neste nível os resultados gerenciais são armazenados em uma base de dados que possui as informações sobre projetos passados e a aplicação de processos de gerenciamento de projetos é reconhecida como fator de sucesso.

No Nível Otimizado, as iniciativas começadas nos níveis 2, 3 e 4 atingem um nível de excelência. A melhoria pode ser visualizada a partir dos resultados obtidos: processos de prazo, custo e qualidade otimizados, modelo de gerenciamento de projetos adequado às necessidades da organização, cultura de gerenciamento amplamente disseminada e praticada, uso rotineiro e eficiente da metodologia de gerenciamento de projetos, estrutura organizacional adequada, harmonia e produtividade nos relacionamentos humanos e total alinhamento com os negócios da empresa.

O Modelo Corporativo, aplicado à organização como um todo, avalia a capacidade de se obter resultados por meio do gerenciamento de portfólio, programas e projetos. O MMGP-Corporativo divide a organização em duas grandes áreas: setores formalmente organizados para o gerenciamento de projetos e setores não-formalmente organizados para o gerenciamento. Também possui cinco níveis, contudo estes possuem um significado um pouco diferente dos modelos CMM/CMMI. Estes níveis são:

- **Inicial:** a empresa não tem nenhuma experiência organizada em gerenciamento de projetos, tanto setorial como corporativo;
- **Projetos Isolados:** existem algumas iniciativas isoladas bem-sucedidas de gerenciamento de projetos;
- **Setorial:** alguns setores da organização se organizaram adequadamente para gerenciar projetos;
- **Portfólio e Programas:** a empresa possui regras para identificar e gerenciar os portfólios e programas corporativos;
- **Corporativo:** a empresa possui uma centralização corporativa para estabelecer regras e acompanhar a evolução de todos os setores.

A avaliação da maturidade setorial e corporativa pode ser realizada através de questionários, cada um contendo 40 questões referentes aos modelos, e a partir da avaliação inicial, a organização pode desenvolver seu plano de crescimento, com ou sem intermédio de consultoria.

P3M3 – *Portfolio, Programme and Project Management Maturity Model*

O *Portfolio, Programme and Project Management Maturity Model* (P3M3) é um modelo europeu desenvolvido pelo *Office of Government Commerce* (OGC) em 2006. O modelo descreve atividades que estão relacionadas ao gerenciamento de projetos, programa e portfólio dentro de áreas que contribuem para alcançar sucesso em projetos.

O P3M3 originou-se de uma extensão do OGC's *Project Management Maturity Model*, o qual foi baseado no framework de maturidade de processos o qual evoluiu para o *Capability Maturity Model* (CMM). Possui três modelos individuais: *Portfolio Management Maturity Model* (PfM3), *Programme Management Maturity Model* (PgM3) e *Project Management Maturity Model* (PjM3). Por possuí três modelos que podem ser visualizados separadamente, o P3M3 permite a avaliação independente de qualquer um deles.

Níveis de Maturidade				
Nível 1 Inicial	**Nível 2 Repetível**	**Nível 3 Definido**	**Nível 4 Gerenciado**	**Nível 5 Otimizado**
O comitê executivo reconhece programas e executa uma lista informal dos seus investimentos em programas e projetos. A organização consegue reconhecer programas e os executa de forma diferente para cada projeto. A organização consegue reconhecer projetos e os executa de forma diferente para cada negócio.	A organização consegue assegurar que em cada programa e/ou projeto no portfólio/programa os processos e procedimentos estão sendo executados de acordo com um padrão mínimo especificado	A organização tem o controle centralizado dos programas/ projetos e consegue individualmente adaptar os processos de forma a se ajustar a um projeto específico	A organização possui e mantém avaliações do desempenho de seus portfólio/ programas/projetos e executa gerenciamento de qualidade organizacional para melhor predizer o desempenho futuro	A organização consegue executar melhoria de processo contínua com reação pró-ativa aos problemas e gerenciamento da tecnologia para os portfólio/ programas/ projetos a fim de melhorar sua habilidade de descrever o desempenho e otimizar processos

Tabela 1. Níveis de Maturidade no P3M3

Seguindo a estrutura do CMM, o P3M3 usa um *framework* de cinco níveis de maturidade para cada um dos modelos individuais, sendo cada um composto pelos níveis: 1 - Inicial, 2 - Repetível, 3 - Definido, 4 - Gerenciado e 5 - Otimizado. A **Tabela 1** apresenta os níveis sumarizados para cada um dos modelos individuais.

Os níveis de maturidade do P3M3 indicam como áreas de processo-chave (KPAs – *Key Process Areas*) podem ser estruturadas hierarquicamente para definir uma progressão de capacidade, ou aumento de maturidade, que a organização pode utilizar a fim de alcançar seus objetivos e planejar melhoria.

Existem sete perspectivas de processo dentro do P3M3 que definem as características chave de uma organização madura. Estas perspectivas se aplicam aos três modelos individuais e a todos os níveis de maturidade. Cada perspectiva descreve os processos e práticas que devem ser implantados em um dado nível e maturidade.

Para cada uma das perspectivas existe um conjunto de atributos que são comuns aos níveis de maturidade. Atributos descrevem o perfil pretendido para cada perspectiva em cada nível de maturidade, e, além disso, os tópicos, processos e práticas que serão desenvolvidos e sofrerão mudanças quando o nível de maturidade muda.

O P3M3 possui dois tipos de atributos: atributos específicos e atributos genéricos. Atributos específicos se relacionam com uma perspectiva de processo particular. Atributos genéricos são utilizados em todas as perspectivas de processo em um dado nível de maturidade e incluem: planejamento, gerenciamento da informação, e treinamento e desenvolvimento.

As organizações podem avaliar seu nível de maturidade pelo P3M3 através do questionário de auto-avaliação disponível no site www.p3m3-officialsite.com ou através de revisão formal.

OPM3 - Organizational Project Management Maturity Model

Após seis anos de pesquisa e desenvolvimento, em 2003 o *Project Management Institute* - PMI apresentou o modelo OPM3 (*Organizational Project Management Maturity Model*). Através de uma pesquisa realizada com mais de 30.000 profissionais, foram analisados os pontos fortes e fracos dos modelos contemporâneos de maturidade em gerenciamento de projetos. De acordo com o PMI (2003), foram avaliados mais de 27 modelos de maturidade em organizações de 35 países.

O OPM3 surge não apenas como um modelo organizacional, mas oferece uma grande base de dados de melhores práticas e indicadores de capacidade. Essas informações permitem que se localize a organização em relação às práticas aplicadas por ela e permite que seja comparada com o padrão proposto.

O modelo foi intencionalmente projetado sem níveis de maturidade. Estabelecer níveis específicos de maturidade pode ser relativamente simples se a progressão da maturidade é unidimensional, contudo o OPM3 acredita que esta progressão é multidimensional. Múltiplas perspectivas para avaliar a maturidade permitem flexibilidade para aplicar o modelo, de acordo com as necessidades de uma organização, além de proporcionar maiores detalhes sobre o suporte a decisões e planos de melhoria.

Uma dimensão envolve visualizar melhores práticas em termos de suas associações com os estágios progressivos de melhoria no processo: padronização, medição, controle e melhora contínua. Outra dimensão envolve o progresso de melhores práticas associado com cada domínio, primeiro endereçando o Gerenciamento de Projeto, depois o Gerenciamento de Programa e, finalmente, o Gerenciamento de Portfólio. Cada uma dessas progressões é um contínuo ao longo do qual a maioria das organizações aspira chegar. Dentro dessas duas dimensões está a progressão das capacidades que leva às melhores práticas. Por último, o OPM3 também categoriza habilidades em termos de suas associações com os cinco grupos de processo de gerenciamento de projeto: Iniciação, Planejamento, Execução, Controle e Conclusão.

Capacidade é definida como uma competência específica que deve existir em uma organização para que ela execute processos de gerenciamento de projeto e entregue produtos e serviços. Capacidades são, portanto, passos incrementais que levam a uma ou mais melhores práticas.

Para se utilizar por completo o conteúdo do modelo, seus diretórios são essenciais. Os diretórios de (i) Melhores Práticas, (ii) Capacidades e (iii) Planejamento de Melhorias são utilizados para se avaliar uma organização diante do OPM3 e para se avaliar o escopo e a seqüência dos possíveis melhoramentos.

A avaliação pode ser realizada através de 151 questões que fazem parte do modelo.

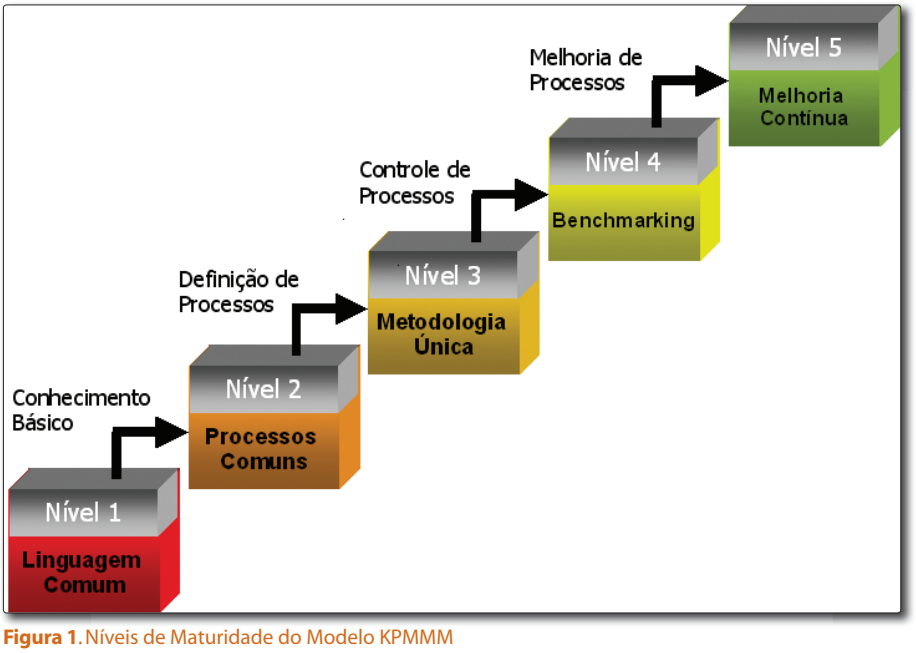


Figura 1. Níveis de Maturidade do Modelo KPMMM

No entanto, se a organização deseja alcançar maiores percentuais de maturidade, deve realizar cinco passos: Preparar-se para a avaliação, Avaliar, Planejar melhorias, Implementar melhorias e Repetir o processo. Estes passos conduzem a organização às capacidades necessárias para avançar no caminho para aumentar sua maturidade. Ainda, sugerem que o processo seja repetido, incentivando a melhoria contínua.

KPMMM – *Kerzner Project Management Maturity Model*

O modelo Kerzner *Project Management Maturity Model*, o KPMMM, foi proposto por Harold Kerzner em 2001. O KPMMM define o estágio de maturidade atual da organização, e a partir daí fornece orientações em relação ao planejamento e às ações para implementação e desenvolvimento gradual do gerenciamento de projetos. É uma aplicação prática do gerenciamento de mudanças, onde se busca minimizar a resistência à implementação do sistema através da disseminação da cultura.

É composto por cinco níveis: Linguagem Comum, Processos Comuns, Metodologia Única, *Benchmarking* e Melhoria Contínua, visualizados na **Figura 1** e apresentados a seguir:

• **Nível 1 - Linguagem Comum:** Caracterizado pelo uso de gerenciamento de projetos de forma esporádica. Não existe investimento em treinamento e a organização não reconhece a importância do gerenciamento de projetos e a necessidade de entender bem os seus conceitos básicos e linguagem.

• **Nível 2 – Processos Comuns:** Neste nível inicia-se o reconhecimento dos benefícios tangíveis de gerenciamento de projetos. Gerenciamento de escopo, prazo e custos começa a ser realizado e assim o sucesso em um projeto consegue ser repetido em outros projetos.

• **Nível 3 – Metodologia Única:** A integração de processos em uma metodologia única caracteriza o nível 3. Começa a surgir apoio de todos os níveis gerenciais da organização em prol de um melhor gerenciamento de projetos e existe treinamento contínuo em gerenciamento de projetos.

• **Nível 4 – *Benchmarking:*** O nível 4 estabelece um PMO (*Project Management Office*), que deve aprimorar os processos para o gerenciamento de projetos. Começam a ser realizados *benchmarks* que devem abranger as análises quantitativa (processos e metodologias) e qualitativa. A organização reconhece que a melhoria de processos é necessária para manter vantagem competitiva.

• **Nível 5 – Melhoria Contínua:** No nível de Melhoria Contínua, a organização registra as lições aprendidas para beneficiar seus projetos futuros. O planejamento estratégico para o gerenciamento de projetos é um processo contínuo e a metodologia adotada pode ser melhorada de acordo com as informações obtidas através de *benchmarking*.

O KPMMM possui um questionário com 183 questões de múltipla escolha que avaliam e enquadram as organizações em um dos cinco níveis de maturidade. A avaliação também pode ser realizada, mediante autorização, através da web pelo site http://www.iil.com/pm/kpmmm/. Kerzner Project Management Maturity Assessment Tool é a ferramenta que faz a avaliação de maturidade, de acordo com as áreas de conhecimento do PMBOK® Guide. A ferramenta, que precisa de uma autorização para uso, fornece o nível de maturidade de acordo com o KPMMM e apresenta um relatório com um plano de ação a ser executado pela organização.

Comparativo dos Modelos de Maturidade

Conforme já comentado neste artigo, os modelos KPMMM, P3M3 e Prado-MMGP possuem uma abordagem estagiada que sugere o embasamento no CMM.

O KPMMM é um modelo de acesso fácil e é simples de ser utilizado. De acordo com o PMI (2008), está entre os três modelos de maturidade mais utilizados no Brasil, contudo é o único dos apresentados que só aborda a dimensão Projeto no estudo da maturidade organizacional.

P3M3 é um conjunto de modelos individuais que evoluiu do CMM. Não cita um conhecimento base em gerenciamento de projetos, como o PMBOK®, e se fundamenta na literatura existente sobre o assunto. Possui um questionário de avaliação de maturidade relativamente grande e de simples aplicação. É um modelo que, segundo o apresentado no *benchmarking* realizado pelo PMI [PMI 2008], não apresenta uma porcentagem significativa para ser mencionado o seu uso.

Os Modelos Prado-MMGP foram desenvolvidos no Brasil e dispõem de avaliação bem mais simples e direta que os demais. A avaliação setorial é realizada através de um questionário pequeno, que pode ser respondido por qualquer funcionário. Esses modelos foram concebidos a partir das experiências práticas bem sucedidas de Darci Prado. Por ser fácil de utilizar e de fácil acesso (www.maturityresearch.com) é uma boa opção para uma avaliação inicial da maturidade. Contudo, o Modelo Corporativo ainda se encontra numa fase inicial e não fornece suporte para elaborar um plano de crescimento das dimensões Programa e Portfólio.

Característica	OPM3	P3M3	KPMMM	Prado-MMGP
Conceitos em Gerenciamento de Projetos	PMBOK® *Guide*	Literatura de gerenciamento de projetos	PMBOK® *Guide*	PMBOK® *Guide*, literatura de gerenciamento de projetos
Domínios	Projeto, Programa e Portfólio	Projeto, Programa e Portfólio	Projeto	Projeto, Programa e Portfólio
Dimensões da Maturidade	Maturidade Corporativa	Maturidade Corporativa	Maturidade Corporativa	Maturidade Setorial e Maturidade Corporativa
Avaliação do Nível de Maturidade	Questionário e diagnóstico efetuado pelo consultor	Questionário e diagnóstico efetuado pelo consultor	Questionário e diagnóstico efetuado pelo consultor	Questionário e diagnóstico efetuado pelo consultor
Medição da Avaliação	Percentual (0 a 100%) desdobrado pelas dimensões de avaliação	Numérica (entre 1 e 5)	Numérica (entre 1 e 5)	Numérica (entre 1 e 5) e Percentual de Aderência a cada nível e a cada dimensão
Simplicidade de uso	Não	Sim	Sim	Sim
Universalidade	Sim	Sim	Sim	Sim
Plano de Melhorias	Fornecido pelo sistema informatizado (disponível em CD)	Produzido pelo consultor e profissionais da empresa	Fornecido pelo sistema informatizado ou livro	Produzido pelo consultor e profissionais da empresa

Tabela 2. Comparativo dos Modelos de Maturidade

Destes quatro, apenas o OPM3 utiliza valores percentuais ou um contínuo de maturidade ao invés da classificação em níveis sugerida pelo CMM. É um modelo mais robusto e possui uma forma de avaliação diferente dos demais. Vem sendo bastante utilizado e, de acordo com PMI (2008), é adotado por 39% das empresas brasileiras que participaram do *benchmarking* de 2007. Apresenta um questionário para uma avaliação preliminar do nível de maturidade, e um número razoável de melhores práticas a serem implementadas. Estas características fazem dele um modelo mais difícil de utilizar, pelo menos num primeiro momento.

Com o intuito de sintetizar os modelos apresentados, algumas características foram observadas por se acreditar que elas descrevem um bom modelo de maturidade:

- **Conceitos em Gerenciamento de Projetos**: a base de conceitos em gerenciamento de projetos utilizada pelo modelo.
- **Domínios**: Projeto, Programa e Portfólio.
- **Dimensões da Maturidade**: Corporativa ou Setorial.
- **Avaliação do Nível de Maturidade:** formato da avaliação.
- **Medição da Avaliação:** medida utilizada pelos modelos (Percentual ou Numérica).
- **Simplicidade de uso**: facilidade de aplicação.
- **Universalidade:** o modelo pode ser utilizado por diferentes tipos de projetos.
- **Plano de Melhorias:** orientações para o desenvolvimento de um plano de melhorias após a realização da avaliação.

