engenharia de software magazine

Edição Especial

DevMedia GROUP

Engenharia de Software
Saiba seu significado e para que serve

Qualidade de Software

Entenda os principais conceitos sobre Testes e Inspeção de Software

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

Requisitos
Conheça os principais conceitos envolvidos na Engenharia de Requisitos

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

Especial >> **Processos**

Melhore seus processos através da análise de risco e conformidade

Veja como integrar conceitos de Modelos Tradicionais e Ágeis

Veja como integrar o Processo Unificado ao desenvolvimento Web

engenharia de software magazine

Ano 1 - 1ª Edição 2007 - - Impresso no Brasil

Corpo Editorial

Apoio

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!

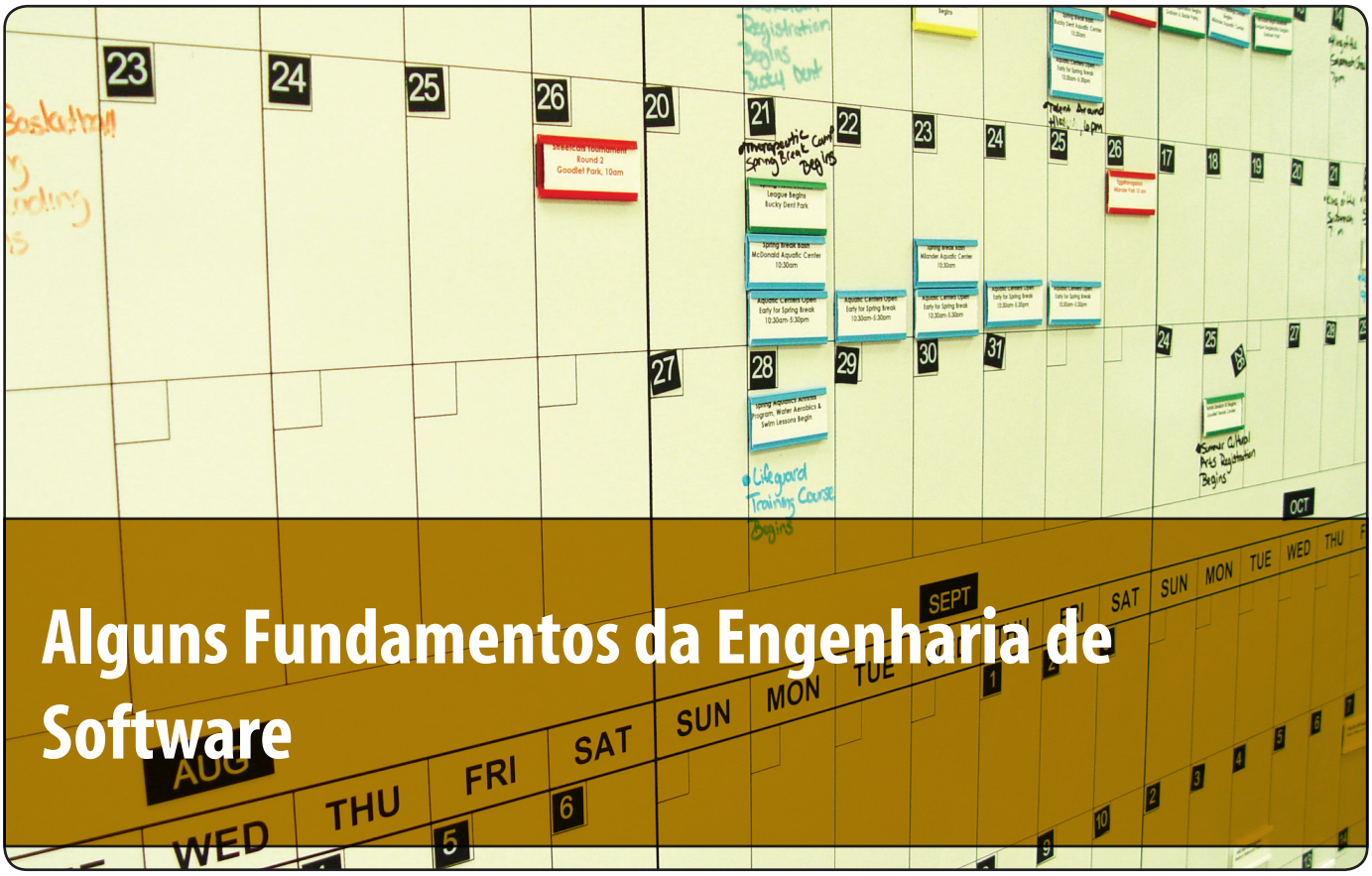
É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!

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

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

Parceiros:

ÍNDICE



Alguns Fundamentos da Engenharia de Software

Wilson de Pádua Paula Filho

(wppf@ieee.org)

Engenheiro Mecânico pelo ITA, Doutor em Engenharia Elétrica pela Escola Politécnica da USP, Professor Titular aposentado do Departamento de Ciência da Computação da UFMG. Autor dos Livros "Engenharia de Software: Fundamentos, Métodos e Padrões" e "Multimídia: Conceitos e Aplicações". Atualmente é consultor em Engenharia de Software e trabalha no Synergia – Laboratório de Engenharia de Software e Sistemas da UFMG.

O que é Engenharia de Software?

É a mesma coisa que Ciência da Computação? Ou é uma entre muitas especialidades da Ciência da Computação? Ou dos Sistemas de Informação, ou do Processamento de Dados, ou da Informática, ou da Tecnologia da Informação? Ou é uma especialidade diferente de todas as anteriores?

Na maioria das instituições brasileiras de ensino superior, o conjunto de conhecimentos e técnicas conhecido como Engenharia de Software é ensinado em uma ou duas disciplinas dos cursos que têm os nomes de Ciência da Computação, Informática ou Sistemas de Informação. Raramente, em mais disciplinas, muitas vezes opcionais, e muitas vezes oferecidas apenas em nível de pós-graduação. Algumas instituições oferecem cursos de pós-graduação em Engenharia de Software, geralmente no nível de especialização.

O uso do termo para designar uma carreira profissional também não é muito comum, mesmo em organizações que produzem grande quantidade de software, ou até naquelas em que o desenvolvimento de software é atividade fim. Programas e exames de certificação em Engenharia de Software são pouco conhecidos, ao contrário do que acontece com algumas linguagens e tecnologias usados por esses profissionais.

Em outros países, a situação é um pouco diferente. Algumas universidades americanas oferecem programas de graduação, mestrado e doutorado na área. O IEEE (*Institute of Electrical and Electronics Engineers*), principal organização internacional de profissionais de Engenharia Elétrica, através da *Computer Society*, que forma o seu ramo em Computação, oferece a qualificação de Profissional Certificado para o Desenvolvimento de Software.

Ciência, Engenharia e Valor

Sem pretender fazer distinções definitivas, vamos explorar o que dizem os dicionários. O Dicionário Aurélio Eletrônico V.2.0 assim define Ciência e Engenharia:

Ciência *- Conjunto organizado de conhecimentos relativos a um determinado objeto, especialmente os obtidos mediante a observação, a experiência dos fatos e um método próprio.*

Engenharia *- Arte de aplicar conhecimentos científicos e empíricos e certas habilitações específicas à criação de estruturas, dispositivos e processos que se utilizam para converter recursos naturais em formas adequadas ao atendimento das necessidades humanas.*

Vê-se que, pelas definições acima, a Ciência focaliza acumulação do conhecimento através do método científico, com base em experimentos e observações. Já a Engenharia aplica esses conhecimentos "ao atendimento das necessidades humanas". Embora o conhecimento seja certamente uma necessidade humana, trata-se de uma entre várias outras, sejam necessidades materiais, como alimentação, moradia, segurança, ou imateriais, como afeição ou auto-estima. A tudo aquilo que satisfaz a necessidades, atribui-se um valor. A Engenharia está, portanto, ligada à noção de valor, e a Engenharia de Software busca gerar valor com o processamento de informação. A noção de valor tem muitas conseqüências práticas importantes, e, de fato, a teoria conhecida como "Engenharia de Software Baseada em Valor" representa uma importante escola de pensamento dentro da área.

Arte, Técnica, Artesanato, Indústria?

Outros termos constantes da definição de Engenharia podem ser explorados de várias formas, com conseqüências interessantes. Por exemplo, é usada a palavra "Arte", que o mesmo dicionário define como a "capacidade que tem o homem de pôr em prática uma idéia, valendo-se da faculdade de dominar a matéria", ou "a utilização de tal capacidade, com vistas a um resultado que pode ser obtido por meios diferentes". Na Engenharia de Software, a matéria dominada pelas faculdades humanas consiste em máquinas de processamento da informação, devidamente configuradas e programadas. Nesse sentido, os conceitos de "Arte" e "Técnica" são bem próximos; aliás, a palavra grega *techné* significa, exatamente, Arte.

O termo "Arte" abre outras discussões. Não poucos programadores se consideram como artistas, no sentido de praticantes das Belas Artes, e valorizam critérios estéticos na criação de seus programas. Isso pode ter conseqüências boas e ruins, do ponto de vista de gerar valor. Por um lado, a busca da elegância pode levar à economia e simplicidade de formas, fazendo com que resultados de melhor qualidade sejam obtidos de maneira mais produtiva. E, principalmente, levando a escrever programas que possam ser mais facilmente reutilizados, mantidos e expandidos. Por outro lado, a auto-satisfação do programador pode ter como preço os interesses de quem está pagando pelo trabalho dele, ou de quem o usará. Seja, por exemplo, produzindo programas que ninguém entende, senão o próprio autor (e, depois de certo tempo, nem ele mesmo). Seja, como outro exemplo, introduzindo funções que o autor achou interessantes, mas não são realmente necessárias, nem foram solicitadas.

E muito próxima de "Arte" está a palavra "Artesanato", que lembra produção caseira, em pequena escala, sem a utilização de métodos industriais, que são caracterizados pela padronização e pela repetição. E, realmente, parece mais difícil aplicar esses métodos industriais na confecção de software, do que nos ramos da engenharia do mundo material. Nestes, as leis físicas impõem limites claramente visíveis ao que pode ser feito. Na Engenharia de Software, a criatividade não é limitada por leis físicas, e sim pela capacidade humana de entender e dominar a complexidade.

Mas não se pode escapar do fato de que a Engenharia de Software tem que resolver muitos problemas de ordem industrial. Raramente é possível construir software profissional sem envolver equipes, às vezes de dezenas ou até centenas de pessoas; raramente é possível trabalhar na área sem a pressão de prazos e orçamentos apertados; freqüentemente defeitos de software podem acarretar prejuízos vultosos, e, em certos casos, até riscos à vida humana; muitas vezes empreendimentos de software são afetados por um contexto econômico, político ou social.

Produtos e Ciclos de Vida

A íntima relação entre a Engenharia de Software e a noção de valor significa que essa profissão trata o software como produto. Estão fora do escopo da Engenharia de Software programas que são feitos com objetivo exclusivamente lúdico: a diversão do programador. Estão fora de seu escopo também pequenos programas descartáveis, feitos por alguém apenas para resolver um problema dessa pessoa, e que não serão utilizados por outros. Mas, se um desses programas interessar a outras pessoas, e assim passar a ter valor, aparecerão demandas para melhorar suas qualidades, aumentar suas funções, prolongar sua vida. E aparecerá quem esteja disposto a investir nisso, com a expectativa de ganhar dinheiro. Nesse momento, o programa entrou no escopo da Engenharia de Software e se tornou um produto. Muitos dos produtos de software mais usados seguiram esse caminho.

A íntima relação entre a Engenharia de Software e a noção de valor significa que essa profissão trata o software como produto.

Todo produto tem usuários: aqueles que efetivamente usam o produto. Alguns produtos são feitos por encomenda de um cliente: aquele que pagará por sua produção. Outros, chamados de produtos de prateleira, são vendidos no mercado aberto, a quem se interessar. Neste caso, quem faz o papel do cliente é quem define que recursos e funções se esperam do produto; talvez um departamento de vendas, ou de marketing, ou de definição de produtos de uma organização, ou até, para produtos menores, os próprios desenvolvedores, colocando o chapéu de investidores de risco. E existem todos os casos intermediários, em que um produto parcialmente pronto é completado, adaptado ou incrementado, por encomenda de um cliente.

Como todo produto industrial, o produ-

to de software tem seu ciclo de vida:

- ele é concebido para tentar atender a uma necessidade;
- é especificado, quando essas necessidades são traduzidas em requisitos viáveis;
- é desenvolvido, transformando-se em um conjunto formado por código e outros itens, como modelos, documentos e dados;
- passa por algum procedimento de aceitação e é entregue a um cliente;
- entra em operação, é usado, e sofre atividades de manutenção, quando necessário;
- é retirado de operação ao final de sua vida útil.

O ciclo de vida compreende muitas atividades, que são assunto das diferentes áreas da Engenharia de Software. A **Figura 1** mostra um modelo do ciclo de vida do software, usando a notação conhecida como UML (*Unified Modeling Language*). Essa notação é usada para descrever muitos aspectos diferentes do desenvolvimento de software, inclusive as seqüências de atividades que compõem esse desenvolvimento. O tipo específico de diagrama que é mostrado na figura é o Diagrama de Atividades, que representa uma evolução dos tradicionais fluxogramas, usados pelos programadores há décadas.

É interessante observar que a codificação, que representa a escrita final de um programa em forma inteligível para um computador, é apenas uma pequena parte do ciclo de vida. Muitas pessoas, inclusive alguns profissionais da informática, acham que a codificação é a única tarefa de um engenheiro de software.

Nos Diagramas de Atividades da UML, a bolinha cheia representa Início; a bolinha com um anel adicional representa fim; cada retângulo simples de cantos arredondados representa uma ação, ou seja, uma atividade simples, da qual não mostram detalhes; cada seta representa uma transição entre atividades; as barras paralelas representam início e término de atividades executadas em paralelo; um retângulo de cantos arredondados com detalhes internos representa uma atividade estruturada, que é composta de outras atividades.

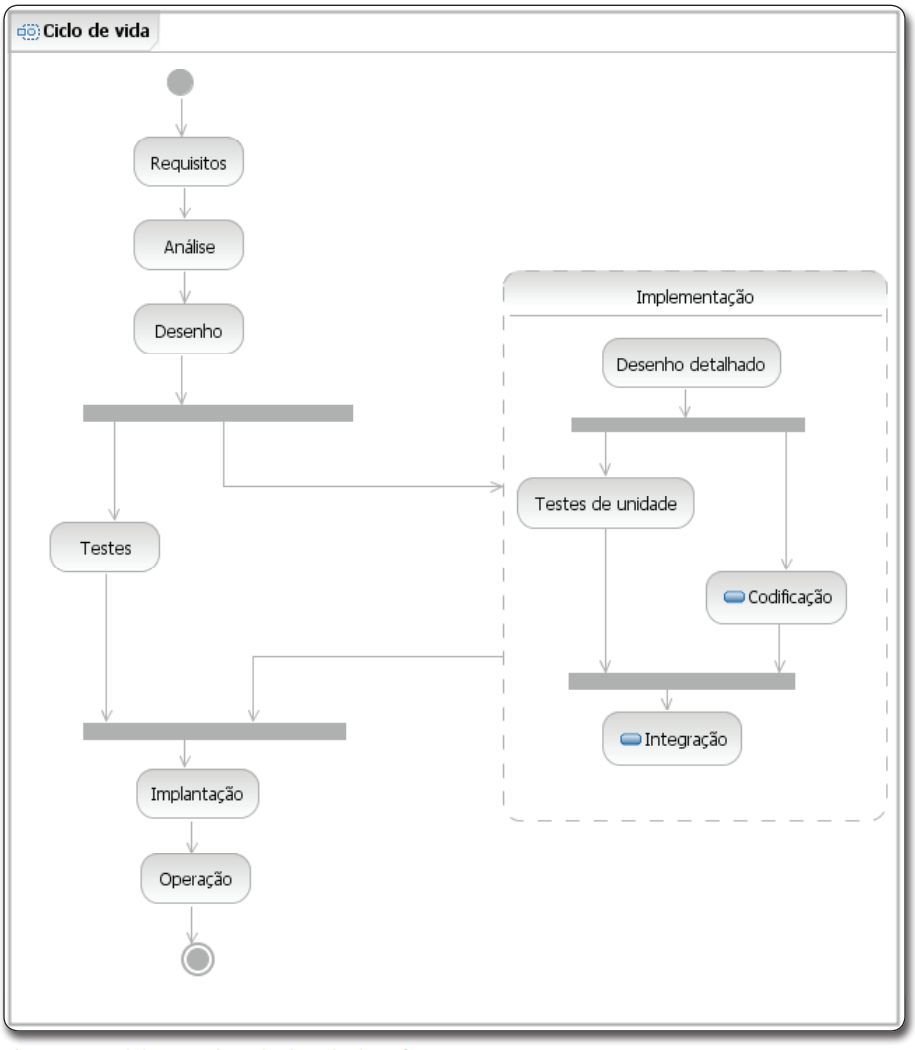


Figura 1. Modelo UML do ciclo de vida do software

Projetos, Atividades e Estruturas Analíticas

Normalmente, o software é desenvolvido dentro de projetos. Todo projeto tem uma data de início, uma data de fim, uma equipe e outros recursos. O responsável por um projeto é chamado de gerente de projeto. O trabalho realizado dentro de um projeto pode ser descrito por um conjunto de atividades, que podem possuir relações de dependência, paralelismo, e decomposição em atividades mais elementares, como no diagrama da **Figura 1**.

As atividades são delimitadas por marcos, isto é, pontos que representam estados significativos do projeto. Geralmente, os marcos são associados a resultados concretos: documentos, modelos ou porções do produto, que podem fazer parte do conjunto prometido aos clientes, ou ter apenas utilização interna ao projeto. O próprio produto é um resultado concreto associado ao marco de conclusão do projeto, que pode ser utilizado sozinho, ou como componente de um sistema.

O PMI (*Project Management Institute*) é uma organização internacional, com seções em muitos países, inclusive o Brasil, que tem o objetivo de promover e difundir boas práticas de gestão de projetos. Para isso, administra programas de certificação de profissionais nessa área, e publica o guia conhecido PMBOK (*Project Management Body of Knowledge*).

Nesse guia, define-se um projeto como um empreendimento temporário realizado para criar um produto, serviço ou resultado distinto. Um produto, por sua vez, é definido como um objeto produzido, quantificável e que pode ser um item final ou um item componente. Uma atividade é definida como um componente de trabalho realizado durante o andamento de um projeto.

Os relacionamentos entre as atividades que compõem um projeto são mostrados em uma estrutura analítica, que o PMBOK define como uma decomposição hierárquica orientada à entrega do trabalho

a ser executado pela equipe do projeto, para atingir os objetivos do projeto e criar as entregas necessárias. Estruturas analíticas de projeto podem ser apresentadas de muitas maneiras. Diagramas de atividade são uma das representações mais usadas atualmente. Outro tipo de representação usual é fornecida por planilhas e cronogramas, como aqueles que são produzidos pela ferramenta Microsoft Project (**Figura 2**).

Modelos de Referência e Fatores de Produção

O PMBOK é um exemplo de modelo de referência: uma estrutura de conhecimento que organiza conceitos, práticas e padrões de uma área. No caso do PMBOK, a área focalizada é a gestão de projetos de qualquer natureza, cobrindo assuntos como integração, escopo, tempo, custos, qualidade, recursos humanos, comunicações, riscos e aquisições.

Outro modelo de referência importante na Engenharia de Software é o CMMI (*Capability Maturity Model Integration*), que foi formulado pelo *Software Engineering Institute* da *Carnegie-Mellon University*. O CMMI foi encomendado e patrocinado pelo Departamento de Defesa americano, popularmente conhecido como Pentágono, que o utiliza para avaliação da capacidade de seus fornecedores de software. Sendo o Pentágono provavelmente a maior organização compradora de software do mundo, o CMMI tem grande aceitação da indústria americana de software, e considerável influência no resto do mundo. A rigor, suas práticas não são restritas à indústria de software, podendo ser aplicadas ao desenvolvimento de outros tipos de produtos.

Os conceitos do CMMI têm raízes em comum com o PMBOK, como se pode observar pela similaridade das definições que adota:

Produto - *Resultado que se pretende entregar a um cliente ou usuário.*

Projeto - *Conjunto gerido de recursos inter-relacionados, que entrega um ou mais produtos a um cliente ou usuário, com início definido e que, tipicamente, opera conforme um plano.*

Estrutura analítica do projeto - *Um arranjo dos elementos do trabalho e dos relacionamentos deles entre si e com o produto final.*

Ao contrário do PMBOK, o CMMI não se limita aos conhecimentos sobre gestão de projetos. Cobre também áreas técnicas, e focaliza principalmente a aplicação dos processos ao desenvolvimento de produtos. Um processo, segundo o IEEE, é uma seqüência de passos executados com um determinado objetivo; segundo o PMBOK, é um conjunto de ações e atividades inter-relacionadas, realizadas para obter um conjunto especificado de produtos, resultados ou serviços.

Um projeto representa a execução de um processo, e um processo é uma receita que é seguida durante a realização um

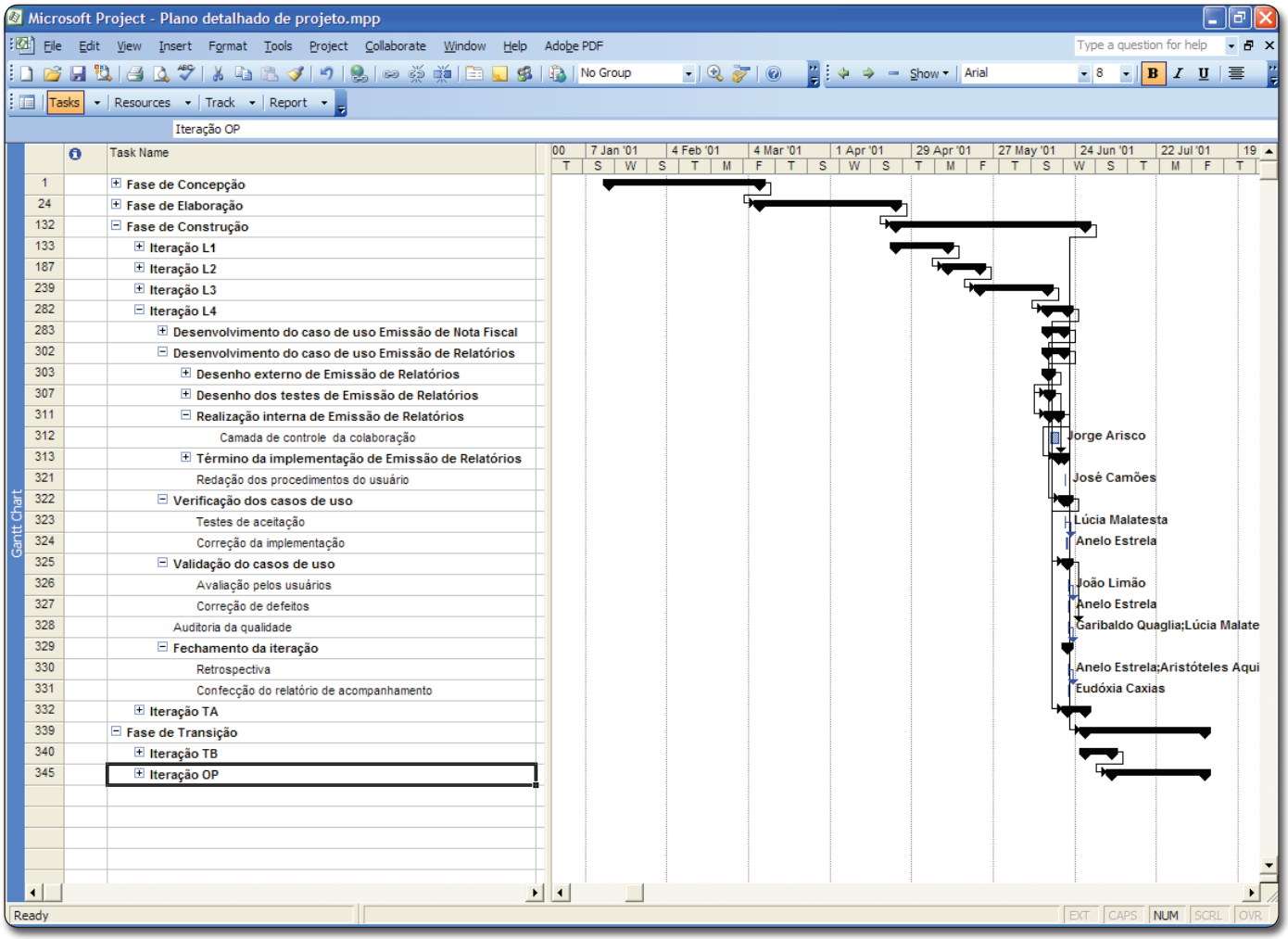


Figura 2. Cronograma de um projeto.

projeto; em outras palavras, o projeto é o empreendimento que concretiza uma abstração, que é o processo. Não se deve confundir um processo (digamos, uma receita de bolo) com o respectivo produto (o bolo) ou com a execução do processo através de um projeto (a confecção de um bolo por determinada pessoa, em determinado dia).

Processos, pessoas e tecnologia constituem os fatores de produção, que determinam o grau de sucesso dos projetos, ou seja: se eles conseguem entregar um produto de qualidade suficiente, dentro de um prazo aceitável e com custos viáveis. Portanto, desses fatores dependem a rentabilidade e a sobrevivência das organizações produtoras.

Para muitos profissionais, a tecnologia é o fator mais atraente; muitas vezes, há um otimismo exagerado quanto aos resultados da aplicação de novas tecnologias. Entretanto, tecnologias aparentemente promissoras podem levar para um beco sem saída, e, às vezes, tecnologias consideradas inferiores pelos especialistas lançam raízes permanentes, graças a forças de mercado. Além disso, a tecnologia só se paga quando colocada nas mãos de pessoas capacitadas, trabalhando dentro de processos adequados. Toda introdução de nova tecnologia tem uma curva de aprendizado: as pessoas precisam ser treinadas, cometem inicialmente muitos erros e, por isso, podem até se tornarem menos produtivas, durante algum tempo. Algumas tecnologias mais complexas só se pagam depois de muitos projetos.

Investir na capacitação das pessoas é certamente necessário. Entretanto, formar pessoas é difícil, caro e demorado. Recrutar pessoas capacitadas também: não há sinais de que a oferta de pessoas com alta qualificação no desenvolvimento de software venha a se igualar à demanda, em futuro próximo. Além disso, muitas pessoas produzem menos que a sua capacidade permitiria, por falta de liderança, por deficiência de ferramentas e suporte, ou por inadequação de processos.

Dos investimentos nos fatores de produção, os investimentos nos processos são geralmente aqueles que podem trazer retorno em prazo mais curto. Processos também não fazem milagres, mas a melhoria dos processos costuma trazer benefícios em prazos relativamente curtos, como é ilustrado por inúmeros relatos de experiência publicados. No mínimo, contribuem para reduzir o desperdício e o retrabalho, o que geralmente já traz ganhos muito significativos.

A melhoria dos três fatores (tecnologias, pessoas e processos) também requer seu próprio processo: ela deve ser feita em etapas bem-definidas e controladas, para que as organizações, em prazos relativamente curtos, possam avaliar se seu investimento está tendo o retorno esperado. E a principal contribuição de modelos de referência como o CMMI é indicar o caminho das pedras e o mapa da mina: onde a experiência coletada indica que o investimento em melhorias é mais rentável, em cada passo da evolução das organizações.

Conclusão

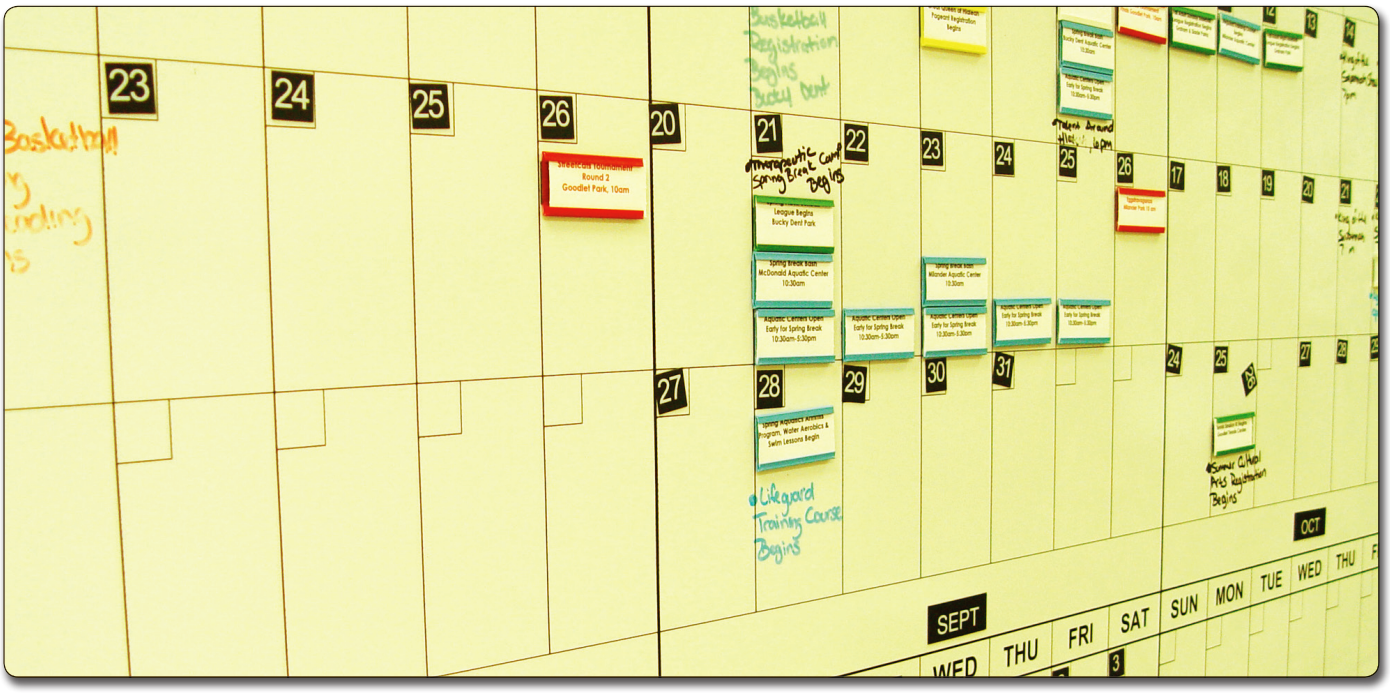
A Engenharia de Software visa à criação de produtos de software que atendam as necessidades de pessoas e instituições e, portanto, tenham valor econômico. Para isso, usa conhecimentos científicos, técnicos e gerenciais, tanto teóricos quanto empíricos. Ela atinge seus objetivos de produzir software com alta qualidade e produtividade quanto é praticada por profissionais treinados e bem informados, utilizando tecnologias adequadas, dentro de processos que tirem proveito tanto da criatividade quando da racionalização do trabalho. ●

Links

SEI
http://www.sei.cmu.edu/

CMMI
http://www.sei.cmu.edu/cmmi/

PMI Brasil
http://www.pmi.org.br/



kali software

Desde 2004 Trazendo Teoria para a Prática

Treinamento e Consultoria em Engenharia de Software

Profissionais com ampla experiência prática (Consusltores Certificados) e acadêmica (Professores Universitários, Mestres e Doutores)

Know-how para implementar processos de software de acordo com modelos de qualidade (MPS e CMMI) e maximizar o retorno de investimento

Cursos 2008: *Inscrições Abertas*

Certificados emitidos pela Kali Software
Aulas aos sábados na Zona Sul do Rio, próximo ao metrô
Para outras localidades, por favor entre em contato

Março | Abril	Processos de Software com ênfase em CMMI e MPS – **32 horas** Teste de Software – **16 horas**
Maio | Junho	Engenharia de Requisitos – **32 horas** Gerência de Configuração de Software – **16 horas**
Julho | Agosto	Gerência de Projetos de Software – **32 horas** Projeto Orientado a Objetos com UML e Padrões – **32 horas**

Confira as ementas, investementos e condições de pagamento: *www.kalisoftware.com*
Oferecemos também cursos de programação (Java, Java para Web e Ajax)
Inscrição e outras informações: *info@kalisoftware.com*

kalisoftware.com | info@kalisoftware.com

Melhorando Processos Através da Análise de Risco e Conformidade

Rafael Espinha

rafael.espinha@primeup.com.br

É Engenheiro de Computação e Mestre em Engenharia de Software pela PUC-Rio. Foi pesquisador do Laboratório de Engenharia a de Software da PUC-Rio e atualmente é consultor e diretor da PrimeUp, onde desenvolve projetos na área de melhoria de processos e qualidade de software. Possui cursos e certificações em CMMI e MPS.BR.

João Sousa

joaomss@primeup.com.br

É Bacharel em Informática pela UERJ e pós-graduado pela UFRJ. Possui mais de 15 anos de experiência no desenvolvimento de sistemas e melhoria de processos. Atualmente é consultor da PrimeUp, onde desenvolve projetos na área de melhoria de processos e qualidade de software.

Um dos fatores importantes para a construção de um software de qualidade é o processo de desenvolvimento utilizado e como este é implantado na organização. A inexistência ou a não utilização de processos bem definidos e de boas práticas de desenvolvimento, mesmo que informais, faz com que o desenvolvimento de software seja realizado de forma *ad-hoc*, ficando altamente dependente da experiência e do conhecimento das pessoas envolvidas. Este cenário resulta na realização de projetos cujos resultados são imprevisíveis, onde cada um realiza as suas atividades da forma que lhe convém, e dificulta a reutilização de boas práticas e de lições aprendidas.

Com a crescente demanda por qualidade dos produtos de software, a adoção de modelos de maturidade, normas de qualidade e guias de boas práticas na definição de processos tem se tornado cada vez mais freqüente. Modelos como CMMI-DEV e MPS.BR e normas como a ISO/IEC 15504 e 12207 definem um conjunto de boas práticas e características que devem estar presentes em um processo para que este possa ser gerenciado e resulte na entrega de produtos de qualidade. Entretanto, como têm o objetivo de serem aplicáveis a quase todo tipo de organização, estes modelos ou normas muitas vezes não definem como estas boas práticas e características devem ser implementadas e implantadas. Uma das maiores dificuldades de organizações que iniciam um programa de melhoria de processos enfrentam é a dificuldade de adaptar este conjunto de boas práticas para a sua realidade, identificando quais áreas são mais relevantes e devem ser abordadas com maior urgência. Cada organização possui políticas, crenças e uma cultura específica, que deve ser levada em conta para que as melhorias sejam bem aceitas e realmente contribuam com um desenvolvimento mais eficiente.

Para orientar a necessidade de adaptação apresentada acima, utilizamos o con-

ceito de risco associado à não utilização das boas práticas de desenvolvimento de software nos projetos e processos da organização. Consideramos também que a qualidade do produto (software) está diretamente relacionada à qualidade dos processos utilizados na sua produção e ao conhecimento técnico que os usuários deste processo têm sobre as práticas definidas (institucionalização do processo). Partindo-se destas premissas pode-se concluir que qualquer risco à qualidade e à institucionalização do processo se reflete em riscos na qualidade do produto que será entregue e, consequentemente, em riscos para a organização. Sendo assim, ações de gerência de risco nos processos podem contribuir diretamente com a garantia da qualidade do produto final e fornecem dados que permitem identificar quais ações devem ser tomadas com maior urgência.

Neste artigo apresentamos uma abordagem de análise de processos na qual é identificado de forma customizada tanto o nível de conformidade (quantidade de recomendações do modelo de referência implementadas nos processos da organização) quanto o nível de risco (quantidade de riscos presentes no processo de desenvolvimento devido às recomendações não implementadas) em cada área de processo. Dessa maneira, uma análise dos processos da organização fornece duas classes de dados para a tomada de decisão e direcionamento de recursos, indicando o que deve ser feito para melhorar o processo (conformidade) e quais ações devem ser tomadas primeiro (risco).

Uma das formas mais indicadas para a definição e implantação de processos de maneira eficiente é a utilização de um ciclo de melhoria contínua. Esta forma pode ser comparada ao desenvolvimento iterativo de um sistema, onde o processo é definido e implantado na organização em fases direcionadas pelas necessidades e características da organização. O modelo IDEAL, desenvolvido pelo Software Engineering Institute (SEI), ilustra a utilização deste conceito. A implantação deste ciclo de melhoria faz com que os processos de uma organização sejam constantemente avaliados e melhorados.

No modelo IDEAL destacam-se duas fases: Diagnóstico e Estabelecimento. A fase de Diagnóstico consiste em avaliar o ambiente produtivo e identificar as oportunidades de melhoria. Em geral, nesta fase é realizado um estudo no qual um cenário esperado é definido e o cenário atual é avaliado segundo algum critério de qualidade ("*Gap Analysis*"). O cenário esperado pode ser definido a partir de um ou mais modelos ou normas de referência, como, CMMI-DEV 1.2 ou ISO 9001:2000. Dessa forma, obtém-se a diferença entre o que se espera dos processos da organização e onde eles realmente estão. A partir daí, elaboram-se planos de ação para que esta distância seja diminuída ou eliminada, a partir da priorização e seleção dos planos de ação que serão implantados (fase de Estabelecimento).

A próxima seção apresenta alguns conceitos básicos sobre riscos e como este conceito é utilizado neste domínio. Em seguida a estratégia e a ferramenta que compõem a abordagem serão apresentadas e um exemplo da sua utilização será discutido.

Análise de Risco

Risco é a "exposição à chance de perdas ou danos". Embora exista também uma chance de alguma coisa sair melhor do que o esperado, o risco geralmente costuma ser associado a possíveis perdas ou danos. O conceito de risco resulta da incerteza quanto a eventos futuros e é parte de

Por mais controlada e precisa que seja a execução de uma atividade de desenvolvimento, sempre existe o risco, mesmo que muito remoto, de algo dar errado.

A abordagem que apresentaremos facilita a realização das fases de Diagnóstico e Estabelecimento de um ciclo de melhoria contínua, identificando claramente os riscos associados aos processos definidos (Diagnóstico) e fornecendo um critério de priorização destes riscos (Estabelecimento). Para isso, são utilizadas uma estratégia de avaliação e atividades de avaliação e melhoria de processos de desenvolvimento de software. A avaliação verifica tanto a dimensão de conformidade entre o processo e modelos de maturidade ou normas de qualidade, quanto à dimensão dos riscos que a não utilização das boas práticas definidas nestas referências oferecem à qualidade do produto desenvolvido pela organização e aos seus objetivos de negócio. Esta ferramenta também indica como estes pontos podem ser solucionados de forma que a organização obtenha uma maior qualidade ou resultados a partir deste ciclo. O diferencial desta abordagem é a utilização da análise de risco como um instrumento de priorização das ações que devem ser tomadas pelas empresas para mitigar (reduzir as chances de ocorrência) os riscos identificados durante a fase de diagnóstico fornecendo um critério concreto para definição do escopo de cada ciclo de melhoria.

todas as atividades de desenvolvimento. Um processo de desenvolvimento bem definido e institucionalizado visa reduzir a chance de ocorrência de ameaças (perdas ou danos). Toda oportunidade de sucesso sempre carrega consigo uma possibilidade de falha, cabendo a cada empresa avaliar a relação risco *versus* retorno e determinar se "estar" sujeito a esta perda é aceitável, se este evento é muito grave, ou ainda se o procedimento para a mitigação não oferece um retorno satisfatório.