A seguir, a **Tabela 2** apresenta os quatro modelos de maturidade discutidos, resumindo as principais características de cada um.

Considerações Finais

Segundo alguns pesquisadores, o sucesso na execução de projetos pode ser associado ao amadurecimento da organização. Tendo em vista esta afirmação, diversos modelos foram desenvolvidos para auxiliar organizações a alcançarem sucesso no gerenciamento de seus projetos.

Os modelos apresentados neste artigo são competitivos entre si. São boas opções para a avaliação e para o planejamento de melhorias que, se aplicadas, levam a maiores níveis de maturidade. Assim, estes modelos contribuem para o sucesso e, em longo prazo, para que as organizações possam caminhar rumo à excelência ●

Referências

[Kerzner 2000] Kerzner, H. (2000), "Gestão de Projetos: As Melhores Práticas". John Wiley & Sons.

[Kerzner 2001] Kerzner, H. (2001) "Strategic Planning for Project Management using a Project Management Maturity Model" 1ª ed. John Wiley & Sons.

[OGC 2008] OGC (2008) "Portfolio, Programme & Project Management Maturity Model (P3M3)". P3M3 Public Consutation Draft Versão 1.0, Junho.

[PMI 2003] PMI – Project Management Institute - USA (2003) "Organizational Project Management Maturity Model (OPM3)", Knowledge Foundation.

[PMI 2008] PMI – Project Management Institute – Brasil (2008) "Estudo de Benchmarking em Gerenciamento de Projetos Brasil 2007", http://ww.pmi.org.br, Agosto.

[Prado e Archibald 2007] Prado, D.; Archibald, R. (2007) "Pesquisa sobre Maturidade e Sucesso em T.I. – Relatório Anual 2006", disponível em: http://www.maturityresearch.com/

[Prado 2008] Prado, D. (2008) "Maturidade em Gerenciamento de Projetos" – Série Gerência de Projetos – Volume 7 INDG Tecs.

[SEI 2001] SEI - Software Engineering Institute. (2001) CMMI - Capability Maturity Model Integration version 1.1 Pittsburgh, PA. Software Engineering Institute, Carnegie Mellon University. USA.

Avaliação Independente de Garantia da Qualidade

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

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

De que se trata o artigo:

Processo de Avaliação Independente de Garantia da Qualidade. Neste artigo apresenta-se o processo de Garantia da Qualidade com base no MPS.BR nível F, por que é necessário a realização de uma avaliação independente de Garantia da Qualidade e fatores críticos de sucesso.

Para que serve:

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

Em que situação o tema é útil:

A Garantia da Qualidade serve para apoiar a gestão e execução dos projetos, proporcionando uma avaliação objetiva dos produtos de trabalho e dos processos em relação aos padrões e procedimentos estabelecidos. Além de apoiar a implementação dos processos na organização.

A Garantia da Qualidade visa avaliar a aderência das atividades executadas e dos produtos de trabalho gerados a padrões, processos, procedimentos e requisitos estabelecidos e aplicáveis, fornecendo uma visão objetiva e independente, tanto para atividades de processo quanto de produto, em relação a desvios e pontos de melhoria, de forma a assegurar que a qualidade planejada não será comprometida [Magalhães, 2006]. Além de verificar se o processo está adequado, sendo seguido e trabalhando a favor da organização (evitando retrabalho, melhorando custos e prazos), busca-se identificar desvios o quanto antes e acompanhar a sua resolução até que seja concluído [Salviano, 2005]. A garantia da qualidade fornece suporte ao controle da qualidade por meio de evidência e confiança na habilidade do processo empregado em produzir um produto de software que atenda aos requisitos especificados [Salviano, 2005]. Ferramentas e técnicas utilizadas pela garantia da qualidade incluem auditorias (de produtos ou processos) e avaliações (appraisals ou assessments) [Kasse, 2004].

Mapeamento - Gerência da Qualidade do Projeto			
PMBOK	**CMMI**	**MPS.BR**	**ISO/IEC 12207**
8.1 Planejamento da Qualidade.	Nível 2 - Gerenciado (Área de processo: Garantia da qualidade do Processo e do Produto)	Nível F - Gerenciado (Processo: Garantia da Qualidade - GQA)	6.3.1 Implementação do processo.
8.2 Realizar Garantia da Qualidade.			6.3.2 Garantia do produto. 6.3.3 Garantia do processo.
8.3 Realizar Controle da Qualidade.			6.3.4 Sistemas de garantia da qualidade.

Figura 1. Mapeamento comparativo entre normas modelos e metodologias [Biondo, 2007]

Desta forma, o processo de Garantia da Qualidade da organização também precisa ter sua aderência verificada de forma objetiva e independente. E, nesta situação, a equipe de garantia da qualidade não pode conduzir sua própria auditoria, sendo necessário que avaliadores externos (da própria organização ou contratados) realizem este avaliação de acordo com uma periodicidade previamente definida. Esta avaliação será tratada neste artigo por "Avaliação Independente de Garantia da Qualidade", porém na indústria de software também pode ser encontrada como "Auditoria Externa de GQA" e "QA do QA".

Este artigo está organizado da seguinte forma: a seção 2 apresenta uma visão geral sobre Garantia da Qualidade; a seção 3 descreve a prática de Avaliação Independente de Garantia da Qualidade; a seção 4 apresenta fatores críticos de sucesso para realização de uma Avaliação Independente de Garantia da Qualidade; e por fim, a seção 5 trata das considerações finais.

Garantia da Qualidade

Garantia quer dizer: tornar seguro; dar proteção segura, proteger; afirmar, afiançar a veracidade de (Priberam); assegurar o cumprimento ou a realização de (Dicionário Editora da Língua Portuguesa 2009 - Acordo Ortográfico). Qualidade: o grau no qual um sistema, componente ou processo satisfaz os requisitos especificados e as necessidades e expectativas do cliente ou usuário [IEEE Std 610.12, 1990]. De acordo com modelos de qualidade, Garantia da Qualidade (GQA) diz respeito a assegurar que os processos e produtos de trabalho estão em conformidade com o plano predefinido. Ou seja, tornar seguro, assegurar o cumprimento da qualidade, dos padrões predefinidos, garantir que aquilo que foi planejado de processo para o desenvolvimento do projeto e produtos de trabalhos gerados será adequadamente utilizado e aplicado no projeto.

Garantia da Qualidade corresponde à área de processo do nível F e nível 2, respectivamente dos modelos MR-MPS.BR e CMMI. As normas ISO/IEC 12207, ISO/IEC 15504 e metodologias, como por exemplo, Project Management Body of Knowledge (PMBOK), também auxiliam na aplicação da Garantia da Qualidade, veja na **Figura 1** o comparativo entre as normas, modelos e metodologias.

De acordo com SOFTEX (2007c), os princípios da Garantia da Qualidade são:

- Avaliar objetivamente os processos executados, produtos de trabalho e serviços em relação à descrição de processos aplicáveis, padrões e procedimentos;
- Identificar e documentar itens de não-conformidades;
- Prover feedback para a equipe do projeto e gerentes como resultados das atividades de Garantia da Qualidade;
- Assegurar que as não-conformidades são corrigidas.

Estes objetivos correspondem ao propósito de Garantia da Qualidade no MPS.BR nível F, que é assegurar que os produtos de trabalho e a execução dos processos estão em conformidade com os planos e recursos predefinidos [Softex, 2007a]. Na **Tabela 1** apresentamos os resultados esperados deste processo.

Este processo serve de apoio para outros processos que serão aplicados na organização e, para isso, deve ser integrado desde o início nas atividades do projeto. Seu planejamento deve ocorrer em paralelo ao planejamento do projeto e um plano de qualidade deve ser estabelecido para que sirvam de referência com relação ao que se aplicará de procedimentos, métodos e padrões no projeto. Desta forma, GQA deve apoiar aos gerentes e suas equipes na utilização dos processos, a forma correta de aplicá-los e na identificação de melhoria nos processos aplicados.

Para assegurar que o planejado está sendo realizado e aplicado no projeto, são necessárias revisões e avaliações objetivas. Uma avaliação objetiva visa minimizar a subjetividade e a influência do revisor. Uma forma muito utilizada de realizar a avaliação objetiva é através de auditorias. Auditoria, ou qualquer tipo de avaliação objetiva, deve ser realizada por um grupo ou pessoa que não tenha envolvimento direto no projeto, ou seja, deve ser independente.

De acordo com Biondo (2007), a garantia da qualidade visa documentar e identificar as não-conformidades de acordo com a qualidade planejada para o projeto. Para tal identificação, uma das técnicas utilizadas é a auditoria de qualidade. A auditoria verificará por meio de *checklists* os produtos e processos para certificar a conformidade com os padrões, diretrizes, especificações e procedimentos baseados nos planos predefinidos para garantir desempenho e objetividade nos seus resultados.

Grupo independente caracteriza-se por não ter envolvimento direto nas atividades que serão verificadas, auditadas, ou seja, a realização de uma auditoria de determinado projeto não poderá ser rea-

GQA 1. A aderência dos produtos de trabalho aos padrões, procedimentos e requisitos aplicáveis é avaliada objetivamente, antes dos produtos serem entregues ao cliente e em marcos predefinidos ao longo do ciclo de vida do projeto.
GQA 2. A aderência dos processos executados às descrições de processo, padrões e procedimentos é avaliada objetivamente.
GQA 3. Os problemas e as não-conformidades são identificados, registrados e comunicados
GQA 4. Ações corretivas para não-conformidades são estabelecidas e acompanhadas até as suas efetivas conclusões. Quando necessário, o escalonamento das ações corretivas para níveis superiores é realizado, de forma a garantir sua solução.

Tabela 1. Resultados esperados GQA

lizada por um integrante da equipe deste projeto, é necessário que um indivíduo que não tenha participação nas atividades do projeto seja o auditor. Isto assegura a autoridade e autonomia organizacional, sem envolvimento direto dos responsáveis pelo desenvolvimento do produto de software ou pela execução do processo no projeto [ISO/IEC 12207:1995].

Para isto, a empresa poderá manter uma equipe específica para realizar auditorias de projetos e, caso não seja financeiramente viável manter uma equipe para este fim, a empresa pode adotar auditorias cruzadas, ou seja, um integrante de cada equipe, ou ser nomeados pelo menos de duas a três pessoas de equipes diferentes a auditores, sendo que o projeto A não poderá ser auditado pelo representante de sua equipe. Exemplo: Projeto A é auditado pelo representante do projeto B, Projeto B é auditado pelo representante do projeto C e Projeto C é auditado pelo representante do projeto A. Assim estará garantida a independência do auditor com relação aos resultados do projeto auditado, para que não haja influência em seu envolvimento.

Outra característica de GQA é garantir que os produtos a serem entregues serão avaliados. Isto assegura a qualidade dos produtos que possuem entregas ao longo do projeto. Nestas avaliações, o auditor verificará o cumprimento da etapa, integridade dos documentos para que seja entregue aquilo que foi planejado. De acordo com o SOFTEX (2007c) a Garantia da Qualidade é uma forma da organização se resguardar de possíveis faltas de qualidade e por isso a importância de ser conduzida ao longo do desenvolvimento do projeto, para que as correções sejam feitas com o mínimo de retrabalho.

Estas avaliações devem ocorrer ao longo do projeto, não somente nos marcos, verificando também o atendimento dos processos no que diz respeito a procedimentos, métodos, padrões definidos para o projeto. As auditorias deverão analisar os produtos de trabalho desde sua concepção, na medida em que serão produzidos. Estas avaliações também devem conter questões para identificar melhorias no processo.

Quando identificadas não-conformidades, estas devem ser registradas, comunicadas e acompanhadas. De acordo com Albertuni (2008), não-conformidades caracterizam falhas na aplicação do processo e nos produtos de trabalho. A falta de aplicação do processo também caracteriza não conformidades. A não-conformidade é composta por causa e efeito, então quando identificada, deve-se:

- Analisar a não-conformidade: verificar seu impacto;
- Identificar e tratar os efeitos: quais são as conseqüências da não-conformidade (pela não realização do processo ou falha em sua aplicação) e tratar seus efeitos;
- Tratar a causa: identificar o motivo pelo qual ocorreu desvio;
- Verificar abrangência: identificar se a não-conformidade não impacta em outros processos;
- Definir um plano de ação: registrar a não-conformidade e uma ação para tratamento, como contingência ou mitigação. Para cada ação deve ser definido um responsável para tratamento e prazo para realização. O prazo para o tratamento pode estar relacionado à severidade da não-conformidade.
- Acompanhar plano de ação: o plano de ação deve ser acompanhado até seu fechamento, ou seja, realização. Desta forma garantirá a resolução da não-conformidade.

Quando uma não-conformidade não é tratada no seu prazo definido, ou então caracterizada como reincidência (problema identificado seqüencialmente), o tratamento deve ser escalonado para um nível superior do responsável pelo tratamento, até que seu tratamento seja atendido. O escalonamento também poderá ocorrer de acordo com o tipo de severidade da não-conformidade e o ciclo de vida do projeto se encontra.

Avaliação Independente de Garantia da Qualidade

De acordo com os atributos de processo (atributos de Processos são constituídos de um conjunto de resultados esperados (RAPs). Um determinado conjunto de atributos de processos demonstra a capacidade de refinamento para realização do processo de acordo com o nível de maturidade. Quanto maior o nível, maior será a capacidade necessária para realização do processo) representados na **Tabela 2**, conforme o nível de maturidade no qual está o GQA, este processo deverá ser aplicado para todos os processos definidos na organização. Isto por que estes resultados correspondem à capacidade do nível F, ou seja, para todos os processos que compõem este nível, ou superior, já que a capacidade dos níveis inferiores são incorporados aos níveis superiores. Isto só não será aplicado ao nível G, pois é um nível abaixo do nível F, no qual não se aplica Garantia da Qualidade e o número de atributos de processo para atendimento também é menor.

Como a aplicação de avaliação objetiva da aderência dos processos executados de acordo com o processo e seus padrões está determinada em um atributo de processo, é necessário que o processo de Garantia da Qualidade também seja avaliado objetivamente numa organização. Isto quer dizer que tudo que se aplica a uma avaliação objetiva de projeto, também é verificável nas atividades de GQA.

Ao verificar GQA, é necessário que um auditor independente faça a avaliação, lembrando que auditor independente (ou grupo independente) caracteriza-se por não realizar a atividade de GQA nos projetos da organização. Nenhum auditor de projeto poderá auditar, verificar, aplicar revisões no processo de GQA, pois estará verificando se as suas próprias atividades de GQA (ou seja, resultados esperados de GQA – **Tabela 1**) estão sendo aplicadas na organização.

Abaixo, segue uma relação dos resultados esperados de GQA com relação a algumas verificações que são realizadas neste tipo de atividade:

AP 2.1 O processo é gerenciado RAP 9. (A partir do Nível F) A aderência dos processos executados às descrições de processo, padrões e procedimentos é avaliada objetivamente e são tratadas as não conformidades.
AP 2.2 Os produtos de trabalho do processo são gerenciados RAP 12. Os produtos de trabalho são avaliados objetivamente com relação aos padrões, procedimentos e requisitos aplicáveis e são tratadas as não conformidades.

Tabela 2 - Atributos de Processo Nível F

GQA 1 - A aderência dos produtos de trabalho aos padrões, procedimentos e requisitos aplicáveis é avaliada objetivamente, antes dos produtos serem entregues ao cliente e em marcos predefinidos ao longo do ciclo de vida do projeto.

• Isto pode ser verificado a partir dos planos de Qualidade e do Projeto, no qual devem expressar as atividades de qualidade em marcos predefinido e antes de entregas. Os registros de realização de avaliações (instancias de *checklists* aplicados, entrevistas, questionários, etc) também são evidências para assegurar a obtenção deste resultado.

GQA 2 - A aderência dos processos executados às descrições de processo, padrões e procedimentos é avaliada objetivamente.

• Na avaliação realizada, o auditor verifica se os processos definidos e seu aparato necessário para seu atendimento estão sendo aplicados e avaliados nos projetos da organização. São encontradas evidências para esta prática também no método utilizado para a avaliação (auditorias por meios de *checklists*, questionários, etc). Deve ficar evidente a avaliação da aplicação dos padrões, procedimentos, métodos definidos para os projetos.

GQA3 - Os problemas e as não-conformidades são identificados, registrados e comunicados

• Deve haver um registro das não-conformidades e tratamento para as mesmas, desde seu registro, motivo, comunicação para os responsáveis para conhecimento do problema encontrado.

GQA 4 - Ações corretivas para não-conformidades são estabelecidas e acompanhadas até as suas efetivas conclusões. Quando necessário, o escalonamento das ações corretivas para níveis superiores é realizado, de forma a garantir sua solução.

• Não basta identificar, registrar e comunicar uma não-conformidade. Para sua completeza é necessário que planos de ações tenham sido definidos e acompanhados até seu fechamento. Neste caso, o auditor deve verificar se para todas as não-conformidades apontadas há um plano de ação para resolução das mesmas. O plano de ação deve ser composto por ação de correção (e também de prevenção), responsável pelo tratamento e prazo para atendimento. Neste resultado é importante verificar em que *status* estão todas não-conformidades apontadas, para avaliar se estão sendo acompanhadas até sua conclusão. Quando o auditor identificar que existem não-conformidades pendentes de resolução, este deve certificar que estas não-conformidades foram escalonadas, ou seja, seu tratamento foi encaminhado para o superior imediato do responsável por seu fechamento.

Além de verificar os resultados esperados de GQA, uma avaliação independente também poderá avaliar os atributos de processos. Neles, resultados como definição de política organizacional para a realização de avaliações objetivas podem ser identificados, entre outras.

A avaliação independente de Garantia da Qualidade deve estar sob total aplicação pelo auditor externo (ou auditor independente) para não ter influência no resultado. Neste caso, desde o questionário à condução de entrevistas devem ser planejados e realizados pelo auditor externo. A equipe envolvida não poderá exigir aplicação de um questionário/checklist desenvolvido internamente para sua avaliação, desta forma a organização garantirá sua objetividade sem influências e/ou tendências.