Por mais controlada e precisa que seja a execução de uma atividade de desenvolvimento, sempre existe o risco, mesmo que muito remoto, de algo dar errado. Este fato decorre do grande número de variáveis que podem influenciar no resultado final e da sua natureza muitas vezes imprevisível. A partir desse cenário, torna-se necessário aprender a conviver com riscos, minimizando suas possíveis conseqüências negativas.

As principais atividades da disciplina de gerência de riscos são:

- **Identificação** corresponde à identificação dos riscos inerentes a uma etapa do desenvolvimento (fase, processo, iteração). Isto é feito através do levantamento das ameaças presentes e do impacto que podem provocar caso se realizem.
- **Análise** corresponde à reflexão so-

bre os riscos identificados, a partir da identificação do nível de exposição de cada projeto. É também realizado um estudo de classificação dos riscos no qual, baseando-se no relacionamento entre a exposição e conseqüência negativa do risco e o benefício da oportunidade, determina-se quais serão eliminados, quais serão mitigados, quais serão aceitáveis e quais serão acompanhados.

• **Planejamento** observa e determina como e quando os riscos serão abordados ao longo do projeto. Comumente são elaborados planos de mitigação, eliminação e acompanhamento de riscos que serão utilizados como base para a gerência de riscos.

A eliminação total dos riscos associados a um projeto ou iniciativa é um conceito utópico.

• **Controle** corresponde à execução e ao acompanhamento dos planos elaborados para o projeto. Os riscos identificados são analisados constantemente para a identificação do seu estado atual e atualização dos planos elaborados. Novos riscos podem ser identificados também.

A gerência de riscos utiliza como base o conceito de exposição de risco. Para cada ameaça ou possível fonte de problema que possa causar perdas ao projeto, a exposição (Exp) é definida como o produto da probabilidade da perda ocorrer (P) e do tamanho da perda (impacto I no projeto, artefato, ativo ou qualquer elemento que seja o foco da análise de risco): $Exp = P * I$.

Na abordagem apresentada, o conceito de exposição foi estendido, permitindo uma melhor determinação do impacto da concretização de um risco. Para cada ativo da organização ou projeto avaliado (pessoa que participa do desenvolvimento ou processos utilizados) é determinado um índice de exposição balanceado. Este índice, denominado PSR, determina a exposição através da probabilidade da ocorrência do risco (P), a severidade dessa ocorrência para o projeto ou para a organização (S) e a relevância do ativo da organização (pessoa ou processo) na qualidade do produto final. Dessa forma é possível obter um retrato mais preciso da distribuição e do impacto dos riscos.

A estratégia de diminuição de riscos utilizada nos planos elaborados durante a atividade de planejamento pode tentar reduzir sensivelmente a probabilidade do risco se realizar, fazendo com que a concretização da perda seja algo muito difícil de ocorrer. É possível tentar reduzir o impacto do risco, fazendo com que a conseqüência da materialização dele seja tão pequena que a sua ocorrência pouco afetará o resultado final do projeto. Por fim, é possível reduzir tanto a probabilidade quanto a severidade do risco, resultando em um balanceamento razoável entre as possíveis perdas e a probabilidade de sua ocorrência no projeto.

A eliminação total dos riscos associados a um projeto ou iniciativa é um conceito utópico. Cada ação de identificação, acompanhamento e controle (redução, mitigação ou contenção de riscos) possui um custo associado, cabendo a cada indivíduo ou organização identificar o ponto de equilíbrio entre o nível de exposição aos riscos e o custo de redução. Para cada projeto ou iniciativa deve-se determinar um índice aceitável de exposição aos riscos, que seja suficiente para as características do projeto e tenha um custo que não comprometa os resultados.

A Estratégia de Avaliação

A estratégia de avaliação que utilizaremos se baseia no conceito de análise de risco na verificação dos processos de uma organização e compreende os conceitos apresentados na ISO/IEC 17799. Esta estratégia recomenda a avaliação de uma equipe de desenvolvimento ou processo de software em duas etapas, onde a primeira (etapa de **Abrangência**) representa uma verificação mais abrangente e menos rigorosa da implementação dos modelos e normas de referência, para identificar quais são as áreas críticas para a organização (onde está o problema). Na etapa seguinte (etapa de **Profundidade**) são definidas quais áreas devem ser o foco de uma avaliação mais rigorosa e da próxima iteração do ciclo de melhoria contínua (qual é o problema e como ele pode ser solucionado). Em cada uma das etapas, são realizadas as fases de Diagnóstico (identificação dos riscos) e Estabelecimento (priorização e definição dos planos de ação). Na etapa de Abrangência o diagnóstico é mais abrangente e os planos de ação são menos detalhados e, na etapa de Profundidade, o diagnóstico é mais específico e os planos de ação são mais detalhados.

O objetivo desta estratégia é complementar métodos como SCAMPI, MA-MPS e ISO/IEC 15504, oferecendo propostas de soluções a alguns potenciais problemas encontrados na aplicação destes métodos. Os princípios que guiam a estratégia são:

• Mapear resultados aos objetivos do negócio da organização;

• Minimizar o esforço de avaliação segundo critérios de importância definidos pela própria organização;

• Obter maior aproveitamento dos resultados gerados;

• Utilizar duas dimensões de análise: conformidade e risco.

O primeiro princípio tem como objetivo sanar uma característica identificada na maioria dos métodos de avaliação de processos de desenvolvimento. De forma geral, a aplicação destes métodos gera conclusões ou resultados que estão restritos ao nível das diretivas do modelo de maturidade ou norma de qualidade utilizada. Este fato faz com que o trabalho de convencimento da alta gerência (geralmente não técnica) acerca da importância dos investimentos em engenharia de software ou qualidade seja dificultado e, consequentemente, o projeto de melhoria de processos fique ameaçado devido à falta de comprometimento dos patrocinadores. O objetivo destes princípios é dar ênfase às necessidades e prioridades da empresa, ao invés de impor uma estrutura que pode não ser a mais adequada a ela. O mapeamento dos resultados do nível de TI para o nível de negócios facilita o seu entendimento, podendo orientar a empresa a como atingir suas metas.

O segundo e terceiro princípio é resultado da constatação de um ponto fraco identificado nos métodos mais utilizados.

A grande maioria realiza inspeções e análises rigorosas, que abrangem todo o modelo de referência e geram relatórios com uma grande quantidade de informações sobre diversas áreas da engenharia de software mas, na maioria dos casos, outra grande quantidade de informações é desperdiçada. Uma avaliação indica o estado atual da organização e aponta as oportunidades de melhoria na implementação das diretivas do modelo de maturidade ou norma de qualidade utilizada como referência. Entretanto, devido a restrições de orçamento e ao impacto que elas representam no ambiente produtivo, apenas uma pequena parte destas recomendações pode ser implementada na próxima iteração do programa de melhoria. Após a implementação da primeira iteração de melhoria, o estado da organização já não é o mesmo de quando a avaliação foi realizada, tornando obsoletos os dados relativos às áreas de processos e as oportunidades de melhoria que não foram abordadas. A utilização de uma estratégia em duas etapas permite identificar, através de uma análise menos elaborada, as áreas mais relevantes e realizar uma análise mais elaborada apenas nestas áreas.

Finalmente, o quarto princípio tem como objetivo oferecer dados de um nível de abstração menos granular para a tomada de decisão. Embora a utilização da capacidade de processo e do nível de maturidade seja o parâmetro mais utilizado no direcionamento de recursos na área de desenvolvimento de software, estes conceitos são um tanto abstratos e em muitos casos dificultam esta atividade (se vários processos apresentam a mesma capacidade e o mesmo *gap* entre a capacidade esperada e a avaliada, qual deve receber os recursos?). A utilização de uma análise de risco oferece um critério de desempate ou uma opção para a priorização de investimentos.

Ferramenta de apoio à avaliação

Para auxiliar a realização da avaliação dos processos de uma organização utilizamos uma ferramenta de apoio à execução de avaliações.

A metodologia de análise de risco implementada pela ferramenta se baseia na avaliação de características de ativos da organização, que podem representar pessoas da organização, processos utilizados, tecnologias e características do ambiente de desenvolvimento. Cada ativo é mapeado em componentes de negócio (para o contexto de desenvolvimento de software foram utilizados objetivos do

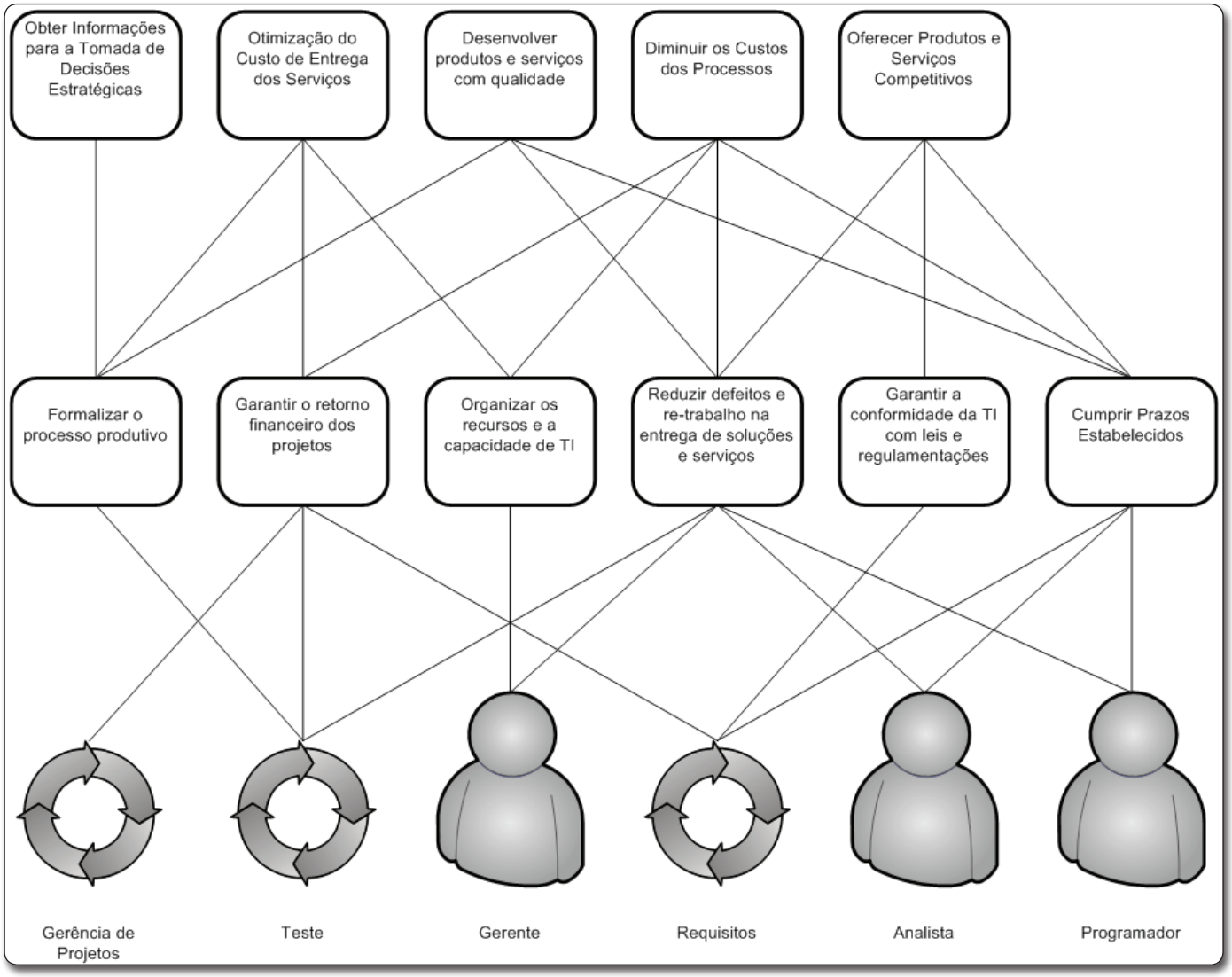


Figura 1. Mapeamento dos ativos em objetivos do negócio da organização

negócio ou de TI) da organização e possui uma relevância que está diretamente relacionada à relevância dos objetivos aos quais ele se relaciona. A **Figura 1** ilustra este conceito.

A cada ativo podem ser associados um ou mais componentes, que representam características que serão avaliadas em um projeto de avaliação que estão relacionadas à participação deste ativo, ou seja, ao adicionar um componente a um ativo estamos dizendo que ele é um coletor de dados que indicam o estado de implementação de um conjunto de boas práticas na organização. Um projeto de avaliação pode utilizar todos os componentes ou apenas um conjunto que seja relevante para esta instância de avaliação. Cada componente possui um *checklist* associado, composto por um ou mais controles, que representam os itens atômicos de verificação da implementação das boas práticas. Cada controle possui uma estrutura com os seguintes elementos:

• **Nome do Controle** indica uma característica (boa prática ou requisito) que deve estar presente no ativo para que o controle seja considerado implementado e seu risco associado seja eliminado.

• **Justificativa** define os termos e conceitos necessários para o entendimento do controle e fornece uma justificativa que explique porque aquele controle deve ser implementado. Neste elemento são apresentadas as vantagens que se obtém com a implementação do controle e as conseqüências da sua não implementação.

• **Ameaças** indicam quais elementos podem se aproveitar da não implementação do controle para se manifestar e causar danos ao negócio da organização.

• **Recomendação** fornece razões e dados para a elaboração de um plano de ação após a realização da avaliação, através de uma sugestão de como o controle pode ser implementado para diminuir a exposição da organização aos riscos e atingir a conformidade desejada com o modelo ou norma de referência.

• **Referências** relacionam referências bibliográficas para que mais informações acerca do controle e da sua implementação possam ser obtidas.

• **Probabilidade** representa a probabilidade de uma ameaça se manifestar caso o controle não esteja implementado na organização. Este elemento é representado por um número de 1 (menor) a 5 (maior probabilidade).

• **Severidade** indica o grau do impacto negativo na organização, caso uma ou mais ameaças se manifestem. Este elemento é representado por um número de 1 (menor) a 5 (maior severidade).

• **Agrupamento** indica a qual agrupamento o controle pertence. Os agrupamentos são comuns a todos os checklists, permitindo verificar o estado da imple-

Controle	Processo - CMMI – Gerência de Configuração – Prática 3.1 - 2 - As versões anteriores de um item de configuração devem ser passíveis de recuperação.		
Justificativa	Uma versão anterior de um item de configuração deve ser recuperada caso uma modificação não seja corretamente implementada, caso os arquivos atuais (na nova configuração do software) estejam corrompidos, caso seja efetuada uma junção (merge) incorretamente entre um ramo (linha temporária de desenvolvimento) e a linha principal de desenvolvimento. Nestas situações, pode ser apropriado restaurar uma versão anterior para que esta se torne a atual.		
Recomendação	Este controle pode ser implementado através dos seguintes procedimentos: - Disponibilizar as versões dos itens de configuração. - Reportar os procedimentos para: (1) recuperar uma versão anterior, (2) verificar as revisões de um item de configuração e (3) analisar as diferenças entre a versão anterior e a atual. Essas informações devem constar no plano de gerência de configuração e nos procedimentos de controle de versões. Caso não estejam, sugere-se alterá-los. - Garantir a integridade e a disponibilidade dos repositórios de configurações. Observação: A ferramenta de controle de versões deve facilitar a visualização e recuperação das versões dos itens de configuração. Portanto, contém funcionalidades que facilitam a implementação deste controle. Exemplos de Artefatos Produzidos: - Lista de versões de itens de configurações - Procedimentos para controle de versões		
Probabilidade	4	Severidade	3
Referências	Std 1042 - IEEE Guide to Software Configuration Management, Institute of Electrical and Electronics Engineers, 1987. ISO/IEC 12207 - Information technology - Software life cycle processes, International Organization for Standardization, 1995. ISO/IEC 15504 - Information technology - Process Assessment, International Organization for Standardization, 2004. CMMI-Dev: Área de Processo: Gerência de Configuração Bibliografia de apoio: H. R. Berlack, Software Configuration Management. New York: John Wiley & Sons, 1992.		
Ameaças	Baixa manutenibilidade Perda de controle de solicitações de mudança		
Agrupamento	Gerência de Configuração		

Tabela 1. Exemplo de controle

mentação de características espalhadas em vários checklists.

A **Tabela 1** apresenta um exemplo de controle de uma prática de gerência de configuração.

A avaliação consiste em responder aos checklists associados aos ativos e selecionados na configuração do projeto de avaliação. Estes podem ser respondidos diretamente ou através de questionários enviados via WEB, onde o conteúdo dos controles pode ser interpretado para um domínio específico, por exemplo, para os diversos papéis de uma organização. A utilização dos questionários permite um ganho de escala e de cobertura da avaliação, ao mesmo tempo em que diminui o impacto da avaliação e aumenta a aceitação das melhorias no processo de desenvolvimento uma vez que todos se sentem parte da avaliação e podem contribuir com comentários e sugestões.

Após a coleta dos dados, é gerado um conjunto de relatórios contendo tabelas e gráficos que indicam o estado da implementação das boas práticas e os riscos presentes na organização e fornecem dados para a tomada de decisão (o que e como deve ser feito). Cada controle não implementado ou implementado parcialmente, contribui com um índice de risco (PSR) que é obtido pela multiplicação da relevância do ativo avaliado (R), da probabilidade da concretização das ameaças possíveis (P) e da severidade desta concretização (S) . Além do PSR, os seguintes indicadores são utilizados:

$$\text{Índice de Segurança} = \frac{PSR_{\substack{\text{controles}\\ \text{implementados}}}(\text{elemento})}{PSR\,(\text{Total})}$$

$$\text{Índice de Conformidade} = \frac{\text{NumControles Implementados}}{\text{NumControlesTotal}}$$

A partir destes índices, pode-se gerar um grande número de interpretações, através da filtragem e agrupamento de dados das áreas de processo, ativos, ameaças, departamentos ou objetivos de negócio.

Utilizando a Abordagem

Para utilização desta abordagem em uma avaliação, utilizamos um conjunto de atividades.

A primeira atividade consiste na criação de um processo padrão para a área de processo de definida em algum modelo ou norma de referência, como Gerência de Configuração e Solução Técnica do CMMI, ou grupo de áreas de processo, que será utilizado como base para a elaboração das recomendações de implementação de controles. Esta atividade possui tarefas de elaboração e revisão de conteúdo e aderência aos modelos ou normas de qualidade utilizadas como referência. Em seguida, atividades de geração e revisão de arquitetura dos checklists são executadas de forma iterativa e concorrente, até que um produto com qualidade assegurada seja desenvolvido. A arquitetura do checklist consiste da identificação dos controles que serão desenvolvidos e dos questionários que serão utilizados como coletores automáticos de dados.

A arquitetura do checklist consiste da identificação dos controles que serão desenvolvidos e dos questionários que serão utilizados como coletores automáticos de dados.

Tendo um conjunto revisado dos controles que devem ser implementados e um questionário que interpreta e responde estes controles do checklist através de uma lógica bem definida para as suas perguntas, inicia-se uma nova etapa de implementação e revisão dos controles, onde o conteúdo dos controles identificados é elaborado e o questionário desenvolvido é refinado. Finalmente, os checklists são homologados e adicionados à ferramenta.

Tendo um guia para a elaboração dos checklists, o segundo passo é a identificação dos checklists que seriam utilizados para a realização das avaliações. Como os checklists são associados aos ativos da organização, através dos componentes, esta atividade foi realizada utilizando como parâmetro os tipos de ativos.

A ferramenta define quatro tipos de ativo para a realização das avaliações: "Pessoa", "Processo", "Ambiente" e "Tecnologia". Para o domínio de processos de desenvolvimento, definiu-se que cada ativo funciona como uma dimensão de coleta de dados sobre os processos da organização, das pessoas, dos fornecedores ou do projeto ao qual ele pertence. Sendo assim, chegou-se a seguinte taxonomia para os ativos:

- **Processos** representam fluxos de trabalho, áreas de processo ou disciplinas da organização. Cada checklists associado a este tipo de ativo possui o escopo de uma área de processo do modelo de maturidade ou norma de qualidade. A utilização destes ativos define uma dimensão da avaliação onde os processos são o principal fator para o alcance dos objetivos da organização e, consequentemente, a principal fonte de evidências da implementação do modelo ou norma de referência.
- **Pessoa** representa os atores da organização. Os checklists associados a este tipo de ativo verificam as diretivas da norma relativas a um papel da organização (desenvolvedor, gerente, analista de requisitos, etc.) que é assumido pela pessoa que é referenciada pelo ativo. A utilização de ativos do tipo Pessoa define uma dimensão da avaliação onde as pessoas e os papéis que elas executam são o principal fator para o alcance dos objetivos da organização e, consequentemente, a principal fonte de evidências da implementação do modelo ou norma de referência.
- **Ambiente** representam o ambiente organizacional, caracterizado pelas acomodações, aspectos inter-pessoais, política motivacional e outros fatores que afetam indiretamente a execução dos processos. Os checklists associados a este tipo de ativo verificam as características gerais da organização e como ela contribui ou não com o alcance dos objetivos da organização.
- **Tecnologia** representa a estrutura tecnológica da organização, definida por suas máquinas e ferramentas. Os checklists associados a este tipo de ativo verificam características da infra-estrutura referenciada pelo ativo, como segurança

em servidores, funcionalidades de ferramentas, entre outros.

Como terceiro passo da customização, a estrutura para a geração de checklists foi elaborada. Esta atividade contou com a definição dos agrupamentos, ameaças e do escopo dos controles.

Como o objetivo da avaliação é obter resultados mapeados nas áreas de processo do modelo ou norma de qualidade de referência, independente de quais ativos foram utilizados para coletar os dados, foi definido que os agrupamentos utilizados seriam as áreas de processo.

Uma área de processo possui características específicas, que determinam a sua implementação, e características comuns a todas as áreas, que indicam a sua capacidade. Para facilitar a utilização de modelos de maturidade que utilizam o conceito de capacidade de processos foi definido também que deve existir um agrupamento para cada nível de capacidade.

A **Figura 2** ilustra este conceito para o CMMI-DEV 1.2, onde um Checklist para o ativo desenvolvedor (papel) contém os agrupamentos Definição do Processo Organizacional, Desenvolvimento de Requisitos, Foco no Processo Organizacional, Garantia da Qualidade, Gerência de Configuração e Gerência de Requisitos que representam as características específicas de cada área de processo e o agrupamento GG2 – Processo Gerenciado representa as características genéricas. Estão também ilustrados controles dos agrupamentos de Gerência de Configuração e Gerência de Requisitos. Cada agrupamento contém um conjunto de controles.

Com a estrutura de agrupamentos definida, os resultados das análises de risco e conformidade podem ser consolidados pelos agrupamentos, resultando em relatórios (exemplificados no próximo tópico), que indicam risco e conformidade de cada área de processo, e que deixam de forma transparente quais tipos de ativo foram utilizados.

Para o escopo dos controles, foi definida a utilização do item de menor granularidade do modelo ou norma de referência. Como em muitos modelos e normas o item de menor granularidade possui mais de um item de verificação em seu escopo, foi definida também a realização de uma análise da atomicidade. Esta análise tem como objetivo quebrar o item em elementos de verificação atômicos, evitando situações de falha no preenchimento do controle (se existem dois ou mais pontos de verificação conectados por um operador "e" ou "ou" em um controle, como respondê-lo no caso de alguns estarem implementados e outras não?).

Finalmente, para uniformizar as análises de risco e conformidade em processos de desenvolvimento realizados com a utilização da ferramenta, foi elaborada uma lista de ameaças.

Tendo em vista que a grande maioria dos métodos de avaliação de processos de desenvolvimento (SCAMPI, MA-MPS, ISO/IEC 15504) utiliza uma estrutura de níveis para a classificação da implementação de uma diretiva do modelo ou norma, utilizamos nesta solução a mesma abordagem. Desta forma, cada controle pode ser classificado como "Implementado", "Não Implementado" ou "Parcialmente Implementado". Partindo do fato de que um controle parcialmente implementado não se encontra implementado, foi determinado que, quando a análise realizada indicasse que um controle não

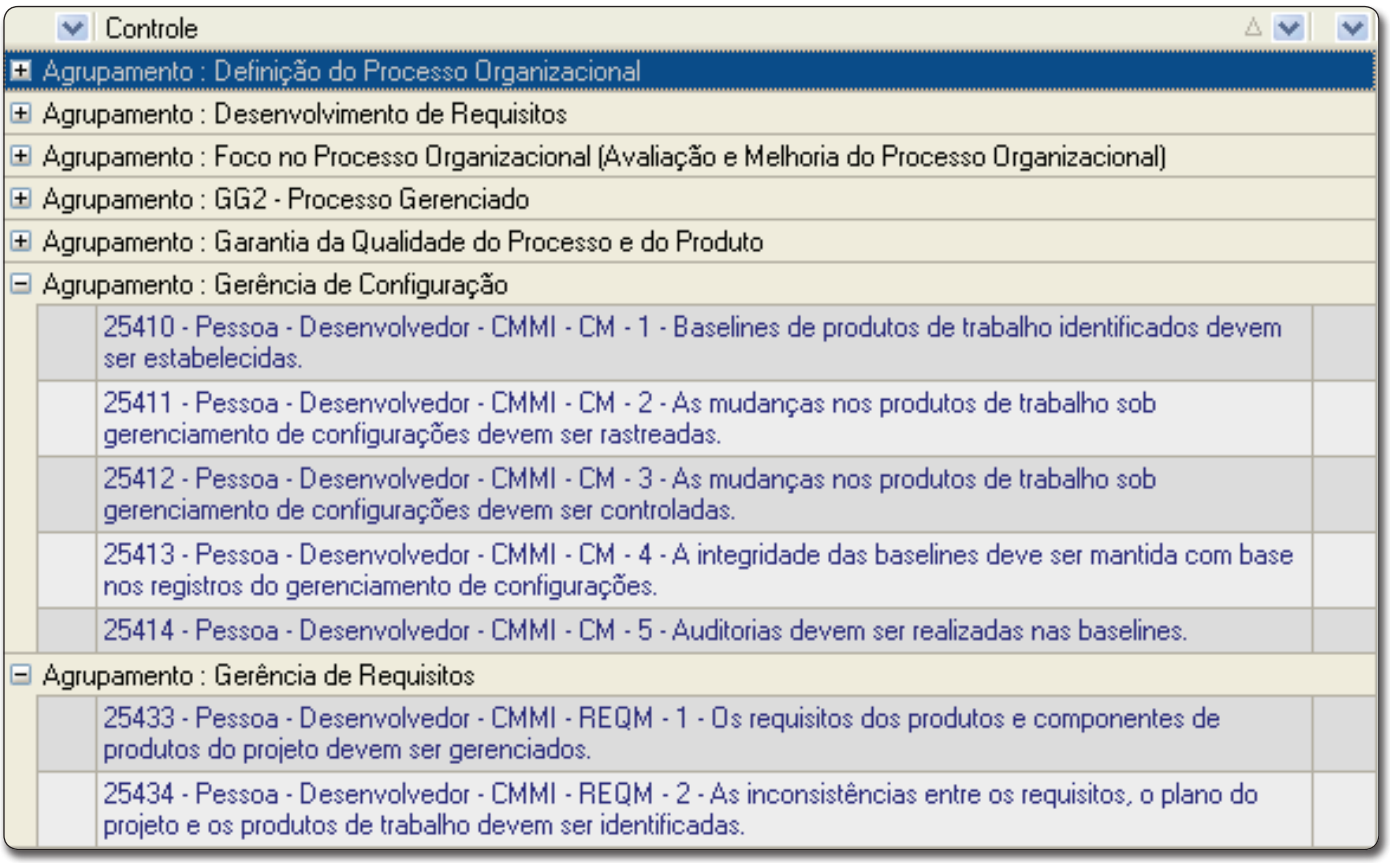


Figura 2. Checklist com agrupamentos para características específicas e genéricas

foi totalmente atendido, este deve ser preenchido como "Não implementado". Utilizando a premissa de que um controle parcialmente implementado possui uma probabilidade menor de causar danos do que um controle não implementado, foi determinado que, para cada nível de implementação atingido pelo controle, a sua probabilidade será diminuída em uma unidade, até chegar ao valor mínimo 1.

Para exemplificar a abordagem, considere um controle cuja probabilidade seja quatro. Em uma avaliação que utiliza o modelo CMMI como referência, um controle classificado como parcialmente implementado será preenchido como Não Implementado e a sua probabilidade será alterada para três. O resultado desta abordagem é que, ao final da análise de risco, os controles parcialmente implementados contribuirão com menos risco do que os controles do mesmo tipo classificados como não implementados.

Utilizando a abordagem

Para demonstrar a aplicabilidade da abordagem proposta, foram realizados vários estudos de caso, nos quais o processo utilizado por equipes de desenvolvimento de software de organizações distintas foi avaliado. A seguir, é apresentado um resumo de uma das avaliações realizadas onde se realizou apenas a etapa de Abrangência. Para preservar a confidencialidade, as informações apresentadas foram descaracterizadas (ver **Tabela 2**).

A avaliação considerou somente o perfil de desenvolvedor, portanto o peso das áreas técnicas do modelo será maior que o peso das áreas de gestão ou das áreas organizacionais. Além disso, nesta etapa, os objetivos do negócio da organização não foram identificados e mapeados nos ativos de software.

Abaixo são apresentados os resultados da avaliação:

Conforme apresentado na **Tabela 3** e na **Figura 3**, as áreas de **Planejamento de Projetos**, **Solução Técnica** e **Definição do Processo Organizacional** têm o maior índice de Segurança indicando que a maior parte das práticas destas áreas estão implementadas. Ao contrário, as áreas de **Treinamento Organizacional**, **Gerência de Requisitos**, **Integração do Produto** e **Gerência de Configura-**

Empresa ABC			
Objetivos: Avaliar o processo utilizado no desenvolvimento dos softwares da organização utilizando o modelo CMMI DEV 1.2 como referência e identificar oportunidades de melhoria nas áreas de processo de maior impacto no desenvolvimento dos softwares.			
Fase de Abrangência			
Escopo Organizacional	**Participantes**	**Escopo do Modelo**	**Duração (h/dias)**
Projetos de desenvolvimento de sistemas internos da organização	- 9 Desenvolvedores do Setor 1 - 9 Desenvolvedores do Setor 2	Áreas de níveis 2 e 3 do CMMI DEV 1.2, exceto : Garantia da Qualidade, Gerencia de Fornecedores, Gerência de Risco, Gerenciamento Integrado e Análise de Decisões e Resoluções	16/4

Tabela 2. Resumo do objetivo e escopo da avaliação

Área de Processo	PSR Controles Implementados	PSR Controles Não Implementados	PSR Total	Índice de Conformidade (%)	Índice de Segurança (%)
Avaliação e Melhoria do Processo	64	194	256	32,16	25,00
Definição do Processo Organizacional	160	96	256	85,13	62,50
Desenvolvimento de Requisitos	304	1328	1632	19,16	18,63
Gerência de Configuração	388	1912	2300	25,34	12,52
Gerência de Requisitos	72	1152	1224	0,00	5,88
Integração do Produto	148	1364	1512	15,05	9,79
Medição e Análise	248	264	512	76,18	48,44
Monitoramento e Controle de Projetos	580	780	1360	74,05	42,65
Planejamento de Projetos	1024	200	1224	78,15	83,66
Solução Técnica	2140	852	2992	67,12	71,52
Treinamento Organizacional	8	400	408	0,00	1,96
Validação	108	468	576	22,25	18,75
Verificação	316	1472	1788	18,15	17,67
Total	**5560**	**10482**	**16040**	**67,15**	**53,04**

Tabela 3. Resultados da avaliação em abrangência por área de Processo

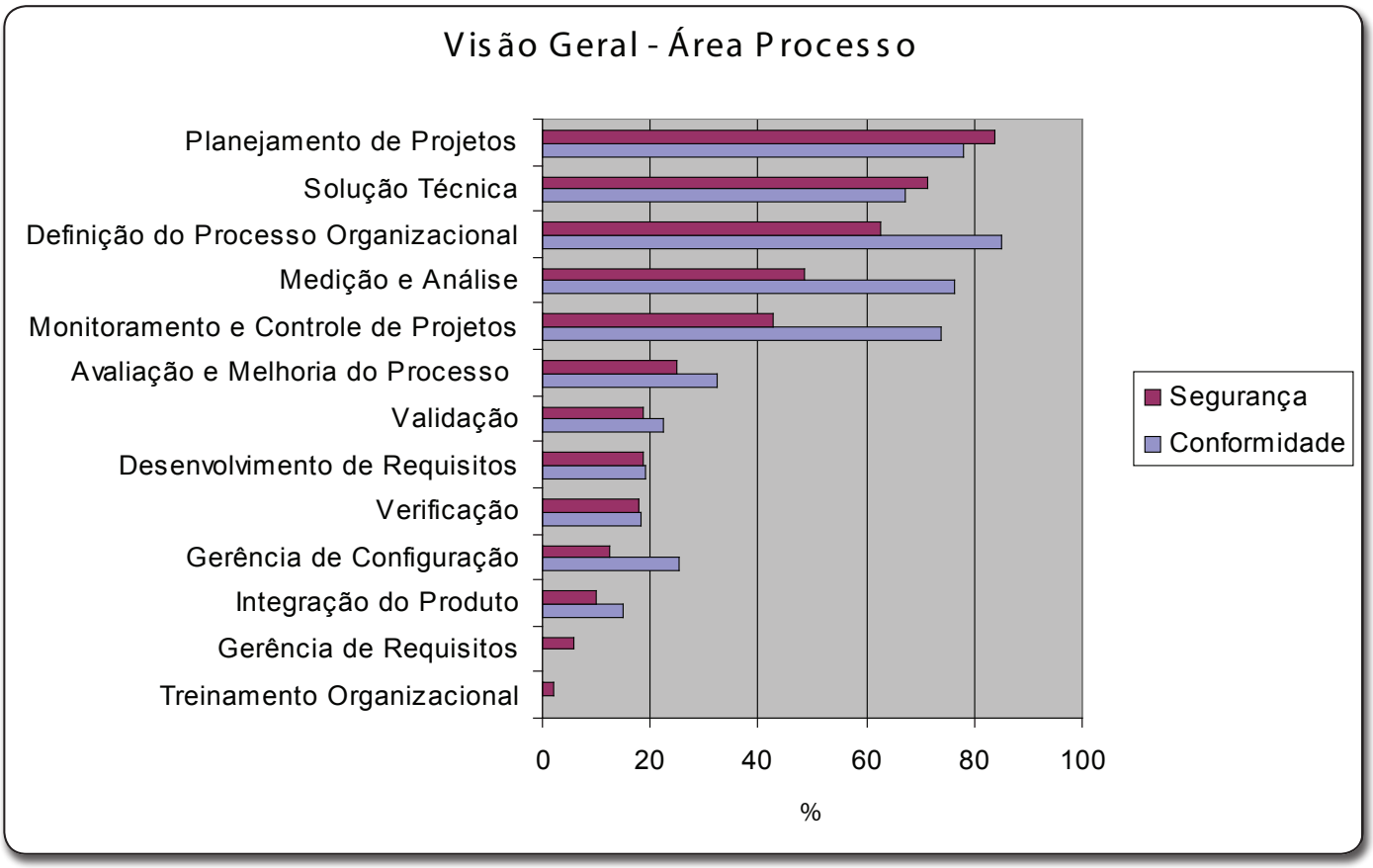


Figura 3. Resultados da avaliação em Abrangência – Gráfico de Índices de conformidade e segurança para cada área de Processo (Ordenação decrescente pelas áreas de maior índice de Segurança)

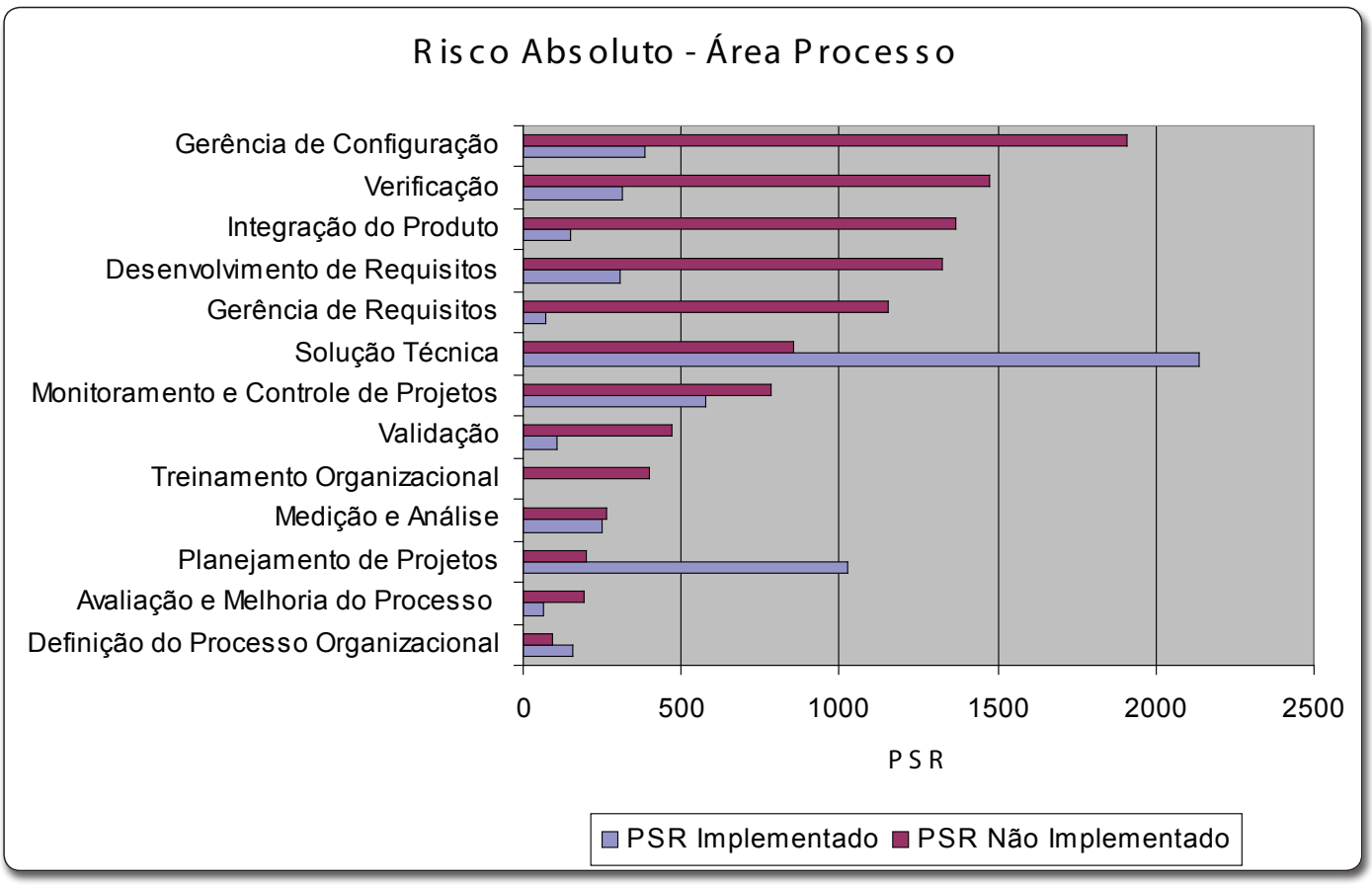


Figura 4. Resultados da avaliação em Abrangência – Gráfico de PSR por área de Processo (Ordenação decrescente pelas áreas de maior PSR)

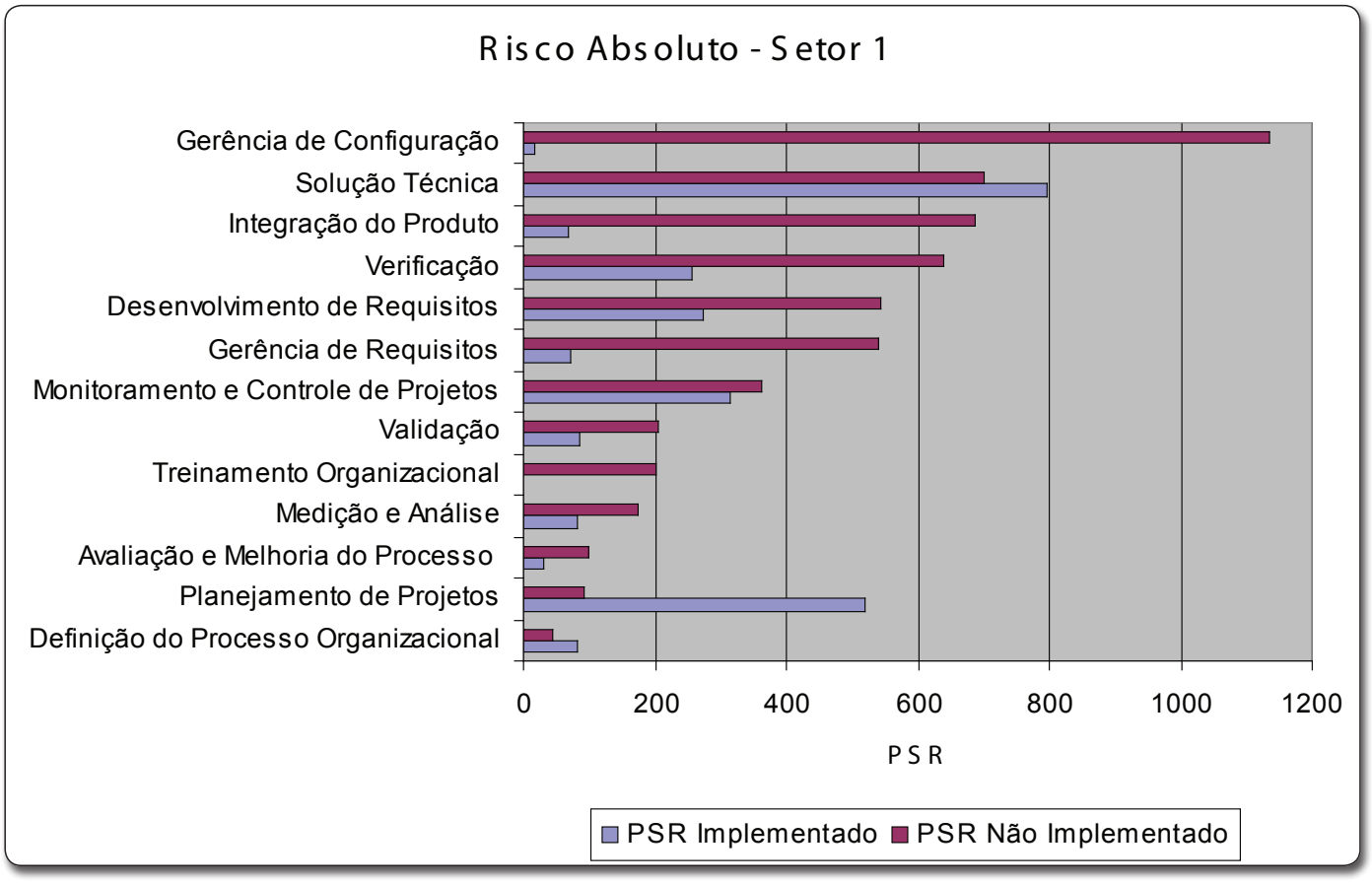


Figura 5. Resultados da avaliação em Abrangência – Gráfico de PSR do Setor 1 por Processo (Ordenação decrescente pelas áreas de maior PSR)

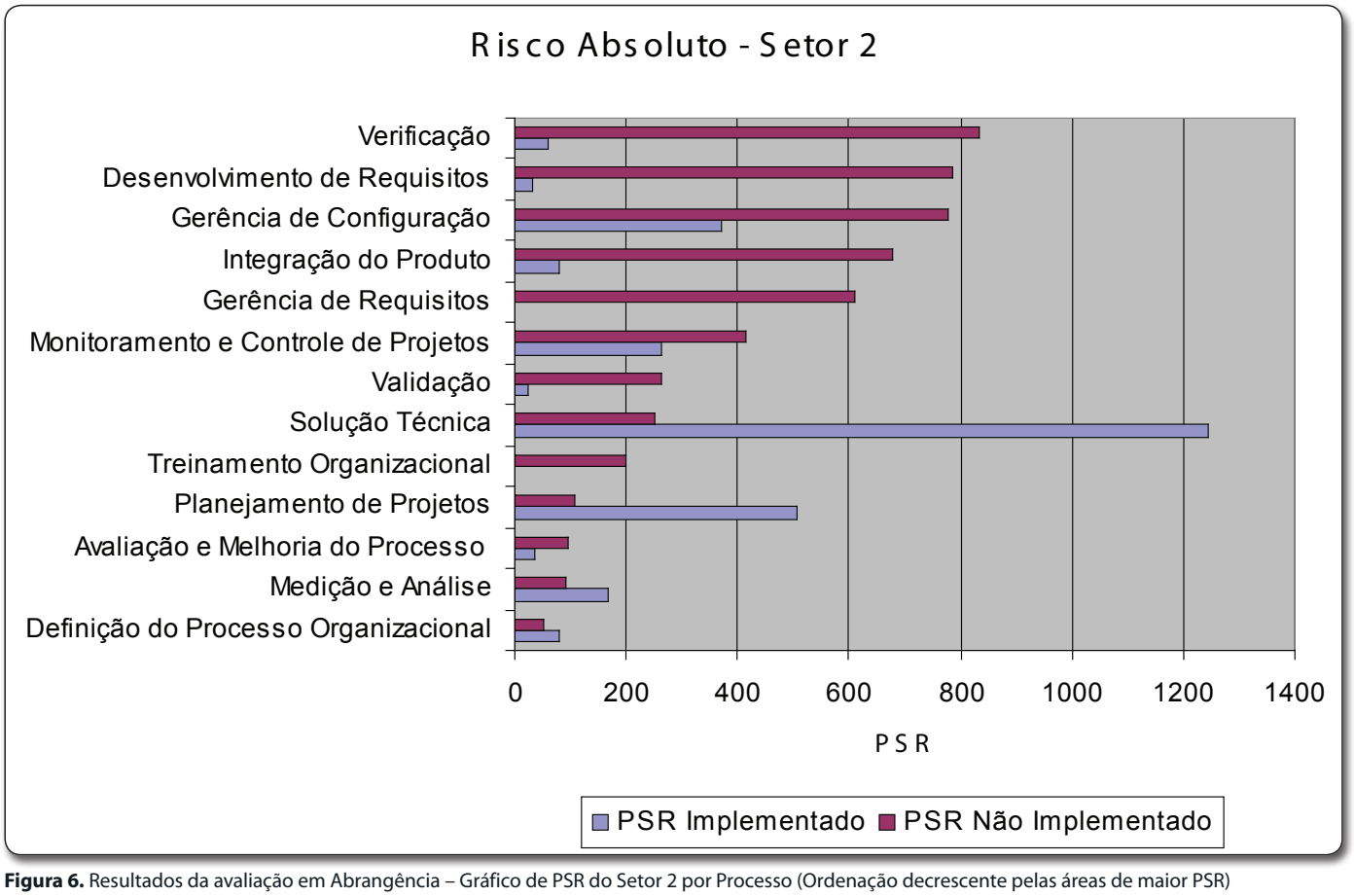


Figura 6. Resultados da avaliação em Abrangência – Gráfico de PSR do Setor 2 por Processo (Ordenação decrescente pelas áreas de maior PSR)

ção têm o menor índice de Segurança indicando que poucas práticas destas áreas estão implementadas. As áreas de **Definição do Processo Organizacional**, **Medição e Análise**, **Monitoramento e Controle de Projetos** e **Gerência de Configuração** têm um índice de Conformidade relativamente maior que o índice de Segurança indicando a implementação de práticas pouco eficientes na mitigação dos riscos.

De acordo com a **Figura 4**, pode-se verificar que as áreas de **Gerência de Configuração**, **Verificação**, **Integração do Produto**, **Desenvolvimento de Requisitos** e **Gerência de Requisitos** têm maior PSR Não implementado indicando maior probabilidade de manifestação dos riscos durante sua execução. Ao contrário, as áreas de **Definição do Processo Organizacional**, **Avaliação e Melhoria do Processo**, **Planejamento de Projetos** e **Medição e Análise** têm menor PSR não implementado, indicando menor probabilidade de manifestação dos riscos durante sua execução.

As áreas de **Solução Técnica**, **Planejamento de Projetos** e **Monitoramento e Controle de Projetos** têm maior PSR implementado, indicando a implementação de práticas que evitaram a manifestação de um maior número de riscos.

Verificando as informações apresentadas na **Tabela 4** e nas **Figuras 5** e **6**, conclui-se que na organização, as áreas de **Gerência de Configuração**, **Verificação**, **Integração do Produto**, **Desenvolvimento de Requisitos** e **Gerência de Requisitos** têm maior probabilidade de manifestação dos riscos durante sua execução. Entretanto, quando os dois setores são avaliados separadamente conclui-se que:

- No Setor 1 as áreas de **Gerência de Configuração**, **Solução Técnica**, **Integração do Produto**, **Verificação**, **Desenvolvimento de Requisitos** e **Gerência de Requisitos** têm maior probabilidade de manifestação dos riscos durante sua execução, ou seja, a área de **Solução Técnica** tem um peso importante nos riscos das atividades do Setor 1;
- No Setor 2, as áreas de **Verificação**, **Desenvolvimento de Requisitos**, **Gerência de Configuração**, **Integração do Produto** e **Gerência de Requisitos** têm maior probabilidade de manifestação dos riscos durante sua execução, ou seja, o mesmo resultado da organização com a inversão de algumas posições.

Pelo apresentado na **Figura 7**, as ameaças mais relevantes estão relacionadas com:

- Produto não atender às necessidades dos clientes - Desenvolvimento de Requisitos, Verificação e Validação;
- Requisitos incompletos, inconsistentes ou incorretos - Gerência e o Desenvolvimento de Requisitos e Verificação;
- Software apresentar falhas – Solução Técnica, Integração do Produto e Verificação;
- Perder controle sobre as solicitações de mudança - Gerência de Requisitos e Gerência de Configuração.

Como resultado final da avaliação, definiu-se um plano priorizando ações de melhoria nas áreas de **Gerência de Configuração**, **Desenvolvimento de Requisitos** e **Gerência de Requisitos**, que apresentaram um alto PSR não implementado, combinado com um baixo índice de segurança. Para um melhor detalhamento das ações de melhoria foi sugerida também a realização de uma avaliação em Profundidade para cada uma das três áreas mencionadas.

Em um segundo momento, foram sugeridas ações de melhoria para as áreas de **Verificação** e **Integração do Produto**, devido ao alto PSR não implementado

	Setor 1			Setor 2		
Área de Processo	**PSR Controles Implementados**	**PSR Controles Não Implementados**	**PSR Total**	**PSR Controles Implementados**	**PSR Controles Não Implementados**	**PSR Total**
Avaliação e Melhoria do Processo	30	98	128	34	94	128
Definição do Processo Organizacional	82	46	128	78	50	128
Desenvolvimento de Requisitos	272	544	816	32	784	816
Gerência de Configuração	16	1134	1150	372	778	1150
Gerência de Requisitos	72	540	612	0	612	612
Integração do Produto	70	686	756	78	678	756
Medição e Análise	82	174	256	166	90	256
Monitoramento e Controle de Projetos	316	364	680	264	416	680
Planejamento de Projetos	518	94	612	506	106	612
Solução Técnica	796	700	1496	1244	252	1496
Treinamento Organizacional	4	200	204	4	200	204
Validação	84	204	288	24	264	288
Verificação	256	638	894	60	834	894
	2598	**5422**	**8020**	**2862**	**5158**	**8020**

Tabela 4. Resultados da avaliação em Abrangência – por Setor e Área de Processo

e o estabelecimento de uma política de **Treinamento Organizacional**, pois apesar do PSR desta área ser menor, o índice de segurança é muito baixo. Da mesma forma foi sugerida uma avaliação em Profundidade para cada uma das três áreas mencionadas.

Foram também sugeridas ações de melhoria para o Setor 1 na área de **Solução Técnica**.

Conclusão

Neste artigo apresentamos uma abordagem que define um critério concreto para priorização de melhorias a serem realizadas nos processos de desenvolvimento de software de uma organização. Este critério utiliza a análise de risco como um instrumento de priorização das ações que têm maior eficiência na mitigação das ameaças identificadas durante o diagnóstico da situação atual. A relevância das ameaças é definida com base no atendimento aos objetivos de negócio da organização e na conformidade com modelos e normas de qualidade de software. ●

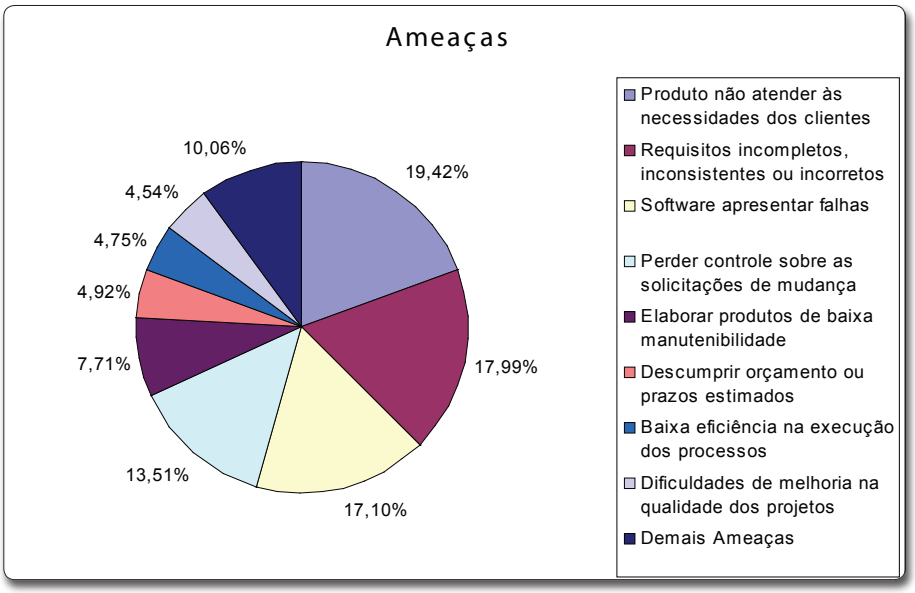
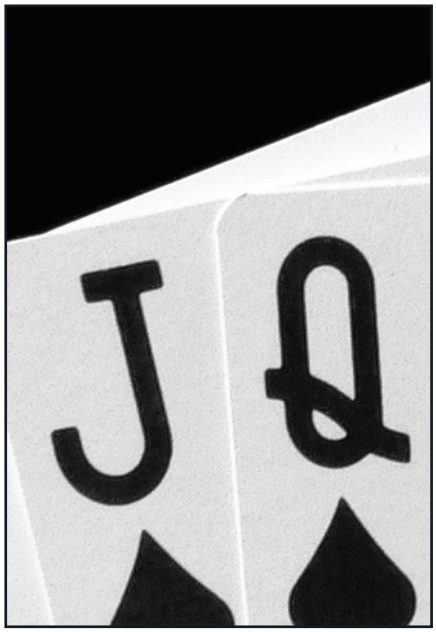


Figura 7. Resultados da avaliação em Abrangência – Ameaças



Referências

BOEHM, B. W. Software Risk Management: Principles and Practices. IEEE Software, v. 8(1), p. 32-41, jan. 1991.

BOEHM, B. W.; DEMARCO, T. Software Risk Management. IEEE Software, v. 14(3), p. 17-19, mai./jun. 1997.

CHRISSIS, M.B.; KONRAD, M.; SHRUM, S. CMMI: Guidelines for Process Integration and Product Improvement. Boston: Addison-Wesley, 2003.

DEMARCO, T; LISTER, T. Waltzing with Bears: Managing Risks on Software Projects. New York: Dorset House Publishing, 2003.

GREMBA, J.;MYERS, C. The IDEALSM Model: A Practical Guide for Improvement. 1997. Disponível em <http://www.sei.cmu.edu/ideal/ideal.bridge.html>. Acesso em 18 nov.2006.

ISO/IEC. Guide 73:2002. Risk management -- Vocabulary -- Guidelines for use in standards, Reference No. ISO/IEC Guide 73:2002(E).

_____. International Standard 12207. Information Technology – Software Life Cycle Processes, Reference No. ISO/IEC 12207: 1995(E): First Edition 1995.

_____. International Standard 15504. Information Technology – Proces Assessment, Reference No. ISO/IEC 15504:2004(E).

_____. International Standard 17799. Information Technology – Security Technics - Code of practice for information security management, Reference No. ISO/IEC 17799:2005(E): Second Edition 2005.

_____. Technical Report 13335 – 1.Information technology - Guidelines for the management of IT Security - Part 1: Concepts and models for IT Security, Reference No. ISO/IEC TR 13335: 1996(E).

IT GOVERNANCE INSTITUTE (ITGI). Control Objectives for Information and Related Technology (COBIT). V4.0, 2005. Disponível em <http://www.itgi.org/>. Acesso em 25 fev 2007.

POULIN, A. Reducing Risk with Software Process Improvement. Boca Raton: Auerbach Publications, 2005.

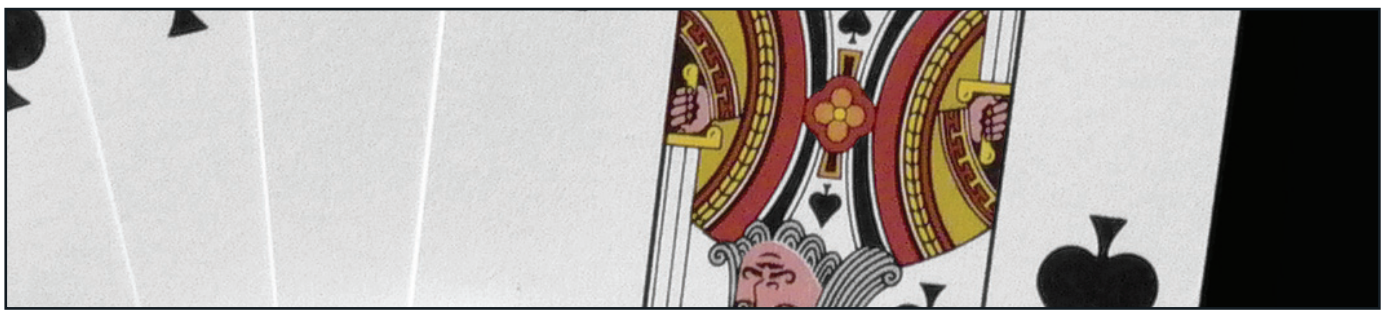
SOFTEX.. MPS.BR – Melhoria de Processo do Software Brasileiro. Guia de Avaliação. Versão 1.0, 2006a.

_____. MPS.BR – Melhoria de Processo do Software Brasileiro. Guia Geral. Versão 1.1, 2006b.

SOFTWARE ENGINEERING INSTITUTE (SEI). Appraisal Requirements for CMMI, Version 1.2 (ARC, V1.2). Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2006a

_____. CMMI for Development, Version 1.2. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2006b.

_____. Standard CMMI Appraisal Method for Process Improvement (SCAMPI[SM]) A, Version 1.2: Method Definition Document. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 2006c



Agilidade ou Controle Operacional? Os dois!

Alexandre Bartie
alexandre_bartie@hotmail.com
Atua há 17 anos no gerenciamento de processos voltados à Qualidade e Engenharia de Software. Pós-Graduado em Capacitação Gerencial pela FEA-USP e em Gestão Empresarial pelo Instituto Trevisan. É Bacharel em Administração de Empresas pela Fundação Santo André. Diretor de Inovação da ALATS (Associação Latino-Americana de Teste de Software) e Engenheiro de Software responsável pelo Framework X-Zone. Autor do Livro Garantia da Qualidade de Software.

Quando os modelos CMMI e RUP foram popularizados, as empresas convenceram-se de que, para gerenciar a crescente complexidade do setor de TI, seria fundamental ampliar o nível do controle operacional sobre todos os projetos existentes.

Apesar de muitas empresas seguirem à risca a cartilha do CMMI, formalizando seus processos, padronizando seus artefatos e reduzindo as flutuações de qualidade e incertezas dos projetos, muitas delas não obtiveram os resultados financeiros desejados. Suas receitas financeiras e lucros operacionais foram reduzidos em função do aumento dos custos e a crescente dilatação dos prazos de entrega.

Infelizmente, as empresas e profissionais deram apenas ênfase ao controle operacional, implementando cada vez mais artefatos, controles e procedimentos internos. Desta forma, os processos tornaram-se mais pesados, provocando uma grande lentidão operacional, sinalizando uma perda contínua de agilidade para seus clientes e o mercado.

O movimento ágil foi o ressurgimento de um ponto de vista que foi esquecido pela TI, quando abraçamos tão fortemente os processos e controles operacionais e negligenciamos a agilidade de nossos projetos.

O modelo ágil orienta empresas, profissionais e metodologistas a pensarem na TI como uma organização veloz, onde o tempo de resposta não pode ser sacrificado. Quanto mais rápida uma empresa, mais entregas são realizadas em um menor espaço de tempo, gerando mais oportunidades de faturamento, o que reverte imediatamente em maior rentabilidade nos negócios.

Porém, a ênfase na agilidade também possui suas restrições - ser rápido leva inevitavelmente a um novo limite operacional. Somente com controles gerenciais bem implementados, poderemos ultrapassar estes limites sem provocarmos um colapso nos serviços da TI.

Desta forma, o grande desafio das organizações do futuro é conseguir conciliar agilidade e controle operacional simultaneamente, possibilitando a área de TI operar com o máximo de qualidade e produtividade.

O objetivo deste artigo é justamente apresentar um modelo que combine **agilidade** e **controle operacional** numa única abordagem, viabilizando o que chamo de **Organizações TI de Alto Desempenho**.

Entendendo o Momento Atual

É crescente o movimento de profissionais e empresas interessadas na adoção dos chamados modelos ágeis, como uma opção aos modelos controlados, liderados mundialmente pelo CMMI e abordagens equivalentes.

Existem inúmeros debates sobre as vantagens e desvantagens dos dois modelos, onde profissionais e especialistas buscam estabelecer critérios que definem em quais contextos uma abordagem deve ou não ser empregada.

Atrás da busca pelo melhor modelo, muitos executivos encontram-se na situação delicada de decidir qual a estratégia a ser seguida pela empresa. Muitos já iniciaram o processo de adequação aos padrões CMMI e agora estas ações passam a ser questionadas por profissionais que defendem as abordagens ágeis.

Para piorar, os executivos deparam-se com uma TI dividida, onde equipes trabalham juntas mais acreditam em abordagens diferentes. Inevitavelmente, o boicote entre as equipes ocorre de uma forma silenciosa, alimentando mais a rivalidade entre aqueles que defendem seus pontos de vista.

Foi neste momento que tomei consciência do período delicado que muitas organizações estão passando. Nunca o momento foi tão favorável a investimentos de TI, porém o excesso de alternativas estão gerando um alto nível de incertezas em nossos executivos, o que provoca uma paralisação nestes investimentos.

Desta forma, elaborei um modelo de gestão de TI que respeitasse as abordagens em andamento, chamado aqui de **Modelo Controlado**, e permitisse uma adoção gradual de novos conceitos que poderiam dar mais velocidade à TI, chamado aqui de **Modelo Ágil**.

Este modelo de gestão baseia-se em indicadores objetivos, não em conceitos ou abordagens de trabalho. Estes indicadores monitoram a agilidade e o controle operacional dos projetos, possibilitando acompanhar o desempenho de todas suas equipes, independente de suas características e restrições.

Antes de abordarmos este modelo de gestão de TI, seria conveniente explorarmos um pouco mais sobre as abordagens controladas e ágeis, para compreendermos melhor como combinar estes elementos nesta proposta.

Na **Tabela 1** é apresentado um pequeno resumo das principais diferenças entres as duas abordagens.

Ressalva Importante

Antes de prosseguirmos com o artigo, é fundamental compreender que as abordagens citadas estão sendo apresentadas de forma resumida, não explorando todos os aspectos, contextos e formatos existentes.

O método de classificação empregado foi pelo critério de prioridades que cada abordagem possui. Modelos ágeis priorizam a agilidade, enquanto que modelos controlados priorizam o controle operacional.

Porém, é interessante ressaltar que abordagens ágeis possuem seus mecanismos de controle, da mesma forma que processos controlados podem ser tão ou mais rápidos que as abordagens ágeis.

Atrás da busca pelo melhor modelo, muitos executivos encontram-se na situação delicada de decidir qual a estratégia a ser seguida pela empresa.

O que podemos aprender com os Modelos Controlados

O propósito inicial do CMM foi estruturar um modelo de avaliação que permitisse estabelecer um nível de risco associado ao grau de maturidade de cada organização. Seu conceito baseia-se no fato de que as organizações que não controlam sua cadeia produtiva possuem maiores chances de fracassarem na condução de seus projetos, pois terão menor controle sobre os fatores que influenciam negativamente a execução dos trabalhos.

Características	Modelo Ágil	Modelo Controlado
Premissa Fundamental	Ênfase na Agilidade	Ênfase no Controle Operacional
Condução dos Trabalhos	Baseado em Processos Empíricos	Baseado em Processos Formais
Escopo da Solução	Centradas no Desenvolvimento	Englobam todas as Disciplinas
Profundidade da Abordagem	Definir apenas o quê deve ser feito	Definir o quê e como deve ser feito
Foco dos Profissionais	Atuação Local (por projeto)	Atuação Global (por disciplina)
Abordagem Estratégica	Atender melhor o Curto Prazo	Atender melhor o Longo Prazo
Palavras Chaves	Pessoas, FeedBack, Adaptação	Maturidade, Estrutura, Padronização
Modelos de Implementação	XP, SCRUM, FDD, APM, Lean, Crystal e DSDM	CMMI, RUP, ITIL, ISO, PMI, MPS.br
Frase que resume sua Filosofia	Aproxime sua equipe do Cliente, simplifique o projeto e aumento sua produtividade	Não podemos melhorar o que não podemos controlar

Tabela 1. Principais diferenças entre Modelos Ágeis e Controlados

A criação do modelo CMM foi gerada a partir da necessidade do Governo dos Estados Unidos. O objetivo foi de acelerar seus processos de terceirização na construção de softwares, sem que ocorresse um descontrole em relação a prazos, custos, escopo e qualidade dos projetos.

Sendo a terceirização uma decisão inevitável, seria necessário estabelecer um modelo que mitigasse os riscos de uma má contratação, através de critérios objetivos que comprovassem a qualificação das empresas.

As empresas que possuem maior nível de maturidade organizacional sinalizam possuir maior controle operacional sobre seus projetos.

Desta forma, todos os fornecedores deveriam ser submetidos ao processo de avaliação CMM, de forma a mensurar seu nível de maturidade organizacional e atestar suas **condições mínimas** para atender determinados projetos em concorrência.

As empresas que possuem maior nível de maturidade organizacional sinalizam possuir maior controle operacional sobre seus projetos e estarão em melhores condições de oferecer seus serviços que as demais. Fornecedores bem avaliados terão acesso a participar de um maior número de projetos, aumentando suas chances de incrementar suas receitas operacionais. Bom para os clientes e melhor ainda para os fornecedores que investiram nos controles operacionais.

Este modelo demonstrou tanto sucesso que foi adotado por várias outras organizações americanas, que basearam seus processos de terceirização de serviços nos níveis de maturidade atestados pela avaliação CMM. Com um modelo estabelecido, a terceirização foi incentivada e intensificada em todo o mundo, tornando extremamente popular o CMM como modelo de avaliação.

As empresas que adotam o modelo controlado buscam estabelecer uma relação duradoura com seus clientes e fornecedores, objetivando sempre ampliar seus controles operacionais e repassar este valor adicional aos clientes. O projeto deixa de contemplar exclusivamente o software que será disponibilizado em produção, mas incorpora todos os elementos gerenciais que permitam manter a evolução do produto ao longo do tempo.

Para as empresas que adotam o modelo controlado, quanto mais controles operacionais forem incorporados, maior será o valor agregado do projeto, proporcionando uma vantagem competitiva em relação às demais organizações.

Atualmente, o CMM foi remodelado para o chamado CMMI. Uma visão mais moderna e ampliada de avaliação das organizações de TI, de modo a incorporar novos temas e oferecer um novo formato de avaliação, baseado nas disciplinas. Isto possibilita que empresas especializadas em determinados serviços de TI (testes de software, gerenciamento de projetos, gerenciamento de requisitos, gerenciamento de ambientes), possam ser avaliados exclusivamente nos itens de seu maior domínio.

Sempre é bom afirmar que o CMM é um modelo de avaliação, pois estabelece apenas os pontos de controle que deverão estar contemplados nos procedimentos de trabalho das empresas. Cada organização tem a liberdade de estabelecer e modelar seus próprios processos internos.

Portanto, apesar do CMM levar inevitavelmente as organizações a trabalharem por processos, seria errado dizer que o CMM é um processo pesado ou burocrático. Na verdade, serão as empresas que determinarão seus processos internos, podendo criar esquemas complicados, pesados, rígidos e burocráticos, não sendo necessariamente culpa do CMM, mas de sua incorreta implementação.

O que podemos aprender com os Modelos Ágeis

Os modelos ágeis fazem parte de um movimento global de profissionais, empresas e instituições, chamado de "**Manifesto Ágil**", que buscam estabelecer os novos paradigmas para as empresas fornecedoras de software, onde a "entrega rápida do software encomendado" é o principal objetivo desta abordagem.

Os modelos ágeis surgem como uma reação natural à expansão do CMMI no mercado mundial, atingindo não apenas as grandes organizações, mas também as pequenas e médias empresas de TI. Dessa forma, estas passam também a conviver com a pressão de organizarem toda sua cadeia produtiva, sob o risco de serem simplesmente excluídos do mercado.

As constantes fusões corporativas favoreceram o surgimento das megacorporações globalizadas, que necessitam de fornecedores que consigam garantir níveis de serviços em escalas nunca antes imaginadas. Sem possuir tantos recursos financeiros disponíveis e prevendo uma dificuldade de adequar-se aos padrões estabelecidos pelo mercado mundial, as pequenas e médias empresas observam suas oportunidades no mercado serem gradativamente reduzidas.

Neste cenário, a abordagem ágil torna-se uma interessante estratégia de sobrevivência das pequenas e médias organizações que não conseguem adequar-se aos rígidos padrões de excelência estabelecidos pelo mercado mundial. Para isto, é necessário convencer o mercado de que seguir os "métodos tradicionais" tornarão suas empresas mais lentas e caras, que a área de TI segue um processo evolutivo diferente de outros setores da economia, e que adotar padrões industriais é um grande equívoco estratégico.

De certa forma, os modelos ágeis ressuscitam a idéia da área de desenvolvimento como o centro da organização TI. Na abordagem ágil, tudo passa a girar em torno dos desenvolvedores, tudo é organizado para dar poder de decisão a estes profissionais que irão encontrar os meios mais adequados para implementar as mudanças no código-fonte e gerar os benefícios esperados no projeto, no menor prazo e custo possível.

As abordagens ágeis chamam atenção de muitas empresas e profissionais pela sua simplicidade de implementação, pois suas regras são simples de serem seguidas. O sucesso da adoção do modelo ágil está intimamente ligado ao comprometimento dos profissionais, pois sua forma de condução dos trabalhos possui um alto grau de informalidade.

A abordagem ágil possui um discurso leve, pautado na redução drástica da complexidade dos projetos de desenvolvimento e de toda a estrutura da TI, reduzindo em poucos elementos gerenciáveis e trabalhando com profissionais com atuação multidisciplinar.

Sua filosofia reforça e incentiva a atuação livre dos profissionais para decidirem como será a melhor forma de implementação, apesar de existirem poucas, mas rígidas regras de controle sobre o projeto. Incentiva a simplicidade do software e uma estrutura mínima para mantê-lo em funcionamento, adicionando flexibilidade e eliminando complexidades ao longo de todo o ciclo de vida do projeto.

Desta maneira, a abordagem ágil torna-se então uma modelo viável de atuação no mercado mundial. Possibilitando criar um novo nicho de oportunidades, atraindo investidores que estarão dispostos a enfrentar os riscos e as condições impostas pelo modelo ágil.

Nos modelos ágeis, o valor de uma organização é medida através de sua capacidade em realizar entregas rápidas a um baixo custo operacional.

Os pontos fracos dos Modelos Ágeis

O modelo ágil possui algumas fraquezas que impedem sua adoção de forma corporativa e podem comprometer sua credibilidade e implementação no mercado mundial.

Falta de uma avaliação para atestar empresas com abordagens ágeis

A abordagem ágil não possui um modelo de avaliação que garanta ao mercado que os projetos serão efetivamente guiados por esta filosofia. Isto significa que qualquer empresa pode pleitear o título de empresa ágil, sem necessariamente seguir esta abordagem em seus projetos.

Mesmo que no futuro exista um modelo de avaliação para empresas ágeis, a informalidade do modelo tornará qualquer avaliação altamente subjetiva, sem quaisquer evidências de que determinadas ações estão realmente acontecendo dentro da filosofia ágil.

As certificações existentes apenas atestam que determinados profissionais estão aptos a aplicar as técnicas estabelecidas pelos modelos ágeis. Porém, não existem garantias que os mesmos as aplicam adequadamente dentro dos projetos das empresas, ou mesmo se esta filosofia é seguida como uma estratégia empresarial ou apenas uma iniciativa isolada de uma equipe dentro da empresa, acentuando ainda mais a fragilidade deste modelo.

Foco excessivo nos prazos e custos inviabiliza o modelo financeiro de longo prazo

Nos **modelos controlados**, o valor da organização é medida pelo total de controles operacionais que uma empresa efetivamente consegue gerenciar em seus projetos, possibilitando que esta seja auditada e facilmente avaliada nestes requisitos. Quanto mais controles forem efetivamente gerenciados, maior será o valor da organização no mercado.

Portanto, as empresas estarão sempre buscando investir mais em controles operacionais para garantir maiores oportunidades de mercado – trata-se de uma estratégia organizacional sustentável de longo prazo.

Nos **modelos ágeis**, o valor de uma organização é medida através de sua capacidade em realizar entregas rápidas a um baixo custo operacional. Para destacarem-se no mercado, as empresas irão concorrer de forma cada vez mais agressiva, oferecendo prazos e custos sucessivamente mais apertados.

No médio e longo prazo, a lucratividade dos projetos será gradualmente reduzida e a pressão no aumento da produtividade das equipes será cada vez maior. Além disso, os salários deverão ser reduzidos para manter uma margem de lucro que compense as organizações seguirem este

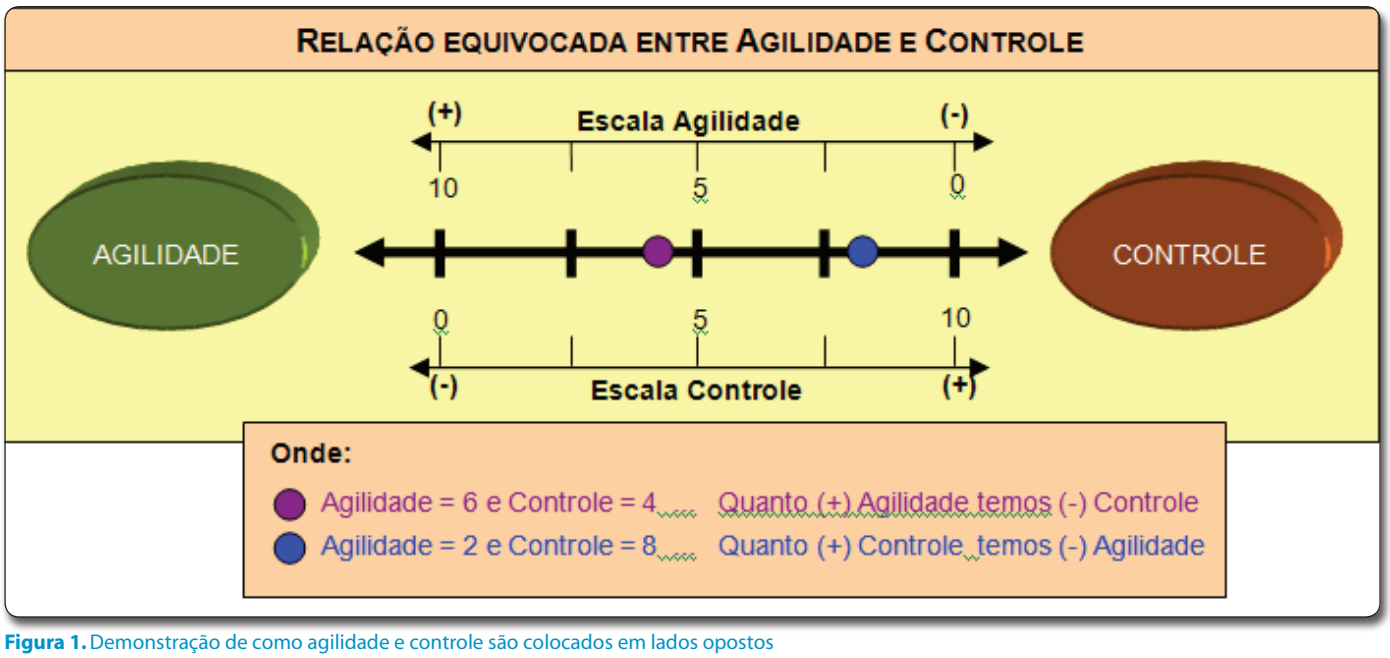


Figura 1. Demonstração de como agilidade e controle são colocados em lados opostos

modelo. Há cada novo projeto, as empresas estarão trabalhando mais próximas de seu limite operacional e financeiro, devendo necessariamente rever sua estratégia de posicionamento no mercado.

Portanto, as abordagens ágeis possuem uma inconsistência estratégica que impedem que suas organizações sejam viáveis financeiramente no longo prazo. Isto não significa que os princípios ágeis não são bons nem interessantes, porém demonstram apenas que, a ênfase exclusiva na agilidade pode levar as organizações a uma estratégia equivocada.

Para a maioria das empresas e profissionais, agilidade e controle operacional são indicadores inversamente proporcionais, estabelecendo um equivocado modelo de causa-efeito.

O Paradigma da Incompatibilidade dos Modelos

Para a maioria das empresas e profissionais, agilidade e controle operacional são indicadores inversamente proporcionais, estabelecendo um equivocado **modelo de causa-efeito.** Isto conduz a uma **decisão binária** entre adotar um modelo ou outro, impossibilitando aplicar de forma combinada estes dois elementos.

Adotando este modelo, estabelecemos que as organizações com maior nível de controle operacional, necessariamente possuem uma limitada capacidade de responder com agilidade as suas demandas de mercado. A razão deste pensamento é que o gerenciamento destes controles consomem tempo e energia dos profissionais, levando uma redução da produtividade (ver **Figura 1**).