Fatores críticos de sucesso

Abaixo, são apresentados os principais fatores críticos para o sucesso de uma avaliação independente de Garantia da Qualidade na visão do auditor externo e também com relação à Organização:

Para o Auditor:

• Independência: Total independência para acesso ao repositório do projeto e todos os artefatos relacionados à Garantia da Qualidade, bem como documentos de apoio para a realização das atividades de um auditor interno. Para isso, a empresa pode providenciar com antecedência um acordo de confidencialidade entre o auditor (que terá acesso a dados de projetos e de seus processos) como forma de prevenção pelo uso de informações inadequadas e antiéticas.

• Transparência e crítica: Se o auditor identificar problemas ou pontos de atenção em determinada atividade, este deve expor para tratamento antes que se torne um ponto fraco. O mesmo pode ser feito para os pontos fortes, salientar o que a organização faz além do esperado, que supera as expectativas deste processo, apresentando os pontos positivos e não somente os problemas encontrados. Os pontos fracos, melhorias e problemas encontrados na execução das atividades do processo também devem ser apontados. Apresentar para organização os pontos necessários para o sucesso na aplicação de Garantia da Qualidade. Não é necessário apresentar o que fazer, mas identificar práticas para que a organização defina a melhor forma e o que pode ser aplicado para sua empresa.

• Saber ouvir: Item muito importante em casos de entrevistas. É necessário saber ouvir o que os entrevistados têm a dizer, de forma neutra, concentrada, não ficar pensando em outras coisas enquanto o entrevistado fala durante a entrevista, pois informações importantes podem ser perdidas em um momento de descontração e fica deselegante solicitar que o entrevistado repita o que acabou de dizer a todo instante.

• Neutralidade: Ao realizar uma avaliação independente, é necessário desligar-se das atividades e práticas que são feitas pelo auditor, ou da forma que o auditor realizaria. Cada empresa/organização define e realiza as atividades de acordo com seu perfil, e neste momento deve ser avaliado se os resultados propostos de Garantia da Qualidade estão sendo atendidos, e não a maneira que são feitos. Claro que se a prática pode ser prejudicada em alguma etapa, o auditor poderá apontar como ponto de atenção ou oportunidade de melhoria.

• Boa Interpretação: saber interpretar os fluxos, processos definidos, *templates* da organização, mantendo a neutralidade, conseguindo compreender com facilidade o processo definido para a organização.

• Clareza ao expor: Críticas nem sempre são bem-vindas e neste caso, a maneira e a clareza ao expor os problemas encontrados deve ser de forma cuidadosa, bem como no esclarecimento de dúvidas.

• Acesso ao processo com antecedência: quando possível, solicitar documentação de processos e projetos que serão avaliados para melhor entendimento do contexto da organização. Desta forma é

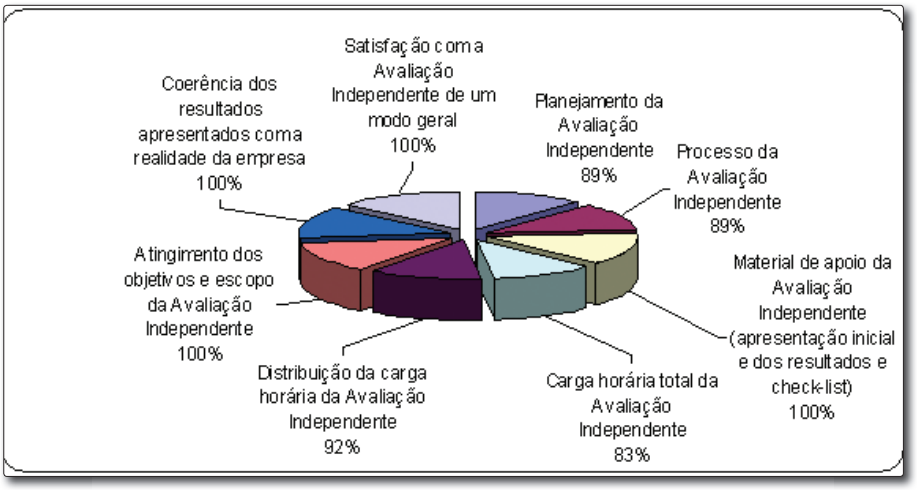


Figura 2. Pesquisa de Satisfação da Avaliação Independente - Resultados Ótimo e Bom

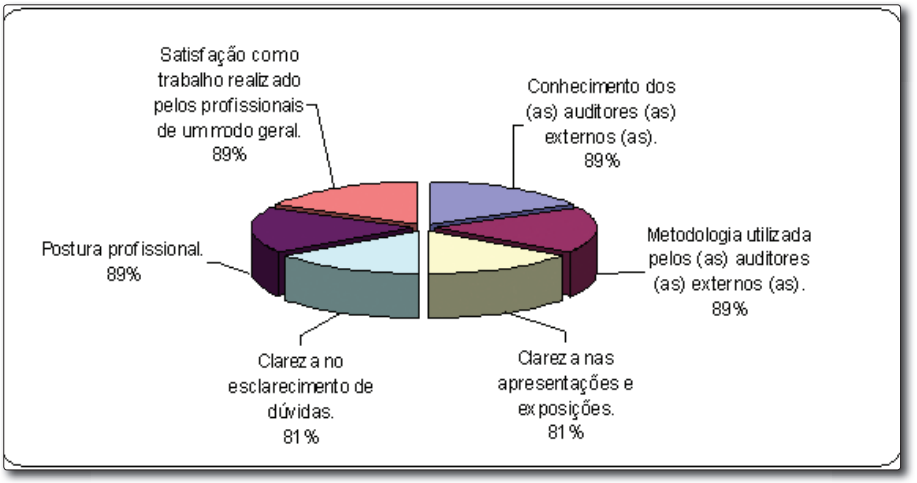


Figura 3. Pesquisa de Satisfação Conduta dos Auditores Externos – Resultados Ótimo

possível levantar questões e dúvidas para esclarecimento antes da avaliação iniciar.

Organização

• Saber ouvir: o recebimento de críticas nem sempre é bem-vindo, ainda mais quando criticadas por uma pessoa externa, que não está habituada as práticas e o perfil da organização. Por isso, saber ouvir o que o auditor tem a trazer de sua experiência e conhecimento é necessário e suas sugestões podem agregar valor ao processo. Ouça e compreenda o que ele tem a dizer. Depois, em outro momento, os responsáveis pela Qualidade da organização podem se reunir e avaliar o que pode e não pode e quando será aplicado. Lembre-se, o auditor externo irá apresentar os pontos fortes, pontos fracos, pontos de atenção e oportunidade de melhoria para melhor atingir os resultados esperados do modelo adotado pela organização.

• Disponibilidade da equipe: durante a avaliação, é importante a disponibilidade da equipe que realiza as atividades de Garantia da Qualidade, para esclarecimento de dúvidas e auxiliar na localização dos artefatos e evidências.

• Repositório da avaliação: manter um repositório exclusivo e atualizado com as evidências de processo e dos projetos, além do instrumento de verificação a ser utilizado durante a avaliação.

• Infra-estrutura: disponibilizar a infra-estrutura necessária para execução da avaliação, conforme seu planejamento e inclusive, executar testes de funcionamento das ferramentas e do ambiente de trabalho. Além disto, a organização deve estar preparada para fazer demonstração das ferramentas adotadas durante a avaliação.

A partir dos resultados de uma pesquisa aplicada em empresas que utilizaram de serviços de avaliação independente entre 2006 a 2008, a partir da contratação de auditor externo podemos avaliar outros fatores que são críticos para o sucesso da avaliação independente, conforme gráficos apresentados nas **Figuras 2** e **3**. A pesquisa estava organizada em questões referentes à condução da avaliação independente e com relação à conduta dos auditores externos.

Em cômputo geral, o resultado da pesquisa contabilizando os resultados obtidos em "Ótimo" e "Bom" é de 91% de satisfação. Apenas três itens foram considerados "Regular", sendo que a distribuição destes na pesquisa está nas questões referentes à "Carga horária total da Avaliação Independente" com 17% regular (seção correspondente à condução da Avaliação Independente); "Clareza nas apresentações e exposições" com 8% regular e "Clareza no esclarecimento de dúvidas" também com 8% regular, ambas na seção correspondente à conduta dos auditores externos. No resultado consolidado, itens regulares representam 3% das respostas. Os demais 6% estão distribuídos em questões que não puderam ser respondidas por alguns participantes (justificado pelo fato de não terem participado desde o planejamento a resultados finais, sendo pontual sua participação). Em nenhuma pesquisa foram constatados resultados como "Ruim".

Entre os principais pontos a melhorar e com certeza com maior dificuldade para seu sucesso está a carga horária da avaliação. Isto porque depende do tempo que o auditor tem para realizar seu trabalho e da disponibilidade dos envolvidos. Este item está diretamente relacionado com o planejamento. O que ocorre muitas vezes é que a avaliação é marcada com base no conhecimento do auditor e sem prévio conhecimento do processo e estágio de maturidade que a organização se encontra, bem como o número de projetos que serão avaliados. Uma forma de agilizar e melhorar o tempo desta atividade é fazer uma análise prévia do processo e dos projetos, para que o auditor possa planejar com maior acurácia.

Como apresentado nos itens acima, é muito importante a clareza do auditor, para apresentar os resultados, bem como

para esclarecer dúvidas. Falhas nestes pontos podem resultar em má interpretação pela organização no momento de atender os problemas apresentados.

Conclusão

Qualquer que seja a organização, a missão da garantia da qualidade e do auditor é deixar o ambiente melhor do que encontrou. Seja qual for o nível de maturidade de uma organização, a garantia da qualidade é essencial para o sucesso de um programa de melhoria [Magalhães, 2006].

Sendo assim, para conduzir uma avaliação independente de garantia da qualidade são necessárias certas habilidades interpessoais e um profundo conhecimento da norma ou modelo de referência adotado como base na organização, além de seus processos organizacionais. Também se deve levar em consideração os fatos e as evidências identificadas, buscando o consenso de todos os envolvidos antes de prover julgamentos e resultados da avaliação.

Pode-se concluir também que uma avaliação independente de garantia da qualidade é considerada bem sucedida quando se percebe uma coerência entre os resultados apresentados com a realidade atual da empresa ●

Links

MPS.BR: Site Oficial
www.softex.br/mpsbr/_home/default.asp

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

Albertuni, Isabel. Implantação da Avaliação e Melhoria do Processo Organizacional Alinhado com Critério Processos (PGQP). Relatório de Estágio Supervisionado. Graduação em Administração com Habilitação em Análise de Sistemas, Faculdades Rio-Grandenses (FARGS), Porto Alegre, 2008.

ASSOCIAÇÃO PARA PROMOÇÃO DA EXCELÊNCIA DO SOFTWARE BRASILEIRO (SOFTEX). MPS.BR – Guia Geral (Versão 1.2), http://www.softex.br/mpsbr/_guias/MPS.BR_Guia_Geral_V1.2.pdf, 2007a.

ASSOCIAÇÃO PARA PROMOÇÃO DA EXCELÊNCIA DO SOFTWARE BRASILEIRO (SOFTEX). MPS.BR – Guia de Implementação – Parte 3: Nível E (Versão 1.1), http://www.softex.br/mpsbr/_guias/MPS.BR_Guia_de_Implementacao_Parte_3_v1.1.pdf, 2007b.

ASSOCIAÇÃO PARA PROMOÇÃO DA EXCELÊNCIA DO SOFTWARE BRASILEIRO (SOFTEX). MPS.BR – Guia de Implementação – Parte 2: Nível F (Versão 1.1), http://www.softex.br/portal/mpsbr/_guias/MPS.BR_Guia_de_Implementacao_Parte_2_V1.1.pdf, 2007c.

Biondo, Aline. Uma Ferramenta para Garantia da Qualidade Aplicada na Implementação de Sistemas. Graduação em Sistemas de Informação, Universidade Luterana do Brasil (ULBRA), Canoas, 2007.

ISO/IEC 12207:1995 - The International Organization for Standardization and the International Electrotechnical Commission. ISO/IEC 12207: Information technology – Software life cycle processes, Geneve: ISO, 1995.

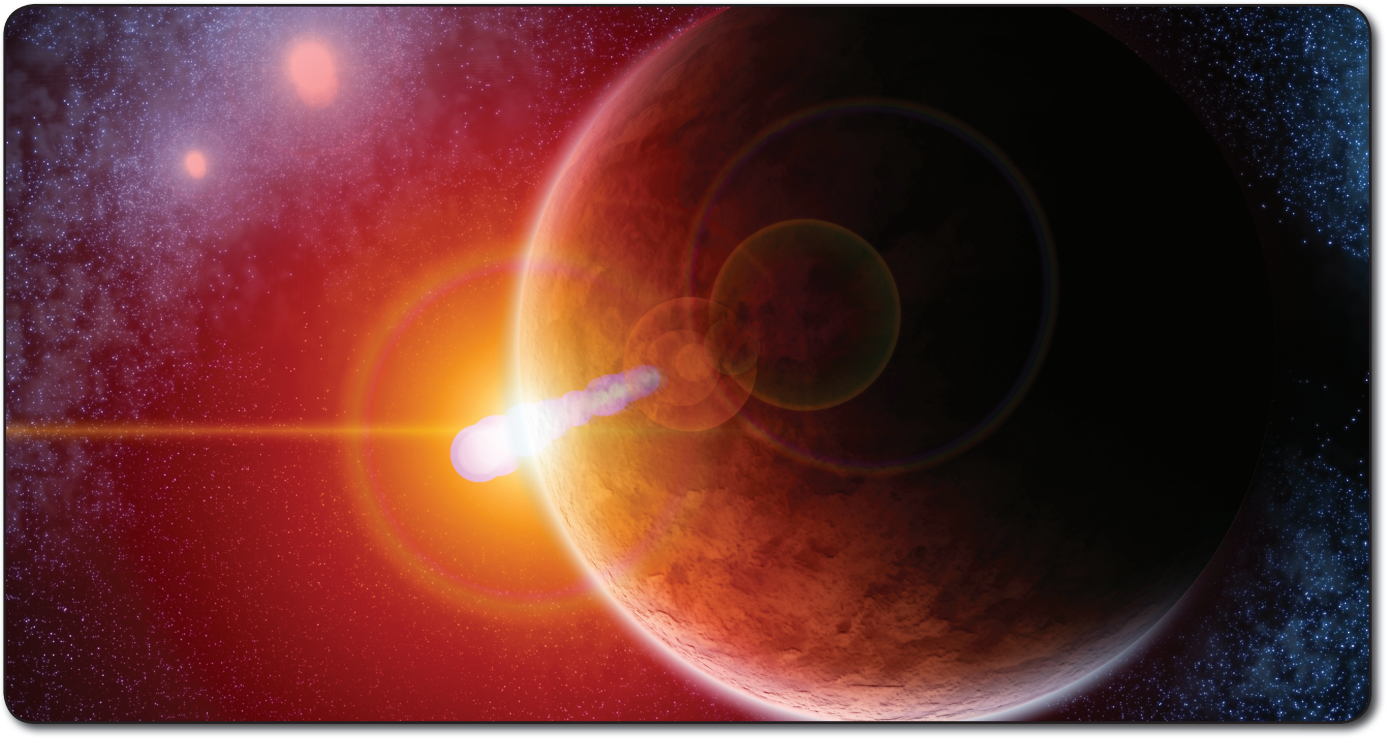
Kasse, Tim. "Practical Insight into CMMI". Artech House Computing Library, 2004.

Magalhães, Ana Liddy Cenni de Castro. A Garantia da Qualidade e o SQA: Sujeito Que Ajuda e Sujeito Que Atrapalha – ProQualiti Vol.2, N.2, novembro 2006.

IEEE Std 610.12, 1990 - IEEE Standard Glossary of Software Engineering Terminology, Institute of Electrical and Electronics Engineers, 1990

Salviano, C. F.; Tsukumo, A. N. Coop-MPS: Um método para projetos cooperativos de melhoria de processo de software e sua aplicação com o Modelo MPS.BR – ProQualiti Vol.1, N.2, novembro 2005.

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



Testes de Mutação com o plug-in MuClipse

Victor Vidigal Ribeiro
victorvidigal@gmail.com
É Graduado do Curso de Bacharelado em Sistemas de Informação pela Faculdade Metodista Granbery e Analista de Sistemas do CAEd (Centro de Políticas Públicas e Avaliação da Educação).

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

De que se trata o artigo:
Apresentação dos principais conceitos sobre testes de mutação e demonstração do funcionamento da técnica utilizando um *plug-in* para Eclipse chamado **MuClipse**.

Para que serve:
Além de testar o software, também pode ser considerada uma técnica para auxiliar no processo de melhoria dos casos de testes para torná-los mais eficientes.

Em que situação o tema é útil:
Testes de mutação podem ser utilizados como técnica para melhorar ou afirmar a qualidade dos casos de testes.

Introdução

O desenvolvimento de sistemas não é uma tarefa simples e, por este motivo, precisa que artifícios sejam utilizados para diminuir os defeitos que podem ser introduzidos nesse processo.

Teste de *software* contribui para a produção de um produto final de maior qualidade e é um processo bastante aceito e difundido atualmente. Porém, uma das questões mais complexas que norteia os testes de *software* é como garantir que os casos de testes gerados são eficientes, ou seja, garantir que os casos de testes encontrem a maior quantidade de defeitos possíveis.

Dentre outras técnicas que podem ser utilizadas para se testar o *software*, ou melhorar os casos de testes, pode-se destacar o Teste de Mutação.

Este artigo tem como finalidade principal demonstrar a utilização de um plug-in desenvolvido para se aplicar testes de mutação chamado **MuClipse**, além de explicar os principais conceitos relacionados a este tipo de teste.