Da mesma maneira, uma organização ágil deveria reduzir drasticamente seus processos e concentrar-se exclusivamente na produção do software, obtendo uma agilidade nunca atingida nas organizações com muitos controles operacionais. A razão deste abrupto aumento de produtividade estaria ligada ao fato de que, com o tempo disponibilizado pela ausência dos controles, os profissionais poderiam dedicar-se ainda mais na construção do software (ver **Figura 1**).

Porém, será que este modelo proposto de causa-efeito é realmente consistente? Como podemos atestar que estamos trabalhando com uma premissa verdadeira? Será que estamos condenados a escolher entre uma ou outra abordagem ou existe uma forma diferente de combinarmos estes dois modelos e potencializarmos suas vantagens?

Existe uma premissa errada na área de TI, onde estabelecemos que, para tornarmos ágeis, precisamos abdicar dos controles operacionais para dar maior atenção aos elementos mais essenciais dos projetos de software. Observando atentamente o comportamento dos demais setores da economia, estes dois elementos são empregados intensamente como estratégia em mercados altamente competitivos.

Seja na indústria (com a produção de carros, aviões e navios), na agricultura e pecuária (produção de soja, milho e gado) ou na medicina (procedimentos cirúrgicos, exames e diagnósticos), a produtividade destes setores vem aumentando consideravelmente ao longo dos anos. Ao mesmo tempo, estes produtos e serviços tornaram-se mais sofisticados, exigindo complexos controles operacionais que garantam a qualidade e níveis de excelência cada vez maiores.

Na verdade, estas organizações investiram pesadamente na automação de processos industriais e administrativos, acelerando drasticamente seu modelo de gestão de fornecimento de serviços. Para isso, empregaram todos os modernos recursos tecnológicos disponíveis para aprimorar seus níveis de agilidade e controle operacional.

Organizações que buscaram apenas agilidade, alcançaram rapidamente seu limite operacional. Com o aumento do volume de negócios e a deficiência de seus controles gerenciais, aumentaram a incidência de defeitos em produção e o nível de retrabalho em toda cadeia produtiva, reduzindo sua lucratividade e participação no mercado.

Organizações que buscaram apenas o controle operacional mantiveram níveis de qualidade e excelência compatíveis com as exigências do mercado, mas com prazos de entregas inviáveis para uma economia cada vez mais dinâmica. Foram então surpreendidas por organizações mais ágeis que ofereciam os mesmos controles operacionais, com prazos e custos muito mais agressivos.

Portanto, todos os setores da economia que sobreviveram e prosperaram, conseguiram combinar agilidade e controle operacional em suas operações. Provando que estes dois elementos formam a verdadeira base de sucesso das organizações do futuro.

Combinando Agilidade e Controle Operacional ao Extremo

Para as **Organizações TI de Alto Desempenho**, combinar agilidade e controle operacional é fundamental para a construção de uma empresa altamente competitiva, que atenda as exigências de mercado de curto, médio e longo prazo.

Acreditar que a área de TI deve seguir um modelo diferente dos demais setores econômicos é alienar-se de um movimento global, onde a Tecnologia da Informação é atualmente o grande gargalo operacional de todos os setores da economia. Será através dela, que os saltos de produtividade e qualidade serão conquistados e superados.

O mais interessante da **Abordagem Combinada** (ágil e controlado) é que a pista para a construção deste modelo foi sugerida por Kent Beck no seu livro "Extreme Programing Explained: embrance change" (um excelente livro), porém aplicada em um contexto diferente, resultando na proposta da filosofia XP ("eXtreme Programing"), umas das abordagens ágeis citadas.

Kent Beck mapeou o que considerava ser as melhores práticas para a construção de um software e idealizou um modelo onde estes controles seriam representados por botões individuais, nos quais poderíamos regular sua intensidade de utilização (mínimo e máximo). Então jogou o seguinte desafio – O que aconteceria a uma equipe

de desenvolvimento se girássemos todos os botões de boas práticas ao máximo? – Deste conceito derivamos o que foi estabelecido como modelos extremos, onde as boas práticas de trabalho devem ser levadas ao seu limite, gerando então o máximo da produtividade e desempenho das equipes.

Adotando a mesma abstração proposta por Kent Beck, podemos visualizar um painel onde existam dois botões individuais, representando respectivamente a agilidade e controle operacional que uma empresa deseja obter. Seguindo este modelo, podemos perguntar: **O que aconteceria com a organização se girássemos estes dois botões ao máximo?** (ver **Figura 2**)

Teríamos uma organização TI altamente competitiva, capaz de oferecer produtos e serviços de TI em prazos e custos cada vez mais reduzidos, sem comprometer a qualidade e a confiabilidade de suas entregas.

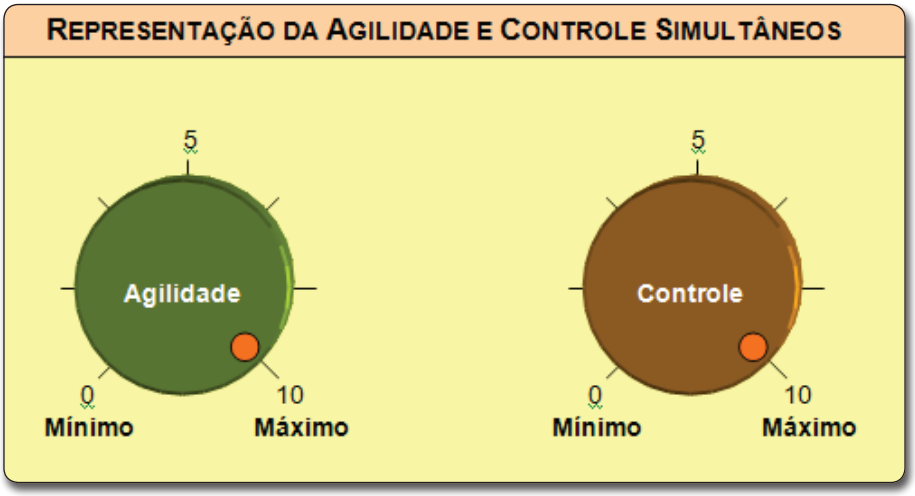


Figura 2. Explorando ao máximo agilidade e controle operacional nos processos TI

Implementando o Modelo Combinado

Com a adoção do **Modelo Combinado**, estaremos alinhando a estratégia da TI a todos os demais setores da economia de mercado, que entendem como necessário serem controlados e ágeis simultaneamente.

A abordagem do **Modelo Combinado** não é tão diferente da abordagem do modelo controlado, baseando-se em processos e suportado por robustos controles operacionais estabelecidos nos moldes do CMMI.

A grande mudança do **Modelo Combinado** é definir como diretriz geral que, qualquer inovação corporativa a ser planejada na cadeia produtiva TI, quando acarretar em perda de agilidade, será **antecipadamente compensada** por uma inovação que recupere ou aprimore o patamar de produtividade das equipes de trabalho.

Isto significa que, as organizações devem implementar medidas que aumentem a produtividade das equipes antes de introduzir novos controles, garantindo um perfeito balanceamento entre os dois indicadores.

Neste momento, as abordagens ágeis trarão consideráveis contribuições para estabelecer mecanismos de resposta mais flexíveis, oferecendo uma oportunidade de **implementação gradual** dos conceitos ágeis no ambiente de TI.

Desta forma, o **Modelo Combinado** requer uma maior consciência de seus profissionais, exigindo um **Plano de Inovação Corporativa** que contemplem inovações que conciliem aumento de produtividade e controle operacional em todos os projetos.

Conclusão

No futuro, novas tendências e filosofias irão continuamente balançar nossas convicções de como devemos conduzir nossos projetos. Sempre seremos questionados por escolher uma determinada direção e abdicarmos das demais alternativas existentes.

Até mesmo experientes executivos sentem-se pressionados em modificar suas estratégias, mesmo quando não se encontram plenamente convencidos dos resultados prometidos pelas inovações.

Mais do que abordagens, precisamos de uma visão de longo prazo, que oriente nossa organização ao futuro, fornecendo elementos gerenciais simples que guiem e avaliem nossas equipes de TI.

Não importa se seguiremos uma linha mais próxima ao CMMI ou se estaremos mais aderentes ao Manifesto Ágil. Quebrar recordes de produtividade e qualidade são os grandes desafios das organizações de TI do futuro.

Portanto, independente do título que daremos aos nossos processos (Ágil, Adaptável, Clássico, Controlado, Formal, Extremo, Burocrático), o que garante que estamos no caminho certo será nossa capacidade de sermos mais ágeis e controlados ao longo do tempo - esta é a proposta do **Modelo Combinado**. ●

O processo unificado integrado ao desenvolvimento Web

Rodrigo S. Prudente de Aquino
rodrigo@wpage.com.br
É bacharel em Ciência da Computação pela PUC-SP e MBA em Engenharia de Software pela USP. Foi analista de sistema na Petrobras e trabalhou como Gerente de Tecnologia Web em uma das maiores agências de marketing direto do Brasil. Escritor de artigos e palestrante em universidades, Rodrigo S. Prudente de Aquino é autor do livro WPage - Padronizando o Desenvolvimento de Web Sites (*www.wpage.com.br*). Atualmente, trabalha como Líder de Projeto no Grupo Totvs.

Com o propósito de auxiliar os fornecedores de soluções de software que utilizam como plataforma a internet, este artigo objetiva formalizar idéias práticas, explicando como o desenvolvimento de sistemas Web pode ser integrado ao Processo Unificado. Serão apresentados alguns artefatos para controlar o desenvolvimento de um Web Site, além das vantagens e os cuidados a tomar com a integração de forma a facilitar a entrega. Serão apresentados também alguns pontos relacionados com a gerência e o plano de execução.

Além disso, explica-se como o Processo Unificado pode ser configurado de acordo com o tempo que uma empresa possui para desenvolver um projeto voltado para internet.

Processo Unificado

O Processo Unificado é um processo de desenvolvimento fortemente ligado à orientação a objetos, porém, pode-se utilizá-lo em qualquer projeto mesmo sendo ele estruturado, sem que perca suas características básicas. Ele utiliza alguns princípios modernos (componentização, revisões, etc) na área de engenharia de software.

Algumas características básicas do Processo Unificado são:

- **Direcionado por casos de uso:** O início do processo deve ser marcado pela utilização dos casos de uso, a fim de se definir uma linguagem entre os usuários e o sistema, facilitando a especificação dos requisitos.
- **Centrado na arquitetura:** O processo procura modelar uma arquitetura através dos aspectos estáticos e dinâmicos de um projeto, que podem ser obtidos junto a um estudo direcionado pelos casos de uso mais significativos.
- **É iterativo e incremental:** Uma das práticas do processo é dividir grandes projetos em mini-projetos. Cada mini-projeto possui uma iteração, que quase sempre abrange todo o fluxo de trabalho. Olhando como um todo, essa iteração

resulta em um incremento para o projeto. É válido lembrar que as iterações são planejadas de acordo com os casos de uso.

O Processo Unificado visa tornar clara a necessidade de atribuições de tarefas a grupos ou indivíduos envolvidos diretamente no desenvolvimento de um projeto. Além disso, deve-se definir o quanto antes, quais as etapas (iterações) e os artefatos que serão envolvidos durante o processo. Com essas características, conclui-se que o Processo Unificado é um modelo configurável, ou seja, deve ser ajustado de acordo com os tipos de projeto que se necessita desenvolver.

A **Figura 1** apresenta a relação entre as fases, iterações e os fluxos de trabalho dentro do Processo Unificado.

Concepção ou iniciação: Essa fase tem como objetivo verificar a viabilidade do projeto, bem como os riscos e um dos fatores não menos importantes: definir os casos de uso mais críticos obtendo as funções chave do sistema. É através do tipo do projeto, dos casos de uso e consequentemente dos requisitos, que se realizará o ajuste de quantas iterações o processo terá. De acordo com os casos de uso, pode-se definir também quais as etapas exigirão maior cuidado.

Elaboração: Durante essa fase, a maioria dos casos de uso são especificados e detalhados. A arquitetura do sistema é projetada utilizando artefatos que podem ser estáticos ou dinâmicos. Neste instante são apresentados, o Baseline completo do projeto, os componentes que formarão a equipe de desenvolvimento, etc. No final dessa fase os envolvidos devem estar aptos a planejar a fase de construção em detalhes.

Construção: A fusão de vários artefatos de software ocorre neste momento, possibilitando que o sistema seja implementado quase que completamente. Tem-se uma visão geral de como o Baseline do projeto está sendo seguido. No final dessa fase, o sistema deve estar totalmente preparado para a transição ao usuário.

Transição: O objetivo dessa fase é garantir que todos os requisitos do projeto foram atendidos e implementados corretamente. O produto final pode ser liberado em uma versão beta. Existem ainda outras atividades que, de acordo com o projeto, podem ocorrer de maneira paralela, por exemplo, a preparação do ambiente, a conclusão do manual do usuário, identificação e correção de defeitos. No final dessa fase deve-se tirar uma conclusão geral do projeto, obtendo os pontos positivos e negativos os quais devem ser utilizados durante a concepção de projetos futuros.

Em relação aos fluxos de trabalho, ou disciplinas, tem-se os seguintes esclarecimentos.

Modelo do negócio: O objetivo principal desse fluxo é que o fornecedor entenda muito bem o problema a ser resolvido, elaborando se necessário uma análise de risco e de viabilidade para o projeto como um todo. Neste momento, existe uma grande interação entre o fornecedor e o cliente. A fim de que possam ser gerados os casos de uso e consequentemente a extração dos requisitos. Entender o modelo de negócio do cliente é peça fundamental antes que um requisito possa ser definido.

Requisitos: Nesse fluxo procura-se extrair os requisitos do sistema a ser desenvolvido. A grande dificuldade nesta etapa e no desenvolvimento de software é capturar requisitos de forma que os clientes possam entender claramente o que o sistema se propõe a fazer. A base para isso é que o fornecedor entenda o domínio do problema e consequentemente construa um bom modelo de casos de uso. A extração dos requisitos, através dos casos de uso, irá compor um artefato que será evoluído durante todo o projeto.

Análise e Projeto: No início desse fluxo de trabalho, desenvolve-se uma visão "arquitetural", incluindo os artefatos significativos para o modelo de projeto. O objetivo aqui é compreender os casos de uso mais importantes, que serão insumos para a elaboração de alguns artefatos, como: um diagrama de classes, de estado, de iteração, de seqüência, de colaboração, etc. É válido lembrar que não é necessária a utilização de todos os artefatos, mas apenas aqueles que sejam relevantes a fim de que o cliente entenda perfeitamente o que será construído. Com artefatos bem elaborados, a equipe de desenvolvimento terá grandes facilidades em realizar a implementação. No início deste fluxo encontra-se, caso necessário, protótipos de funcionalidade e de interface, como também uma descrição da arquitetura básica do sistema. Durante o desenvolvimento do projeto alguns artefatos poderão sofrer ajustes de acordo com as implementações realizadas.

Implementação: No início desse fluxo, os desenvolvedores poderão buscar componentes (funções) que foram utilizados em outro sistema. Ainda na fase de concepção, pode-se ter um protótipo de funcionalidade como um produto final em primeira instância. No decorrer deste fluxo, procura-se ter um sistema executável a cada iteração, além da implementação baseada nos artefatos criados no modelo de análise e projeto. O conceito de componentização deve ser sempre levado em consideração, com o

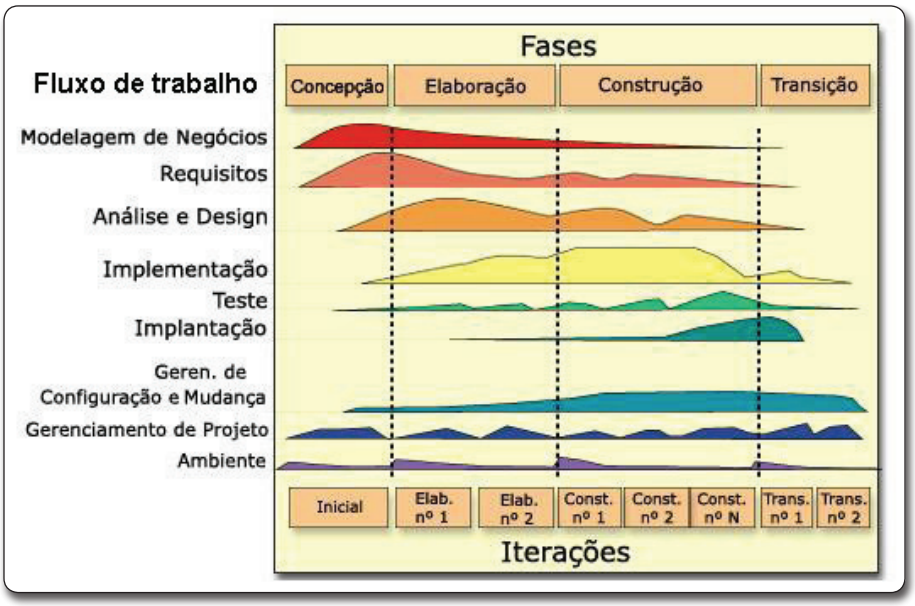


Figura 1. Overview do Processo Unificado

intuito de que estes segmentos de códigos possam ser aproveitados mais tarde por outros sistemas.

Testes: Neste fluxo, um plano de teste deve ser elaborado, definindo e identificando qual procedimento e quais tipos de testes serão realizados. Esse plano poderá ser alterado de acordo com a melhor definição dos requisitos do sistema. Ele também poderá ser utilizado durante todo o projeto, sendo modificado a cada iteração, mostrando a situação do executável que foi entregue ao cliente. Nas fases de concepção e de elaboração têm-se os testes de módulos e na fase de construção têm-se os testes de integração. O número de testes de integração poderá se repetir de acordo com a quantidade de alterações nos requisitos do sistema.

Implantação: Descreve-se nesse fluxo de trabalho, a instalação do sistema no ambiente do cliente. Durante toda a fase de elaboração, até o meio da fase de construção, um simples documento especificando algumas características do ambiente do cliente poderá ser realizado. Este artefato pode conter, por exemplo, especificações técnicas sobre a infraestrutura de rede e de sistemas suportada pela empresa contratante. Além disso, algumas dicas de instalação podem ser acrescentadas nesse artefato de forma a reduzir mais tarde, o número de erros de instalação e consequentemente o tempo de testes. No final da fase de construção, inicia-se a migração do sistema para o ambiente de testes do cliente. Posteriormente, no final da fase de transição, já se pode observar a completa migração e configuração do sistema no ambiente de produção do cliente.

Gerência de configuração e mudança: É durante esse fluxo de trabalho que são controlados todos os artefatos do projeto, bem como suas versões. Antes de realizar uma mudança, deve-se fazer uma análise em relação ao que deve ser modificado e saber em quais artefatos e áreas da implementação isso irá afetar. Um bom controle de mudança é crucial para garantir o sucesso e a qualidade do projeto. À medida que o projeto entra na fase de construção, a dificuldade no controle de mudança e gerência de configuração aumenta. Isso ocorre porque o projeto está maior, com mais requisitos implementados e com maiores chances de que uma alteração possa afetar outras áreas do sistema. Ter rastreabilidade e saber relacionar os requisitos é uma tarefa importante do engenheiro de software. Após uma modificação, necessita-se de novos testes em várias áreas do sistema, garantindo que a mudança foi implementada corretamente. Não menos importante, a alteração da documentação deve estar completamente condizente com o que foi implementado.

Não menos importante, a alteração da documentação deve estar completamente condizente com o que foi implementado.

Gerenciamento de projeto: Nesse fluxo se escolhe os artefatos a serem utilizados no desenvolvimento da aplicação, de acordo com o tipo do projeto e o entendimento do cliente. O gerente deve ter uma visão clara do que o cliente deseja, do que está documentado e do que está sendo implementado. A atividade de gerenciamento de projeto é constante durante todo o ciclo de vida do software, elaborando reuniões com RTF (Revisão Técnica Formal), garantindo a correta mudança dos artefatos, além da necessidade de manter um bom relacionamento com o cliente.

Ambiente: Esse fluxo representa o ambiente de trabalho da empresa que desenvolverá o projeto. Ele pode ser caracterizado pelo tipo de plataforma, pela rede, pela organização dos diretórios no qual ficarão os artefatos e os códigos fonte, pelo sistema de backup etc. Pode-se perceber na **Figura 1** que no final de cada iteração, têm-se ajustes no ambiente. Esses ajustes podem ser do tipo: criação de diretórios, o backup das versões do software, etc.

As iterações, nada mais são do que marcos durante a construção de um sistema utilizando o Processo Unificado. Um aspecto muito importante é que o número de iterações deve ser definido logo no início de cada projeto (elas podem variar de número de acordo com o tamanho do sistema a ser desenvolvido). Uma iteração normalmente é marcada pela entrega de uma versão executável do sistema e uma reunião formalizada através de uma RTF (Revisão Técnica Formal). Em geral, o resultado de uma iteração é um incremento para o sistema. Entende-se também que uma iteração é como se fosse uma "foto" tirada da aplicação num determinado instante. É um marco indicando o final de um mini-projeto.

Artefatos específicos utilizados no desenvolvimento de projetos Web

Durante a construção de aplicações web pode-se utilizar inúmeros tipos de artefatos. Serão citados a seguir, alguns documentos que poderão ser utilizados no Processo Unificado.

Código	Requisito	Categoria	Prioridade	Dificuldade	Atendido	Comentários
Req0001	O sistema deve permitir receber e-mails de dúvidas dos usuários	Estável	Média	Baixa	Não	
Req0002	O sistema deve avisar o usuário de suas horas conectadas na web	Estável	Alta	Média	Não	
Req0003	O sistema deve validar o e-mail de entrada do usuário	Derivado - Req001	Média	Baixa	Sim	
Req0004	O sistema deve possuir uma área de produtos	Estável	Alta	Alta	Não	
Req0005	O sistema deve mostrar informações das contas dos usuários	Estável	Alta	Média	Sim	

Figura 2. Planilha de requisitos

Planilha de requisitos

Para elaborar um sistema Web, é necessário um levantamento dos requisitos. Neste contexto, precisa-se de um artefato para armazenar estas informações. Utilizaremos neste artigo uma planilha Excel como artefato, de forma a explicar simplificadamente a organização dos requisitos. A planilha Excel terá as características representadas na **Figura 2**.

Nesta planilha temos:

- **Código:** Identifica unicamente um requisito a fim de que se possa controlá-lo através do projeto.
- **Descrição:** Nesta coluna descreve-se o requisito.
- **Categoria:** Indica qual é o tipo do requisito (ver **Tabela 1)**.
- **Prioridade:** Indica o nível de importância que o requisito possui para o sistema em geral, podendo ser baixa, média ou alta.
- **Dificuldade:** Indica o nível de dificuldade para implementar este requisito, podendo ser baixa, média ou alta.
- **Atendido:** Representa o status do requisito, indicando se o mesmo foi ou não implementado no sistema.
- **Comentários:** Fornece informações sobre o requisito, dizendo, por exemplo, o motivo que um determinado requisito ainda não foi implementado (indicando mais especificamente, quais são as dificuldades).

A planilha de requisitos é um artefato "vivo" no ciclo de vida do projeto e deve ser incorporado à área de SCM (Software Configured Management) do Processo Unificado. A expressão artefato "vivo" indica que a planilha está apta a sofrer alterações no decorrer do projeto.

Projeto Linear

Além da planilha de requisitos, esse é um dos artefatos mais importantes para o desenvolvimento de um sistema Web. Nele poderão ser mapeados os requisitos do sistema com as áreas ou páginas de uma aplicação. Cada página receberá um código, que por sua vez será relacionado com nenhum, um ou mais requisitos.

Através deste documento busca-se um maior controle do sistema, pois se houver quaisquer modificações nos requisitos o fornecedor saberá quais áreas devem sofrer mudança. Este também é um artefato "vivo" e deve ser incorporado ao fluxo de trabalho de gerência de configuração e mudança (SCM - Software Configured Management). A **Figura 3** apresenta um exemplo de como seria uma simples representação de um Projeto Linear, mostrando algumas áreas do site, com seus respectivos requisitos relacionados.

Web Content

O Web Content é um artefato de software responsável pelo armazenamento de todo o conteúdo textual utilizado em um site. Não existe um documento

Requisitos Funcionais	
Requisitos Estáveis: São aqueles que derivam da atividade fim da organização e são relativos diretamente ao domínio do sistema.	
Requisitos Voláteis: São requisitos que mudam ao longo do desenvolvimento ou após o início da operação. Dentro dele, existem:	
	Requisitos Mutáveis: São requisitos que se alteram em razão das mudanças no ambiente no qual está operando.
	Requisitos Emergentes: São requisitos que não podem ser completamente definidos quando o sistema está em desenvolvimento.
	Requisitos Conseqüentes: São requisitos baseados em premissas de como o sistema será usado. Quando o sistema é colocado em operação, ocorrem mudanças.
Requisitos Não Funcionais	
Requisitos de Produto: São aqueles específicos do comportamento do produto. Dentro dele, existem:	
	Requisitos de usabilidade
	Requisitos de eficiência
	Requisitos de disponibilidade
	Requisitos de portabilidade
	Requisitos de confiabilidade
Requisitos Organizacionais: São aqueles derivados de políticas e procedimentos organizacionais do cliente e dos desenvolvedores. Existem os seguintes tipos de requisitos organizacionais:	
	Requisitos de versão: definindo o produto e quais os documentos são necessários para liberar uma versão para o usuário.
	Requisitos de implementação: envolve linguagens de programação, banco de dados, etc.
	Requisitos de padrões: envolve os padrões a serem usados.
Requisitos Externos: São aqueles derivados de fatores externos ao sistema e ao seu processo de desenvolvimento.	
	Requisitos de interoperabilidade: definição de como o sistema interage com outros sistemas.
	Requisitos étnicos: assegurando que o sistema será aceito pelos usuários e pelo público em geral.
	Requisitos de legislação: devem ser seguidos para assegurar que o sistema vai operar de acordo com normas vigentes. Pode ser dividido em: requisito de privacidade e requisito de segurança.

Tabela 1. Tipos de requisitos

padrão de Web Content. Normalmente cada empresa que desenvolve aplicações web possui o seu.

O Web Content é formado de acordo com os requisitos do sistema e entende-se que o mesmo pertence ao fluxo SCM (Software Configured Management) do Processo Unificado. Na **Figura 4** exemplifica-se como seria uma página de um Web Content.

É muito importante lembrar que esse artefato é formado não só de uma, mas várias seções, onde cada uma indica o conteúdo de cada página do site. Com o Web Content, o fornecedor consegue agrupar e gerenciar melhor o conteúdo de um site.

FDD (Wireframes)

O FDD (Functional Design Document) é um conjunto de Wireframes onde cada um representa uma página da aplicação. Um Wireframe é uma maquete da página Web que se dirige somente à disposição de elementos, não à estética. Ele é o esboço de como seria uma página, desprezando cores e imagens. A vantagem em utilizar um Wireframe como guia para implementação, é que ele trabalha representando o fluxo da informação estabelecido anteriormente no Projeto Linear. Ele pode ser desenvolvido pelo arquiteto de informação.

O uso de um FDD estabelece uma forte ligação da arquitetura da informação com a estrutura do site, colocando a informação no seu respectivo local. Além disso, um FDD bem organizado pode oferecer fortes soluções para os problemas de usabilidade. Outra característica importante deste artefato é que ele pode informar onde encontrar o conteúdo para aquela respectiva página dentro do Web Content.

A desvantagem do FDD é que ele não apresenta uma solução gráfica para o projeto, apesar de ter um papel muito importante em conduzir a proposta de layout a ser construída pelo designer. Em relação ao desenvolvimento de um Web Site, o FDD torna-se um dos artefatos mais completos, que auxiliam muito os programadores, pois eles criam uma relação entre a página a ser implementada e o conteúdo a ser aplicado. A **Figura 5** apresenta como seria um Wireframe dentro de um FDD - representando uma determinada página de um site.

Projeto Linear

Código	Áreas do site	Requisitos
1	Área de pré cadastro	Req0001, Req0002, Req0007
1.1	Formulário	Req0031
1.1.1	Mensagem de erro	Req0011, Req0012, Req0013
1.1.2	Mensagem de sucesso	Req0051, Req0032
2	Home page personalizada	Req0011, Req0042, Req0047, Req0048, Req0057, Req0017, Req0027
2.1	Login	Req0011, Req0042, Req0047
2.1.1	Mensagem de erro	Req0023, Req0028, Req0029
3	Esqueci minha senha	Req0068
3.1	Formulário	Req0011, Req0055, Req0098, Req0048, Req0057, Req0017, Req0027
3.1.1	Mensagem de erro	Req0033, Req0039, Req0050
3.1.2	Mensagem de sucesso	Req0051, Req0032

Figura 3. Projeto Linear e requisitos

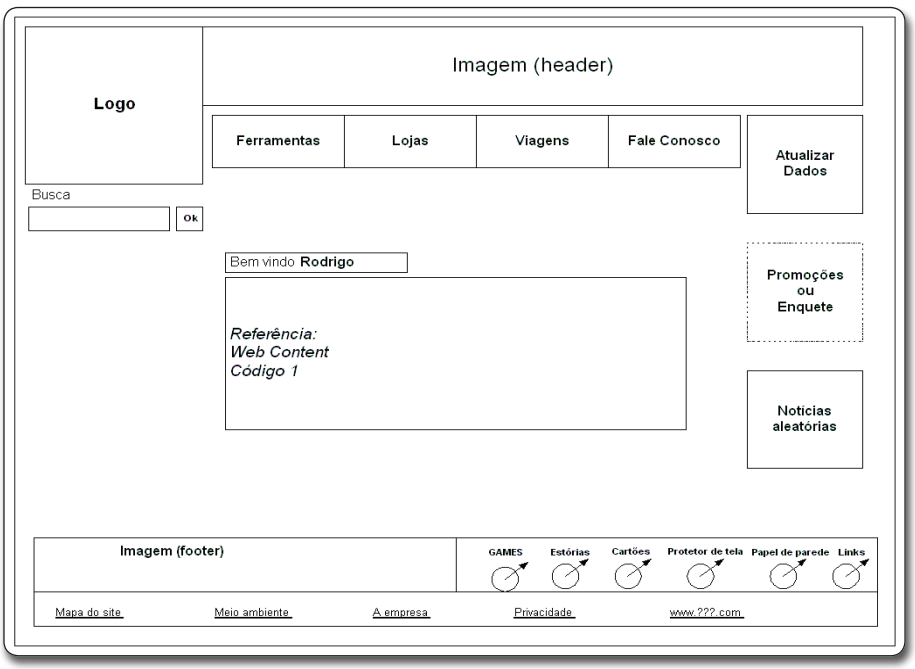


Figura 5. Wireframe (página do FDD)

Protótipo de interface

O protótipo pode ser uma parte da aplicação implementada, protótipo de funcionalidade, ou uma proposta de layout, protótipo de interface, feita pelo designer e aprovada pelo cliente. Este item fornece algumas informações apenas sobre o protótipo de interface. Para chegar até o protótipo, o designer precisa utilizar o FDD ou pelo menos uma parte dele para ter noções de como será a divisão do site. A principal função deste artefato é fornecer ao cliente quais serão as cores básicas

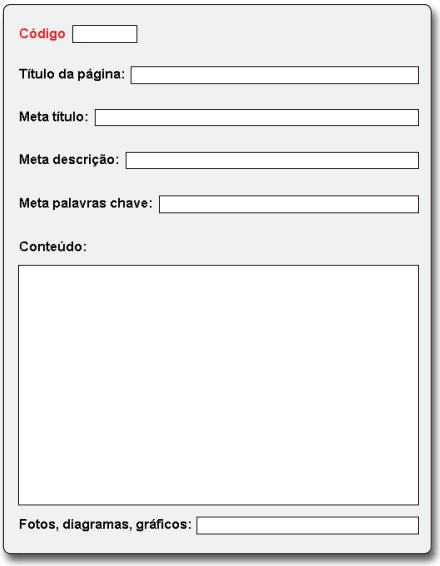


Figura 4. Ilustração de uma página do Web Content

da aplicação, uma parte da arquitetura de informação e como ficarão disponibilizadas as informações aos usuários dentro do site. A vantagem na utilização deste artefato é direcionar totalmente a equipe de análise e projeto, bem como a equipe de implementação.

O Processo Unificado integrado ao desenvolvimento Web

Neste item, explica-se como configurar o Processo Unificado de acordo com o sistema a ser desenvolvido, obtendo uma determinada quantidade de iterações. Além disso, descreve-se o esforço gasto para construir cada artefato Web em razão das fases do processo.

Configurando o Processo Unificado

Antes de iniciar o desenvolvimento de qualquer projeto utilizando o Processo Unificado, é necessário determinar os fluxos de trabalho mais utilizados, o número e o tempo de cada iteração dentro das fases. Para um sistema Web, normalmente consideram-se todos os fluxos de trabalho do processo, ou seja, modelo de negócios, requisitos, análise e projeto, implementação, teste e implantação. Com o objetivo de esclarecer melhor a configuração do Processo Unificado imagina-se, para este artigo, o desenvolvimento de um Web Site contendo um prazo de três meses.

Ao ser definido o prazo de entrega, o processo começa a ser modelado à medida que o Baseline é construído. Considerando que o fornecedor tenha conhecimento da visão de negócio do cliente e do sistema a ser desenvolvido como um todo, pode-se, por exemplo, dividir as fases do projeto conforme apresentado na **Tabela 2**.

É importante destacar que a **Tabela 2** é apenas um exemplo baseado na ilustração do Processo Unificado, presente no segundo tópico deste artigo. A quantidade de tempo e iterações que um sistema terá irá variar muito em razão do tipo do projeto, do número de profissionais envolvidos, do prazo de entrega, das funcionalidades, etc. O tempo de experiência da equipe de desenvolvimento é um fator importante, de forma a identificar e combater os pontos críticos durante a implementação da aplicação.

A **Tabela 3** apresenta uma aproximação da quantidade de esforço gasto em cada artefato de software voltado para Web, relacionando-os ao Processo Unificado.

Fases	Tempo	Iterações
Concepção	1 semana	1
Elaboração	2,5 semanas	1
Construção	6 semanas	2
Transição	2,5 semanas	1

Tabela 2. Divisão das fases do projeto

Artefatos	Concepção	Elaboração	Construção	Transição
Planilha de requisitos	30%	50%	15%	5%
Projeto Linear	20%	70%	10%	0%
Web Content	15%	70%	15%	0%
FDD (Wireframes)	10%	60%	30%	0%
Protótipo de Interface	100%	0%	0%	0%

Tabela 3. Mensuração de esforço X fases

Relacionando artefatos, fases do processo e fluxos de trabalho

Descreve-se detalhadamente neste item, o que deve ser feito em todos os fluxos de trabalho, através do número de iterações definidas na configuração do processo unificado.

Fase Concepção - 1ª. Iteração

A 1ª iteração ocorre praticamente depois de toda a fase de concepção do projeto, tendo como referência a **Figura 1**. No final dessa iteração, deixa-se claro quais os artefatos farão parte da gerência de configuração e mudança, ou seja, aqueles artefatos que ainda sofrerão algum tipo de alteração no decorrer do desenvolvimento do projeto.

Modelo de negócio: Neste fluxo, se o fornecedor achar necessário, pode-se realizar um documento indicando a análise de viabilidade e de risco do projeto. Caso esteja difícil para o fornecedor entender o domínio do problema, deve-se elaborar um diagrama de casos de uso do negócio, bem como sua descrição, ficando mais fácil chegar ao domínio da solução. Os casos de uso do negócio darão suporte ao diagrama e a descrição de casos de uso do sistema. Nesse fluxo, o Baseline do projeto começa a ser construído, contemplando custos, prazos, cargos e número de pessoas envolvidas. É válido destacar que às vezes, o Baseline pode ser modificado de acordo com as iterações do processo. Um cliente pró-ativo em resolver as dúvidas do fornecedor, consegue diminuir as chances de impactar o desenvolvimento de algumas partes do projeto, como também uma possível alteração no Baseline.

" Ao ser definido o prazo de entrega, o processo começa a ser modelado à medida que o Baseline é construído. "

Requisitos: A extração dos requisitos deve ser feita à medida que os casos de uso do sistema são realizados e validados. Conclui-se que a montagem da planilha de requisitos aumenta à medida que os casos de uso são aprovados pelo cliente. Mesmo sem a arquitetura de informação do site, o próprio cliente, por exemplo, já pode obter informações sobre o que deseja, em relação ao conteúdo que será apresentado em sua aplicação.

Análise e Projeto: Este fluxo utilizará até o momento, todos os requisitos construídos e aprovados dentro da planilha. A arquitetura de informação descrita no Projeto Linear já pode ser montada se baseando nos requisitos. É no Projeto Linear que serão mapeados os códigos de cada página do site, a descrição da área e a identificação dos requisitos. Note que uma página do site poderá estar relacionada a nenhum, um ou a vários requisitos. Com este artefato garante-se mais tarde a rastreabilidade. A estru-

turação quase que definitiva do Web Content pode ser feita pelo fornecedor e pelo cliente, ainda neste fluxo, no final da 1ª iteração.

Mediante a construção do Projeto Linear e do Web Content, inicia-se a montagem FDD. O FDD servirá como guia para o designer montar o protótipo de interface do site. A finalização do protótipo e a aprovação do cliente marcam o final da 1ª iteração.

Implementação: Nesse momento os desenvolvedores podem, por exemplo, buscar funções e componentes já desenvolvidos em outros projetos, os quais servirão para a realização desta aplicação. A preparação do ambiente de desenvolvimento também pode ser feita, como a instalação dos softwares e ferramentas necessárias para a implementação. Neste fluxo, dá-se a construção do diagrama de classes, bem como outros diagramas UML que os engenheiros de software acharem necessários, para o entendimento e validação do sistema pelo cliente. Objetivando diminuir as chances de algum requisito ser implementado de forma incorreta, é válido que alguns desenvolvedores participem de reuniões com os clientes, tirando suas dúvidas, como também revalidando os requisitos.

Teste: Esse fluxo pode ser marcado com o início da construção de um artefato chamado plano de teste. Essa construção deverá ser direcionada pelos requisitos do sistema obtidos até o momento.

Implantação: Não há.

A **Figura 6** apresenta um overview da construção do sistema, levando-se em consideração os artefatos utilizados no desenvolvimento de projetos Web. Conclui-se que o Web Content (representado pelo círculo vermelho), o Projeto Linear (representado pelo círculo preto) e o FDD (representado pelo círculo azul) estão em fase de formação, por isso eles estão tracejados. A área com cor cinza claro da **Figura 6** representa que poucos requisitos foram encontrados nesta iteração do processo. Os círculos em amarelo em volta do FDD e do Projeto Linear representam que estes documentos necessitam de um conhecimento em arquitetura de informação para que possam ser elaborados.

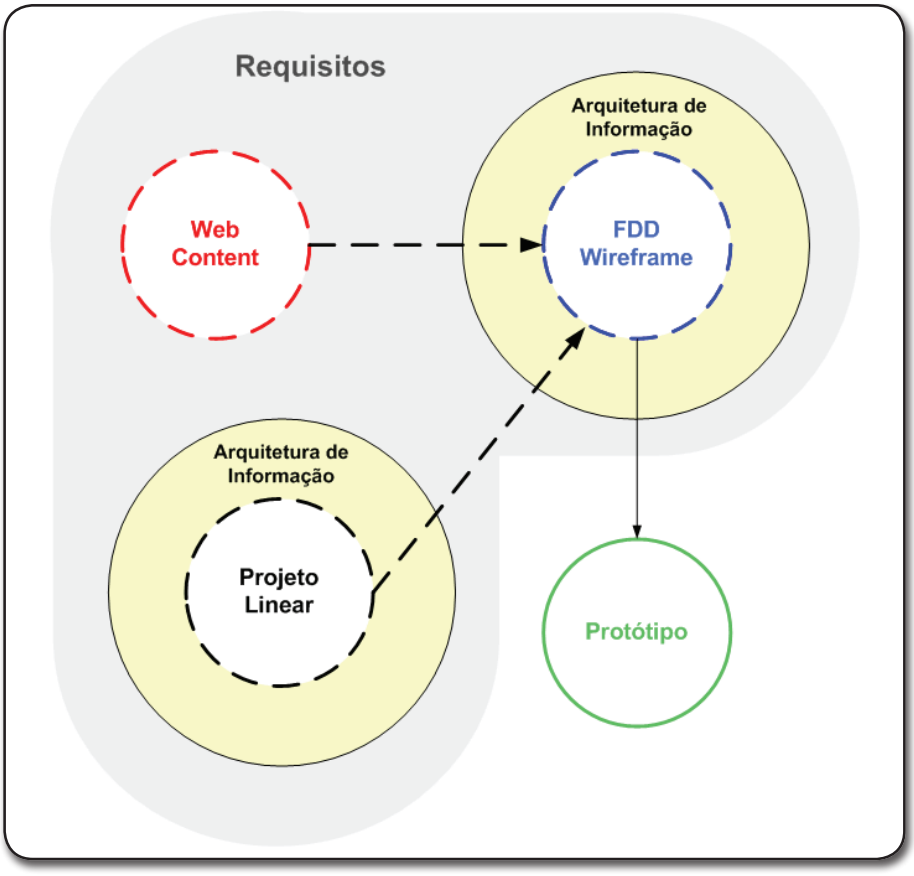


Figura 6. Overview da 1ª iteração

Após a primeira iteração, um Wireframe (uma parte do FDD) deverá ser enviado ao designer, que se responsabilizará pela construção da proposta de layout. A proposta de layout não terá o papel de mostrar interações e funcionalidades do sistema ao cliente.

Ao final dessa iteração, têm-se os seguintes artefatos sob gerência de configuração e mudança: FDD, Projeto Linear, Web Content, Planilha de requisitos, descrição dos casos de uso, plano de teste, documento de Baseline e quaisquer outros artefatos da UML que podem ser incluídos mediante a necessidade do projeto. A RTF e o protótipo de interface, aprovado pelo cliente, estabelecidos no final dessa iteração, não farão parte da gerência de configuração e mudança, pois são artefatos "mortos", os quais não sofrerão mais modificações.

Fase Elaboração - 2ª. Iteração

No contexto apresentado, a 2ª iteração abrange toda a fase de elaboração do projeto. Já não existem mais esforços voltados para o protótipo de interface. Ele servirá apenas para guiar a montagem da estrutura dos templates do site. Essa iteração levará mais tempo para acontecer do que a primeira, pois neste instante os esforços vão se concentrando e os envolvidos no projeto precisam entender e resolver os problemas mais críticos que começarão a aparecer.

Modelo de negócio: Análises de viabilidade e de riscos podem e muitas vezes devem continuar sendo feitas. O domínio do problema deve ser entendido completamente e uma solução deve ser descrita através dos casos de uso que estarão 80% finalizados no final desse fluxo de trabalho. A planilha de requisitos cresce na mesma proporção em que os casos de uso são validados pelo fornecedor e cliente.

Requisitos: Os requisitos continuam sendo extraídos dos casos de uso e compondo a planilha. No final dessa iteração tem-se 80% dos requisitos já documentados e aprovados pelo cliente. Os requisitos são a base para a construção dos artefatos, como o FDD, Web Content e o Projeto Linear. Por essa razão tem-se grande parte da formação desses documentos ainda nesta iteração.

Análise e Projeto: Os requisitos aprovados até o momento servirão como

base para a quase completa formação do Projeto Linear, do Web Content e consequentemente do FDD. A **Figura 1** fornece uma idéia da quantidade de esforço gasto para construir cada artefato. Como explicado anteriormente, o Web Content e o Projeto Linear são documentos que possuem forte ligação com o FDD, e este depende muito das informações dos dois primeiros artefatos, a fim de que seja corretamente construído. Alguns artefatos da UML poderão ser desenvolvidos nessa fase, com o objetivo de ajudar o cliente a entender o sistema.

Implementação: Inicia-se a junção de todas as funções e componentes pesquisados no início do projeto, com os artefatos desenvolvidos até o momento. Os desenvolvedores precisam estar aptos a entender não só os artefatos como o FDD, Web Content e o Projeto Linear, mas também os possíveis diagramas da UML e, principalmente os requisitos gerados até o instante. Deve-se ter cuidado com padrões, de forma que o código seja construído seguindo o conceito de componentização, para que seja facilmente reutilizado mais tarde.

Teste: Esse fluxo pode apresentar alterações no plano de teste devido ao número de requisitos já extraídos. Em conjunto com a fase de implementação, são realizados testes de módulos, com objetivo de verificar o que está sendo feito. É importante lembrar que estes testes não irão validar um requisito, mas apenas verificar se ele foi implementado corretamente.

Implantação: De forma a verificar se o que está sendo feito em relação à codificação irá funcionar do lado do cliente, no final dessa iteração, tem-se a implantação do que já foi codificado até o momento. De acordo com o resultado, algumas funções poderão exigir um cuidado especial e serem modificadas. Durante esse fluxo, poderão surgir alguns requisitos não funcionais, não encontrados durante a análise do sistema.

A **Figura 7** mostra que o objetivo agora não é mais entregar o protótipo de interface e sim, finalizar os documentos para que a equipe de desenvolvimento possa codificar o sistema de maneira rápida e correta. As linhas tracejadas, com menos espaços em relação às linhas da **Figura 6,** representam que os artefatos estão quase completos. Isso ocorre à medida que os requisitos são extraídos (área em cinza mais escura em relação às da **Figura 6**, indicando que mais requisitos estão sendo extraídos).

No final desta iteração, os artefatos estarão quase que totalmente concluídos. Eventuais ajustes podem ocorrer na Baseline e devem ser feitos pelo gerente do projeto. A RTF é construída avaliando e formalizando toda a iteração, servindo de aprendizado e preparando os envolvidos para a próxima fase do projeto.

Fase Construção - 3ª e 4ª Iteração

Neste contexto, a 3ª e 4ª iterações irão compor toda a fase de construção do projeto. Mais especificamente, a 3ª iteração indica o meio da fase de construção, enquanto que a 4ª, marca o final dessa fase. Eventuais ajustes nos artefatos surgirão em razão da implementação.

Modelo de negócio: As análises de viabilidade e de risco diminuem e servem agora apenas para pequenas tarefas realizadas no decorrer do projeto. Dificilmente, durante essas iterações, tem-se de reconstruir um modelo de casos de uso do negócio, a não ser que o cliente solicite uma nova característica para o projeto e que os gerentes julguem a necessidade de construção para melhor entender o problema.

Requisitos: Os requisitos ainda continuam sendo extraídos dos casos de uso e compondo a planilha. No final dessa iteração, tem-se em torno de 95% dos requisitos já documentados e aprovados pelo cliente. Dessa forma, o FDD, o Web Content e o Projeto Linear, estarão também, quase completamente elaborados.

Análise e Projeto: Alguns artefatos da UML, escolhidos para melhor representar as características do sistema, serão finalizados durante essas iterações. É válido lembrar que, exatamente nesse momento, o desenvolvedor poderá necessitar de algum outro artefato da UML o qual não havia escolhido anteriormente para representar alguma parte do sistema. Normalmente, essas escolhas são feitas devido a alguns requisitos críticos.

Implementação: Os programadores deverão possuir um suporte quase completo dos artefatos Web citados anteriormente. A maioria dos esforços do projeto são voltados agora para implementação e

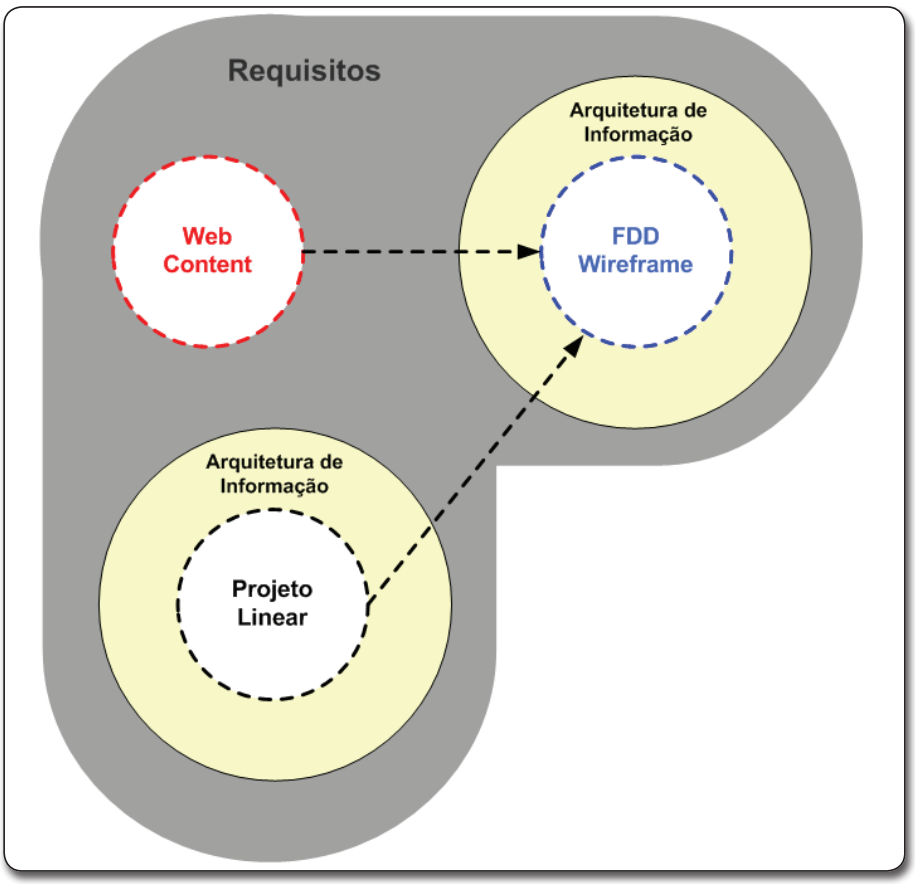


Figura 7. Overview da 2ª iteração

para a gerência das possíveis mudanças nos requisitos e consequentemente nos artefatos. Cuidados especiais devem ser tomados nesse momento, de forma a garantir que uma mudança não afete outra parte do sistema.

Teste: Nesse fluxo, consegue-se entender os pontos críticos de implementação e elaborar o plano de teste quase que totalmente. Os testes "caixa preta" são muito importantes nestas iterações, pois eles irão verificar a conformidade do sistema com as exigências do cliente. Testes de módulos continuam sendo feitos e neste instante os desenvolvedores farão também os testes de integração.

Implantação: À medida que a codificação é finalizada, uma versão executável do sistema poderá ser implantada no ambiente do cliente. A implantação é também uma forma de verificar se o que está sendo feito funcionará do lado do cliente.

A **Figura 8** mostra que os documentos estão sendo quase finalizados (pode-se notar pelas linhas tracejadas com menos espaços em relação à **Figura 7**) à medida que os requisitos ficam mais consistentes (cor mais escura na área que abrange os requisitos). Artefatos como o FDD, Web Content, Projeto Linear, descrição e diagramas de casos de uso etc, continuam a fazer parte da gerência de configuração e mudança.

As duas RTFs construídas nessa fase irão acrescentar muito para a experiência dos desenvolvedores, ensinando-os que, sempre existe a possibilidade da alteração de um requisito. O gerente continua a trabalhar atentamente na gerência de configuração e mudança, que serve tanto para o Baseline quanto para todos os artefatos "vivos" do projeto.

Fase Transição - 5ª Iteração

Esta é a última iteração do exemplo apresentado neste artigo, integrando o desenvolvimento Web com o Processo Unificado. O final dessa iteração marca o término do projeto, bem como a construção completa de todos os artefatos. A gerência de configuração e mudança, ainda continua trabalhando durante os fluxos para deixar os artefatos condizentes com o sistema desenvolvido.

Modelo de negócio: Dificilmente nesta etapa existirão tarefas que exijam análises de viabilidade e de risco. Devem-se armazenar os casos de uso do projeto com boas descrições representativas, a fim de que sejam facilmente encontrados, de forma a servir como modelo para projetos futuros.

Requisitos: Os requisitos nesse fluxo são mínimos. É mais comum encontrar alguns requisitos de engenharia (não funcionais) devido à parte do sistema já implantada no cliente. Os artefatos devem estar finalizados de acordo com todos os requisitos funcionais e não funcionais encontrados até o momento.

Análise e Projeto: Nesse fluxo, tem-se a finalização do FDD, Web Content e do Projeto Linear. Os artefatos da UML também serão finalizados durante essa iteração.

Implementação: O ideal é que os desenvolvedores façam ajustes no sistema, apenas para a adaptação ao ambiente do cliente. Pouquíssimas alterações nos requisitos funcionais devem ser realizadas nesse fluxo e consequentemente, poucas modificações no sistema.

Teste: Nesse fluxo, o documento de plano de teste será finalizado. Testes de sistemas e de validação podem ser realizados também pelo cliente. Se necessário, poderá ocorrer a contratação de uma empresa terceirizada para realizar os testes funcionais e não funcionais na aplicação.

Implantação: A versão executável final do sistema deverá ser colocada no ambiente de teste e posteriormente no ambiente de produção do cliente, mediante aprovação do mesmo. É válido lembrar que, o gerente ou o engenheiro de software responsável pelo controle de configuração e mudança, continuará a realizar o seu trabalho, pois no final dessa fase, alguns artefatos poderão ser ajustados.

A **Figura 9** mostra que os requisitos estão completos (cor escura) e consequentemente os documentos estão finalizados (linhas contínuas em torno do Web Content, Projeto Linear e do FDD). Outros artefatos, como a descrição e diagramas de casos de uso, plano de teste, planilha de requisitos estarão completos no final dessa iteração.

A RTF indicará os pontos fortes e fracos do projeto, ocorridos durante essa fase. Tem-se o backup de todas as versões do software realizadas até o momento, bem

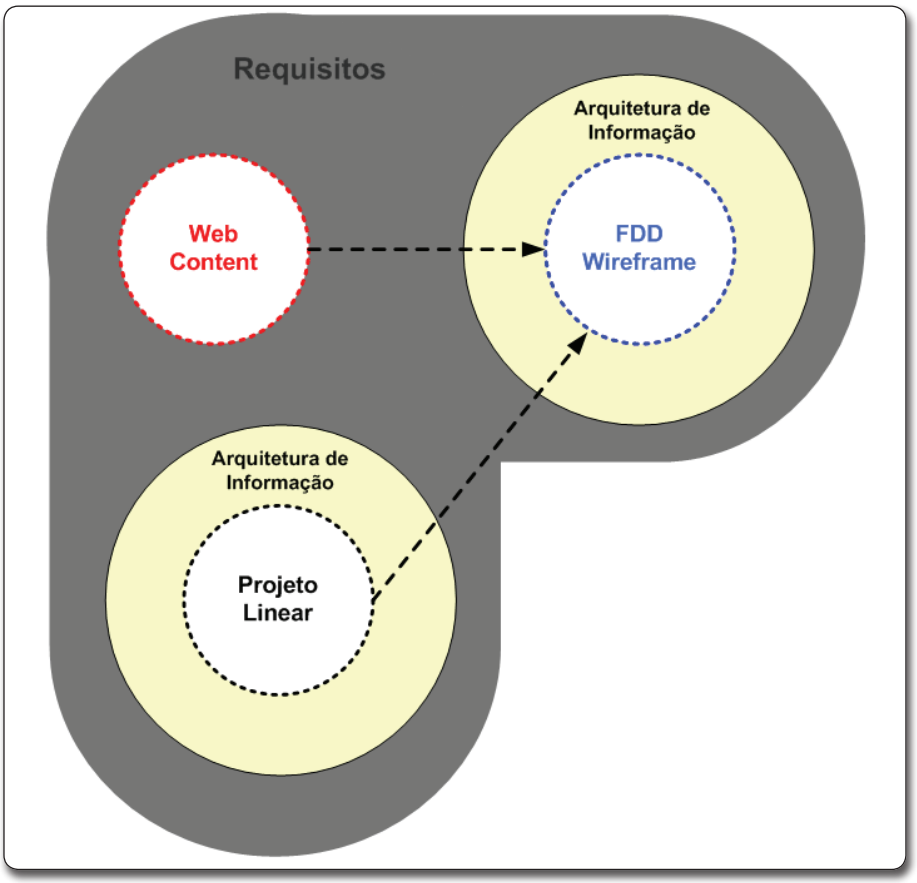


Figura 8. Overview da 3ª e 4ª iteração

como o armazenamento dos componentes desenvolvidos nos seus respectivos diretórios. Uma boa documentação é importante, de forma a facilitar a recuperação dos componentes para projetos futuros.

Um requisito mudou, e agora?

Uma das grandes preocupações do gerente do projeto é saber exatamente o que fazer quando um requisito é alterado pelo cliente. Explica-se a seguir, o comportamento dos artefatos quando um requisito sofre alguma modificação. A planilha de requisitos e o modelo de casos de uso são os primeiros artefatos que o gerente terá de verificar e se necessário, fazer a alteração imediata. É na planilha que estão todos os requisitos do sistema, bem como seus respectivos códigos indicadores. O código que indica o requisito alterado precisa ser identificado pelo gerente, que em seguida, deve abrir o Projeto Linear e verificar quais as áreas do site usam o requisito modificado. Dessa maneira, ele poderá obter os códigos de várias áreas da aplicação. Utilizando os códigos identificadores de cada área do site, o gerente deve pesquisar no Web Content, a fim de saber quais páginas do site sofrerão alteração. Perceba que não é objetivo do gerente de projeto efetuar as alterações, mas identificar o impacto que as solicitações de alteração podem causar.

Caso o projeto sofra uma mudança no conteúdo, essa identificação será feita facilmente. Em relação a uma mudança de funcionalidade, esta deverá impactar também na alteração de outros artefatos, como o digrama de classes, diagrama de componentes, diagrama de seqüência ou qualquer outro diagrama no qual esteja sendo usado no projeto. A alteração de funcionalidade envolve o desenvolvedor desde o início, que terá uma visão de como essa alteração deverá ser implementada. Com isso, consegue-se maior controle para que outras partes do sistema continuem se comunicando e funcionando normalmente.

Conclusão

Neste artigo, procurou-se mostrar como artefatos específicos, utilizados no desenvolvimento de projetos Web, podem ser usados durante a construção de um sistema baseado no Processo Unificado. Além disso, mostrou-se um exemplo prático indicando como o processo pode ser ajustado de acordo com o tipo e tamanho de um projeto. A configuração do processo a cada projeto mostra um acumulo de conhecimento armazenado durante a entrega de cada sistema, fazendo parte de uma melhoria contínua. ●

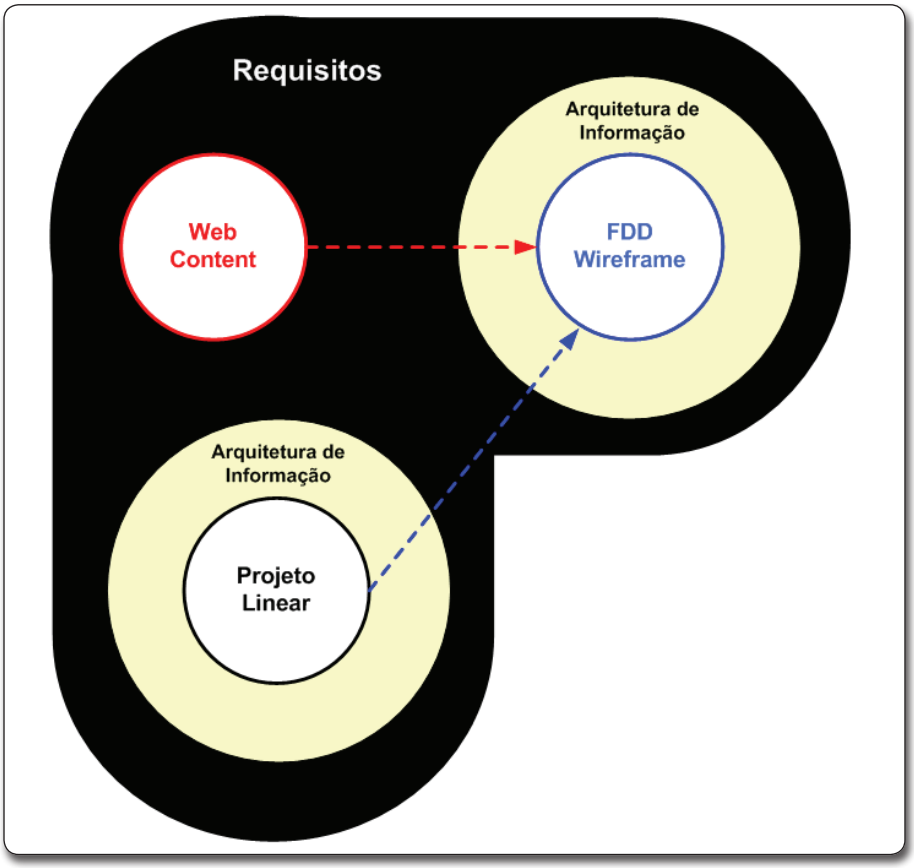


Figura 9. Overview da 5ª iteração

Arquitetura de Software

Desenvolvimento orientado para arquitetura

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 dos 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.

Software, de modo genérico, é uma entidade que se encontra em quase constante estado de mudança. As mudanças ocorrem por necessidade de corrigir erros existentes no software ou de adicionar novos recursos e funcionalidades. Igualmente, os sistemas computacionais (isto é, aqueles que têm software como um de seus elementos) também sofrem mudanças frequentemente. Essa necessidade evolutiva do sistema de software o torna 'não confiável' e predisposto a defeitos, atraso na entrega e com custos acima do estimado. Concomitante com esses fatos, o crescimento em tamanho e complexidade dos sistemas de software exige que os profissionais da área raciocinem, projetem, codifiquem e se comuniquem por meio de componentes de software. Como resultado, qualquer concepção ou solução de sistema passa então para o nível arquitetural, onde o foco recai sobre os componentes e relacionamentos entre eles num sistema de software.

Arquitetura de software

Quase cinco décadas atrás software constituía uma insignificante parte dos sistemas existentes e seu custo de desenvolvimento e manutenção eram desprezíveis. Para perceber isso, basta olharmos para a história da indústria do software (ver seção Links). Encontramos o uso do software numa ampla variedade de aplicações tais como sistemas de manufatura, software científico, software embarcado, robótica e aplicações Web, dentre tantas. Paralelamente, observou-se o surgimento de várias técnicas de modelagem e projeto bem como de linguagens de programação. Perceba que o cenário existente, décadas atrás, mudou completamente.

Antigamente, os projetos de sistemas alocavam pequena parcela ao software. Os componentes de hardware, por outro lado, eram analisados e testados quase exaustivamente, o que permitia a produção rápida de grandes quantidades de subsistemas e implicava em raros erros de

projetos. Entretanto, a facilidade de modificar o software, comparativamente ao hardware, tem servido como motivador para seu uso. Além disso, a intensificação do uso do software numa larga variedade de aplicações o fez crescer em tamanho e complexidade. Isto tornou proibitivo analisá-lo e testá-lo exaustivamente, além de impactar no custo de manutenção.

Um reflexo dessa situação é que as técnicas de abstração utilizadas até o final da década de 1980 (como decomposição modular, linguagens de programação de alto nível e tipos de dados abstratos) já não são mais suficientes para lidar com essa necessidade.

Diferentemente do uso de técnicas que empregam algoritmos e estruturas de dados e das linguagens de programação que implementam tais estruturas, o crescimento dos sistemas de software demanda notações para conectar componentes (módulos) e descrever mecanismos de interação, além de técnicas para gerenciar configurações e controlar versões. A **Tabela 1** apresenta o contexto da arquitetura de software. Na programação estruturada, é feito uso de estruturas de seqüência, decisões e repetições como 'padrões' de controle nos programas. Já a ocultação de informações é um recurso do paradigma orientado a objetos que permite ao programador, por exemplo, ocultar dados tornando-os seguros de qualquer alteração acidental. Além disso, na programação orientada a objetos, dados e funções podem ser 'encapsulados' numa entidade denominada objeto, o que resulta em mais simplicidade e facilidade na manutenção de programas. Por outro lado, os estilos arquiteturais capturam o 'padrão' de organização dos componentes de software num programa, caracterizando a forma na qual os componentes comunicam-se entre si.

Perceba que a categorização, apresentada na **Tabela 1**, teve o objetivo de capturar uma visão geral de abordagens aplicadas a sistemas de software. Nada impede, por exemplo, o uso da programação estruturada em sistemas de grande porte ou da ênfase de um estilo arquitetural num sistema de pequeno porte. Entretanto, essa prática não é comum.

Note que à medida que tamanho e complexidade dos sistemas de software aumentam, o problema de projeto extrapola as estruturas de dados e algoritmos de computação. Ou seja, projetar a arquitetura (ou estrutura geral) do sistema emerge como um problema novo. Questões arquiteturais englobam organização e estrutura geral de controle, protocolos de comunicação, sincronização, alocação de funcionalidade a componentes e seleção de alternativas de projeto. Por exemplo, nos sistemas Web, uma solução que tem sido empregada faz uso de múltiplas camadas separando componentes cliente, servidores de aplicações, servidores Web e outras aplicações (que possam ter acesso a esse sistema). Essa estruturação em camadas objetiva facilitar a alocação da funcionalidade aos componentes. O uso de camadas oferece suporte à flexibilidade e portabilidade, o que resulta em facilidade de manutenção. Outro aspecto a destacar da arquitetura em camadas é o uso de interfaces padrões visando facilitar reuso e manutenção. Interfaces bem definidas encapsulam componentes (com funcionalidades definidas) já testados, prática que permite o reuso e também auxilia na manutenção, já que toda e qualquer alteração necessária estaria confinada àquele componente.

Importância da Arquitetura de software

Todos esses fatores compreendem o projeto no nível arquitetural e estão diretamente relacionados com a organização do sistema e, portanto, afetam os atributos de qualidade (também chamados de requisitos não funcionais) como desempenho, portabilidade, confiabilidade, disponibilidade, entre outros. Se fizermos uma comparação entre arquitetura de software (caracterizada, por exemplo, pelo estilo em camadas) e arquitetura 'clássica' (relativa à construção de edificações), podemos observar que o projeto arquitetural é determinante para o sucesso do sistema.

A **Tabela 2** destaca aspectos da representação de projeto que captura elementos característicos da arquitetura enquanto que as restrições estão associadas a atributos de qualidade e, portanto, servem como determinantes nas decisões do projeto arquitetural. Por exemplo, embora o uso de múltiplas camadas facilite a manutenção de um sistema de software, também contribui para degradar o desempenho do sistema. Uma tática tem sido reduzir o nível de acoplamento entre componentes para não comprometer o desempenho do sistema. Dessa forma, se adotarmos uma redução no nível de acoplamento dos componentes, eles terão menor necessidade de comunicação entre si, o que resulta num melhor desempenho.

Hoje em dia, os processos de engenharia de software requerem o projeto arquitetural de software. Por quê?

- É importante poder reconhecer as estruturas comuns existentes de modo que arquitetos de software (ou engenheiros de software realizando o papel de arquiteto de software – conforme **Tabela 3**) possam entender as relações existentes nos sistemas em uso e utilizar esse conhecimento no desenvolvimento de novos sistemas.
- O entendimento das arquiteturas permite aos engenheiros tomarem decisões sobre alternativas de projeto.

Abordagem	Foco	Padrões
Programação estruturada	Sistemas de pequeno porte	Estruturas de controle
Abstração e modularização	Sistemas de médio porte	Encapsulamento e ocultação de informações
Componentes e conectores	Sistemas de grande porte	Estilos arquiteturais

Tabela 1. Contexto da arquitetura de software.

Categorias arquiteturais	Representações de projeto	Restrições
Arquitetura Clássica	Modelos, desenhos, planos, elevações e perspectivas	Padrão de circulação, acústica, iluminação e ventilação
Arquitetura de Software	Modelos para diferentes papéis, múltiplas visões	Desempenho, confiabilidade, escalonamento e manutenibilidade

Tabela 2. Comparação de aspectos arquiteturais.

• Uma especificação arquitetural é essencial para analisar e descrever propriedades de um sistema complexo, permitindo o engenheiro ter uma visão geral completa do sistema.

• O conhecimento de notações para descrever arquiteturas permite engenheiros comunicarem novos projetos e decisões arquiteturais tomadas a outros membros da equipe.

Cabe destacar que, para que haja o entendimento da arquitetura, faz-se necessário ao engenheiro de software conhecer os estilos arquiteturais existentes, conforme apresentado adiante. As propriedades de cada arquitetura, portanto, são dependentes do estilo arquitetural adotado. Por exemplo, o uso de uma notação padrão como a UML ajuda na representação de componentes e compartilhamento de informações do projeto.

Esses aspectos servem como indicadores de uma maturidade inicial de engenharia de software. Outros aspectos compreendem uso e reuso de soluções existentes no desenvolvimento de novos sistemas. Para tanto, a prototipagem tem sido usada em projetos de natureza inovadora (bem antes da implementação ou aceitação de um produto acontecer).

Além disso, o aumento da complexidade e quantidade de requisitos dos sistemas dificulta cada vez mais atender às restrições de orçamento e cronograma. Atualmente, empresas têm procurado incorporar estratégia de reuso de software, enfatizando o reuso centrado na arquitetura para obter melhores resultados no desenvolvimento de sistemas. Note que a arquitetura de software serve como uma estrutura através da qual se tem o entendimento dos componentes de um sistema e de seus inter-relacionamentos. Em outras palavras, ela define a estrutura do sistema, de modo consistente para implementações, já que está diretamente relacionada aos atributos de qualidade como confiabilidade e desempenho. A organização dos componentes num sistema de software impacta sobre a qualidade apresentada por ele. Por exemplo, a adoção de uma arquitetura em camadas serve para modularizar o sistema bem como facilitar modificações. Entretanto, um número elevado de camadas (4 ou 5) pode degradar o desempenho do sistema se houver um elevado grau de acoplamento entre os componentes.

Diversos benefícios decorrem da incorporação da arquitetura de software como 'elemento norteador' do processo de desenvolvimento de software. Cabe destacar que a arquitetura pode:

• Prover suporte ao reuso – seus componentes definidos e testados podem ser reaproveitados em novas aplicações.

• Servir de base à estimação de custos e gerência do projeto – a existência de uma arquitetura bem definida permite ao gerente de projeto adequadamente alocar tarefas de, por exemplo, implementação de componentes e melhor estimar o tempo e tamanho de equipe necessária para realização de um projeto.

• Servir de base para análise da consistência e dependência – o arquiteto de software pode verificar se a arquitetura de software adotada suporta os atributos de qualidade desejados de modo consistente e avaliar o nível de dependência dos atributos de qualidade em relação à arquitetura. Para tanto, ele faz a análise arquitetural que verifica o suporte oferecido pela arquitetura a um conjunto de atributos de qualidade (como desempenho, portabilidade e confiabilidade).

• Ser utilizada para determinar atributos de qualidade do sistema – o arquiteto de software faz a análise arquitetural a fim de determinar os atributos de qualidade. Trata-se de um processo iterativo.

Habilidades desejadas	Tarefas atribuídas
Conhecimento do domínio e tecnologias relevantes	Modelagem
Conhecimento de questões técnicas para desenvolvimento de sistemas	Análise de compromisso e de viabilidade
Conhecimento de técnicas de levantamento de requisitos, e de métodos de modelagem e desenvolvimento de sistemas	Prototipagem, simulação e experimentação
Conhecimento das estratégias de negócios da empresa	Análise de tendências tecnológicas
Conhecimento de processos, estratégias e produtos de empresas concorrentes	'Evangelizador' de novos arquitetos

Tabela 3. Habilidades e tarefas de um arquiteto de software.

• Atuar como uma estrutura para atender os requisitos do sistema – a arquitetura ajuda a definir os requisitos funcionais, que compreendem o conjunto de funcionalidades do sistema de software, e requisitos não funcionais (ou atributos de qualidade) que determinam as características visíveis ao usuário como desempenho e confiabilidade.

Uma questão que você deve estar se fazendo é: Por que apenas recentemente houve o foco na arquitetura de software?

A resposta é simples: economia e reuso.

Anteriormente não havia forte ênfase na disciplina de engenharia de software, fato este ocorreu com o amadurecimento desta nova área ao longo de toda a década de 1990. Tudo motivou o surgimento de um novo profissional: o arquiteto de software.

Habilidades do Arquiteto de software

Perceba que o arquiteto de software tem um papel de suma importância para estratégia adotada pela empresa. Ele precisa ter profundo conhecimento do domínio, das tecnologias existentes e de processos de desenvolvimento de software. Uma síntese de um conjunto desejado de habilidades para um arquiteto de software e das tarefas atribuídas a ele são apresentados na **Tabela 3**. Note que a prototipagem é uma tarefa comum onde o arquiteto desenvolve um protótipo para 'testar' uma possível solução. Já a simulação pode ser empregada quando ele necessita avaliar o suporte oferecido a determinado atributo de qualidade como o desempenho. Por outro lado, a experimentação pode ocorrer quando o arquiteto precisa testar um novo componente recém implementado.

Entendo o Estilo Arquitetural

O estilo arquitetural serve para caracterizar a arquitetura de software de um sistema, possibilitando a:

• Identificação de componentes – o arquiteto identifica quais os principais elementos que tem funcionalidades bem definidas como, um componente de cadastro de (informações de) usuários e um componente de autenticação de usuário numa aplicação Web.

• Identificação de mecanismos de interação – a comunicação entre objetos por

meio da troca de mensagens constitui uma forma através da qual os componentes de software interagem entre si.

• Identificação de propriedades – o arquiteto pode analisar as propriedades oferecidas por cada estilo baseado na organização dos componentes e nos mecanismos de interação, conforme discutido abaixo.

O estilo arquitetural considera o sistema por completo, permitindo o engenheiro ou arquiteto de software determinar como o sistema está organizado, caracterizando os componentes e suas interações. Em outras palavras, ele determina uma estrutura para todos os componentes do sistema. O estilo arquitetural compreende o vocabulário de componentes e conectores, além da topologia empregada. Mas, você pode estar se questionando: Por que saber o estilo arquitetural é importante?

Os sistemas de grande porte exigem níveis de abstração mais elevados (justamente onde se têm os estilos) que servem de apoio à compreensão do projeto e comunicação entre os participantes do projeto. Ele é determinante no entendimento da organização de um sistema de software.

Mas, o que se ganha em saber o estilo arquitetural? Ele oferece:

• Suporte a atributos de qualidade (ou requisitos não funcionais);
• Diferenciação entre arquiteturas;
• Menos esforço para entender um projeto;
• Reuso de arquitetura e conhecimento em novos projetos.

Conhecer o estilo arquitetural permite ao engenheiro antecipar, por meio da análise (arquitetural), o impacto que o estilo (isto é a classe de organização do sistema) terá sobre atributos de qualidade. Adicionalmente, facilita a comunicação do projeto, além do reuso da arquitetura (da solução).

A caracterização e existência de estilos arquiteturais constituem sinais de amadurecimento da engenharia de software, uma vez que permite ao engenheiro organizar e expressar o conhecimento de um projeto de modo sistemático e útil. Note que uma forma de codificar conhecimento é dispor de um vocabulário de um conjunto de conceitos (terminologia, propriedades, restrições), estruturas (componentes e conectores) e padrões de uso existentes. Conectores são empregados na interação entre componentes como, tubo (pipe) no estilo tubos e filtros, e mensagens no estilo de objetos.

Exemplificando o estilo pipes (tubos) e filtros

O estilo arquitetural de tubos e filtros considera a existência de uma rede através da qual dados fluem de uma extremidade à outra. O fluxo de dados se dá através tubos e os dados sofrem transformações quando são processados nos filtros.

O tubo provê uma forma unidirecional do fluxo de dados uma vez que ele atua como uma espécie de 'condutor' para o fluxo de dados entre a fonte até um destino. O exemplo mais comumente conhecido do estilo tubos e filtros é aquele usado no sistema operacional Unix, isto é: **who | sort**

A linha de comando acima executa o comando **who** (uma única vez) e encaminha sua saída ao programa **sort**, conforme ilustrado na **Figura 1**. O resultado da execução do programa who é uma lista de todos os usuários que se encontram conectados (a um servidor específico) naquele momento, enquanto que o programa sort ordena essa lista de usuários em ordem alfabética (de login).

Outro exemplo compreende a arquitetura básica de um compilador como ilustrado na **Figura 2**. Observe que cada etapa do processo de compilação é realizada separadamente por um componente (ou seja, um filtro).

Um compilador tem duas funções básicas: análise e síntese. A função de análise é implementada por três componentes: analisadores léxico, sintático e semântico. Já a função de síntese compreende os componentes de otimização e geração de código. Observe que essa arquitetura oferece suporte à portabilidade e reuso.

Entretanto, essa arquitetura evoluiu com a introdução de um componente gerador intermediário de código, conforme ilustrado na **Figura 3**, a fim de tornar a arquitetura do compilador 'mais portável' a várias plataformas com o objetivo de reduzir custos no desenvolvimento de diferentes produtos, ou seja, compiladores para diferentes plataformas.

Uma nova evolução da arquitetura de compiladores a fim de atender a necessidade de integração (do compilador) com outras ferramentas, tais como editor e depurador, resultou na arquitetura mostrada na **Figura 4**.