Testes de Mutação

Por mais simples que um *software* possa ser, os caminhos de execução possíveis na sua utilização podem ser muito grandes, com isto é praticamente impossível testar um sistema por completo e afirmar que o sistema não contém defeitos.

Por outro lado, os tipos de problemas que podem ser encontrados em um *software* podem ser finitos. É nesta premissa que os testes de mutação se embasam.

Visão geral

A técnica de teste de mutação é divida em quatro etapas. Primeiro, operadores de mutação são utilizados sobre o programa original realizando pequenas alterações sobre esse programa e gerando variações, ou seja, mutantes. Depois disto, o programa original é testado e deve ser aprovado pelos casos de testes. Após a execução do programa original, os mutantes são executados com o caso de teste e, se forem reprovados pelo caso de teste são considerados mortos. Caso algum mutante não seja reprovado, esse mutante é considerado vivo e precisa ser analisado. Para os mutantes vivos pode-se considerar que este mutante é equivalente ao programa original ou que os casos de testes não são bons o suficiente para testar o programa. Neste último caso, os casos de testes devem ser alterados e, com isso, se tornam cada vez mais eficientes.

Basicamente, testes de mutação propõem a criação de variações de um programa anteriormente desenvolvido, ou seja, mutantes, posteriormente à construção de casos de testes com a finalidade de provar que as variações do programa original não estão corretas. Assim, por eliminação, pode-se constatar que o programa original está correto.

O primeiro passo para a execução dos testes de mutação é a criação dos mutantes que são criados executando uma série de operadores chamados operadores de mutação. Esses operadores alteram partes do programa original, como, por exemplo, trocar um sinal de ">" por "<", e assim formar os mutantes.

O segundo passo que deve ser feito é a execução dos casos de testes sobre o programa original. Os casos de testes devem aprovar o programa original concluindo-se assim que ele está correto com base nos casos de testes executados.

No próximo passo, os mutantes gerados devem ser executados com os mesmos casos de testes que o programa original foi executado. Caso algum caso de teste falhe significa que são abrangentes o suficiente para demonstrarem a diferença entre o mutante e o programa original. Neste caso, o mutante que fez com que o caso de teste falhasse está morto e, por conseqüência, é descartado.

Pode acontecer de algum mutante sobreviver aos testes. Neste caso, podem ser consideradas duas situações: 1) Caso de testes não é bom o suficiente, pois não identificou a modificação no mutante; 2) Equivalência entre o programa original e o mutante.

O último passo da técnica é a análise dos mutantes que sobreviveram. Esse passo exige uma maior intervenção humana para decidir se o mutante sobreviveu porque os casos de testes não são abrangentes o suficiente e assim criar mais casos de testes, ou modificar os existentes. Ou ainda se o mutante é equivalente ao programa original e com isto descartar o mutante. Mutantes ainda são descartados quando gerarem situações impossíveis de serem executadas, como uma condição que nunca se torna verdadeira ou um laço infinito.

Neste último passo, é possível perceber claramente como os testes de mutação ajudam a melhorar os casos de testes caso algum mutante não seja morto.

Alguns passos como, por exemplo, a geração dos mutantes e execução dos mutantes com os casos de testes, são tarefas repetitivas e praticamente inviáveis sem apoio de uma ferramenta para automatizar essas tarefas. Neste artigo será exposto um exemplo prático da utilização de testes de mutação utilizando o *plug-in* **MuClipse** versão 1.2 instalado no **Eclipse** juntamente com o *framework* de testes **JUnit 4.0**.

Estudo de Caso

O **MuClipse** é um *plug-in* para o IDE (Ambiente de Desenvolvimento Integrado) **Eclipse** que tem por finalidade auxiliar no processo de execução de testes de mutação em programas desenvolvidos em Java. Ele funciona como uma ponte entre o **MuJava**, um sistema para testes de mutação em Java, e o **Eclipse**.

Será criado um projeto chamado **Aprovacao** que conterá uma classe **Aluno** com os atributos **nota1**, **nota2**, **notaFinal**, **frequencia**, os métodos *gets* e *sets* desses atributos e um método chamado **calcularAprovacao**. Também será criado um caso de teste chamado **TestarAprovacao** que tem por finalidade executar o método **calcularAprovacao** da classe **Aluno**.

A instalação do **MuClipse** é bastante simples, basta abrir o **Eclipse** e acessar o menu **Help / Software Updates...**, clicar sobre a aba **Available Software** e depois sobre o botão **Add Site**. Neste momento uma janela é exibida onde deve ser digitado **http://muclipse.sourceforge.net/** no campo **Location** e clicar sobre o botão **Ok**. Depois disto, deve-se marcar o endereço adicionado anteriormente, como pode ser visto na **Figura 1**, e clicar sobre o botão **Install**.

Para iniciar o caso de teste deve ser criado um projeto chamado **Aprovacao**. Depois disto deve ser criada a estrutura do projeto que conterá dois *Source Folders* chamados **src** e **testset** cada um contendo

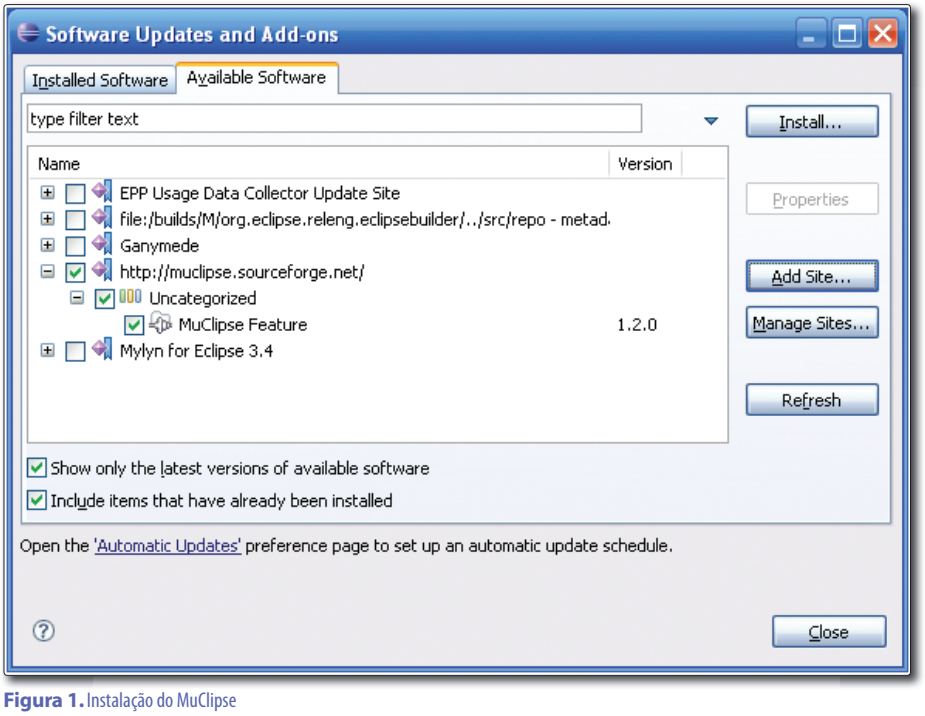


Figura 1. Instalação do MuClipse

Listagem 1. Classe Aluno

```
package br.com.esmagazine;

public class Aluno {

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

    public float getNota1() {
        return nota1;
    }
    public void setNota1(float nota1) {
        this.nota1 = nota1;
    }
    public float getNota2() {
        return nota2;
    }
    public void setNota2(float nota2) {
        this.nota2 = nota2;
    }
    public float getNotaFinal() {
        return notaFinal;
    }
    public void setNotaFinal(float notaFinal) {
        this.notaFinal = notaFinal;
    }
    public int getFrequencia() {
        return frequencia;
    }
    public void setFrequencia(int frequencia) {
        this.frequencia = frequencia;
    }


    public boolean calcularAprovacao(){
        float media;

        if (frequencia < 75) {
            return false;
        } else {
            media = (nota1 + nota2) / 2;
            if (media < 3) {
                return false;
            } else {
                if (media >= 7) {
                    return true;
                } else {
                    if (((media + notaFinal) / 2) >= 5) {
                        return true;
                    } else {
                        return false;
                    }
                }
            }
        }
    }
}
```

o pacote **br.com.esmagazine**. Além disto, deve ser criado os *Folders* **lib** e **result**. Com isto, é obtido um projeto com uma estrutura semelhante à da **Figura 2**.

Depois de criar a estrutura do projeto como demonstrado anteriormente, deve ser configurado onde os arquivos **.class** serão gerados. Isto é necessário devido a uma exigência do **MuClipse** que não aceita que o arquivo **.class** dos casos de testes fiquem no mesmo diretório do arquivo **.class** da aplicação.

Para isto, deve-se clicar com o botão direito sobre **src** e, logo em seguida, escolher a opção **Build Path** / **Configure Output Folder**. Uma janela é exibida onde a opção **Specific output folder (path relative to 'Aprovacao')** deve ser selecionada e posteriormente digitar **bin**, como mostrado na **Figura 3**.

O mesmo passo deve ser seguido para o *Source Folder* **testset**, porém deve-se direcionar o Output Folder para **testset**.

Depois disto, a classe **Aluno** deve ser criada dentro do pacote **br.com.esmagazine** do Source Folder **src**. Para isto clique sobre este pacote e escolha a opção **New** / **Class**. Uma janela será exibida onde **Aluno** deve ser digitado no campo **Name**. Essa classe deve conter o código apresentado na **Listagem 1**.

Esta classe contém o método **calcularAprovacao** que retorna verdadeiro caso o aluno seja aprovado e falso caso contrário. Para isto o método primeiro verifica a freqüência do aluno. Caso o valor da freqüência seja menor que 75 o aluno é reprovado por infreqüência. Caso seja maior, então é calculada a média das duas primeiras notas do aluno. Se esta média for menor que três então o aluno é reprovado, porém se esta média for maior ou igual a sete o

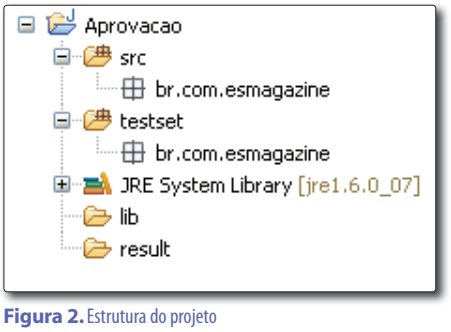


Figura 2. Estrutura do projeto

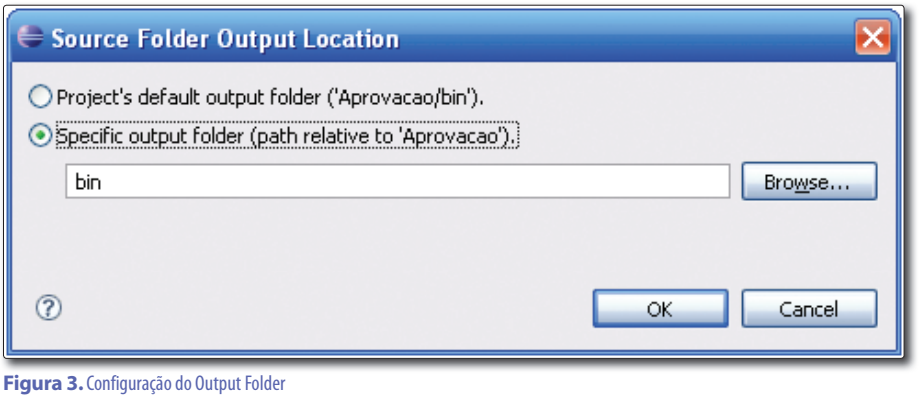


Figura 3. Configuração do Output Folder

aluno é aprovado. Caso a média entre as duas primeiras notas for maior ou igual a três e menor que sete então a nota final é considerada. A média entre a nota final e a média das duas primeiras notas deve ser maior ou igual a cinco para que o aluno seja aprovado pela nota final, caso contrário o aluno é reprovado.

Depois de criar a classe **Aluno**, os casos de testes para executar essa classe devem ser criados dentro do pacote **br.com.esmagazine** no *Source Folder* **testset**. Para isto, clique com o botão direito sobre esse pacote e escolha a opção **New / JUnit Test Case**. Com isto, uma janela é exibida onde deve ser digitado **TestarAprovacao** no campo **Name** e, em seguida, clicar sobre o botão **Finish**. Com isto uma classe que estende a classe **TestCase** do framework **JUnit** é criada.

A classe **TestarAprovacao** deve conter os casos de testes **testReprovadoFrequencia**, **testAprovadoNota**, **testReprovadoNota**, **testReprovadoFinal** e **testAprovadoFinal**. Além disto, o **MuClipse** exige que os métodos **setUp**, chamado antes da execução de cada caso de teste, e **tearDown**, chamado depois da execução de cada caso de teste, sejam sobrescritos utilizando o modificador de acesso **public**. Originalmente esses métodos são declarados como **private**.

A **Listagem 2** demonstra como a classe **TestarAprovacao** deve ficar após completamente construída, que deve ser executada em relação ao programa original para verificar sua conformidade com estes casos de teste.

Listagem 2. Casos de teste TestarAprovacao

```
package br.com.esmagazine;

import junit.framework.TestCase;

public class TestarAprovacao extends TestCase {

   Aluno aluno;

   public void setUp(){
    aluno = new Aluno();
   }

   public void testReprovadoFrequencia(){
    aluno.setFrequencia(74);
    assertFalse(aluno.calcularAprovacao());
   }

   public void testAprovadoNota(){
    aluno.setNota1(7);
    aluno.setNota2(7);
    aluno.setFrequencia(75);
    assertTrue(aluno.calcularAprovacao());
   }

   public void testReprovadoNota(){
    aluno.setNota1(2);
    aluno.setNota2(3);
    aluno.setFrequencia(75);
    assertFalse(aluno.calcularAprovacao());
   }

   public void testReprovadoFinal(){
    aluno.setNota1(7);
    aluno.setNota2(6);
    aluno.setNotaFinal(3);
    aluno.setFrequencia(75);
    assertFalse(aluno.calcularAprovacao());
   }

   public void testAprovadoFinal(){
    aluno.setNota1(3);
    aluno.setNota2(3);
    aluno.setNotaFinal(7);
    aluno.setFrequencia(75);
    assertTrue(aluno.calcularAprovacao());
   }

   public void tearDown(){
    System.out.println("Teste executado");
   }
}
```

Para continuar o exemplo é preciso atender a mais uma exigência do **MuClipse**: importar um arquivo chamado **extendedOJ.jar**. Este arquivo pode ser baixado através do site da ferramenta e deve ser copiado para a pasta **lib** criada anteriormente. Logo em seguida, deve-se adicionar este arquivo ao *Path* da aplicação clicando com o botão direito sobre ele e depois escolhendo a opção **Build Path / Add to Build Path.**

Gerando os Mutantes

Neste ponto, a classe a ser testada e os casos de testes estão criados e é possível iniciar a criação dos mutantes. Para isto, clique com o botão direito sobre o projeto e escolha a opção **Run As / Run Configurations.** Uma janela é exibida onde as configurações de execução do projeto podem ser configuradas. Depois de instalado, o **MuClipse** cria duas novas opções nesta janela, **MuClipse: Mutants**, que fornece a funcionalidade de gerar os mutantes e **MuClipse: Tests**, onde os mutantes podem ser executados com os casos de testes.

Clique com o botão direito sobre a opção **MuClipse: Mutants** e escolha a opção **New**. Com isto, uma janela semelhante a da **Figura 4** é exibida, sendo possível configurar a maneira como os mutantes serão gerados.

O primeiro campo que deve ser configurado nesta janela é o campo **Name**, onde deve ser digitado o nome desta configuração. O segundo campo, **Project**, deve ser preenchido com o nome do projeto.

Uma exigência do **MuClipse** é que o diretório em que os mutantes serão gerados tenha o nome de **result** e que este diretório não esteja no *Path* da aplicação. Portanto, o campo **Mutants Output** deve ser preenchido com result.

O campo **Classes Folder** deve apontar para o diretório onde estão os arquivos *bytecode*. Anteriormente foi configurado o **Output Folder** para **bin**, por este motivo, esse campo deve ser configurado para o mesmo valor **bin**.

Os dois próximos campos, **Testset Folder** e **Source Folder**, devem ser preenchidos respectivamente com o diretório onde estão os casos de testes e o diretório onde

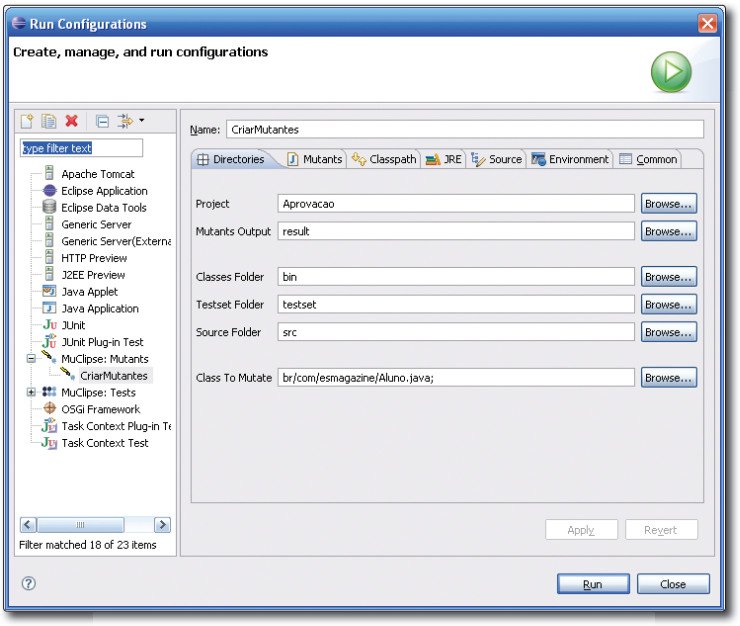


Figura 4. Criação de Mutantes

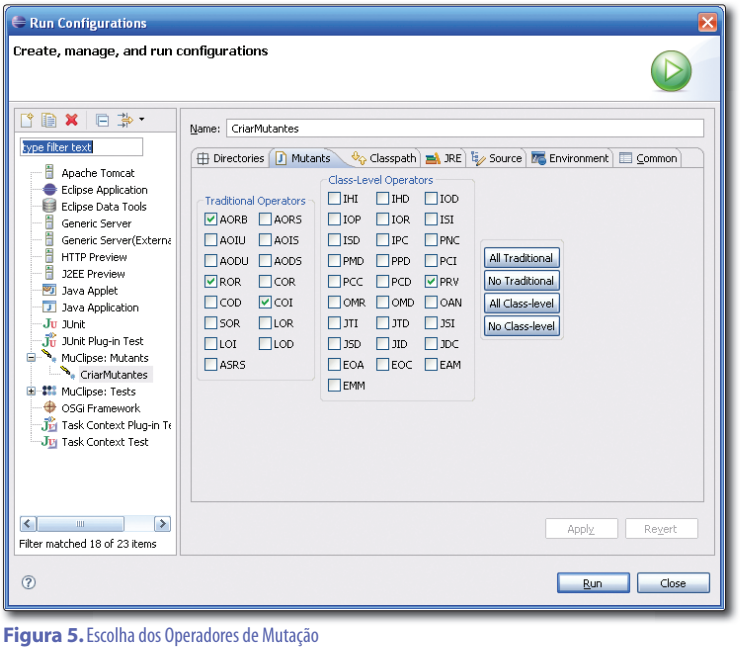


Figura 5. Escolha dos Operadores de Mutação

está o código fonte de aplicação, ou seja, **testset** e **src**.

O último campo deve ser preenchido com a classe a ser testada, no caso a classe **Aluno**.

Depois de configurar os diretórios da aplicação, é preciso escolher quais os operadores de mutação deseja-se utilizar para gerar os mutantes. Na aba **Mutants** desta mesma janela existem vários operadores que podem ser selecionados, como pode ser visto na **Figura 5**.

Como podem ser observados, os operadores são divididos em dois grupos. Em **Traditional Operators** estão os operadores mais comuns que agem a nível de método e, em **Class-Level Operators**, encontram-se os operadores que agem a nível de toda a classe.

Foram escolhidos quatro operadores para demonstrar as funcionalidades do **MuClipse**: **AORB** (**A**rithmetic **O**perator **R**eplacement) que faz alterações em operações aritméticas básicas como soma, subtração divisão e multiplicação; **ROR** (**R**elational **O**perator **R**eplacement) que altera operadores relacionais como >, >=, <, <=, == e !=; **COI** (**C**onditional **O**perator **I**nsertion) que insere o operador de negação ! (not) em cada expressão de condição; e o operador a nível de classe PRV (**R**eference assignment with other comparable **v**ariable) que, no exemplo, altera as variáveis dos métodos sets.

Uma descrição completa com todos os operadores tradicionais pode ser obtida em http://cs.gmu.edu/~offutt/mujava/mutopsMethod.pdf assim como em http://cs.gmu.edu/~offutt/mujava/mutopsClass.pdf pode ser obtida uma descrição de todos os operadores a nível de classe.

Depois de escolher os operadores desejados, o botão **Run** deve ser pressionado iniciando a geração dos mutantes. É mostrado no **Console** cada etapa da geração dos mutantes e, ao final, a mensagem **Mutants have been created!** é exibida.

É possível observar que a pasta **result** não está mais vazia. Agora ela contém todos os mutantes gerados. Como pode ser observado na **Figura 6**, os mutantes são divididos em duas pastas. A pasta **class_mutants** onde se encontram os mutantes a nível de classe e a pasta **traditional_mutants** onde encontra-se os mutantes gerados com os operadores tradicionais. Além dessas duas pastas, há também uma pasta chamada **original** onde é armazenado o fonte e o arquivo *bytecode* original da classe.

As **Figuras 7**, **8**, **9** e **10** demonstram, respectivamente, exemplos de mutantes gerados com os operadores **AORB**, **ROR**, **COI**, **PRV**.

Executar Mutantes

Depois de ter gerado os mutantes é preciso executá-los com os mesmos casos de testes que foram executados sobre o programa original. Para iniciar a execução dos testes, primeiramente é preciso informar ao **MuClipse** algumas configurações. Para isto deve-se clicar com o botão direito sobre o projeto e escolher a opção **Run As / Run Configurations**. Uma janela semelhante à **Figura 11** é exibida.

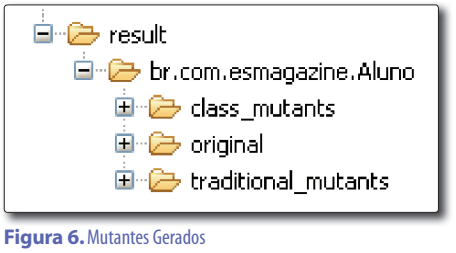


Figura 6. Mutantes Gerados

Em **Name** deve ser digitado o nome da configuração como, por exemplo, **ExecutarMutantes**. O Campo **Project** deve ser preenchido com o nome do projeto e **Mutants Output** deve ser preenchido obrigatoriamente com o diretório **result**.

Em **Class Folder** deve ser digitado o diretório **bin** e em **Source Folder** deve ser digitado **src**, pois essas pastas contêm respectivamente o arquivo *bytecode* e o arquivo de código fonte da classe **Aluno**.

Em **Testset Folder** deve ser digitado **testset** que é o diretório onde se encontra a classe que contém os testes. Já em **Target Class** deve ser digitado o pacote e o nome da classe que se deseja executar os testes que, no exemplo, é **br.com.esmagazine.Aluno**.

O campo **Test** pode ser considerado um campo problemático desta configuração talvez por um *bug* do *plug-in*. Não é possível selecionar o valor deste campo através do botão **Browse**, como pode ser feito com os outros campos. Se isto for feito, o **MuClipse** não preenche este campo corretamente. Portanto, em **Test** deve ser preenchido o pacote e o nome da classe de teste manualmente que no exemplo demonstrado ficaria **br.com.esmagazine.TestarAprovacao**.

Essas não são as únicas configurações necessárias para se executar os testes, a aba **Testing Options** também deve ser configurada.

Nesta aba pode-se configurar o **Timeout** de execução dos mutantes para, caso o mutante entre em laço infinito, não fique agarrado para sempre nos testes. É possível configurar também quais tipos de mutantes deseja-se executar nos testes como pode ser visto na **Figura 12**. No exemplo demonstrado foi mantido o **Timeout** padrão e os dois tipos de mutantes serão executados.

Depois disto, o botão **Run** deve ser pressionado para que os mutantes comecem a ser executados. Ao final na execução, é apresentado no **Console**, além de outras coisas, um resumo sobre a execução dos mutantes, como pode ser visto na **Listagem 3**.

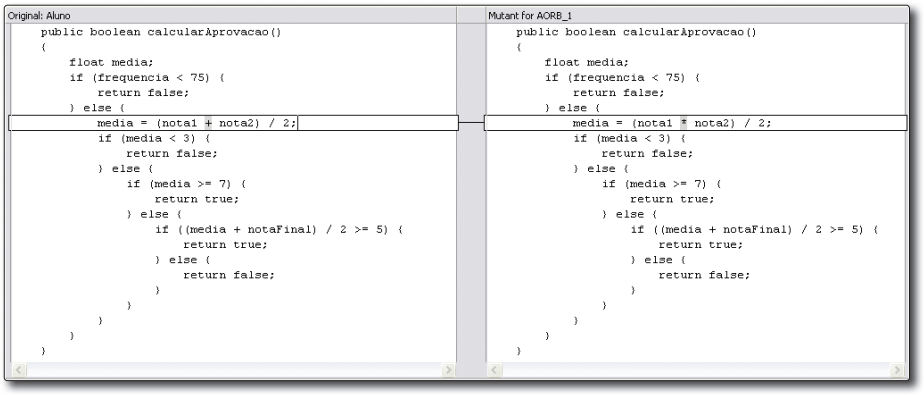


Figura 7. Operador AORB

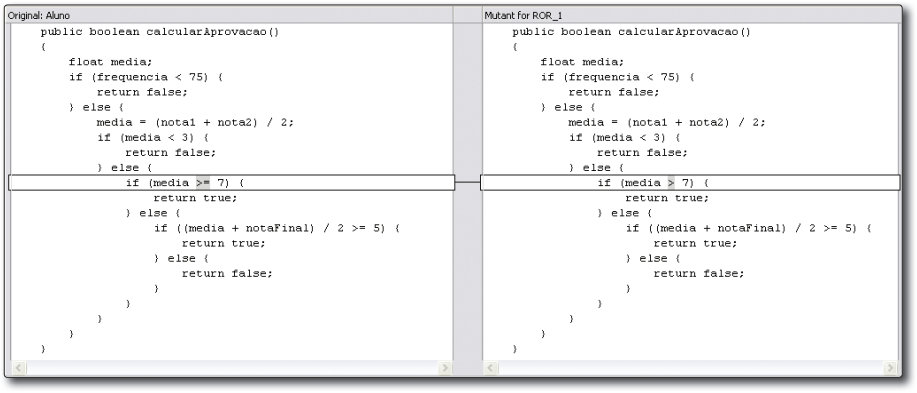


Figura 8. Operador ROR

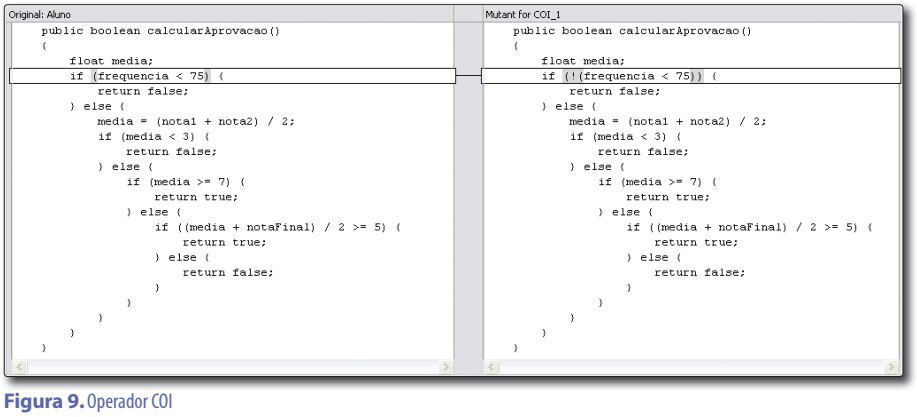


Figura 9. Operador COI

Listagem 3. Resumo da execução dos mutantes

```
Live mutants: 2
Killed mutants: 34
Mutation Score: 94.0
```

Live mutants é o número de mutantes que sobreviveram aos testes e **Killed mutants** é o número de mutantes que foram mortos pelos testes. **Mutation Score** é um número que representa o quanto os testes foram eficientes e, quanto mais próximo de 100 este número for, mais bem sucedido foram os testes.

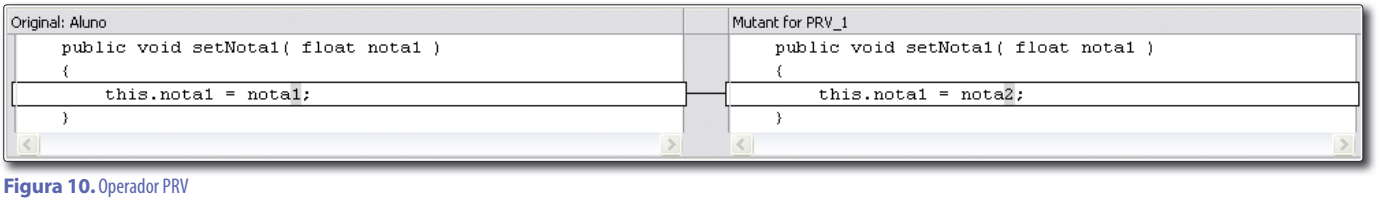


Figura 10. Operador PRV

Análise dos Mutantes Sobreviventes

Depois de criar os mutantes e executá-los com os casos de testes é preciso analisar os mutantes que ainda sobreviveram para decidir se os casos de testes devem ser alterados por não serem suficientemente adequados ou se algum mutante é equivalente ao programa original.

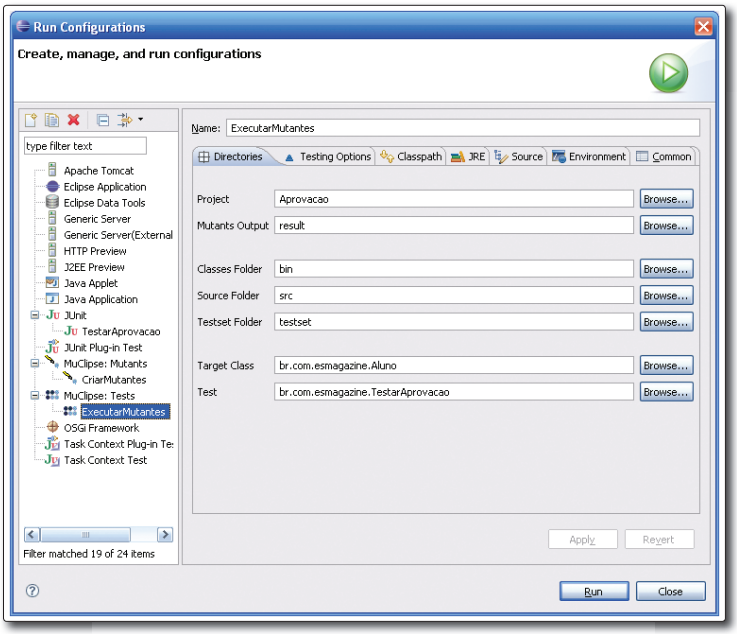
O **MuClipse** disponibiliza uma funcionalidade onde é possível saber exatamente qual mutante continua vivo e qual mutante foi morto pelos casos de testes. Para acessar esta funcionalidade deve-se clicar com o botão direito sobre o diretório **result** e acessar a opção **MuClipse / View Mutants and Results**. Com isto, uma janela é exibida onde pode ser visto cada mutante criado e, à sua direita, a palavra **killed** ou **alive** caso ele esteja morto ou vivo, respectivamente.

A **Figura 13** mostra uma parte desta janela onde há um mutante ainda vivo selecionado.

Não é necessário se preocupar com os mutantes mortos, porém os mutantes que permanecem vivos devem ser analisados. Para isto, é preciso primeiramente saber onde o programa original foi alterado para gerar esse mutante com um duplo clique sobre o mutante desejado. Com isto, uma janela semelhante à **Figura 14** é exibida mostrando uma comparação entre a classe original e o mutante.

Com esta funcionalidade da ferramenta é possível verificar qual alteração foi realizada para gerar o mutante e com isto avaliar se é necessário alterar o caso de teste ou se trata de um mutante equivalente.

No exemplo demonstrado, pode-se perceber que foi trocado o sinal de "**>=**" pelo sinal de "**==**" e que, mesmo com esta alteração, os casos de testes não reprovaram este mutante. Como mencionado anteriormente, para um aluno ser aprovado pela nota final, a soma entre essa nota e a média das duas primeiras notas deve ser maior ou igual a cinco. Portanto, este mutante não deve ser considerado um mutante equivalente e os casos de testes devem ser alterados para conseguir matar esse mutante.



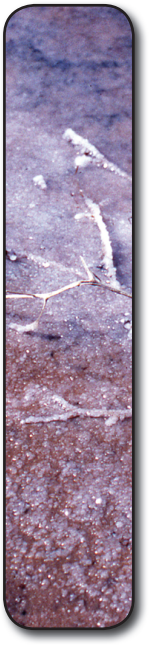


Figura 11. Execução de Mutantes

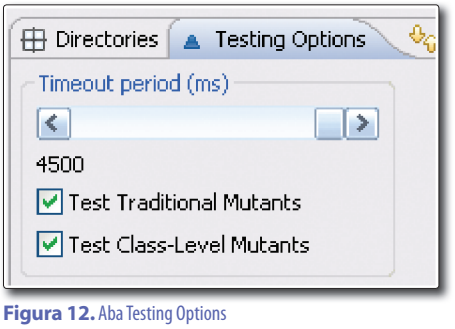


Figura 12. Aba Testing Options

Este mutante não foi morto porque o caso de teste **testAprovadoFinal** faz com que o resultado da expressão **(media + notaFinal) / 2** seja igual a 5. Porém, este não é o único resultado válido. Para resolver este problema um outro caso de teste deve ser criado. Este caso de teste vai ser chamado de **testAprovadoFinalNotaMaxima**, conterá o código demonstrado na **Listagem 4** e deve ser adicionado depois do último caso de teste.

Depois de adicionar este novo caso de teste, é preciso rodar novamente os mutantes com os casos de testes e este mutante será morto pelo último caso de teste criado.