É importante salientar que a evolução da arquitetura de compilador foi resultado, principalmente, da necessidade de dar suporte ao requisito de portabilidade. Nesse sentido, pode-se destacar como vantagens:

• Problema ou sistema pode ser decomposto de forma hierárquica;

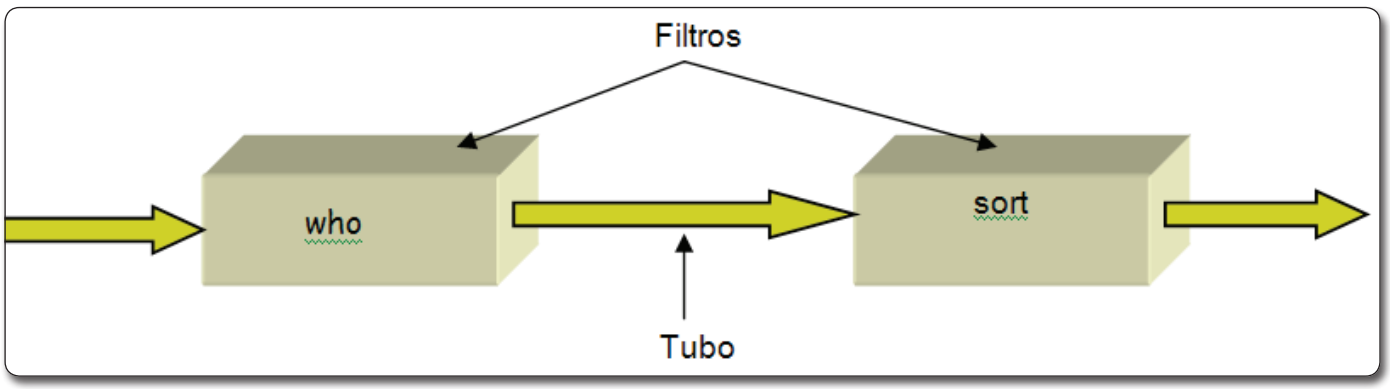


Figura 1. Exemplo do estilo arquitetural de tubos e filtros

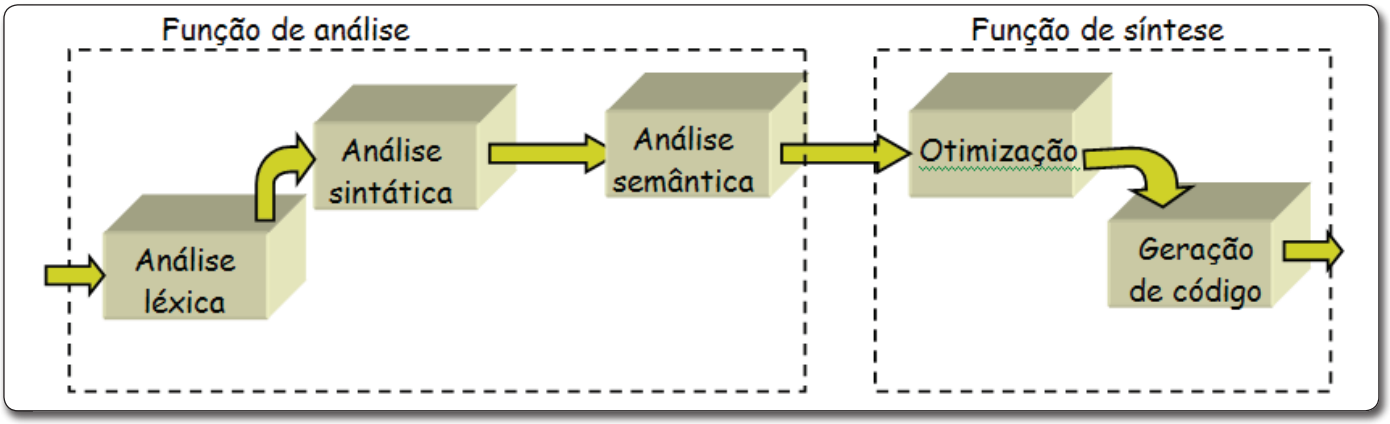


Figura 2. Arquitetura básica de um compilador (seguindo estilo arquitetural de tubos e filtros)

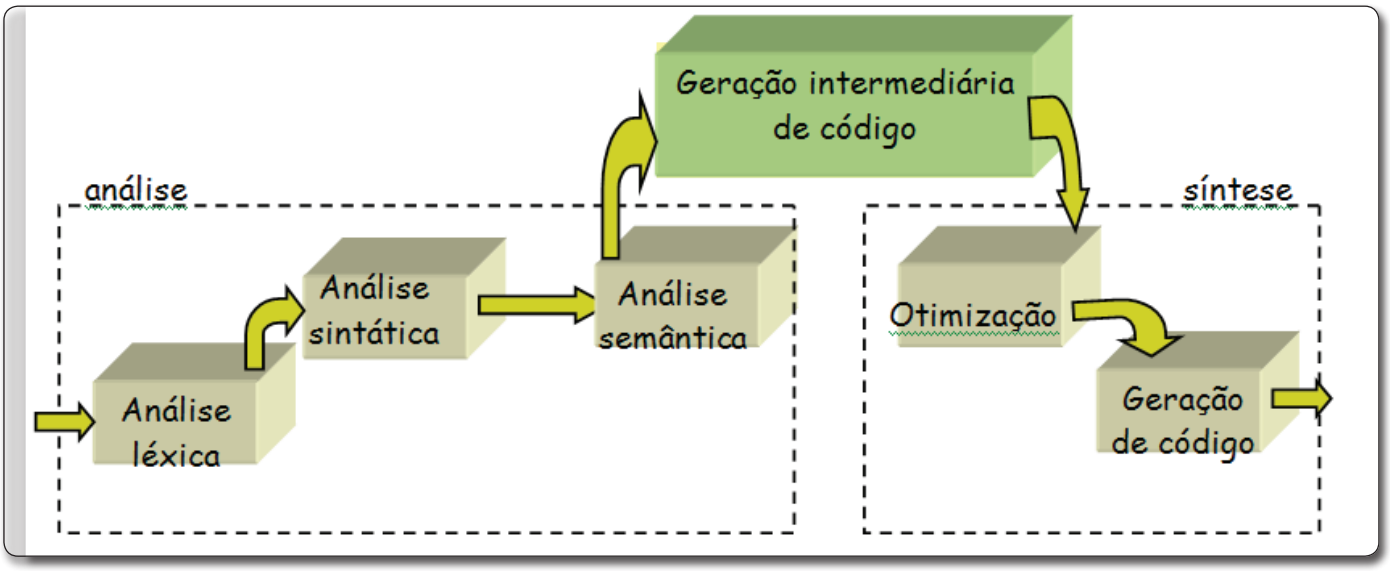


Figura 3. Evolução inicial da arquitetura de um compilador.

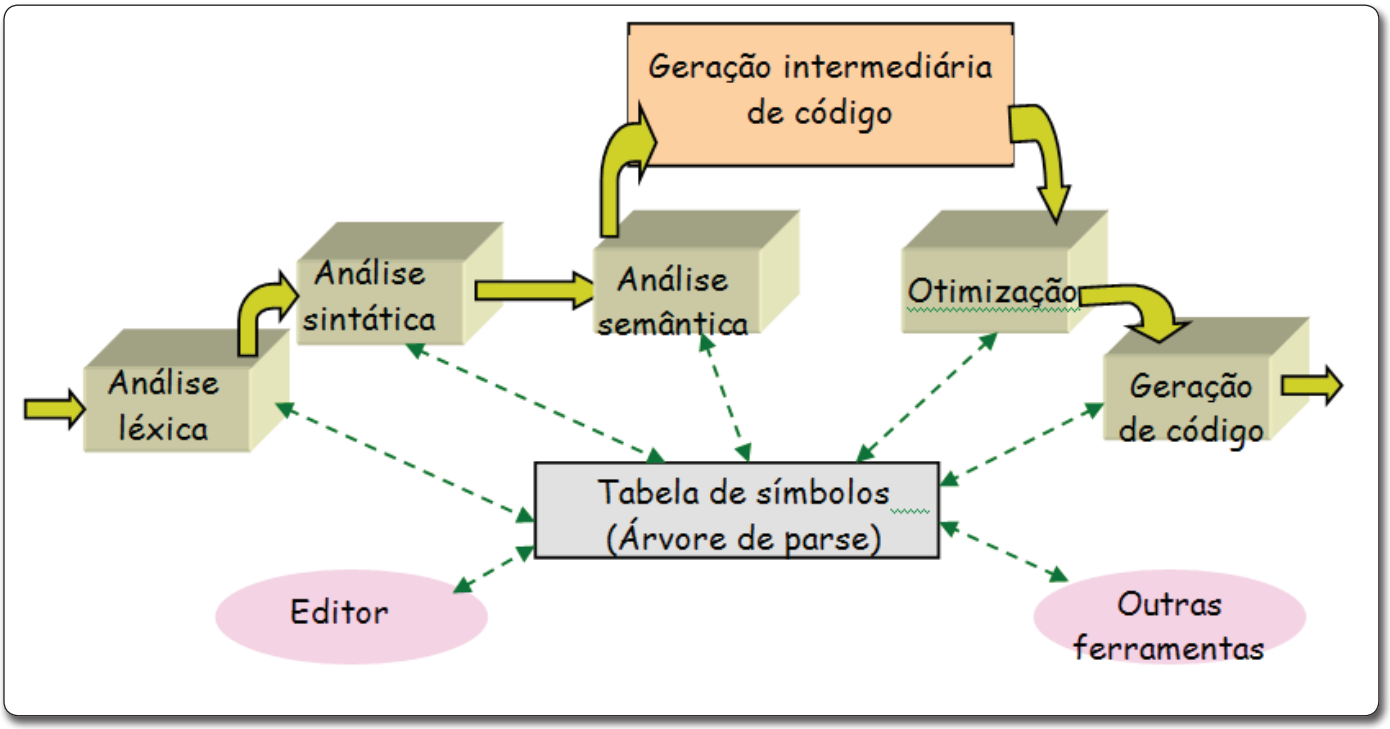


Figura 4. Nova evolução da arquitetura de um compilador.

• Função do sistema é vista como composição de filtros;
• Facilidade de reuso, manutenção e extensão, que emprega abordagem caixa preta, onde cada componente tem funcionalidade e interface bem definida, facilitando alterações nos mesmos;
• Desempenho pode ser incrementado através do processamento paralelo de filtros, já que a ativação e uso do componente ocorre com o fluxo de dados, permitindo que componentes com funcionalidades independentes sejam executados de forma concorrente.

Apesar das vantagens acima apontadas, o estilo de tubos e filtros coloca ênfase no modo 'batch', tornando difícil seu uso em aplicações interativas e em situações que exija ordenação de filtros. Outra questão técnica a ser observada é a possibilidade de haver deadlock com o uso de buffers finitos (para armazenamento temporário de dados). Esse estilo arquitetural tem sido empregado em razão das vantagens anteriormente destacadas.

Exemplos de outros estilos arquiteturais compreendem:

• *Camadas* – a arquitetura do sistema de software é organizada num conjunto de camadas, oferecendo maior flexibilidade e suporte a portabilidade. A identificação do nível de abstração nem sempre é evidente e perde-se desempenho à medida que o número de camadas cresce. Exemplo desse estilo compreende os sistemas Web de múltiplas camadas que separa cliente, servidores de aplicação, servidores Web e outros clientes Web.

• *Objetos* – essa arquitetura combina dados com funções numa única entidade (objeto), facilitando a decomposição do problema, manutenção e reuso. É comum utilizar a arquitetura orientada a objetos em sistemas de informação como sistemas de consulta e empréstimos online de bibliotecas de instituições de ensino que dispõem de componentes de cadastro de usuários e componentes de autenticação de usuários. Note que componentes similares existem em outros sistemas de informações, tais como sites de conteúdos (jornais e revistas) que exigem cadastro e autenticação de qualquer usuário antes de disponibilizar o conteúdo.

• *Invocação implícita* – diferentemente do estilo baseado em objetos, no qual um componente invoca outro diretamente por meio da passagem de mensagens, a invocação implícita requer que componentes interessados em receber ou divulgar eventos se registrem para receber ou enviar. Um exemplo de sistema empregando mensagens são listas de notícias e fórum que possuem componentes de registro de novos usuários acoplados ao componente de autenticação. Perceba que esse tipo de sistema apenas permite que o usuário tenha acesso ao conteúdo se este for devidamente autenticado e registrado.

• (Sistemas orientados a) *Eventos* – trata-se de um estilo no qual os componentes podem ser objetos ou processos e a interface define os eventos de entrada e saída permitidos. Conectores são implementados através do 'binding' evento-procedimento. Assim, eventos são registrados junto com eventos e os componentes interagem entre si através do envio de eventos. Dessa forma, quando um evento é recebido, o(s) procedimento(s) associado(s) a este evento é(são) invocado(s). Um interessante exemplo deste estilo são jogos online, como discutido na seção seguinte.

• Quadro negro (ou Blackboard) – esse estilo faz uso de um repositório central de dados circundado por um conjunto de componentes (ou células) de informações.

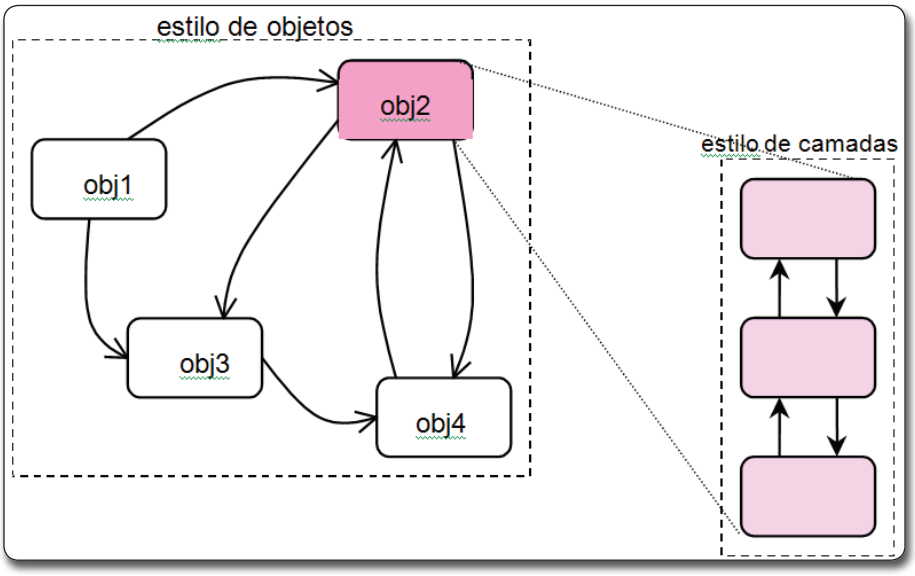


Figura 5. Combinação de estilos.

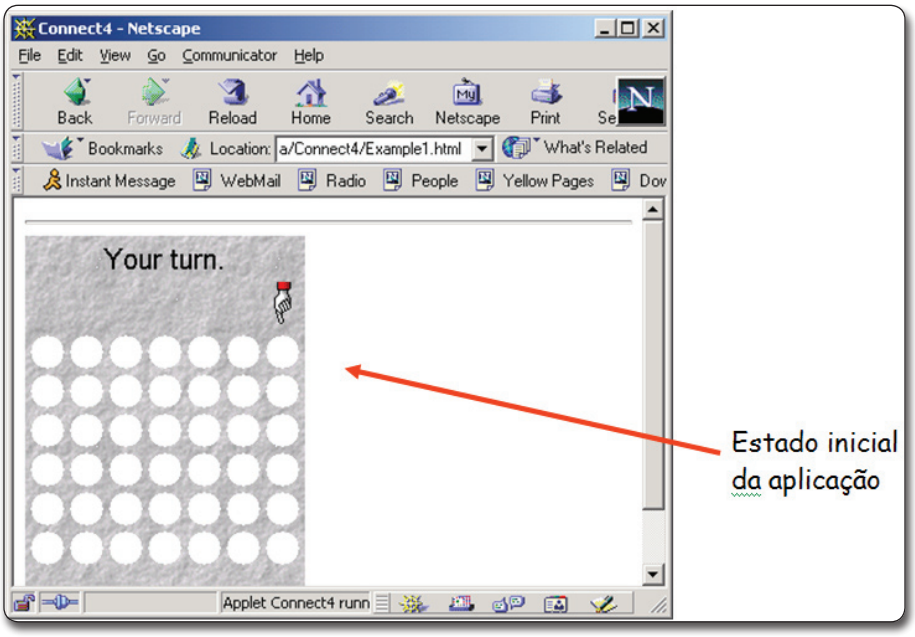


Figura 6. Jogo Connect4.

Esses componentes contêm informações necessárias à solução de problemas. Os dados da solução de problemas são mantidos na base de dados compartilhada (o repositório), o qual é denominado de quadro negro. O exemple mais comum desse estilo é um sistema especialista.

• Combinação de estilos – Outros sistemas, na prática, combinam estilos arquiteturais resultando numa heterogeneidade, como ilustrado na **Figura 5,** que agrega o estilo de objetos e camadas.

Exemplificando o estilo combinado de objetos e eventos

Jogos online e de computador têm se tornado comuns atualmente com a popularidade da Internet. Costuma-se categorizar os jogos em:

• Baseados na vez – são jogos nos quais cada ação é baseada na vez do jogador como jogo da velha e xadrez.

• Baseados em eventos – são jogos onde eventos podem ocorrer em qualquer instante e estes ditam o ritmo do jogo. Exemplos compreendem simuladores de vôo e corridas de carro.

Por exemplo, quando os jogos são disponibilizados na Internet, costuma-se denominá-los de jogos online ou jogos para Internet, possibilitando ao usuário jogar contra a máquina (computador). Um exemplo desse tipo de jogo de computador é o Connect4 que tem como meta para cada jogador conectar quatro peças da mesma cor, verticalmente, horizontalmente ou diagonalmente. Cada jogador deve depositar uma peça na parte superior da coluna selecionada e esta cai até preencher a lacuna inferior da coluna (selecionada), conforme ilustrado na **Figura 6**. Note que o quadro contém 7 colunas e 6 linhas, um indicador de status do jogo e um seletor manual (utilizado para selecionar a coluna na qual uma peça será depositada).

A arquitetura de software dessa aplicação é apresentada na **Figura 7**. O componente jogador computacional (que dispõe de recursos de inteligência artificial para simular um jogador humano) contém uma classe Connect4State que trata a maioria das requisições para verificar se algum jogador venceu o jogo, e também dispõe de mecanismo para atualizar o estado do jogo após uma jogada do jogador computacional.

O componente (máquina) de inferência contém uma classe para tratar as jogadas feitas pelos jogadores bem como determinar a melhor jogada para o jogador computacional.

O componente de interface de usuário contém uma classe que carrega os arquivos de imagem e áudio, trata as requisições feitas através do mouse e invoca um método que checa se houve vitória, derrota ou empate, além de fazer a atualização da interface gráfica. Um possível estado final desse jogo é mostrado na **Figura 8**, onde na terceira linha há um conjunto de quatro peças na cor azul, indicando que o computador completou primeiro a conexão de quatro peças na mesma cor (a cor do usuário humano é vermelha).

A combinação estilo arquitetural orientado para eventos e objetos permite a decomposição de um sistema em termos de objetos (componentes de inferência, componente de interface gráfica e componente jogador computacional) que são mais independentes além de possibilitar que atividades de computação e coordenação (de eventos) sejam realizadas separadamente. Há ainda a facilidade de reuso e manutenção, já que novos objetos podem ser facilmente adicionados. Essa característica, e interfaces bem definidas, facilitam ainda a integração.

O mecanismo de invocação é não determinístico (isto é, ocorre de forma aleatória) uma vez que considera a recepção de eventos. Adicionalmente, os componentes têm seus dados preservados de qualquer modificação acidental já que essas informações são encapsuladas em objetos, facilitando também a integração.

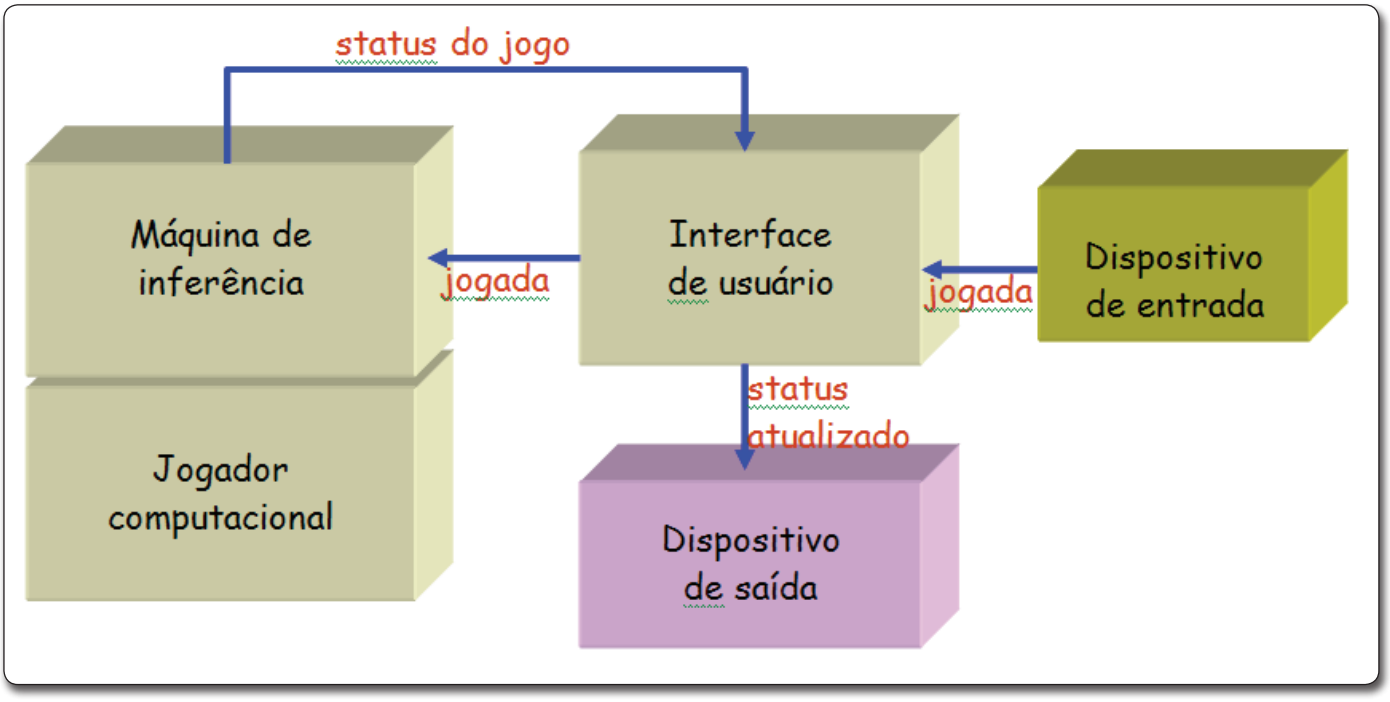


Figura 7. Arquitetura do Connect4

Importância do Reuso

É importante perceber de tudo o que foi apresentado e discutido que um modelo arquitetural oferece suporte ao reuso de várias formas. Por exemplo, pode se ter arquiteturas reusáveis que forneçam a organização e o modelo de coordenação, permitindo seu uso em diversos sistemas. Além disso, podem-se reutilizar componentes, já testados, em mais de um sistema. Disso tudo, o mais importante é o reuso do conhecimento que tem impacto direto na definição da arquitetura de software candidata à solução de um problema (ou sistema).

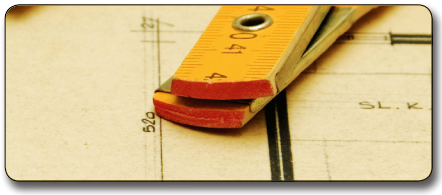
A ampla variedade de plataformas e utilitários, juntamente com a pressão do mercado para reduzir o tempo de entrega de produtos de software e elevar a produtividade, faz com que o reuso seja apontado como uma das chaves de sucesso para empresas.

O reuso de artefatos (ou componentes) é possível quando o projeto arquitetural está incorporado e orienta o processo de desenvolvimento de software. Isto, como discutido, permite antever os atributos de qualidade que o sistema suporta e também administrar o cronograma de execução do projeto. Portanto, impactando positivamente o reuso e economia do sistema.

O quadro apresentado na **Tabela 4** sumariza as principais características da arquitetura de software e pontos importantes na capacitação e reciclagem de engenheiros e arquitetos de software.

Conclusão

Embora arquitetura de software seja um tema relevante no contexto atual para desenvolvimento de sistemas de software que têm seu foco no reuso e, portanto, consideram tanto o aspecto econômico quanto a produtividade, sua incorporação nos processos de desenvolvimento de software tem sido observada apenas em empresas de grande porte e em poucas de médio porte. Entretanto, esse cenário começa a mudar dada a crescente necessidade de integração de sistemas a qual tem a arquitetura de software como premissa. Assim, o fator econômico tem sido e será o determinante da sobrevivência da empresas de software. Novas estratégias de desenvolvimento de sistemas devem ser *para* o reuso e *com* reuso (de componentes de software) e o pilar principal dessas estratégias é a arquitetura de software. Há, portanto, uma necessidade premente de formação de capital humano com tais qualificações. ●



Links

História da Indústria de Software
www.softwarehistory.org

An introduction to software architecture
http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html

SEI's Software Architecture Technology Initiative
www.sei.cmu.edu/architecture/sat_init.html

Worldwide Institute of Software Architects
www.wwisa.org/wwisamain/index.htm

The Software Architecture Portal
http://www.softwarearchitectureportal.org/

Características de arquitetura de software	Uso prático da arquitetura de software
Constitui um artefato reutilizável	Como um arquiteto de software pode organizar o projeto e código de um sistema
Dispõe de mecanismos de interconexão	Como um arquiteto avalia e implanta arquiteturas de software em sistemas
Oferece um vocabulário de projeto e separa funcionalidades	Como um arquiteto de software atua no processo de desenvolvimento de software
Vincula o projeto a atributos de qualidade	Como um arquiteto avalia a qualidade do código baseada em métricas de produto
Suporta o desenvolvimento baseado em componentes e linha de produto (quando os requisitos são considerados para uma família de sistemas)	Como usar arquitetura como parâmetro para reduzir custos de manutenção e amortizar custos de desenvolvimento

Tabela 4. Características e uso prático da arquitetura de software

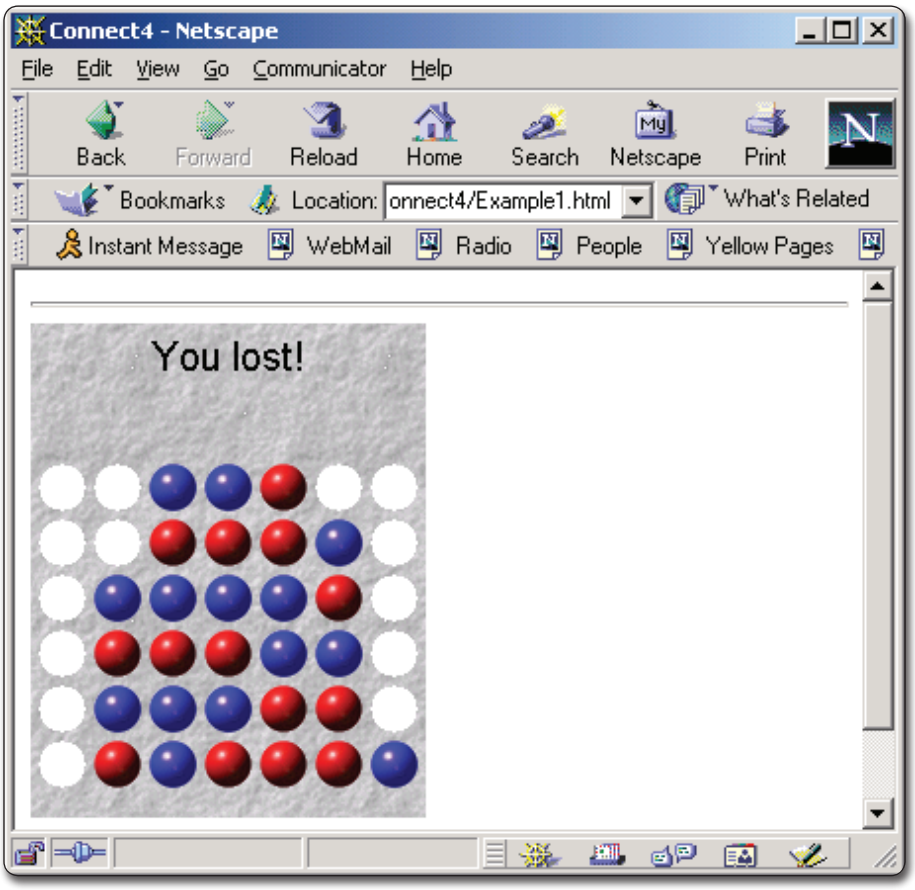


Figura 8. Possível estado final do jogo Connect4

Introdução à Engenharia de Requisitos

Ana Luiza Ávila
analuizaavila@yahoo.com.br
É bacharel em Ciências da Computação pela Universidade Salvador (UNIFACS) e Mestre em Ciências da Computação pela PUC-Rio na linha de Engenharia de Software.

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 em projetos de consultoria na COPPE/UFRJ. É colaborador da Engenharia de Software Magazine.

Desenvolver software é uma atividade complexa por natureza. Uma das razões para esta afirmação é que não existe uma única solução para cada cenário de desenvolvimento. Além disso, lidamos o tempo todo com pessoas, o que torna o sucesso do projeto bastante relacionado à competência da equipe e à forma como trabalham, e, para dificultar ainda mais, muitas vezes não fazemos uso de um processo bem definido para apoiar as atividades do projeto.

Entende-se por processo, neste contexto, como sendo um conjunto de atividades bem definidas com os respectivos responsáveis por execução, ferramentas de apoio e artefatos produzidos. Ou seja, define-se como a equipe deverá trabalhar para alcançar o objetivo: desenvolver software com qualidade dentro de prazos, custos e requisitos definidos.

A boa notícia é que muitas empresas estão se movimentando no sentido de definirem detalhadamente seus processos para apoiarem suas atividades de desenvolvimento. Uma recente matéria publicada na revista Exame relata o crescimento do número de empresas que atingiram níveis de maturidade considerando modelos como MPS.BR e CMMI. Este resultado é um forte indicador de que as empresas nacionais estão se preocupando com a qualidade dos serviços que oferecem, conseguindo, dessa forma, uma inserção maior no mercado internacional de desenvolvimento de software.

Neste artigo, faremos uma introdução à Engenharia de Requisitos, atividade base para as demais tarefas associadas ao desenvolvimento de software.

Engenharia de Software e Requisitos

Vimos na introdução que se busca cada vez mais o apoio dos fundamentos da engenharia de software no desenvolvimento de sistemas. Entendemos engenharia de software como sendo, de acordo com o IEEE, a aplicação de uma abordagem sistemática, disciplinada e quantificável

no desenvolvimento, operação e manutenção de software. Sistemática por que parte do princípio de que existe um processo de desenvolvimento definindo as atividades que deverão ser executadas. Disciplinada por que parte do princípio de que os processos definidos serão seguidos. Quantificável por que se deve definir um conjunto de medidas a serem extraídas do processo durante o desenvolvimento de forma que as tomadas de decisão relacionadas ao desenvolvimento do software (por exemplo, melhoria de processo) sejam embasadas em dados reais, e não em "achismos". Alguns de seus principais objetivos são:

- Qualidade de software;
- Produtividade no desenvolvimento, operação e manutenção de software;
- Permitir que profissionais tenham controle sobre o desenvolvimento de software dentro de custos, prazos e níveis de qualidade desejados.

Entretanto, o cenário de desenvolvimento de software atual e o cenário idealizado junto à engenharia de software ainda estão distantes. Vários fatores contribuem para isso, podemos citar dois:

- O **não** uso dos fundamentos da engenharia de software para apoiar as atividades do desenvolvimento;
- O **mau** uso dos fundamentos da engenharia de software para apoiar as atividades do desenvolvimento.

Isso tem diversas conseqüências. Gostaríamos de destacar neste artigo o crescente custo com manutenção dos sistemas. Consideramos como manutenção neste artigo como sendo qualquer retrabalho (em nível de requisitos, projeto, codificação, teste) causado por uma definição do domínio do problema mal elaborada nas fases iniciais do desenvolvimento.

Neste ponto, é importante analisamos os dados da **Tabela 1**. Perceba que o custo das atividades relacionadas à análise de requisitos é baixo. Por outro lado, é nesta fase que grande parte dos defeitos são inseridos. Podemos perceber ainda analisando a primeira linha que o custo de correção destes problemas nesta fase é baixo. Continuando a análise, percebemos que estes defeitos não são tratados no momento devido, o que pode aumentar bastante o custo com o projeto.

Embora simples, esta análise nos permite concluir que é importante que seja dada maior importância às atividades relacionadas à especificação dos requisitos do software.

Reforçando a importância que as atividades relacionadas a requisitos devem possuir na indústria de software, estudo realizado pelo Standish Group, considerando 350 companhias e 8.000 projetos de software, em 1995 revelou que (ver **Figura 1**):

- 16,2% dos projetos são finalizados com sucesso, ou seja, cobre todas as funcionalidades em tempo e dentro do custo previsto;
- 52.7% dos projetos são considerados problemáticos, ou seja, não cobre todas as funcionalidades exigidas, custo aumentado e está atrasado.
- 31,1% dos projetos fracassam, ou seja, o projeto é cancelado durante o desenvolvimento.

O Standish Group ainda fez uma análise sobre os fatores críticos para sucesso dos projetos de software. O resultado dos dez mais lembrados pode ser visto na **Tabela 2**. Podemos perceber que três dos principais fatores estão relacionados às atividades de requisitos: (1) Requisitos Incompletos; (2) Falta de Envolvimento do Usuário; (6) Mudança de Requisitos e Especificações.

Neste ponto, sabemos que um trabalho mais criterioso na área de requisitos é fundamental para o sucesso de projetos de software. A partir da próxima seção conheceremos a definição de requisitos e algumas de suas definições relacionadas antes de discutirmos mais profundamente a engenharia de requisitos.

Fatores Críticos	%
1. Requisitos Incompletos	13.1%
2. Falta de Envolvimento do Usuário	12.4%
3. Falta de Recursos	10,6%
4. Expectativas Irreais	9,9%
5. Falta de Apoio Executivo	9,3%
6. Mudança de Requisitos e Especificações	8,7%
7. Falta de Planejamento	8,1%
8. Sistema não mais necessário	7,5%

Tabela 2. Fatores críticos do sucesso.

	% do Custo de Desenvolvimento	% dos erros introduzidos	% dos erros encontrados	Custo relativo de correção
Análise de Requisitos	5	55	18	1
Projeto	25	30	10	1 – 1.5
Códificação e teste de unidade	50			
Teste	10	10	50	1 – 5
Validação e Documentação	10			
Manutenção		5	22	10 – 100

Tabela 1. Cenário atual de desenvolvimento.

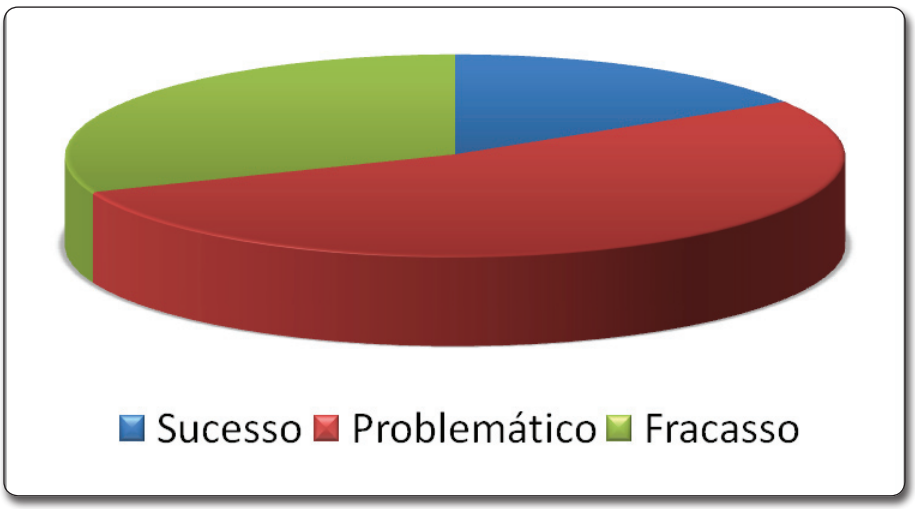


Figura 1. Distribuição da conclusão dos projetos de software.

Requisitos

Existem diferentes definições encontradas na literatura técnica para requisitos:

- Um requisito é uma característica do sistema ou a descrição de algo que o sistema é capaz de realizar para atingir os seus objetivos;
- As descrições das funções e restrições são os requisitos do sistema;
- Um requisito é uma propriedade que o software deve exibir para resolver algum problema no mundo real;
- Uma condição ou uma capacidade que deve ser alcançada ou estar presente em um sistema para satisfazer um contrato, padrão, especificação ou outro documento formalmente imposto...

Percebe-se que as citações encontradas definem o mesmo conceito sob diferentes perspectivas. Podemos entender requisitos como sendo o conjunto de necessidades explicitadas pelo cliente que deverão ser atendidas para solucionar um determinado problema do negócio no qual o cliente faz parte. É importante estar atento para esta definição: embora o requisito seja definido pelo cliente, nem sempre o que o cliente quer é o que o negócio precisa. Cabe à equipe de consultores identificar a real necessidade do negócio.

Neste contexto, requisitos são importantes para:

- Estabelecer uma base de concordância entre o cliente e o fornecedor sobre o que o software fará;
- Fornecer uma referência para a validação do produto final;
- Reduzir o custo de desenvolvimento (como vimos anteriormente, requisitos mal definidos causam retrabalho).

Entendida a definição de requisitos, é preciso conhecer seus tipos.

Requisitos funcionais

São requisitos diretamente ligados a funcionalidade do software, descrevem as funções que o software deve executar. Alguns exemplos são:

- O software deve permitir o cadastro de clientes;
- O software deve permitir a geração de relatórios sobre o desempenho de vendas no semestre;
- O software deve permitir o pagamento das compras através de cartão de crédito.

Requisitos não funcionais

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

- O software deve ser compatível com os browsers IE (versão 5.0 ou superior) e Firefox (1.0 ou superior);
- O software deve garantir que o tempo de retorno das consultas não seja maior do que 5 segundos.

Requisitos de domínio

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

- O calculo da média final de cada aluno é dado pela fórmula: (Nota1 * 2 + Nota2 * 3)/5;
- Um aluno pode se matricular em uma disciplina desde que ele tenha sido aprovado nas disciplinas consideradas pré-requisitos.

A partir da próxima seção apresentaremos os conceitos envolvidos na engenharia de requisitos.

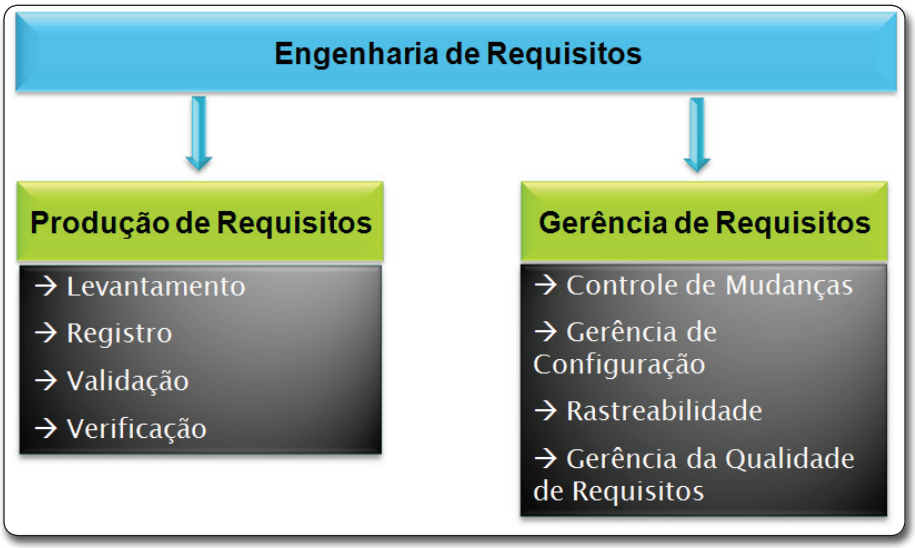


Figura 2. Engenharia de Requisitos.