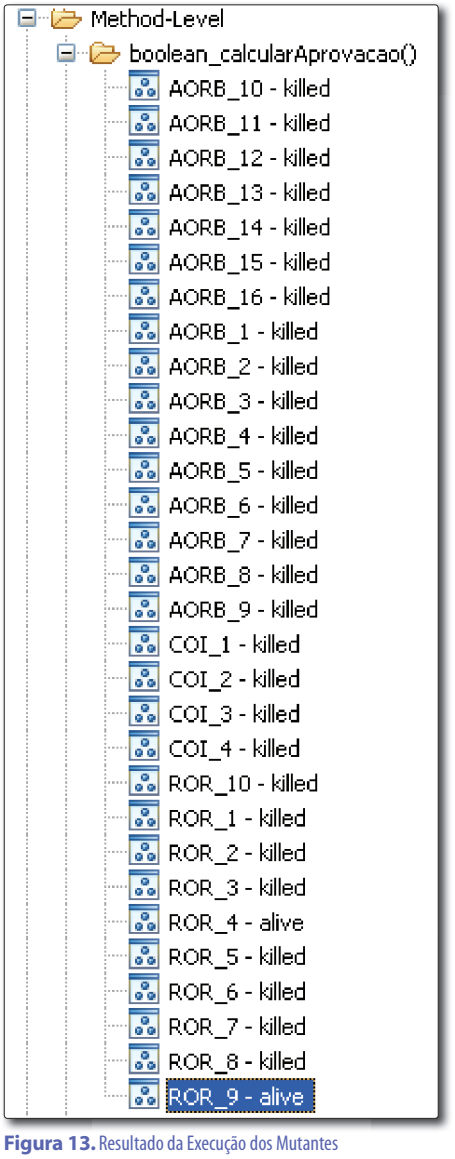


Figura 13. Resultado da Execução dos Mutantes

Listagem 4. Caso de teste testAprovadoFinalNotaMaxima

```
public void testAprovadoFinalNotaMaxima(){
   aluno.setNota1(3);
   aluno.setNota2(3);
   aluno.setNotaFinal(10);
   aluno.setFrequencia(75);
   assertTrue(aluno.calcularAprovacao());
}
```

É possível perceber que o fato deste mutante não ter sido morto mostra uma ineficiência do conjunto de casos de testes utilizados que aprovariam o método **calcularAprovacao()**, mesmo ele não estando correto. É importante notar que a inclusão do caso de teste **testAprovadoFinalNotaMaxima** pode ser considerada como uma melhoria dos casos de testes atestando assim a capacidade dos testes de mutação em tornar melhores os casos de testes.

Porém, ainda há um mutante que não foi morto e que é preciso analisá-lo. Deve-se clicar novamente sobre o diretório **result** e escolher a opção **MuClipse / View Mutants and Results** e encontrar o mutante chamado **ROR_4** que ainda está vivo. A **Figura 15** mostra que nesse mutante foi alterado o operador lógico ">=" para "==".

Como mencionado anteriormente, para um aluno ser aprovado apenas com as duas primeiras notas, a média entre essas notas deve ser maior ou igual a sete. Caso isso não ocorra e a média entre essas notas seja maior ou igual a três, então a nota final é avaliada.

Como o método não retorna qual tipo de aprovação ou reprovação o aluno teve (aprovado/reprovado por infreqüência, nota ou nota final) a alteração realizada na condição destacada na **Figura 15** não está influenciando no retorno do método. Portanto, este mutante pode ser considerado equivalente ao programa original e, com isto, pode ser ignorado.

Pode-se concluir, portanto, que os casos de testes elaborados para a classe **Aluno** são abrangentes o suficiente para demonstrar defeitos nesta classe.

Conclusão

Este artigo procurou expor os principais conceitos que fundamentam a técnica de testes de mutação e posteriormente aplicar essas técnicas com o auxílio do *plug-in* **MuClipse**.

É possível perceber que os testes de mutação ajudam a melhorar a qualidade dos casos de testes principalmente pelo último passo da técnica onde os mutantes sobreviventes são analisados podendo levar a um aperfeiçoamento dos casos de teste.

Além disso, esta técnica pode se tornar inviável sem o auxílio de uma ferramenta, principalmente ao gerar os mutantes e executá-los com os casos de testes. Isso porque esses passos são extremamente repetitivos e fazê-los manualmente pode inserir erros no processo ●

Links

Site oficial do MuClipse
http://muclipse.sourceforge.net/

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

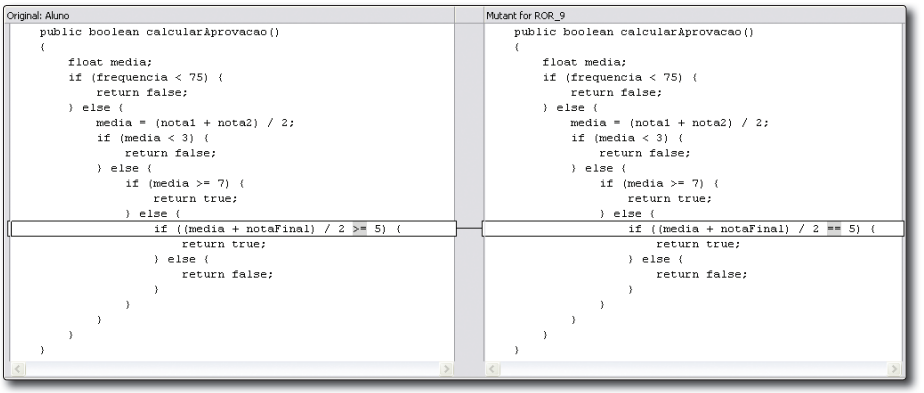


Figura 14. Comparativo entre Classe Original e Mutante

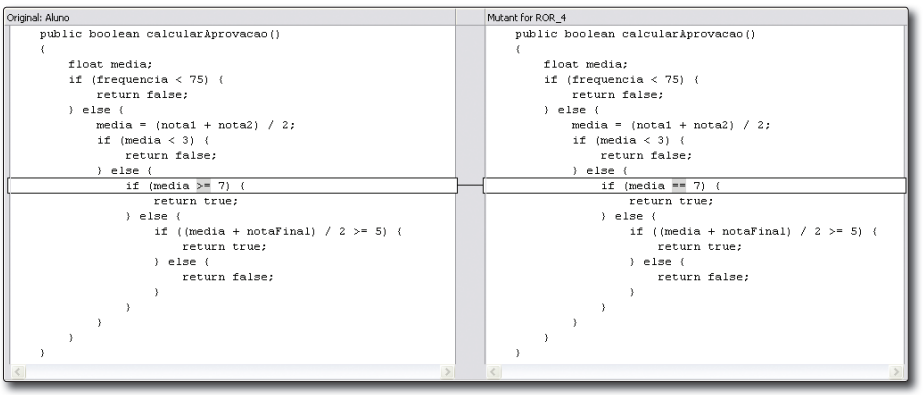


Figura 15. Comparativo entre classe original e mutante sobrevivente

Testes Automatizados de Software em Web Services

Introdução à Ferramenta SoapUI

Marcelo Santos Daibert
marcelo@daibert.net
É professor do Curso de Bacharelado em Ciência da Computação da FAGOC - Faculdade Governador Ozanam Coelho na graduação e pós-graduação (especialização), Mestrando e Especialista em Ciência da Computação pela Universidade Federal de Viçosa e Bacharel em Sistemas de Informação pela Faculdade Metodista Granbery. Gerente técnico da Optical Soluções em Informática Ltda.

Jenifer Vieira Toledo
jenifer@jenifer.eti.br
É graduada em Ciência da Computação pela Faculdade Governador Ozanam Coelho (FAGOC) e Técnica de Hosting da empresa Optical Soluções em Informática LTDA.

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

De que se trata o artigo:
Utilização de testes automatizados de software em WebServices com a ferramenta SoapUI. O artigo apresenta a utilização da ferramenta para testes funcionais e de desempenho.

Para que serve:
Fornecer conhecimentos para se testar, na prática, e de forma automatizada, WebServices com a ferramenta SoapUI. Estabelecer uma visão crítica sobre a necessidade de se desenvolver software de qualidade e a relação da abordagem de desenvolvimento baseado em teste com a busca por qualidade.

Em que situação o tema é útil:
Além de ser considerada uma boa prática, o uso de testes de software reduz o número de defeitos, aumentando assim a qualidade do produto de software e satisfação dos usuários.

Notoriamente, e principalmente nos últimos anos, é possível observar uma profunda evolução nas nossas vidas e nas relações com o mundo. Certamente, nos últimos vinte anos houve uma evolução, neste sentido, maior que de décadas anteriores. As comunicações foram uma das áreas mais impactadas, mudando a forma com a qual interagimos com o mundo. Como apoio a esta evolução foram desenvolvidas várias tecnologias, principalmente na área da engenharia de software. Novas técnicas de desenvolvimento de software, novos métodos, formas de documentação, linguagens, entre outros, foram desenvolvidas. A necessidade de aspectos de usabilidade nos sistemas começaram a fazer diferença e a palavra qualidade cada vez mais foi se destacando na pauta dos desenvolvedores. A qualidade no software passou a ser uma busca constante.

O software, hoje em dia, está presente nas mais variadas áreas e tarefas. Desde equipamentos presentes na cozinha, como um microondas ou geladeira, até equipamentos mais sofisticados, como um avião, são gerenciados e até mesmo comandados por um aplicativo embarcado. Sem contar os softwares mais conhecidos, como sistemas operacionais e aplicativos comerciais, entre

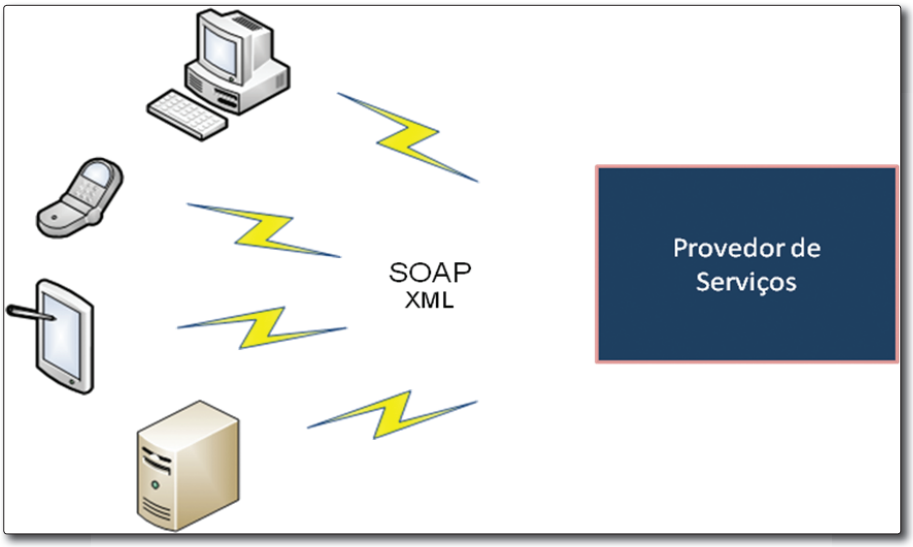


Figura 1. Tecnologia SOA

outros. Por isso, a busca pela qualidade passou a ser uma busca constante para os desenvolvedores, já que uma falha nestes softwares pode resultar em grandes perdas financeiras, de tempo e, dependendo da aplicação, até em vidas humanas.

Neste sentido, o tema Teste de Software vem sendo muito discutido. Atualmente existem várias abordagens de teste e cada vez mais surgem outras novas. A principal abordagem utilizada atualmente em muitas empresas de desenvolvimento, principalmente as de médio para pequeno porte, é a utilização dos testes manuais. Nela é atribuída a função de testar as aplicações desenvolvidas a um membro da equipe de forma manual. No entanto, esta abordagem é muito ineficaz e demorada. As melhores técnicas de teste são baseadas em algum processo automatizado, onde é possível executar uma maior quantidade de testes, buscando assim testar ao máximo os requisitos de um software. Nesta estratégia, existem algumas técnicas, entre elas testes unitários, testes funcionais, testes de regressão, teste de desempenho, testes em banco de dados, testes em WebService, entre outros.

O objetivo deste artigo é apresentar de forma prática a abordagem de desenvolvimento de software baseada em testes, conceituar a técnica de programação por intenção e exibir a utilização prática da estratégia de teste funcional e de desempenho em WebServices, usando a ferramenta SoapUI, aplicados ao desenvolvimento de WebServices na arquitetura SOA (Service Oriented Architecture – Arquitetura Orientada a Serviços).

O teste funcional, apresentado neste artigo no contexto de WebServices, é um tipo de teste baseado em técnicas de caixa-preta, isto é, verifica-se o comportamento da aplicação através de sua estrutura interna.

Arquitetura Orientada a Serviços

SOA vem fazendo muito sucesso com altos índices de utilização. A proposta é uma arquitetura baseada em serviços e na WEB. Ou seja, uma aplicação poderá consumir vários serviços de um servidor de serviços baseado em uma estrutura WEB e através do protocolo da camada de aplicação HTTP. A **Figura 1** busca ilustrar esta tecnologia. Há um provedor de serviços, onde um WebService é implementado. Estes serviços podem então ser consumidos por uma grande gama de dispositivos, linguagens e outras tecnologias, garantindo assim a interoperabilidade de aplicações e linguagens. Uma aplicação pode inclusive ser desenvolvida toda nesta arquitetura. Sempre que uma função for executada por um usuário, a aplicação consome o seu WebService para cumprimento da funcionalidade. As bases para a construção de um WebService são os padrões XML e o protocolo SOAP (Simple Object Access Protocol, baseado em XML). O transporte dos dados é realizado normalmente via protocolo HTTP, ou HTTPS para conexões seguras. Os dados são transferidos no formato XML e encapsulados pelo protocolo SOAP.

Cada WebService apresenta um documento descritor dos serviços disponibilizados, contendo informações de como acessá-los, como deve ser a entrada de dados e qual será o formato da saída. Este documento descritor é conhecido como WSDL (Web Service Description Language). É uma linguagem baseada em XML utilizada para descrever um WebService.

Como apresentado no início deste arti-

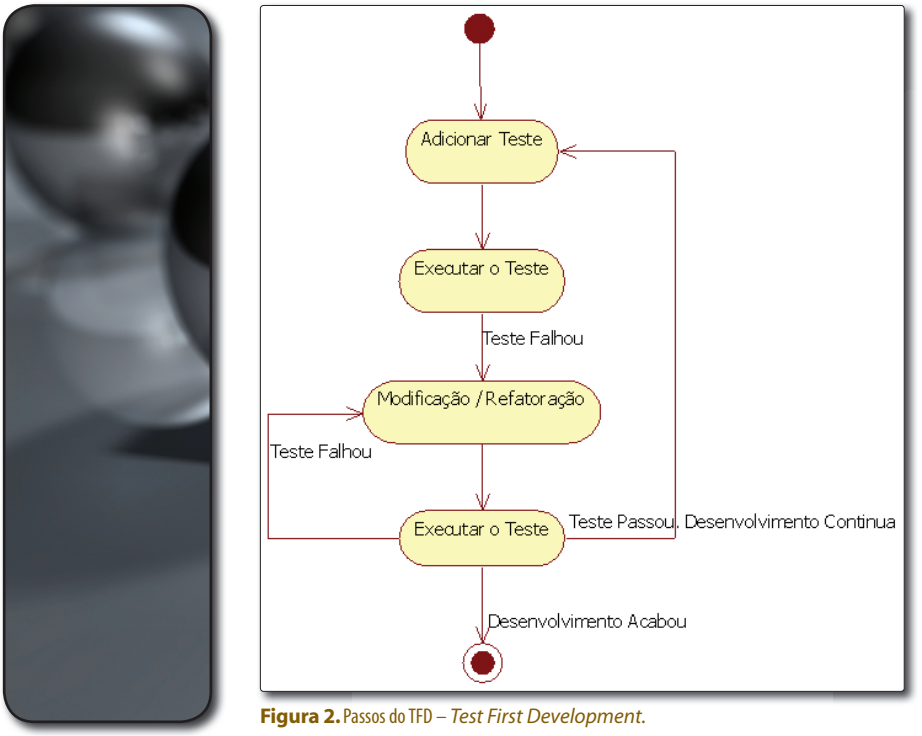


Figura 2. Passos do TFD – *Test First Development.*

Listagem 1. Código do WebService Conversao

```
1. <?php
2.   require_once('./lib/nusoap.php');
3.   $server = new soap_server;
4.   $server->configureWSDL('conversao','urn:conversao');
5.   $server->wsdl->schemaTargetNamespace = 'urn:conversao';
6.   $server->wsdl->addComplexType(
      'TipoEntrada',
      'complexType',
      'struct',
      'all',
      '',
      array(
        'vEntrada' => array(
                       'name' => 'vEntrada',
                       'type' => 'xsd:float'
                     )
      )
  );
 7.  $server->wsdl->addComplexType(
      'TipoSaida',
      'complexType',
      'struct',
      'all',
      '',
      array(
        'vSaida' => array(
                       'name' => 'vSaida',
                       'type' => 'xsd:float'
                     )
      )
  );
 8.  $server->register(
      'Km2Milha',
      array(
        'TipoEntrada' => 'tns:TipoEntrada'
      ),
      array(
        'TipoSaida' => 'tns:TipoSaida'
      ),
      'urn:conversao',
      'urn:conversao#Km2Milha',
      'rpc',
      'encoded',
      'Converte Quilometros em Milhas'
  );
9.   function Km2Milha($vKm){
10.       if (is_string($vKm['vEntrada'])){
11.           return new soap_fault('Client','','O Parametro não deve ser
              uma String. Ele deve ser um Float');
12.       }else if($vKm['vEntrada'] < 0){
13.           return new soap_fault('Client','','O Parametro não deve ser
              Negativo. Ele deve ser Positivo');
14.       }else{
15.           $retorno = $vKm['vEntrada'] * 3.10;
16.           return array(
                   'vSaida' => $retorno
                );
17.       }
18.   }
19.   $server->service($HTTP_RAW_POST_DATA);
20.?>
```

go, várias evoluções são observadas nos últimos anos. Hoje a arquitetura orientada a serviços já pode ser considerada como uma importante evolução destas, mudando os paradigmas de desenvolvimento de software. Desta forma, adequar o desenvolvimento de software para o SOA e buscar qualidade neste processo deve ser algo a ser levado em consideração. Assim, as abordagens de teste de software não deixaram de contemplar o desenvolvimento de WebServices.