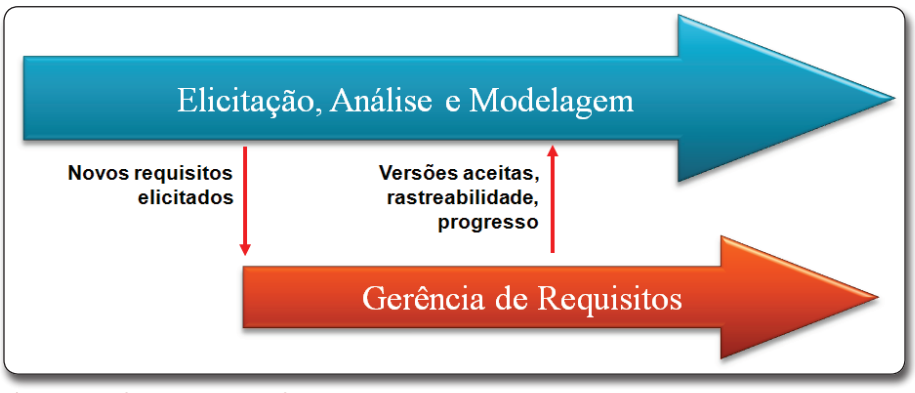


Figura 3. Produção e Gerência de Requisitos.

Engenharia de Requisitos

Existem diferentes definições encontradas na literatura técnica para engenharia de requisitos:

- Termo usado para descrever as atividades relacionadas à investigação e definição de escopo de um sistema de software;
- Processo sistemático de desenvolvimento de requisitos através de um processo cooperativo de análise onde os resultados das observações são codificados em uma variedade de formatos e a acurácia das observações é constante-

mente verificada;

• Processo de descobrir, analisar, documentar e verificar as funções e restrições do sistema.

Embora coerentes, estas definições podem ser melhoradas. Perceba que elas referem-se apenas às atividades relacionadas à produção de requisitos. Entretanto, nada é dito a respeito da gerência destas atividades, também conhecida como gerência de requisitos. Com isto em mente, podemos evoluir a definição de engenharia de requisitos para: termo usado para descrever as atividades relacionadas à produção (levantamento, registro, validação e verificação) e gerência (controle de mudanças, gerência de configuração, rastreabilidade, gerência de qualidade dos requisitos) de requisitos. A **Figura 2** representa essa definição.

Desta forma, os dois conceitos base (produção e gerência) devem ser considerados em conjunto ao se definir estratégias de trabalho com requisitos nas organizações (ver **Figura 3**).

Neste ponto podemos citar alguns dos principais objetivos da engenharia de requisitos:

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

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

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

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

Para apoiar o alcance destes objetivos, é importante que se tenha um processo de engenharia de requisitos bem definido. Um processo de engenharia de requisitos é um conjunto estruturado de atividades a serem seguidas para criar, validar e manter um documento de requisitos. Poucas organizações têm um processo de ER explicitamente definido e padronizado. Entretanto, sugere-se que cada organização deva desenvolver um processo adequado à sua realidade. A **Figura 4** apresenta um modelo genérico de atividades que pode descrever a maioria dos processos de engenharia de requisitos. Perceba que ele está concentrado nas atividades de produção dos requisitos.

Apesar do aparente fluxo entre as atividades, não existe uma fronteira explícita elas. Na prática existe muita sobreposição e interação entre uma atividade e outra.

Como entradas para o processo são consideradas:

• Descrições do que os stakeholders necessitam para suportar suas atividades;

• Informações a respeito do sistema que será substituído ou de qualquer sistema com o qual o sistema sendo definido terá que interagir;

• Padrões vigentes na organização a respeito de práticas de desenvolvimento de sistemas, gerência de qualidade, etc.;

• Regulamentos externos, tais como leis, regulamentos de segurança ou saúde;

• Informações gerais sobre o domínio de aplicação.

Em paralelo à execução das atividades definidas no processo da **Figura 5**, está o processo de gerência dos requisitos, voltado ao endereçamento de modificações nos requisitos. Os aspectos relacionados às atividades de gerência podem ser vistos na **Figura 3**.

A partir de agora conheceremos cada um dos conceitos que, em conjunto, definem a engenharia de requisitos.

Produção de Requisitos

A cada fase do ciclo de vida do software produzimos um documento contendo uma representação distinta do software a ser construído. Cada um desses documentos representa o software em um determinado nível de abstração. A tendência é diminuirmos o nível de abstração através da inclusão de mais e mais detalhes, até que, sua última representação seja o código fonte na linguagem escolhida.

Um dos artefatos produzidos no início do processo de desenvolvimento de software é a sua especificação de requisitos. Ele é base para as demais atividades de desenvolvimento e sua qualidade é fundamental para o sucesso do projeto. Uma especificação de requisitos bem elaborada é pré-requisito para um software de qualidade, embora não seja garantia disso. Desta forma, durante a produção de requisitos devemos possuir, além das atividades essenciais de levantamento e

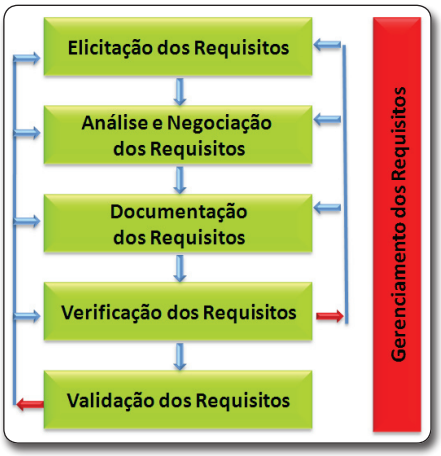


Figura 4. Processo de Engenharia de Requisitos.

especificação, atividades relacionadas à garantia da qualidade. Conheceremos nesta seção as quatro atividades base relacionadas com a produção de requisitos.

Levantamento

Esta atividade relaciona-se à obtenção dos requisitos do software. Para isto, analistas e engenheiros de software trabalham com clientes e usuários finais para descobrir o problema a ser resolvido, os serviços do sistema, o desempenho necessário, restrições de hardware e outras informações.

Existem algumas técnicas que apóiam as atividades de levantamento de requisitos. Uma breve descrição de algumas delas é:

• **Entrevista:** esta técnica resume-se em "conversas" realizadas com o usuário (entrevistado) para levantar os requisitos do sistema a ser desenvolvido. Podemos decompor esta técnica nas seguintes atividades:

o Ler material de suporte;

o Estabelecer os objetivos da entrevista;

o Decidir quem entrevistar;

o Preparar o entrevistado;

o Decidir os tipos de questões e a sua estrutura.

Uma entrevista pode ser estruturada de três diferentes formas:

o Estrutura em pirâmide: iniciamos as entrevistas com perguntas mais especificas sobre o sistema e fecha-

mos com perguntas mais genéricas. Geralmente utilizadas com usuários mais relutantes;

o Estrutura em funil: iniciamos as entrevistas com perguntas mais genéricas sobre o sistema e fechamos com perguntas mais especificas. Geralmente utilizadas com usuários que tem uma relação mais afetiva com o assunto;

o Estrutura em diamante: esta estrutura combina as duas estruturas anteriores e é utilizadas para manter a usuário entrevistado interessado no assunto e para isto se utiliza de perguntas variadas.

• **Prototipação**: é uma versão inicial de um sistema para experimentação. Permite aos utilizadores identificar os pontos fortes e fracos do sistema por ser algo concreto que pode ser criticado. Temos dois tipos de protótipos:

o Protótipos "Throw-away": ajudam o levantamento e desenvolvimento dos requisitos e suportam os requisitos mais difíceis de perceber;

o Protótipos Evolutivos: ajudam o desenvolvimento rápido de uma versão inicial do sistema e suportam os requisitos bem definidos e conhecidos.

Algumas das desvantagens da prototipação são os custos de aprendizagem e os custos de desenvolvimento.

• JAD (Joint Application Development): é uma técnica que permite a interação entre pessoas que necessitam tomar decisões que afetem múltiplas áreas de uma organização. Esta técnica envolve atividades de preparação para as reuniões, sessões de workshop com os participantes, agenda para as reuniões, participantes assumindo papeis de facilitador / condutor e documentador além de facilidades visuais, como a utilização de flipchart, quadro negro.

Esta técnica deve ser utilizada nos casos onde existe a necessidade de consenso entre diversos usuários, pois possibilita a todos os envolvidos ter uma visão global do sistema, ajudando a consolidar interesses de diversos usuários quanto ao sistema a ser desenvolvido.

O objetivo desta técnica é aumentar o comprometimento e participação do usuário e obter subsídios para elaborar o documento de Especificação de Requisitos para o sistema com consenso de todos de forma a ser uma validação formal dos requisitos do sistema.

O JAD é divido quem quatro fases. A **Figura 5** apresenta estas fases e os artefatos produzidos por cada uma delas.

A atividade de levantamento de requisitos não é trivial. Existe um conjunto grande e variado de fatores que a tornam complexa, por exemplo:

• Falta de conhecimento do usuário das suas reais necessidades

o Usuário com vaga noção do que precisa e do que um produto de software pode oferecer.

• Falta de conhecimento do desenvolvedor do domínio do problema

o Desenvolvedor sem conhecimento adequado do domínio, o que leva a decisões erradas.

• Domínio do processo de levantamento de requisitos pelos desenvolvedores

o Desenvolvedor não ouve o que os usuários têm a dizer e força suas próprias visões e interpretações.

• Comunicação inadequada entre os desenvolvedores e usuários

o Usuários incapazes de expressar suas necessidades apropriadamente;

o Significados diferentes a termos comuns.

• Dificuldade do usuário tomar decisões

o Falta de entendimento sobre as conseqüências das decisões ou as alternativas possíveis.

• Problemas de comportamento

o O levantamento de requisitos é um processo social;

o Conflitos e ambigüidades nos papéis que os usuários e desenvolvedores desempenham.

• Questões técnicas

o Complexidade crescente dos sistemas atuais.

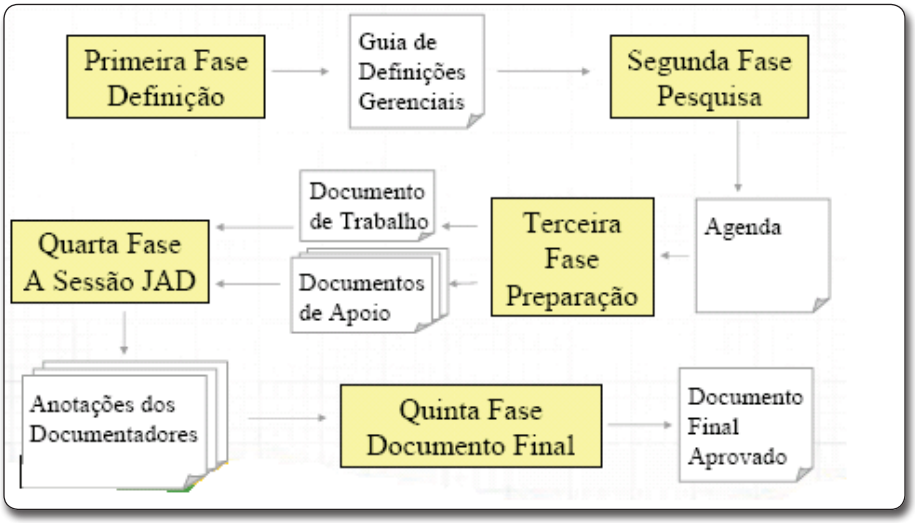


Figura 5. Fases da tecnica para levantamento de requisitos JAD.

Registro

Uma vez identificados e negociados, os requisitos devem ser documentados para que possam servir de base para o restante do processo de desenvolvimento.

Entre os muitos problemas que enfrentamos na documentação de requisitos, certamente, administrar o grande volume de informações gerado pelo processo de requisitos é um dos principais.

Os requisitos são documentados em um nível apropriado de detalhe. Em geral é produzido um documento de especificação de requisitos, de forma que todos os stakeholders possam entendê-lo.

O registro dos requisitos num documento próprio facilita o controle de alterações de todos os envolvidos na manutenção dos requisitos, bem como a geração de versões do documento e a facilidade de acesso por todos os envolvidos.

Verificação

Esta atividade examina a especificação do software, de forma a assegurar que todos os requisitos foram definidos sem ambigüidades, inconsistências ou omissões, detectando e corrigindo possíveis problemas ainda durante a fase de definição dos requisitos.

Neste contexto, sabe-se que o custo da correção de defeitos aumenta na medida em que o processo de desenvolvimento

progride. Revisões de artefatos de software têm se mostrado uma abordagem eficiente e de baixo custo para encontrar defeitos logo após terem sido introduzidos, reduzindo o retrabalho e melhorando a qualidade dos produtos. Não é em vão que modelos de maturidade de processo de software, como o CMMI e o MPS BR exigem a condução de revisões.

Um tipo particular de revisão de software são as inspeções. Inspeções possuem um processo de detecção de defeitos rigoroso e bem definido. Os benefícios de se aplicar inspeções são maiores para artefatos desenvolvidos no início do processo, como o documento de requisitos. Conheceremos um pouco mais sobre inspeção de software em um segundo artigo a ser publicado na segunda edição da Engenharia de Software Magazine.

Validação

A validação representa a atividade em que obtemos o aceite do cliente sob determinado artefato. No cenário de engenharia de requisitos, esta atividade significa aprovar junto ao cliente os requisitos que foram especificados. Embora aparentemente simples, esta atividade pode ser bastante dificultada pelo cliente ou mesmo por um processo de validação inadequado utilizado pela empresa.

Gerência de Requisitos

Requisitos são por natureza voláteis. Diversos fatores contribuem para sua instabilidade ao longo do tempo. Mudanças externas no ambiente (mudanças de legislação, mudanças no mercado, mudança no posicionamento estratégico da empresa), erros incorridos no processo de requisitos, entre outros. Todos esses fatores fazem com que seja necessário alterar os requisitos. Tais alterações precisam ser conduzidas de forma ordenada para que não se perca controle sobre o prazo e o custo do desenvolvimento.

Denominamos a atividade de administrar os requisitos ao longo do tempo de gerenciamento de requisitos. Os benefícios desta atividade são percebidos no médio prazo, sendo que são necessários investimentos no curto prazo. Assim, a atividade é, muitas vezes, tida como um fardo desnecessário à condução do processo. Contudo, sua não implementação faz com que as economias de curto prazo sejam logo suplantadas pelas despesas no longo prazo, verificadas com superação de custo e prazo nos projetos conduzidos.

Veremos a partir de agora algumas das atividades que devem ser consideradas durante a gerência dos requisitos.

Controle de Mudanças

Conforme foi citado anteriormente, os requisitos são voláteis e, portanto sofrem mudanças ao logo do tempo, para conduzir estas mudanças recomenda-se preparo e planejamento. Uma das maneiras bastante utilizadas para organizar estas mudanças é a "baseline" de requisitos que nos permite diferenciar o que era o requisito original, o que foi introduzido e o que foi descartado. Além disto, é interessante estabelecer um único canal para controle de mudanças, bem como utilizar um sistema para este controle.

Apresentamos a seguir uma sugestão de passos que devem ser seguidos para um processo de controle de mudança:

- Checar validade da solicitação de mudança;
- Identificar os requisitos diretamente

afetados com a mudança;

• Identificar dependências entre requisitos para buscar os requisitos afetados indiretamente;

• Assegurar com solicitante a mudança a ser realizada;

• Estimar custos da mudança;

• Obter acordo com usuário sobre o custo da mudança.

Após a realização desta análise, podemos aceitar ou rejeitar a mudança, conforme os impactos e custos que ela pode ter no sistema.

Gerência de Configuração

Durante o ciclo de vida do desenvolvimento, o software passa por uma série de modificações, desde a especificação dos requisitos até a implantação do sistema. A gerência de configuração de software existe no intuito de definir critérios que permitam realizar tais modificações mantendo-se a consistência e a integridade do software com as especificações, minimizando problemas decorrentes ao processo de desenvolvimento, através de um controle sistemático sobre as modificações.

A criação de um plano de gerência de configuração consiste em estabelecer normas, ferramentas e templates que permitam gerenciar de maneira satisfatória os itens de configuração de um sistema, que são definidos por Pressman como: "cada um dos elementos de informação que são criados durante o desenvolvimento de um produto de software, ou que para este desenvolvimento sejam necessários, que são identificados de maneira única e cuja evolução é passível de rastreamento".

Em cada fase do processo de desenvolvimento, um conjunto bem definido de itens de configuração deve ser estabelecido. A este conjunto é dado o nome de baselines. Estas baselines servem como marco no processo de desenvolvimento, pois ao final de cada fase é estabelecida uma nova baseline e, portanto, todos os itens de configuração estão sob total controle de seus desenvolvedores. Desta forma, nesta baseline é guardada "uma foto" dos artefatos criados até aquele momento:

• tornando mais fácil realizar modificações nos artefatos de maneira controlada;

• possibilitando ter um controle sistemático sobre todos os itens de configuração abordados em cada fase do ciclo de vida do software, e;

• tornando possível que sejam realizadas, mais facilmente, avaliações sobre seu grau de evolução.

Rastreabilidade

No centro da atividade de gerenciamento de requisitos está a rastreabilidade. Esta é definida como a habilidade de se acompanhar a vida de um requisito em ambas as direções (por exemplo: partindo de requisitos e chegando a projeto ou, partindo de projeto e chegando a requisitos) do processo de software e durante todo o seu ciclo de vida.

A dificuldade envolvida com a rastreabilidade está no grande volume de informações gerado. As informações do processo de requisitos devem ser catalogadas e associadas aos outros elementos de forma que possam ser referenciadas através dos diversos itens de informação registrados. É um trabalho extenso que, sem o auxílio de ferramentas adequadas, muito provavelmente não poderá ser feito

Para implementar a rastreabilidade, usualmente é confeccionado em conjunto com a especificação de requisitos um artefato chamado matriz de rastreabilidade, que tem como objetivo mapear os rastros dos requisitos descritos na especificação.

Os rastros dos requisitos podem ser de dois tipos:

• Pré-rastreabilidade: os rastros (artefatos: plano de negocio, atas de reunião com o usuário) que fundamentaram a criação do requisito.

• Pós-rastreabilidade: os rastros (artefatos: código, documentos) que se formaram a partir do requisito criado.

Gerência de Qualidade de Requisitos

Segundo o padrão IEEE 830, devemos considerar alguns critérios de qualidade ao trabalharmos com requisitos:

• Correção: um documento de requisitos é considerado correto se todos os requisitos representam algo que deve estar presente no sistema que está sendo desenvolvido, ou seja, os requisitos reais do usuário devem coincidir com os requisitos identificados. Esta não é uma tarefa trivial e parte de seu sucesso está associada a uma boa atividade de validação dos requisitos.

• Não ambigüidade: um conjunto de requisitos é não ambíguo quando somente pode ser interpretado por todos os envolvidos em um projeto de uma única maneira.

• Completude: um conjunto de requisitos é dito completo quando descreve todas as demandas de interesse dos usuários. Estas demandas incluem requisitos funcionais, de desempenho, restrições, atributos e interfaces externas.

• Consistência: um conjunto de requisitos é dito consistente se nenhum subconjunto destes requisitos entra em conflito com os demais requisitos do sistema.

• Verificabilidade: um requisito é verificável se existe uma forma efetiva, em termos de tempo e custo, para que pessoas ou ferramentas indiquem se um sistema cumpre o requisito (IEEE). Em quase todas as situações, é difícil provar de forma conclusiva que um requisito é cumprido por um software. Entretanto, escrever bem o requisito pode ajudar a aumentar a confiança na avaliação.

• Modificabilidade: um conjunto de requisitos é modificável quando seu estilo e estrutura é tal que as alterações podem ser realizadas de forma simples e consistente com os demais requisitos.

A gerência, neste cenário, é responsável por manter uma infra-estrutura necessária para atividades de verificação que tornem possível investigarmos a qualidade dos requisitos que estamos definindo.

Conclusão

A engenharia de requisitos define, sem dúvida, um dos mais importantes conjuntos de atividades a serem realizadas em projetos de desenvolvimento de software. Embora não garanta a qualidade dos produtos gerados, é um pré-requisitos básico para que obtenhamos sucesso no desenvolvimento do projeto. Este artigo é apenas uma breve introduç ãoao tema, que deverá ser bastante explorado em edições futuras da Engenharia de Software Magazine. ●

Introdução a Teste de Software

Arilo Cláudio Dias Neto

(ariloclaudio@gmail.com)

É Bacharel em Ciência da Computação formado na Universidade Federal do Amazonas, Mestre em Engenharia de Sistemas e Computação formado na COPPE/UFRJ, e atualmente está cursando doutorado na área de Engenharia de Software da COPPE/UFRJ. Possui 5 anos de experiência em análise, desenvolvimento e teste de software. É editor técnico das Revistas SQL Magazine e WebMobile, gerenciadas pelo Grupo DevMedia.

Teste de software é o processo de execução de um produto para determinar se ele atingiu suas especificações e funcionou corretamente no ambiente para o qual foi projetado. O seu objetivo é revelar falhas em um produto, para que as causas dessas falhas sejam identificadas e possam ser corrigidas pela equipe de desenvolvimento antes da entrega final. Por conta dessa característica das atividades de teste, dizemos que sua natureza é "destrutiva", e não "construtiva", pois visa ao aumento da confiança de um produto através da exposição de seus problemas, porém antes de sua entrega ao usuário final.

O conceito de teste de software pode ser compreendido através de uma visão intuitiva ou mesmo de uma maneira formal. Existem atualmente várias definições para esse conceito. De uma forma simples, testar um software significa verificar através de uma execução controlada se o seu comportamento corre de acordo com o especificado. O objetivo principal desta tarefa é revelar o número máximo de falhas dispondo do mínimo de esforço, ou seja, mostrar aos que desenvolvem se os resultados estão ou não de acordo com os padrões estabelecidos.

Ao longo deste artigo, iremos discutir os principais conceitos relacionados às atividades de teste, as principais técnicas e critérios de teste que podem ser utilizados para verificação ou validação de um produto, assim como exemplos práticos da aplicação de cada tipo de técnica ou critério de teste.

Conceitos básicos associados a Teste de Software

Antes de iniciarmos uma discussão sobre teste de software precisamos esclarecer alguns conceitos relacionados a essa atividade. Inicialmente, precisamos conhecer a diferença entre Defeitos, Erros e Falhas. As definições que iremos usar aqui seguem a terminologia padrão para Engenharia de Software do IEEE – Institute of Electrical and Electronics

Engineers – (IEEE 610, 1990).

• Defeito é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta. Por exemplo, uma instrução ou comando incorreto.

• Erro é uma manifestação concreta de um defeito num artefato de software. Diferença entre o valor obtido e o valor esperado, ou seja, qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa constitui um erro.

• Falha é o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversos erros e alguns erros podem nunca causar uma falha.

A **Figura 1** expressa a diferença entre esses conceitos. Defeitos fazem parte do universo físico (a aplicação propriamente dita) e são causados por pessoas, por exemplo, através do mal uso de uma tecnologia. Defeitos podem ocasionar a manifestação de erros em um produto, ou seja, a construção de um software de forma diferente ao que foi especificado (universo de informação). Por fim, os erros geram falhas, que são comportamentos inesperados em um software que afetam diretamente o usuário final da aplicação (universo do usuário) e pode inviabilizar a utilização de um software.

Dessa forma, ressaltamos que teste de software revela simplesmente falhas em um produto. Após a execução dos testes é necessária a execução de um processo de depuração para a identificação e correção dos defeitos que originaram essa falha, ou seja, Depurar não é Testar!

A atividade de teste é composta por alguns elementos essenciais que auxiliam na formalização desta atividade. Esses elementos são os seguintes:

• **Caso de Teste.** descreve uma condição particular a ser testada e é composto por valores de entrada, restrições para a sua execução e um resultado ou comportamento esperado (CRAIG e JASKIEL, 2002).

• **Procedimento de Teste.** é uma descrição dos passos necessários para executar um caso (ou um grupo de casos) de teste (CRAIG e JASKIEL, 2002).

• **Critério de Teste:** serve para selecionar e avaliar casos de teste de forma a aumentar as possibilidades de provocar falhas ou, quando isso não ocorre, estabelecer um nível elevado de confiança na correção do produto (ROCHA et al., 2001). Os critérios de teste podem ser utilizados como:

o **Critério de Cobertura dos Testes:** permitem a identificação de partes do programa que devem ser executadas para garantir a qualidade do software e indicar quando o mesmo foi suficientemente testado (RAPPS e WEYUKER, 1982). Ou seja, determinar o percentual de elementos necessários por um critério de teste que foram executados pelo conjunto de casos de teste.

o **Critério de Adequação de Casos de Teste:** Quando, a partir de um conjunto de casos de teste T qualquer, ele é utilizado para verificar se T satisfaz os requisitos de teste estabelecidos pelo critério. Ou seja, este critério avalia se os casos de teste definidos são suficientes ou não para avaliação de um produto ou uma função (ROCHA et al., 2001).

o **Critério de Geração de Casos de Teste:** quando o critério é utilizado para gerar um conjunto de casos de teste T adequado para um produto ou função, ou seja, este critério define as regras e diretrizes para geração dos casos de teste de um produto que esteja de acordo com o critério de adequação definido anteriormente (ROCHA et al., 2001).

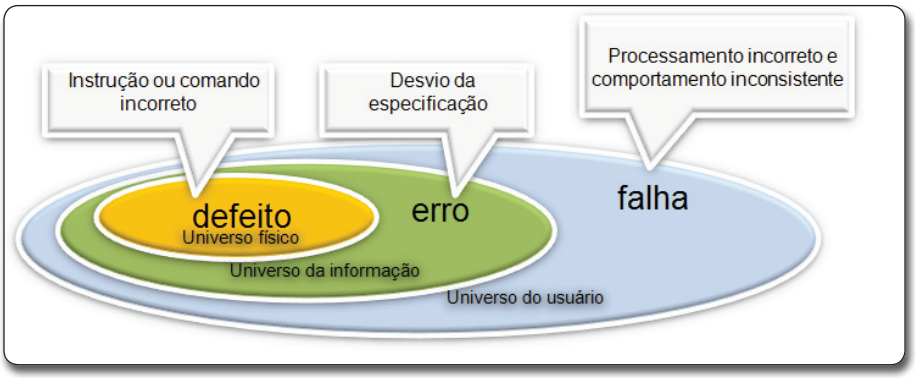


Figura 1. Defeito x erro x falha (Uma versão similar pode ser obtida em *http://www.projectcartoon.com/cartoon/611*)

Definidos os elementos básicos associados aos testes de software, iremos a seguir discutir a origem dos defeitos em um software.

Defeitos no desenvolvimento de software

No processo de desenvolvimento de software, todos os defeitos são humanos e, apesar do uso dos melhores métodos de desenvolvimento, ferramentas ou profissionais, permanecem presentes nos produtos, o que torna a atividade de teste fundamental durante o desenvolvimento de um software. Já vimos que esta atividade corresponde ao último recurso para avaliação do produto antes da sua entrega ao usuário final.

O tamanho do projeto a ser desenvolvido e a quantidade de pessoas envolvidas no processo são dois possíveis fatores que aumentam a complexidade dessa tarefa, e consequentemente aumentam a probabilidade de defeitos. Assim, a ocorrência de falhas é inevitável. Mas o que significa dizer que um programa falhou? Basicamente significa que o funcionamento do programa não está de acordo com o esperado pelo usuário. Por exemplo, quando um usuário da linha de produção efetua consultas no sistema das quais só a gerência deveria ter acesso. Esse tipo de falha pode ser originado por diversos motivos:

• A especificação pode estar errada ou incompleta;

• A especificação pode conter requisitos impossíveis de serem implementados devido a limitações de hardware ou software;

• A base de dados pode estar organizada de forma que não seja permitido distinguir os tipos de usuário;

• Pode ser que haja um defeito no algoritmo de controle dos usuários.

Os defeitos normalmente são introduzidos na transformação de informações entre as diferentes fases do ciclo de desenvolvimento de um software. Vamos seguir um exemplo simples de ciclo de vida de desenvolvimento de software: os requisitos expressos pelo cliente são relatados textualmente em um documento de especificação de requisitos. Esse documento é então transformado

em casos de uso, que por sua vez foi o artefato de entrada para o projeto do software e definição de sua arquitetura utilizando diagramas de classes da UML. Em seguida, esses modelos de projetos foram usados para a construção do software em uma linguagem que não segue o paradigma orientado a objetos. Observe que durante esse período uma série de transformações foi realizada até chegarmos ao produto final. Nesse meio tempo, defeitos podem ter sido inseridos. A **Figura 2** expressa exatamente a metáfora discutida nesse parágrafo.

Essa série de transformações resultou na necessidade de realizar testes em diferentes níveis, visando avaliar o software em diferentes perspectivas de acordo com o produto gerado em cada fase do ciclo de vida de desenvolvimento de um software. Esse será o foco da seção seguinte.

Níveis de teste de software

O planejamento dos testes deve ocorrer em diferentes níveis e em paralelo ao desenvolvimento do software (**Figura 3**). Buscando informação no Livro "Qualidade de Software – Teoria e Prática" (ROCHA et al., 2001), definimos que os principais níveis de teste de software são:

- **Teste de Unidade:** também conhecido como testes unitários. Tem por objetivo explorar a menor unidade do projeto, procurando provocar falhas ocasionadas por defeitos de lógica e de implementação em cada módulo, separadamente. O universo alvo desse tipo de teste são os métodos dos objetos ou mesmo pequenos trechos de código.
- **Teste de Integração:** visa provocar falhas associadas às interfaces entre os módulos quando esses são integrados para construir a estrutura do software que foi estabelecida na fase de projeto.

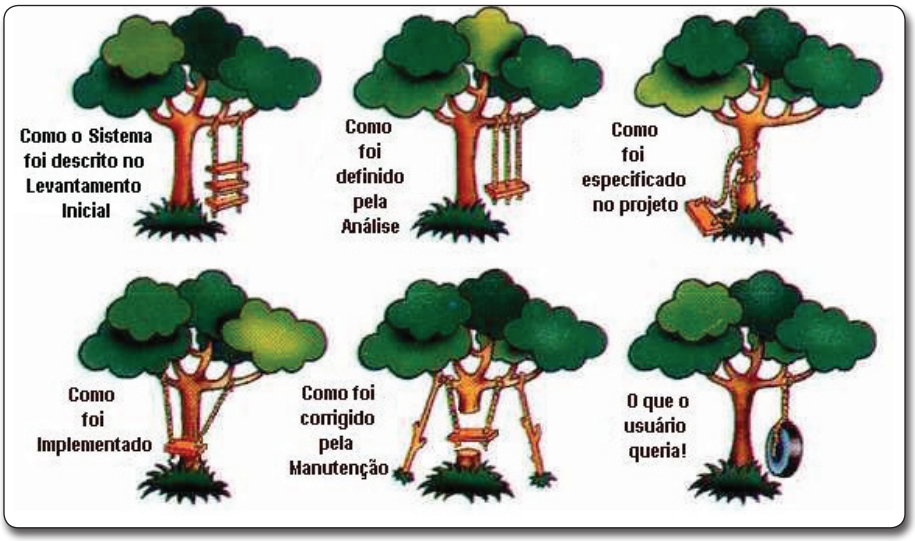


Figura 2. Diferentes Interpretações ao longo do ciclo de desenvolvimento de um software

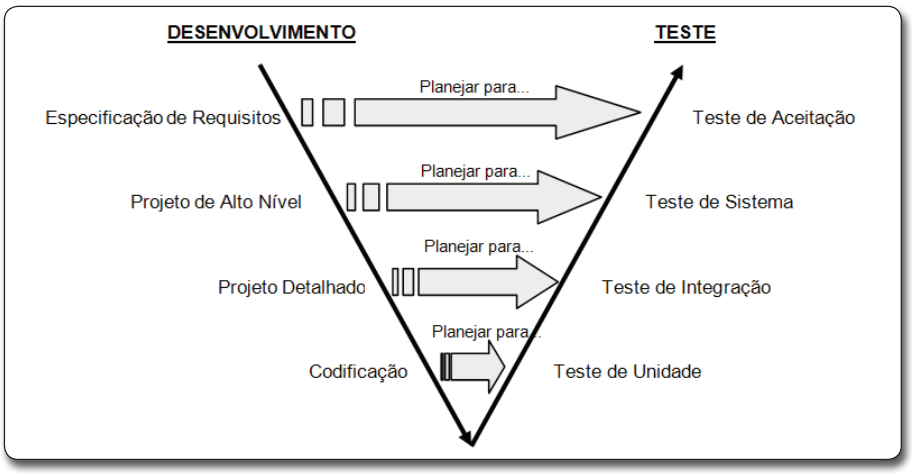


Figura 3. Modelo V descrevendo o paralelismo entre as atividades de desenvolvimento e teste de software (CRAIG e JASKIEL, 2002)

- **Teste de Sistema:** avalia o software em busca de falhas por meio da utilização do mesmo, como se fosse um usuário final. Dessa maneira, os testes são executados nos mesmos ambientes, com as mesmas condições e com os mesmos dados de entrada que um usuário utilizaria no seu dia-a-dia de manipulação do software. Verifica se o produto satisfaz seus requisitos.
- **Teste de Aceitação:** são realizados geralmente por um restrito grupo de usuários finais do sistema. Esses simulam operações de rotina do sistema de modo a verificar se seu comportamento está de acordo com o solicitado.
- **Teste de Regressão:** Teste de regressão não corresponde a um nível de teste, mas é uma estratégia importante para redução de "efeitos colaterais". Consiste em se aplicar, a cada nova versão do software ou a cada ciclo, todos os testes que já foram aplicados nas versões ou ciclos de teste anteriores do sistema. Pode ser aplicado em qualquer nível de teste.

Dessa forma, seguindo a Figura 3, o planejamento e projeto dos testes devem ocorrer de cima para baixo, ou seja:

1. Inicialmente é planejado o teste de aceitação a partir do documento de requisitos;
2. Após isso é planejado o teste de sistema a partir do projeto de alto nível do software;
3. Em seguida ocorre o planejamento dos testes de integração a partir o projeto detalhado;
4. E por fim, o planejamento dos testes a partir da codificação.

Já a execução ocorre no sentido inverso.

Conhecidos os diferentes níveis de teste, a partir da próxima seção descreveremos as principais técnicas de teste de software existentes que podemos aplicar nos diferentes níveis abordados.

Técnicas de teste de software

Atualmente existem muitas maneiras de se testar um software. Mesmo assim, existem as técnicas que sempre foram muito utilizadas em sistemas desenvolvidos sobre linguagens estruturadas que ainda hoje têm grande valia para os sistemas orientados a objeto. Apesar de os paradigmas de desenvolvimento serem diferentes, o objetivo principal destas técnicas continua a ser o mesmo:

encontrar falhas no software.

As técnicas de teste são classificadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste. Elas contemplam diferentes perspectivas do software e impõe-se a necessidade de se estabelecer uma estratégia de teste que contemple as vantagens e os aspectos complementares dessas técnicas. As técnicas existentes são: técnica funcional e estrutural.

A seguir conheceremos um pouco mais sobre cada técnica, mas iremos enfatizar alguns critérios específicos para a técnica funcional.

Técnica Estrutural (ou teste caixa-branca)

Técnica de teste que avalia o comportamento interno do componente de software (**Figura 4**). Essa técnica trabalha diretamente sobre o código fonte do componente de software para avaliar aspectos tais como: teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos (PRESSMAN, 2005).

Os aspectos avaliados nesta técnica de teste dependerão da complexidade e da tecnologia que determinarem a construção do componente de software, cabendo, portanto, avaliação de outros aspectos além dos citados anteriormente. O testador tem acesso ao código fonte da aplicação e pode construir códigos para efetuar a ligação de bibliotecas e componentes.

Este tipo de teste é desenvolvido analisando-se o código fonte e elaborando-se casos de teste que cubram todas as possibilidades do componente de software. Dessa maneira, todas as variações originadas por estruturas de condições são testadas. A **Listagem 1** apresenta um código fonte, extraído de (BARBOSA et al., 2000) que descreve um programa de exemplo que deve validar um identificador digitado como parâmetro, e a **Figura 5** apresenta o grafo de programa extraído a partir desse código, também extraído de (BARBOSA et al., 2000). A partir do grafo deve ser escolhido algum critério baseado em algum algoritmo de busca em grafo (exemplo: visitar todos os nós, arcos ou caminhos) para geração dos casos de teste estruturais para o programa (PFLEEGER, 2004).

Um exemplo bem prático desta técnica de teste é o uso da ferramenta livre JUnit para desenvolvimento de casos de teste para avaliar classes ou métodos desenvolvidos na linguagem Java. A técnica de teste de Estrutural é recomendada para os níveis de Teste da Unidade e Teste da Integração, cuja responsabilidade principal fica a cargo dos desenvolvedores do software, que são profissionais que conhecem bem o código-fonte desenvolvido e dessa forma conseguem planejar os casos de teste com maior facilidade. Dessa forma, podemos auxiliar na redução dos problemas existentes nas pequenas funções ou unidades que compõem um software.

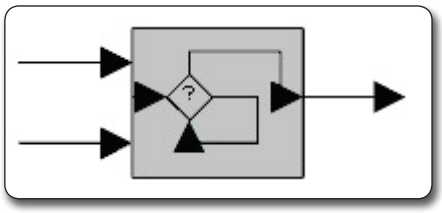


Figura 4. Técnica de Teste Estrutural.

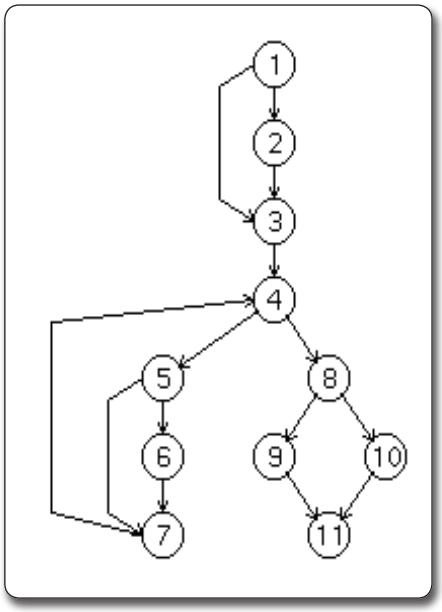


Figura 5. Grafo de Programa (Identifier.c) (BARBOSA et al., 2000).

Teste Funcional (ou teste caixa-preta)

Técnica de teste em que o componente de software a ser testado é abordado como se fosse uma caixa-preta, ou seja, não se considera o comportamento interno do mesmo (Figura 6). Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um re-

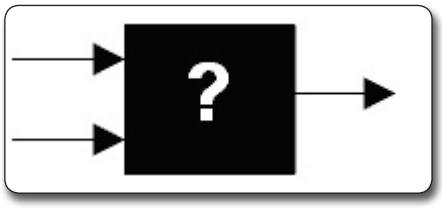


Figura 6. Técnica de Teste Funcional.

Listagem 1. Código fonte do programa identifier.c (BARBOSA et al., 2000)

```
/* 01 */  {
/* 01 */  char  achar;
/* 01 */  int   length, valid_id;
/* 01 */  length = 0;
/* 01 */  printf ("Digite um possível identificador\n");
/* 01 */  printf ("seguido por <ENTER>: ");
/* 01 */  achar = fgetc (stdin);
/* 01 */  valid_id = valid_starter (achar);
/* 01 */  if (valid_id)
/* 02 */            length = 1;
/* 03 */  achar = fgetc (stdin);
/* 04 */  while (achar != '\n')
/* 05 */  {
/* 05 */            if (!(valid_follower (achar)))
/* 06 */                      valid_id = 0;
/* 07 */            length++;
/* 07 */            achar = fgetc (stdin);
/* 07 */  }
/* 08 */  if (valid_id && (length >= 1) && (length < 6) )
/* 09 */            printf ("Valido\n");
/* 10 */  else
/* 10 */            printf ("Invalido\n");
/* 11 */   }
```

sultado esperado previamente conhecido. Haverá sucesso no teste se o resultado obtido for igual ao resultado esperado. O componente de software a ser testado pode ser um método, uma função interna, um programa, um componente, um conjunto de programas e/ou componentes ou mesmo uma funcionalidade. A técnica de teste funcional é aplicável a todos os níveis de teste (PRESSMAN, 2005).