Desenvolvimento Baseado em Testes

O desenvolvimento baseado em testes (TDD – Test Driven Development) (ler **Nota 1**) é uma abordagem que faz uso das técnicas de teste de software para minimizar a quantidade de falhas na aplicação desenvolvida. Para isso, todo o processo de desenvolvimento de software é baseado em testes, a começar pela técnica de desenvolvimento de testes antes da codificação (TFD – Test First Development).

Nesta técnica, o desenvolvedor deve planejar e construir o teste para um trecho ou módulo do sistema que está sendo construído (em SOA, no desenvolvimento de WebServices), antes mesmo de sua codificação. Com isso, a qualidade dos casos de teste é aperfeiçoada, já que o mesmo é feito de forma incremental, juntamente com a codificação, como pode ser visualizado em um diagrama de atividades na **Figura 2**. Esta técnica faz uso de uma estratégia chamada programação por intenção.

Na **Figura 2**, é possível identificar claramente os passos do TFD. A primeira atividade observada é a Adicionar Teste. Nela o teste é escrito e então executado no segundo passo. Certamente o teste irá falhar, já que não existe codificação da função, o que justifica a terceira atividade, chamada Modificação / Refatoração. Nesta atividade o desenvolvedor irá codificar a função a fim de fazer o teste ser executado com sucesso. Após, na ultima atividade, o teste é executado novamente. Se ele passar, o desenvolvimento continua, ou finaliza de acordo com a necessidade, e caso o teste falhe, há a necessidade de uma nova iteração na atividade Modificação / Refatoração até que o teste seja executado com sucesso.

A programação por intenção é uma abordagem de programação que induz a codificação usando classes, métodos ou módulos do sistema que ainda não existem e serão criados futuramente para atender às necessidades de compilação da aplicação. No contexto de desenvolvimento baseado a testes, a programação por intenção auxilia a criação dos casos de testes, quando utilizada a técnica de desenvolvimento de testes antes da codificação, já que o teste produzido verifica um

Nota 1. Desenvolvimento baseado em testes

A abordagem de desenvolvimento baseado em testes é um dos principais conceitos dos métodos de desenvolvimento ágil. A abordagem de testes é inclusive um dos temas do documento chamado manifesto ágil, onde vários autores propõem as características, conceitos e princípios da metodologia de desenvolvimento, onde um exemplo de método ágil é o XP (*eXtreme Programming*). Além disso, modelos que definem padrões de maturidade para o desenvolvimento de software também incluem a realização de testes, como o caso do CMMI (*Capability Maturity Model Integration*) e do MPS. BR (*Melhoria de Processos do Software Brasileiro*) onde, para ambos, a utilização de testes é um item essencial, mesmo em seus níveis iniciais.

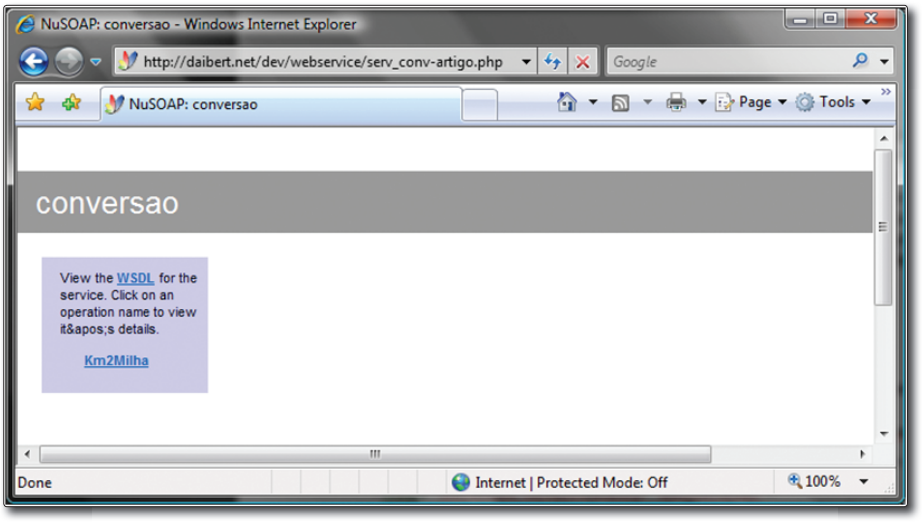


Figura 3. Execução do webservice.

método ainda inexistente. E este método, por sua vez, pode fazer uso de recursos da aplicação também inexistentes.

A idéia central da programação por intenção é comunicação com as intenções do desenvolvedor ao codificar a aplicação. Mesmo gerando um código fonte não compilável, é possível definir e limitar o escopo de ação de determinado método, além de traçar os passos de desenvolvimento para atender às necessidades de compilação, inclusive utilizando técnicas de refatoração de código fonte. Para isso, muitas vezes é necessário utilizar os chamados objetos Mock, com o objetivo de simplificar a utilização da programação por intenção e a criação dos casos de teste.

Objetos Mock são objetos falsos, ou de fachada, com o objetivo de substituir recursos não disponíveis ou inadequados, possibilitando a criação dos casos de teste e execução de testes unitários no sistema. No desenvolvimento de WebServices, esta abordagem pode ser muito bem utilizada e empregada. Inclusive a ferramenta alvo deste artigo, a SoapUI, suporta a utilização de Mocks para os WebServices. Este recurso é chamado pela SoapUI de WebService Simulation - MockServices from WSDL. Ou seja, simula funcionalidades do WebService que ainda não estão implementadas para propiciar a implementação de alguma funcionalidade baseada na idéia de programação por intenção.

Independente da utilização desta abordagem de desenvolvimento baseado em testes, a utilização das técnicas de teste se configuram como uma importante ferramenta para buscar a qualidade das aplicações e a minimização dos defeitos no software.

Criando um WebService

Para exemplificar a utilização da ferramenta SoapUI, este artigo apresenta a construção de um WebService para estabelecer um estudo de caso. Este WebService apresenta um serviço para transformação de quilômetros para milhas. Várias são as linguagem que poderiam implementar o WebService. A escolhida para este artigo foi a linguagem PHP utilizando um framework para desenvolvimento de WebServices chamado nuSOAP.

O nuSOAP disponibiliza toda uma infra-estrutura para a construção de WebServices usando a linguagem PHP. É gratuito e pode ser baixado no site oficial da ferramenta (ver seção Links). Esta seção do artigo apresenta o processo de criação de um WebService utilizando o nuSOAP, onde somente é desenvolvido o WebService. Não é objetivo deste artigo o desen-

Listagem 2. WSDL gerado pelo nuSOAP.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<definitions xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSche-
ma" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="urn:conversao" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:wsdl="http://schemas.xmlsoap.org/
wsdl/" xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="urn:conversao">
<types><xsd:schema targetNamespace="urn:conversao">
 <xsd:import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
 <xsd:import namespace="http://schemas.xmlsoap.org/wsdl/"/>
 <xsd:complexType name="TipoEntrada">
  <xsd:all>
   <xsd:element name="vEntrada" type="xsd:float"/>
  </xsd:all>
 </xsd:complexType>
 <xsd:complexType name="TipoSaida">
  <xsd:all>
   <xsd:element name="vSaida" type="xsd:float"/>
  </xsd:all>
 </xsd:complexType>
</xsd:schema>
</types>
<message name="Km2MilhaRequest"><part name="TipoEntrada" type="tns:TipoEntrada"/></message>
<message name="Km2MilhaResponse"><part name="TipoSaida" type="tns:TipoSaida"/></message>
<portType name="conversaoPortType"><operation name="Km2Milha"><documentation>Converte Quilometros em Milhas</
documentation><input message="tns:Km2MilhaRequest"/><output message="tns:Km2MilhaResponse"/></operation></portType>
<binding name="conversaoBinding" type="tns:conversaoPortType"><soap:binding style="rpc" transport="http://sche-
mas.xmlsoap.org/soap/http"/><operation name="Km2Milha"><soap:operation soapAction="urn:conversao#Km2Milha"
style="rpc"/><input><soap:body use="encoded" namespace="urn:conversao" encodingStyle="http://schemas.xmlsoap.org/soap/
encoding/"/></input><output><soap:body use="encoded" namespace="urn:conversao" encodingStyle="http://schemas.xmlsoap.
org/soap/encoding/"/></output></operation></binding>
<service name="conversao"><port name="conversaoPort" binding="tns:conversaoBinding"><soap:address location="http://
daibert.net/dev/webservice/serv_conv-artigo.php"/></port></service>
</definitions>
```

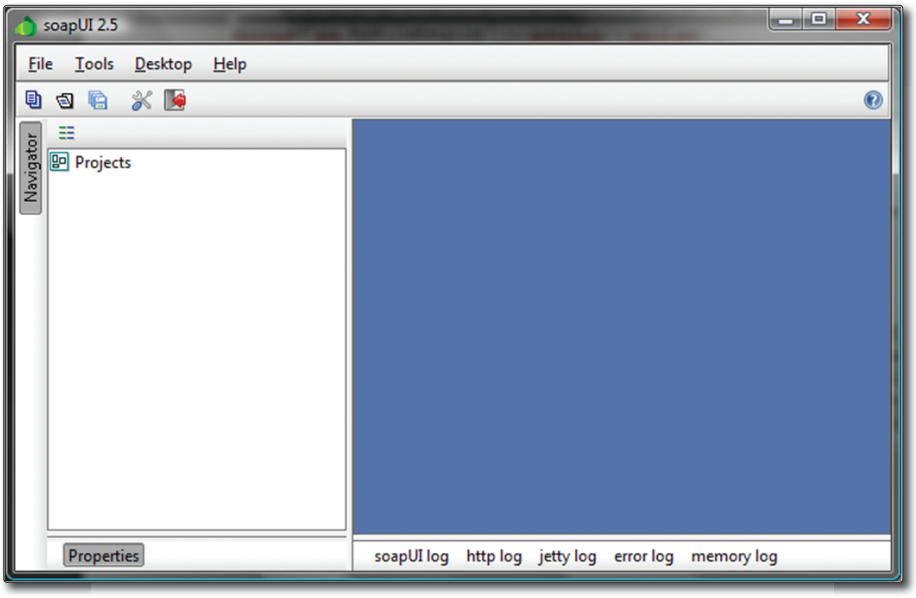


Figura 4. Interface principal da ferramenta SoapUI.

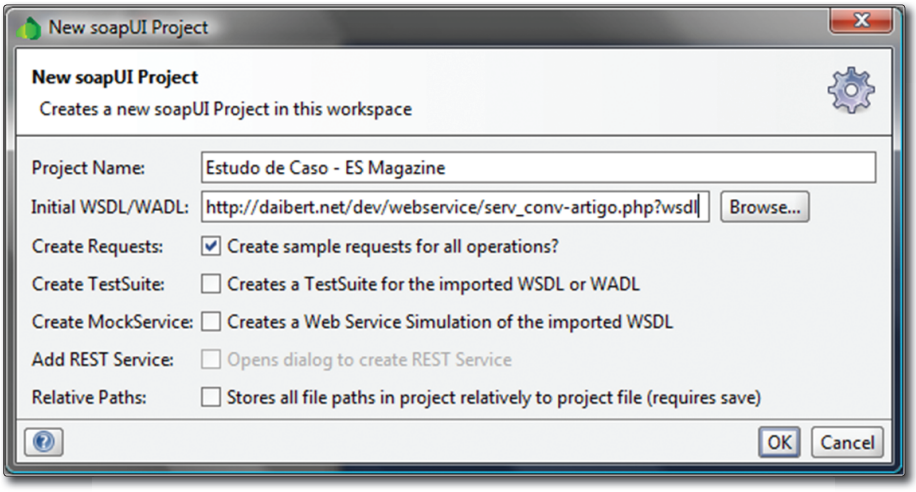


Figura 5. Criação de um novo projeto.

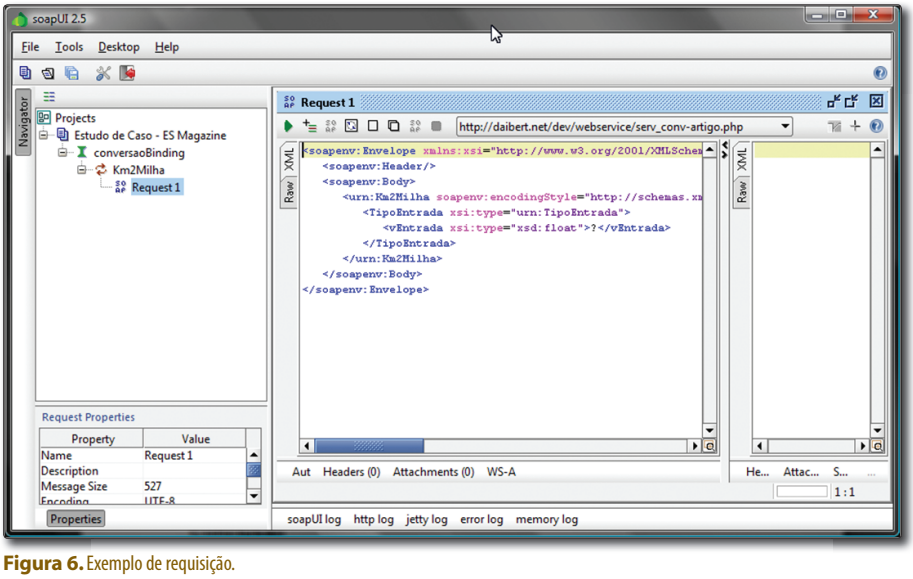


Figura 6. Exemplo de requisição.

volvimento de nenhum cliente associado ao WebService para consumi-lo.

Os arquivos do nuSOAP devem ser adicionados ao projeto do WebService a ser desenvolvido. Para o estudo de caso, foi adicionado no diretório lib, representando um diretório de bibliotecas. A **Listagem 1** apresenta o código fonte do WebService "Conversao".

Na linha 2, é invocado o método require_once com o objetivo de incluir o arquivo nusoap.php. Assim é possível utilizar os métodos e classes do framework nuSOAP. Na linha 3 é instanciado um objeto do tipo soap_server, responsável por estabelecer um servidor de WebServices. Na linha 4 é invocado o método configureWSDL, responsável por definir o arquivo WSDL. É definido o nome do WebService no primeiro parâmetro e, no segundo, é definido o namespace. A linha 5 define qual será o namespace alvo do WSDL para então, nas linhas 6 e 7 serem definidos os tipos de entrada e saída, respectivamente. Foram definidos dois tipos complexos: o primeiro chamado TipoEntrada, representado por uma estrutura complexa chamada vEntrada, que possui o nome vEntrada com o tipo float. O segundo, chamado TipoSaida, também representado por uma estrutura complexa, esta chamada de TipoSaida, igual à TipoEntrada. Estas estruturas foram definidas para estabelecer o tipo de entrada do WebService (no caso ele aceitará uma entrada chamada vEntrada com o tipo float e retornará um vSaida, também com um tipo float).

Neste caso, o WebService receberá um valor float em quilômetros e retornará outro float com o resultado da conversão para milhas. Na linha 8, o serviço Km2Milha é registrado no WebService. Ele representará a função do WebService estabelecido por este estudo de caso. Entre as linhas 9 e 18 é definido o método Km2Milha, onde a função realmente é implementada. São feitos dois testes, onde o primeiro identifica se a entrada recebida é uma String. Se for, é retornado um falta SOAP, representada pelo objeto soap_fault do nuSOAP. O segundo identifica se a entrada é negativa. Se for, age da mesma forma que o primeiro. Caso não seja verdadeira nenhuma das duas condições acima, o fluxo entra no else e define a variável retorno com o cálculo de conversão de quilômetros para milhas, definindo assim o tipo complexo vSaida. Por

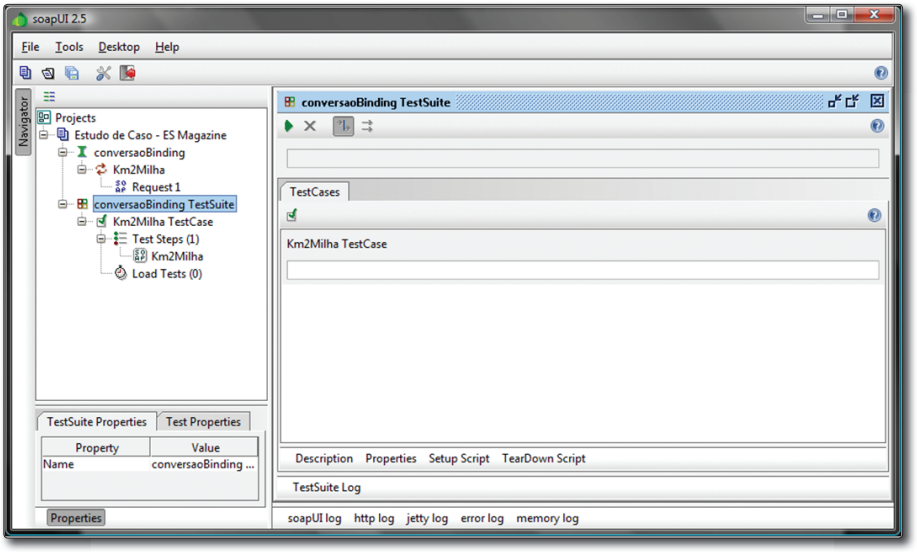


Figura 7. Criação do TestSuite.

fim, na linha 19, o WebService é declarado e apresentado.

A execução deste código em um servidor devidamente configurado com o PHP é apresentada na **Figura 3**. É disponibilizada uma interface com um link para informações do serviço registrado (Km2Milha) e outro link para o WSDL, este apresentado pela **Listagem 2**. O interessante é que o WSDL é gerado pelo nuSOAP de acordo com o configurado no código fonte desenvolvido na **Listagem 1**.

SoapUI

A SoapUI é uma ferramenta para testes em WebServices. Escrita em JAVA, esta ferramenta roda nos principais sistemas operacionais, entre eles o Windows e Linux. É necessário ter configurada a máquina virtual Java (J2RE) para que a SoapUI execute. A ferramenta possui uma versão gratuita e outra paga. A diferença básica entre as duas é o suporte que a versão paga oferece. A SoapUI é uma excelente ferramenta para teste em WebServices e, inclusive, apoiar o desenvolvimento de WebServices. Entre suas principais funcionalidades, destacam-se:

- Importação e geração automática das requisições descritas no WSDL;
- Capacidade de gerenciar um número ilimitado de requisições para cada operação;
- Gerenciamento de múltiplos endpoints para cada WebService;
- Validação das requisições e respostas contra as suas definições no WSDL;
- Testes funcionais, carga e stress;
- Execução de diversos testes em paralelo;
- Editores com syntax highlight e formatação automática;
- Suporta expressões XPATH;
- Suporta criação de testes complexos utilizando scripts Groovy;
- Invocação de funcionalidades ainda não implementadas através de Mocks, objetivando a simulação de WebServices ou serviços;
- Geração de código fonte cliente para diversas linguagens.

A **Figura 4** exibe a interface principal da SoapUI ao ser executada. Acionando o menu File -> New SoapUI Project é exibida a interface de definição de um novo projeto, como apresentado pela **Figura 5**. Foram configurados os parâmetros de nome e WSDL inicial (Initial WSDL). É pelo WSDL que a ferramenta irá criar o projeto para testar e invocar o WebService. É então configurada a URL do WSDL observada na sessão anterior deste artigo. Foi ativada somente a opção de criar exemplos de request ao WebService (Create sample requests for all operations). As demais opções nos check boxes não foram ativadas.

Com isso, a SoapUI cria uma estrutura de projeto com um exemplo de requisição ao WebService. Assim já é possível testar de forma manual o funcionamento do WebService. A **Figura 6** apresenta a interface com a visualização do exemplo de requisição criado (Request 1). Esta opção permite submeter requisições ao WebService. Para isso, basta alterar diretamente no documento XML, onde é exibida a tag vEntrada, que está com o valor "?". Alterando por um valor, é então definido um envelope SOAP. Após, deve-se clicar na seta verde no menu superior da aba do Request 1. Este botão irá submeter o envelope ao WebService, consumindo o serviço configurado. Na janela do lado direto será exibido o resultado.

O próximo passo é a criação de um ambiente de testes na SoapUI, com o objetivo de inserir os testes, para que eles sejam executados de forma automática. É importante analisar que este tipo de teste que está sendo configurado é um tipo de teste baseado nas técnicas de caixa-preta, isso é, verifica-se o sistema e seus proces-

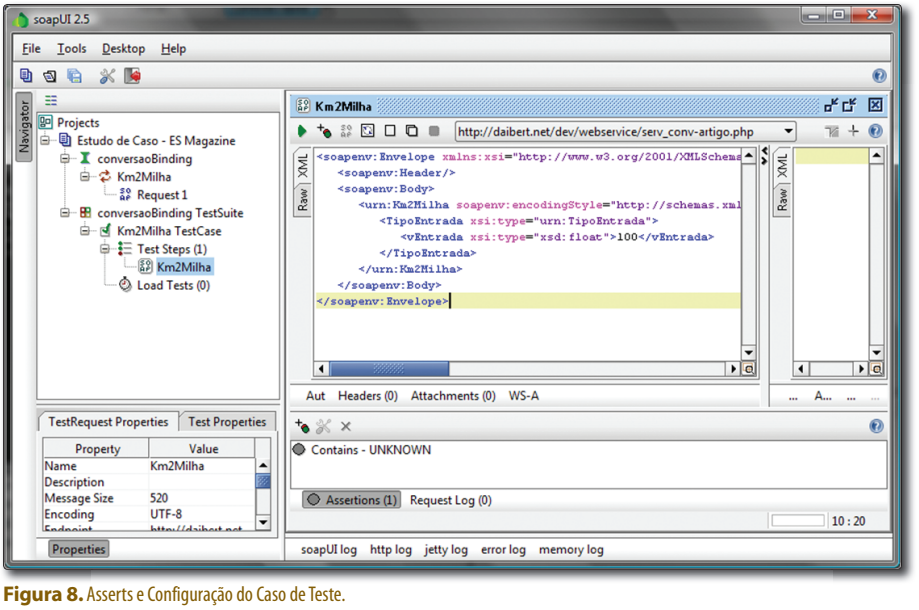


Figura 8. Asserts e Configuração do Caso de Teste.

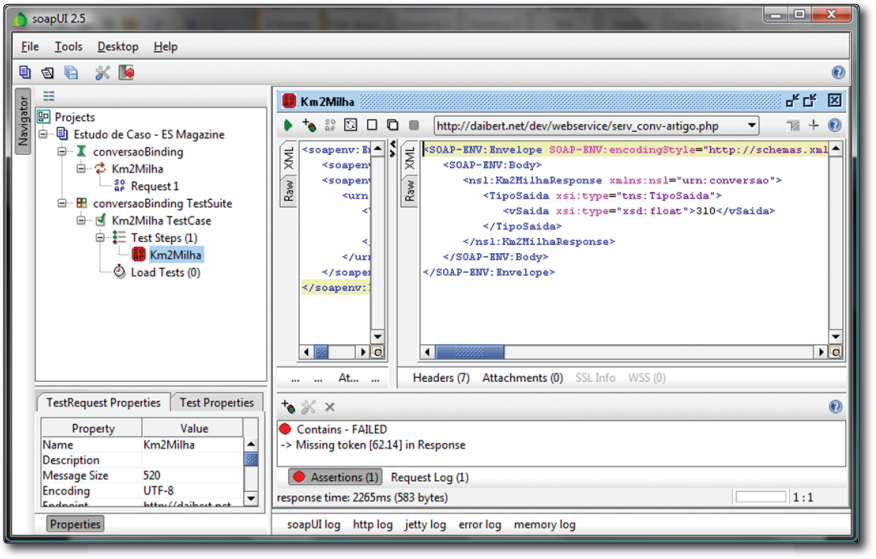


Figura 9. Erro após a execução do Caso de Teste.

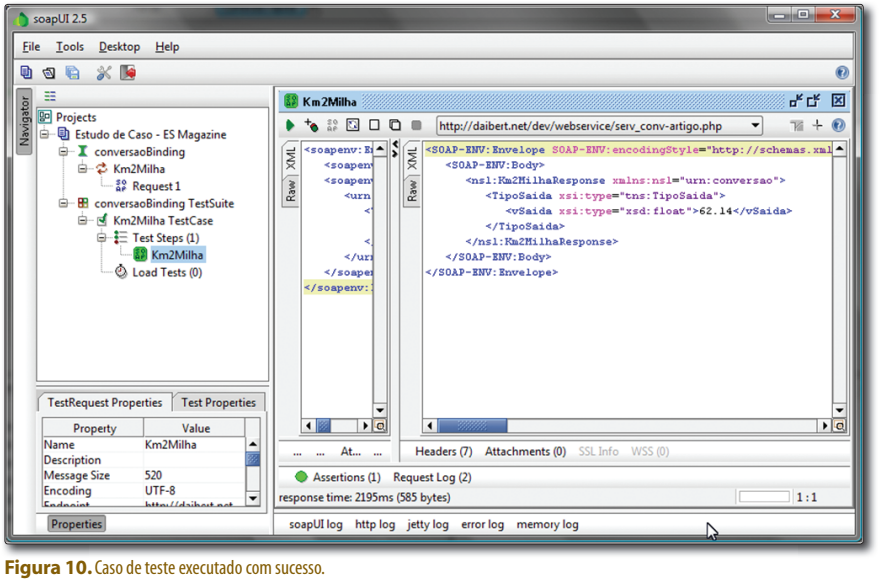


Figura 10. Caso de teste executado com sucesso.

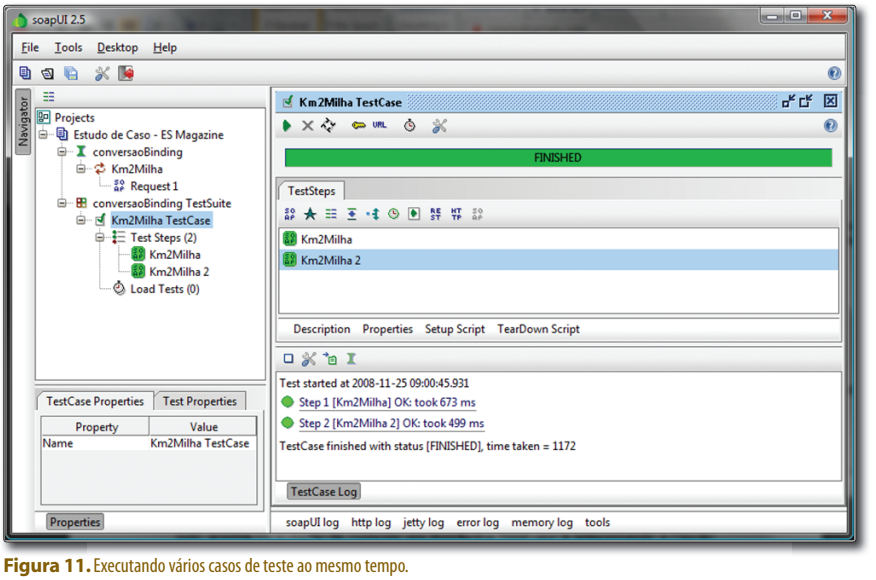


Figura 11. Executando vários casos de teste ao mesmo tempo.

sos internos através de suas interfaces e da análise de suas saídas ou resultados.

Para isso, deve-se criar dentro da SoapUI um ambiente para testes, chamado pela ferramenta de Test Suite. A opção de criação deve ser acessada acionando o menu de contexto sobre a opção conversaoBinding e invocando Generate TestSuite. É então exibida uma interface para criação de um Test Suite, com opção de selecionar as operações a serem testadas (Km2Milha neste estudo de caso). Seleciona-se também a opção um caso de teste por cada operação (One TestCase for Each Operation) onde, desta forma, a SoapUI irá criar automaticamente um caso de teste para cada operação selecionada, facilitando o desenvolvimento inicial dos casos de teste. Após esta operação, é criado o TestSuite com o caso de teste para a operação Km2Milha, como exibido na **Figura 7**.

Após isso, clica-se duas vezes em Km2Milha, no item Test Steps da árvore de opções. É exibida uma interface de execução e configuração do teste. Nesta interface é possível configurar o valor que será submetido ao WebService e qual será o valor esperado. Observe na **Figura 8** que é configurado o valor 100 (quilômetros) para ser submetido. Para definir o valor esperado, 62.14 (milhas) neste caso, é necessário acionar a opção "Adds an Assertion on this item" e então selecionar a opção "Contains", definindo qual texto é esperado, no caso 62.14.

Com estas configurações é possível já executar o teste e verificar o resultado, como exibido pela **Figura 9**. Para executar o teste deve-se invocar a seta verde no menu da janela Km2Milha. Foi possível identificar que o teste não foi bem sucedido. O WebService retornou o resultado 310 milhas para a solicitação. A SoapUI identifica a falha apresentando os resultados com detalhes em vermelho.

Com este resultado é possível identificar alguma falha. O primeiro local a procurar a falha é no código fonte, mais especificamente na parte onde o cálculo da conversão é feito, na linha 15 da **Listagem 1** ($retorno = $vKm['vEntrada'] * 3.10;). É possível observar que existe um erro no código fonte, já que a conversão de quilômetros para milha é expressa pela multiplicação dos quilômetros por 0.6214. A linha do código fonte foi então alterada para "$retorno = $vKm['vEntrada'] * 0.6214;". Após esta alteração, o caso de teste foi novamente

executado e nenhum erro foi observado, como apresentado na **Figura 10**.

É possível criar quantos casos de teste foram necessários. A SoapUI disponibiliza uma opção de executar uma série de casos de teste de uma única vez, otimizando o processo de execução automática dos testes. Esta opção é útil especialmente para execução automática de vários testes, inclusive para executar testes de regressão após alguma alteração no código do WebService. A **Figura 10** exibe esta interface de execução de vários testes ao mesmo tempo. Foi adicionado mais um caso de teste para exemplificar a utilização deste recurso, que é invocado pelo acionando do menu de contexto em Test Steps e selecionando a opção Show Test Case Editor.

A SoapUI possui uma série de funcionalidades. Uma outra destacada aqui é a possibilidade de efetuar testes de desempenho e stress no WebService. O objetivo deste teste é avaliar o desempenho do servidor que hospeda o WebService e o desempenho da codificação. Para criar um teste desses, é necessário criar um novo Load Test. Nas imagens já apresentas é possível ver esta opção na árvore de menu da interface da SoapUI, logo abaixo da opção Test Steps, dentro do TestCase.

Para criar um novo teste de desempenho (performance), aciona-se o menu de contexto da opção Load Tests e seleciona-se New Load Test. Deve-se configurar o nome para o novo teste de desempenho, que para este caso foi definido como Teste de Performance. Com isso é exibida uma interface de execução destes testes. Por padrão, a SoapUI irá ficar consumindo o WebService por 60 segundos após a execução do teste (clicando na seta verde da interface). No entanto, este parâmetro é personalizável. Inclusive é possível definir quantas instâncias irão ficar testando o WebService (quantas threads irão executar - o padrão estabelece 5 threads). A **Figura 12** apresenta a interface de resposta do teste, divulgando resultados como tempo máximo e mínimo de execução, média, quantidade de erros, entre outros. Gráficos também podem ser exibidos, como apresentado pela **Figura 13**.

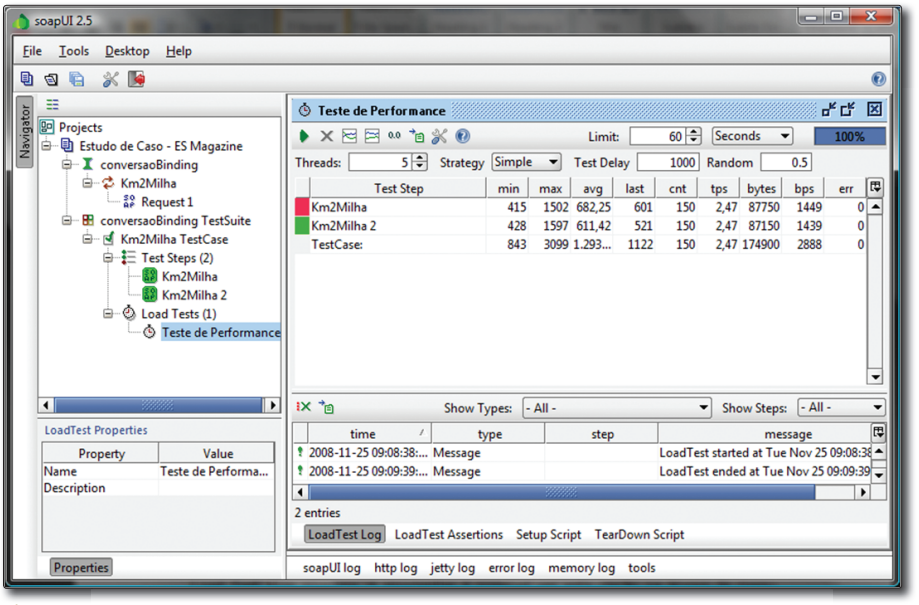


Figura 12. Executando teste de desempenho.

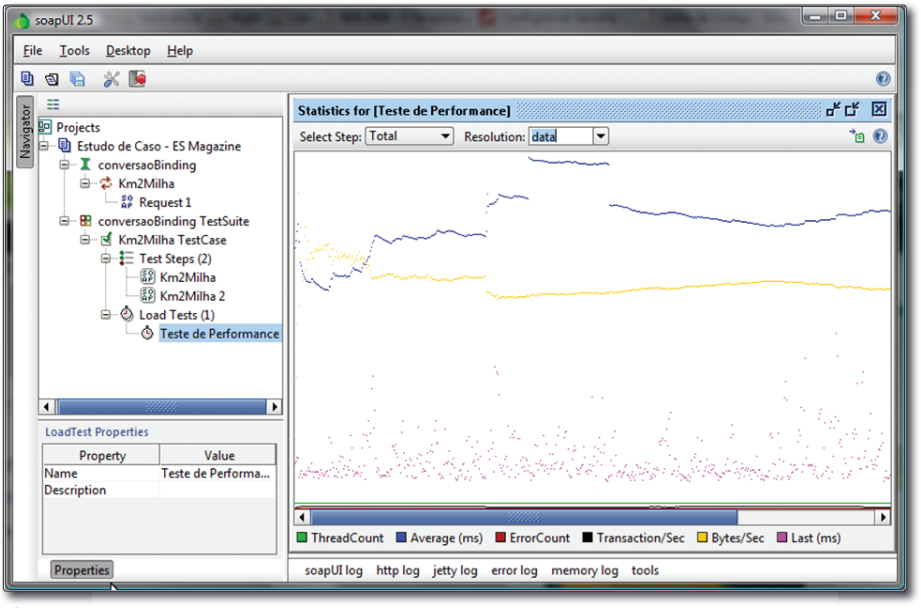


Figura 13. Gráfico do teste de desempenho.

Conclusão

Foi apresentada neste artigo a ferramenta SoapUI para criação e execução de testes funcionais de forma automatizada em WebServices. A utilização dessa abordagem é extremamente importante para minimizar a quantidade de defeitos nos WebServices desenvolvidos. A utilização conjunta de outras técnicas baseadas em testes, como testes unitários, de cobertura, testes de regressão, integração, entre outros, busca melhorar ainda mais este índice de confiança na aplicação desenvolvida. O teste funcional deve ser tratado como um teste de caixa preta, onde se devem testar as interfaces e os resultados de processamento causados pelos dados inseridos na aplicação. A SoapUI auxilia na criação dos testes, contando com uma interface fácil e intuitiva e com recursos de gravação de testes ●

Links

Site Oficial da Ferramenta SoapUI
http://www.soapui.org

Site Oficial do Framework nuSOAP
http://sourceforge.net/projects/nusoap/