Um conjunto de critérios de teste pode ser aplicado aos testes funcionais. A seguir conheceremos alguns deles.

Particionamento em classes de equivalência

Esse critério divide o domínio de entrada de um programa em classes de equivalência, a partir das quais os casos de teste são derivados. Ele tem por objetivo minimizar o número de casos de teste, selecionando apenas um caso de teste de cada classe, pois em princípio todos os elementos de uma classe devem se comportar de maneira equivalente. Eventualmente, pode-se também considerar o domínio de saída para a definição das classes de equivalência (ROCHA et al., 2001).

Uma classe de equivalência representa um conjunto de estados válidos e inválidos para uma condição de entrada. Tipicamente uma condição de entrada pode ser um valor numérico específico, uma faixa de valores, um conjunto de valores relacionados, ou uma condição lógica. As seguintes diretrizes podem ser aplicadas:

- Se uma condição de entrada especifica uma faixa de valores ou requer um valor específico, uma classe de equivalência válida (dentro do limite) e duas inválidas (acima e abaixo dos limites) são definidas.
- Se uma condição de entrada especifica um membro de um conjunto ou é lógica, uma classe de equivalência válida e uma inválida são definidas.

Devemos seguir tais passos para geração dos testes usando este critério:

1. Identificar classes de equivalência (é um processo heurístico)
   - condição de entrada
   - válidas e inválidas
2. Definir os casos de teste
   - enumeram-se as classes de equivalência
   - casos de teste para as classes válidas
   - casos de teste para as classes inválidas

Em (BARBOSA et al., 2000) é apresentado a aplicação do critério de particionamento por equivalência para o programa identifier.c. Iremos apresentá-lo como exemplo do uso deste critério de teste. Relembrando, o programa deve determinar se um identificador é válido ou não.

> "Um identificador válido deve começar com uma letra e conter apenas letras ou dígitos. Além disso, deve ter no mínimo 1 caractere e no máximo 6 caracteres de comprimento. Exemplo: "abc12" (válido), "cont*1" (inválido), "1soma" (inválido) e "a123456" (inválido)."

O primeiro passo é a identificação das classes de equivalência. Isso está descrito na Tabela 1.

A partir disso, conseguimos especificar quais serão os casos de teste necessários. Para ser válido, um identificador deve atender às condições (1), (3) e (5), logo é necessário um caso de teste válido que cubra todas essas condições. Alem disso, será necessário um caso de teste para cada classe inválida: (2), (4) e (6). Assim, o conjunto mínimo é composto por quatro casos de teste, sendo uma das opções: T0 = {(a1,Válido), (2B3, Inválido), (Z-12, Inválido), (A1b2C3d, Inválido)}.

Análise do valor limite

Por razões não completamente identificadas, um grande número de erros tende a ocorrer nos limites do domínio de entrada invés de no "centro". Esse critério de teste explora os limites dos valores de cada classe de equivalência para preparar os casos de teste (Pressman, 2005).

Se uma condição de entrada especifica uma faixa de valores limitada em a e b, casos de teste devem ser projetados com valores a e b e imediatamente acima e abaixo de a e b. Exemplo: Intervalo = {1..10}; Casos de Teste → {1, 10, 0,11}.

Como exemplo, extraído de (BARBOSA et al., 2000), iremos considerar a seguinte situação:

> "... o cálculo do desconto por dependente é feito da seguinte forma: a entrada é a idade do dependente que deve estar restrita ao intervalo [0..24]. Para dependentes até 12 anos (inclusive) o desconto é de 15%. Entre 13 e 18 (inclusive) o desconto é de 12%. Dos 19 aos 21 (inclusive) o desconto é de 5% e dos 22 aos 24 de 3%..."

Aplicando o teste de valor limite

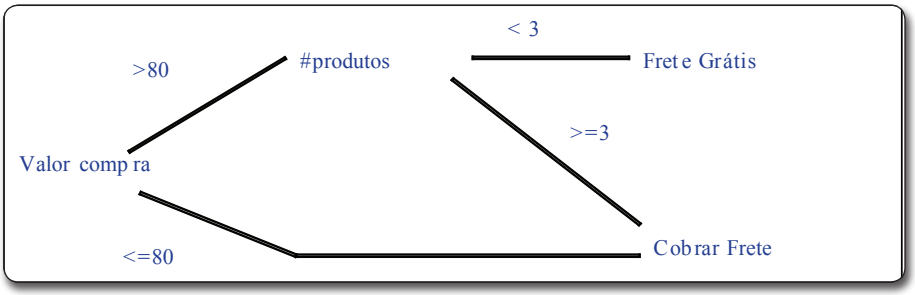


Figura 7. Árvore de Decisão – Grafo de Causa-Efeito.

Condições de Entrada	Classes	Classes
Tamanho *t* do identificador	(1) $1 \leq t \leq 6$	(2) $t > 6$
Primeiro caractere *c* é uma letra	(3) Sim	(4) Não
Só contém caracteres válidos	(5) Sim	(6) Não

Tabela 1. Classes de Equivalência do programa identifier.c.

Causa	Valor da compra	> 60	> 60	<= 60
	#Produtos	< 3	>= 3	--
Efeito	Cobrar frete		V	V
	Frete grátis	V		

Tabela 2. Tabela de decisão para o programa de compra pela Internet.

convencional serão obtidos casos de teste semelhantes a este: {-1,0,12,13,18,19,21,22,24,25}.

Grafo de causa-efeito

Esse critério de teste verifica o efeito combinado de dados de entrada. As causas (condições de entrada) e os efeitos (ações) são identificados e combinados em um grafo a partir do qual é montada uma tabela de decisão, e a partir desta, são derivados os casos de teste e as saídas (ROCHA et al., 2001).

Esse critério é baseado em quatro passos, que exemplificaremos utilizando o exemplo, também extraído de (BARBOSA et al., 2000):

> "Em um programa de compras pela Internet, a cobrança ou não do frete é definida seguindo tal regra: Se o valor da compra for maior que R$ 60,00 e foram comprados menos que 3 produtos, o frete é gratuito. Caso contrário, o frete deverá ser cobrado."

1. Para cada módulo, **Causas** (condições de entrada) e **efeitos** (ações realizadas às diferentes condições de entrada) são relacionados, atribuindo-se um identificador para cada um.
   - Causa: valor da compra > 60 e #produtos < 3
   - Efeito: frete grátis
2. Em seguida, um grafo de causa-efeito (árvore de decisão) é desenhado (**Figura 7**).
3. Neste ponto, transforma-se o grafo numa tabela de decisão (**Tabela 2**).
4. As regras da tabela de decisão são, então, convertidas em casos de teste.

Para a elaboração dos casos de teste, devemos seguir todas as regras extraídas da tabela. Esse critério deve ser combinado com os dois outros já apresentados neste artigo para a criação de casos de teste válidos (extraídos das regras) e inválidos (valores foras da faixa definida). Os casos de teste definidos a seguir refletem somente as regras extraídas da tabela de decisão. Fica como exercício pensar nos casos de teste inválidos para este problema.

- *Casos de Teste (valor, qtd produtos, resultado esperado) = {(61,2,"frete grátis"); (61,3,"cobrar frete"); (60, qualquer quantidade,"cobrar frete")}.*

Outras técnicas

Outras técnicas de teste podem e devem ser utilizadas de acordo com necessidades de negócio ou restrições tecnológicas. Destacam-se as seguintes técnicas: teste de desempenho, teste de usabilidade, teste de carga, teste de stress, teste de confiabilidade e teste de recuperação. Alguns autores chegam a definir uma técnica de teste caixa cinza, que seria um mesclado do uso das técnicas de caixa preta e caixa branca, mas, como toda execução de trabalho relacionado à atividade de teste utilizará simultaneamente mais de uma técnica de teste, é recomendável que se fixem os conceitos primários de cada técnica.

Conclusões

O teste de software é uma das atividades mais custosas do processo de desenvolvimento de software, pois pode envolver uma quantidade significativa dos recursos de um projeto. O rigor e o custo associado a esta atividade dependem principalmente da criticalidade da aplicação a ser desenvolvida. Diferentes categorias de aplicações requerem uma preocupação diferenciada com as atividades de teste.

Um ponto bastante importante para a viabilização da aplicação de teste de software é a utilização de uma infraestrutura adequada. Realizar testes não consiste simplesmente na geração e execução de casos de teste, mas envolvem também questões de planejamento, gerenciamento e análise de resultados. Apoio ferramental para qualquer atividade do processo de teste é importante como mecanismo para redução de esforço associado à tarefa em questão, seja ela planejamento, projeto ou execução dos testes. Após ter sua estratégia de teste definida, tente buscar por ferramentas que se encaixem na sua estratégia. Isso pode reduzir significantemente o esforço de tal tarefa.

Além disso, é importante ressaltar que diferentes tipos de aplicações possuem diferentes técnicas de teste a serem aplicadas, ou seja, testar uma aplicação web envolve passos diferenciados em comparação aos testes de um sistema embarcado. Cada tipo de aplicação possui características especificas que devem ser consideradas no momento da realização dos testes. O conjunto de técnicas apresentado neste artigo cobre diversas características comuns a muitas categorias de software, mas não é completo.

Para finalizar, podemos destacar outros pontos importantes relacionados às atividades de teste que podemos abordar em próximos artigos, tais como processo de teste de software, planejamento e controle dos testes e teste de software para categorias específicas de software, como aplicações web. Até a próxima!

Agradecimentos

Agradecemos aos professores José Carlos Maldonado e Ellen Barbosa por terem gentilmente autorizado a publicação deste material, cujos exemplos utilizados estão fundamentados em seus trabalhos.

Referências

BARBOSA, E.; MALDONADO, J.C.; VINCENZI, A.M.R.; DELAMARO, M.E; SOUZA, S.R.S. e JINO, M.. "Introdução ao Teste de Software. XIV Simpósio Brasileiro de Engenharia de Software", *2000*.

CRAIG, R.D., JASKIEL, S. P., "Systematic Software Testing", Artech House Publishers, Boston, *2002*.

IEEE Standard 610-1990: IEEE Standard Glossary of Software Engineering Terminology, IEEE Press.

PFLEEGER, S. L., "Engenharia de Software: Teoria e Prática", Prentice Hall- Cap. 08, *2004*.

PRESSMAN, R. S., "Software Engineering: A Practitioner's Approach", McGraw-Hill, 6th ed, Nova York, NY, *2005*.

RAPPS, S., WEYUKER, E.J., "Data Flow analysis techniques for test data selection", In: International Conference on Software Engineering, p. 272-278, Tokio, Sep. 1982.

ROCHA, A. R. C., MALDONADO, J. C., WEBER, K. C. et al., "Qualidade de software – Teoria e prática", Prentice Hall, São Paulo, 2001.

Gestão de defeitos

Ferramentas Open Source e melhores práticas na gestão de defeitos

Cristiano Caetano
c_caetano@hotmail.com
É certificado CBTS pela ALATS. Consultor de teste de software sênior com mais de 10 anos de experiência, já trabalhou na área de qualidade e teste de software para grandes empresas como Zero G, DELL e HP Invent. É colunista na área de Teste e Qualidade de software do site linhadecodigo.com.br e autor dos livros "CVS: Controle de Versões e Desenvolvimento Colaborativo de Software" e "Automação e Gerenciamento de Testes: Aumentando a Produtividade com as Principais Soluções Open Source e Gratuitas". O autor mantém um blog sobre testes e qualidade de software, confira no seguinte endereço: *http://spaces.msn.com/softwarequality/*.

O principal objetivo do teste de software é medir o nível de qualidade de um sistema, seja a aplicação executável ou os artefatos utilizados na sua construção. A qualidade de um sistema pode ser medida, essencialmente, pelo número de falhas encontradas durante a execução dos testes.

Falha, nesse contexto, é a conseqüência de um erro, defeito ou engano. Ou seja, é o desvio entre o que foi solicitado pelo usuário por meio dos requisitos e o comportamento apresentado pela aplicação executável. Em virtude da complexidade e tamanho de um sistema ou para atender normas de qualidade ou processos de maturidade, se faz necessário utilizar um processo de gestão de defeitos integrado ao ciclo de vida de desenvolvimento e teste.

Neste contexto, entender as atividades de um processo de gestão de defeitos e escolher a ferramenta adequada para automatizar a gestão do ciclo de vida de um defeito são tarefas fundamentais, como veremos mais adiante.

Neste artigo, você conhecerá os conceitos, atividades e terminologia de um processo de gestão de defeitos. Também será apresentado um exemplo prático utilizando o Mantis, ferramenta Open Source para gestão de defeitos. E, por fim, serão apresentadas outras alternativas Open Source e comerciais caso o Mantis não atenda as suas necessidades.

Processo de Gestão de defeitos

Um processo de gestão de defeitos tem o objetivo de definir práticas para prevenir os defeitos e minimizar os riscos de um projeto. A utilização de uma ferramenta automatizada, além de oferecer uma base comum para a entrada de informações, também oferece um meio para fomentar a integração entre o time de desenvolvimento e o time de testes. Além disso, por meio dos relatórios de gestão e métricas geradas por essas ferramentas, os gestores do projeto poderão promover a melhoria contínua do processo estabelecido.

Genericamente, o termo Erro (*Error*)

é utilizado para indicar uma diferença entre valor computado, observado ou medido em relação ao esperado. No entanto, o padrão IEEE 610.12-1990 (IEEE Standard Glossary of Software Engineering Terminology) distingue a terminologia da seguinte forma:

• Defeito (*Fault*): Passo, processo ou definição de dados incorretos. Por exemplo, uma instrução incorreta no código ou uma falta num artefato estático;

• Engano (*Mistake*): Ação humana que produz um resultado incorreto, como por exemplo, uma ação incorreta tomada pelo desenvolvedor ou analista;

• Falha (*Failure*): Desvio entre o resultado/comportamento apresentado pela aplicação em relação aos requisitos. A falha ocorre em conseqüência de um erro, defeito ou engano gerando um comportamento incorreto da aplicação.

Neste artigo, o termo defeito será usado de forma genérica, podendo representar um erro, engano, defeito ou falha. Segundo o livro, "Base de Conhecimento do CSTE" do QAI, os elementos chave de um processo de gestão de defeitos são (**Figura 1**):

• Prevenção de defeitos: Com base nos levantamento dos riscos críticos do projeto, devem ser promovidas ações de prevenção e planejamento de contingências para minimizar o impacto caso os riscos tornem-se problemas;

• Linha base entregável: Estabelecimento formal de linhas base (*baselines*) por meio da Gerência de Configuração de Software. Cada linha base deve determinar quais requisitos/artefatos serão liberados e submetidos ao teste;

• Identificação do defeito: Definição das técnicas necessárias para encontrar, reportar e classificar os defeitos, assim como, os critérios para reconhecê-los;

• Solução do defeito: Definição das atividades para a correção e posterior notificação da resolução do defeito. Muitas destas atividades são definidas pela Gerência de Configuração de Software para garantir o histórico e rastreamento das modificações por meio do controle de versões;

• Melhoria do processo: Análise das métricas e relatórios de gestão para entender a causa raiz dos problemas e promover a melhoria contínua do processo;

• Relatório de gestão: Geração de relatórios com dados relevantes para acompanhar o progresso dos testes e a qualidade do sistema, assim como, a geração de métricas para alimentar a atividade de melhoria do processo.

Ciclo de vida de um defeito

Como discutimos anteriormente, a qualidade de um sistema pode ser medida, essencialmente, pelo número de defeitos encontrados durante a execução dos testes.

Podemos afirmar que os defeitos são encontrados por meio da execução formal dos testes (testes estruturais ou funcionais), durante a utilização do sistema em produção ou, até mesmo, por acidente. A priori, podemos classificar os defeitos nas seguintes categorias:

• Faltante (*Missing*): O defeito ocorre em virtude da falta parcial ou total de um requisito;

• Errado (*Wrong*): O defeito ocorre porque o requisito foi implementado incorretamente;

• Acréscimo (*Extra*): O defeito ocorre em virtude de um comportamento ou elemento que foi implementado, mas foi não especificado no requisito.

Uma vez que o defeito for encontrado, seja por intenção ou por acidente, o próximo passo deverá ser o relato (ou reporte) desse defeito por meio de algum mecanismo estabelecido no processo de gestão de defeitos.

Este mecanismo poderá ser, desde uma simples planilha, até uma ferramenta automatizada. Por motivos óbvios, uma ferramenta automatizada e construída para esse propósito será, sem sombra de dúvida, muito mais eficiente do que uma simples planilha ou solução alternativa.

De qualquer forma, tão logo o defeito seja relatado, ele deverá ser submetido a um ciclo de vida pré-definido pelo processo de gestão de defeitos. Este ciclo de vida define os fluxos que o defeito deverá percorrer até o seu fechamento.

A **Figura 2** descreve genericamente um ciclo de vida de um defeito aderente ao processo de gestão de defeitos apresentado anteriormente. Devemos lembrar que esse é um exemplo didático, existem fluxos alternativos e papéis que não estão presentes nesta figura.

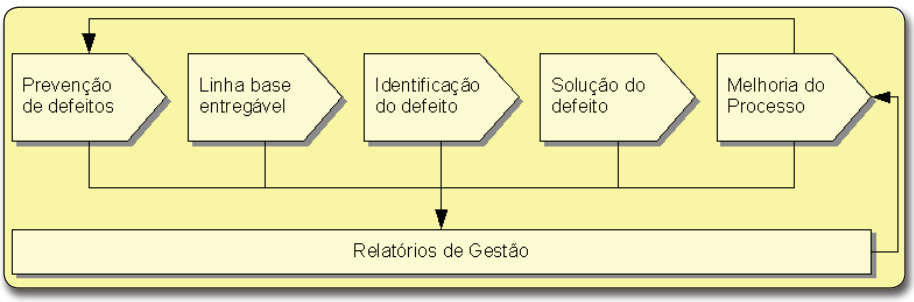


Figura 1. Elementos chave de um processo de gestão de defeitos.

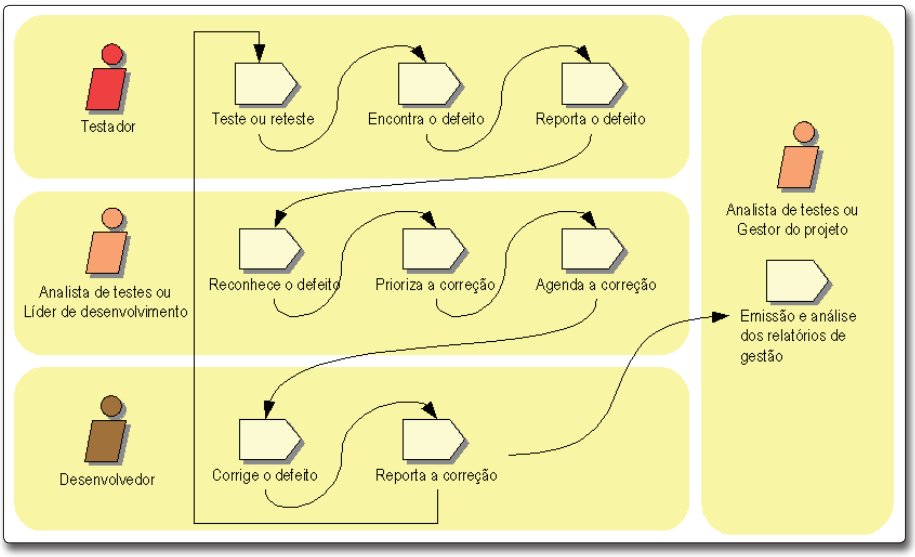


Figura 2. Ciclo de vida de um defeito genérico.

Recomendações para o relato de defeitos

Notamos que, muito embora o relato de um defeito seja um dos passos fundamentais do processo de gestão de defeitos, normalmente ele é relegado para segundo plano. A listagem abaixo apresenta as diretrizes que devem ser seguidas durante o relato de um defeito com base nas recomendações descritas no livro "Base de conhecimento em teste de software":

- Resumir: Descreva claramente o defeito, mas de forma resumida;
- Precisão: Certifique-se que o defeito identificado realmente é um desvio do comportamento esperado e não uma falha de entendimento;
- Neutralizar: Relate apenas os fatos, evitando manifestações de humor, emoção, etc;
- Generalizar: Procure entender o problema de forma genérica, em virtude de que este problema também pode acontecer em outras situações ou funcionalidades.
- Reproduzir: Garanta que o defeito seja reproduzível e descreva os passos necessários para a sua reprodução;
- Evidenciar: Evidencie a existência do defeito encontrado por meio de arquivos de saída, *printscreens* das telas, etc;
- Revisar: Revise a descrição e os passos para reproduzir o defeito. Lembre-se que o relato do defeito é um documento do projeto, assim como um caso de uso, um plano de testes, etc. Trate-o como tal;

Severidade	Descrição
Alta	Bloqueia completamente a utilização de uma funcionalidade básica ou da aplicação inteira
Média	Bloqueia a utilização de uma funcionalidade. Mas, no entanto, a funcionalidade pode ser usada por meio da utilização de um contorno (*workaround*) conhecido
Baixa	Problemas cosméticos e solicitações de melhorias

Tabela 1. Classificação da prioridade.

Prioridade	Descrição
1	O defeito deve ser corrigido imediatamente (até um dia útil). A aplicação não pode ser liberada sem a correção deste defeito
2	É altamente desejável que o defeito seja corrigido tão logo seja possível (até cinco dias úteis). A aplicação não pode ser liberada sem a correção deste defeito
3	Defeito de baixa prioridade. A aplicação pode ser liberada com esse defeito

Tabela 2. Classificação da severidade.

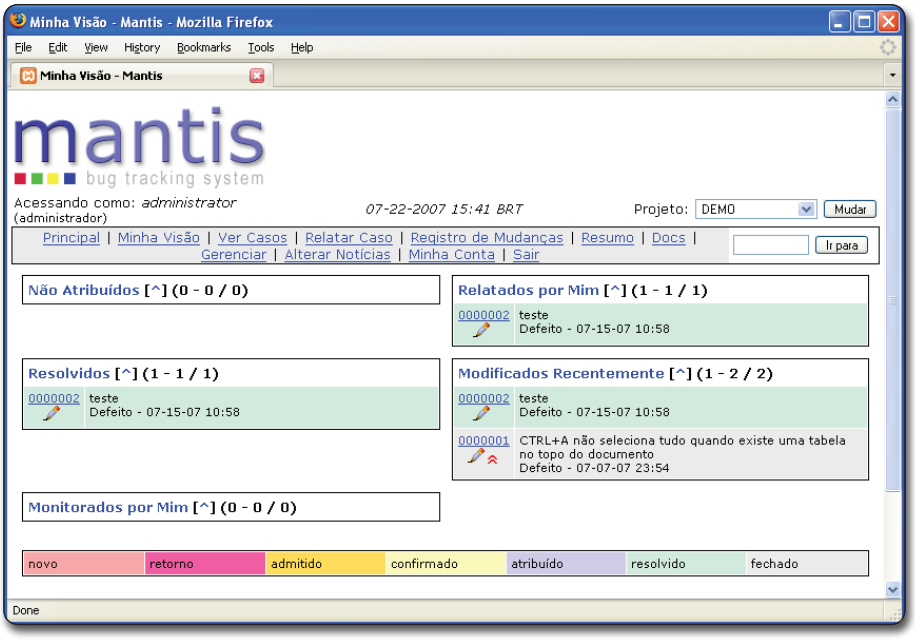


Figura 3. Página inicial do Mantis.

Severidade X Prioridade

A classificação da severidade e prioridade dos defeitos é, de forma similar ao relato de defeitos, um ponto normalmente negligenciado e foco de interpretações incorretas. A classificação incorreta da severidade e da prioridade normalmente contribui para a ineficiência da priorização e agendamento da correção dos defeitos. O efeito colateral direto desse problema é a má utilização dos recursos em função do enfoque na correção de defeitos menos prioritários.

A grosso modo, podemos afirmar que a severidade de um defeito define o impacto do defeito no funcionamento da aplicação. Por outro lado, a prioridade indica a ordem de correção do defeito (defeitos com alta prioridade são corrigidos imediatamente ou num curto prazo de tempo). De modo geral, defeitos com alta severidade são classificados com alta prioridade. No entanto, podem existir diversas situações onde não podemos aplicar essa regra. Por exemplo, a corrupção dos dados de uma aplicação no Windows 3.11 é um defeito com alta severidade mas, no entanto, deve ter baixa prioridade em virtude de que 99,9% dos usuários da aplicação utilizam versões mais atuais do Windows. A justificativa para esse critério é: Por que priorizar a correção de um defeito que vai beneficiar apenas 0,01% dos usuários?

Para evitar a subjetividade da classificação, sugere-se que os critérios para cada nível de prioridade e severidade sejam definidos na documentação do processo de gestão de defeitos. Para fins didáticos e de entendimento, serão apresentados na **Tabela 1** e **Tabela 2** exemplos de critérios para classificar a severidade e a prioridade dos defeitos respectivamente.

Mantis

O Mantis é uma ferramenta Open Source automatizada escrita em PHP cujo principal objetivo é dar suporte ao processo de gestão de defeitos. O Mantis controla o ciclo de vida de um defeito, desde o seu relato até o seu fechamento, por meio de fluxos (*workflows*) personalizáveis, como pode ser observado na **Figura 3**.

O endereço do site do Mantis está disponível na seção **Links**. Caso você tenha interesse de conhecê-lo com maior profundidade, os passos para a instalação

são os seguintes:

1. As pré-condições para a instalação são (PHP 4.0.6 ou superior, MySQL 3.23.2 ou superior, Apache ou IIS).
2. Faça o download do Mantis no site disponível na seção **Links**.
3. Descompacte o arquivo zip do Mantis na pasta www ou htdocs do servidor WEB (Apache ou IIS).
4. Abra o seu navegador e acesse o seguinte endereço: (http://localhost/mantis_1.0.5/).
5. Na janela Pre-Installation Check, preencha os campos username com o nome da conta de usuário para acesso ao banco de dados MySQL e o campo password com a senha.
6. Deixe os valores default nos demais campos.
7. Pressione o botão Install/Upgrade Database.
8. Ao final do processo, abra o seu navegador e acesse o seguinte endereço: (http://localhost/mantis_1.0.5/login_page.php).
9. Faça o primeiro login com o usuário padrão (administrator/root). Lembre-se de mudar a senha deste usuário.

O Mantis é instalado em inglês por padrão. Caso você prefira utilizar o Mantis com os textos traduzidos para a língua portuguesa, então siga os passos descritos abaixo:

1. Faça o login normalmente com o seu usuário e senha.
2. Clique no menu "My Account" e então selecione a opção "Preferences".
3. No campo Language selecione a opção "portuguese_brazil".
4. Pressione o botão "Update Prefs".

Entre as diversas funcionalidades oferecidas pelo Mantis, devemos destacar as seguintes:

- Pode ser executado em qualquer plataforma que suportar PHP/Apache/Mysql (Windows, Linux, Mac, Solaris, AS400/i5, etc);
- Suporta vários bancos de dados (MySQL, MS SQL, PostgreSQL);
- Suporta múltiplos mecanismos de autenticação (Interna, LDAP, HTTP Basic, Active Directory);
- Traduzido em 68 línguas diferentes (incluindo "portuguese_brazil");
- Criação ilimitada de projetos e relatos de defeitos;
- Controle de acesso e níveis de permissões para os usuários;
- Ciclo de vida dos defeitos (*worflow*) personalizável;
- Gerador interno de relatórios e gráficos (possibilidade para exportar os dados nos formatos CSV, Excel e Word);
- Mecanismo para a criação de campos personalizáveis (*custom fields*);
- Notificações por email automáticas ou por meio de RSS Feeds;
- Integração com ferramentas de controle de versões (Subversion e CVS);
- Interface Webservice (SOAP) para integração com outras ferramentas;
- MantisWAP – Suporte a dispositivos móveis (funcionalidade paga);

A fim de dar ao leitor uma exposição sobre as principais funcionalidades do Mantis e como elas atendem os principais fluxos de um ciclo de vida de um defeito, vamos simular na prática o relato de um defeito e acompanhar todos os passos até o seu fechamento.

Tão logo um testador ou usuário encontre um defeito, seja por intenção ou por acidente, o próximo passo deverá ser o relato (ou reporte) desse defeito.

No nosso cenário, vamos relatar um defeito (problema no recurso de seleção de textos quando existe uma tabela no topo do documento) do OpenOffice Writer (processador de texto Open Source).

Para realizar tal tarefa, você deverá selecionar o menu "Relatar Caso" do Mantis. Uma vez que a janela de relato for exibida, você deverá informar o resumo do defeito, a descrição, os passos para reproduzir e assim por diante, como pode ser observado na **Figura 4**.

É importante lembrar que o correto preenchimento dos campos severidade (gravidade) e prioridade neste passo é de extrema importância para a priorização e agendamento da correção do defeito.

Assim que o defeito for relatado, o analista de teste (ou líder de desenvolvimen-

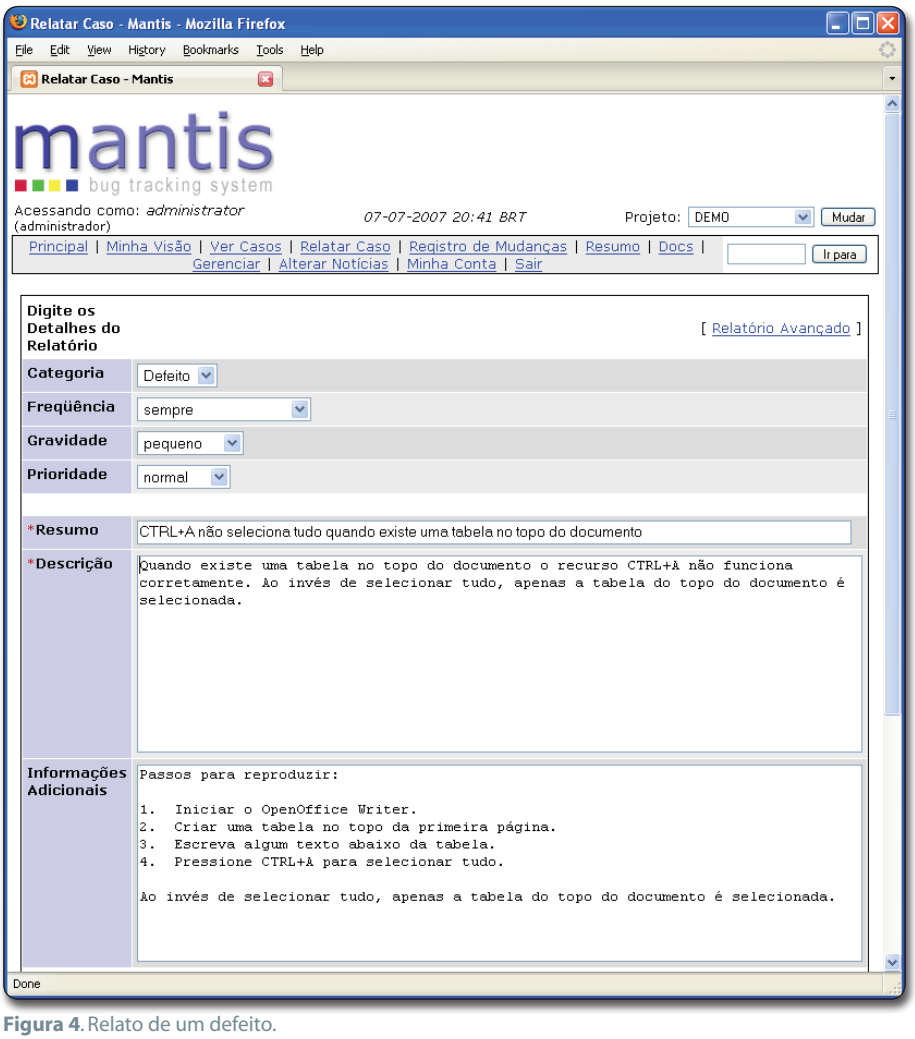


Figura 4. Relato de um defeito.

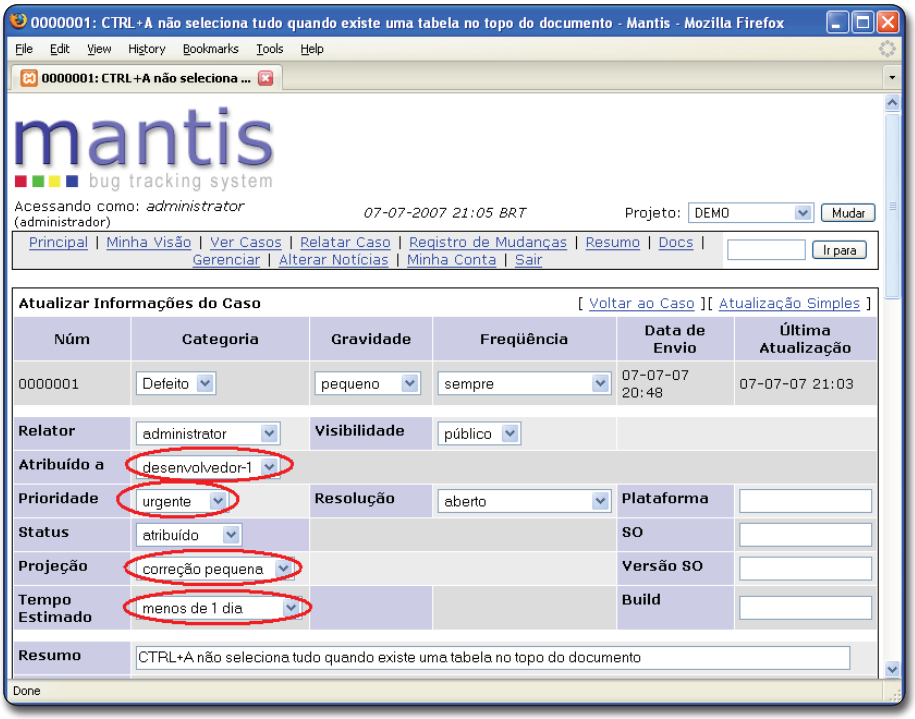


Figura 5. Reconhecimento, priorização e agendamento da correção de um defeito.

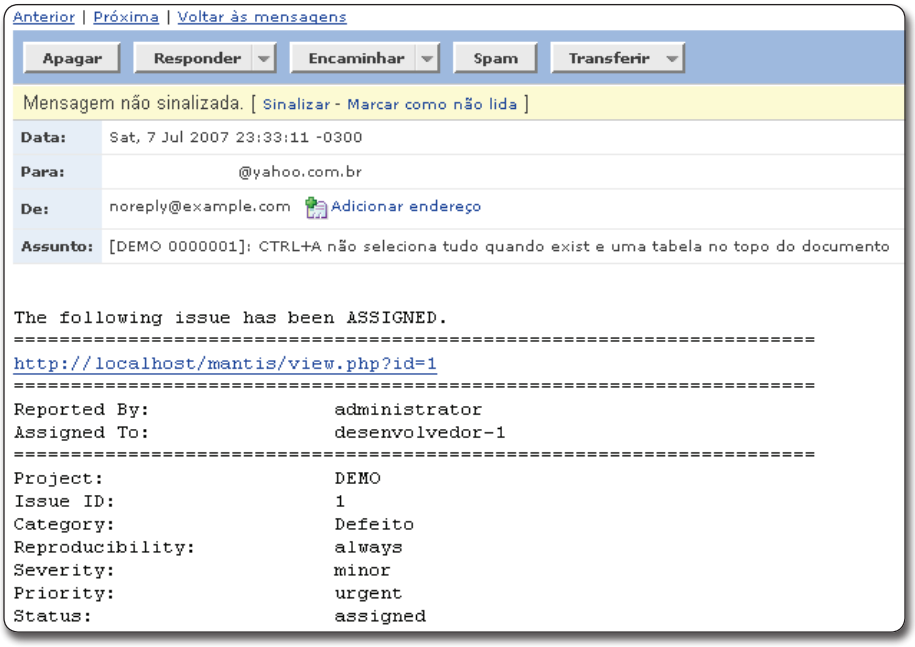


Figura 6. E-mail enviado pelo Mantis ao desenvolvedor.

to) poderá filtrar os defeitos cadastrados recentemente para confirmar e reconhecer se o que foi relatado realmente é um defeito ao invés de um mal entendimento do comportamento esperado.

Se for confirmado que o relato não é um problema, ele é imediatamente fechado. Caso contrário, o analista de teste (ou líder de desenvolvimento) deverá confirmar se a severidade (gravidade) e prioridade foram classificadas corretamente e atribuir o defeito a um desenvolvedor, como pode ser visto na **Figura 5**.

Nesta altura, também poderão ser realizadas a projeção da complexidade e a estimativa do tempo necessário para finalizar a correção do defeito.

O desenvolvedor, por sua vez, receberá um e-mail automaticamente conforme o fluxo (*workflow*) definido no Mantis, como pode ser observado no exemplo apresentado na **Figura 6**. De qualquer forma, o desenvolvedor poderá visualizar os defeitos atribuídos a ele diretamente no Mantis por meio da página "Minha Visão". Nesta página (**Figura 7**), são apresentados e consolidados todos os defeitos, inclusive os defeitos associados ao usuário logado no Mantis.

Assim que o desenvolvedor receber o e-mail notificando que a correção do defeito foi programada e atribuída a ele, o próximo passo será a execução da correção propriamente dita.

Dessa forma, tão logo a correção seja finalizada, o desenvolvedor deverá reportar a correção do defeito por meio do Mantis. Para tal tarefa, o desenvolvedor deverá mudar a resolução do defeito para "Corrigido" e descrever quais foram as modificações necessárias para corrigir o defeito no campo "Anotação", como pode ser visto na **Figura 8**.

Por fim, assim que for reportada a correção do defeito, o testador deverá executar o re-teste. Durante o re-teste, o

testador poderá identificar que o defeito não foi corrigido corretamente ou foi parcialmente corrigido. Neste caso, o defeito deverá ser reaberto e o ciclo de vida do defeito é reiniciado.

Por outro lado, o defeito deverá ser fechado caso ele tenha sido corrigido corretamente. Para realizar tal tarefa, o testador deverá mudar o status do defeito para "Fechado" e descrever algum comentário sobre o fechamento no campo "Anotação", como pode ser visto na **Figura 9**. Dessa forma, chegamos ao final da simulação de um ciclo de vida de um defeito e a apresentação das principais funcionalidades do Mantis.

Métricas e relatórios de gestão

A norma IEEE Std 829-1998 (IEEE Standard for Software Test Documentation) define a documentação e relatórios necessários para a execução de um projeto de teste de software. Dentre os relatórios sugeridos, existe um relatório chamado "Relatório de Incidente de Teste". Este relatório registra e consolida as ocorrências que precisam de algum tipo de investigação. Ele é normalmente utilizado para registrar os defeitos encontrados durante a execução dos testes.

Ferramentas automatizadas de gestão de defeitos, tais como o Mantis, geram diversos relatórios e gráficos com métricas que poderão fornecer dados para alimentar o Relatório de Incidente de Teste.

O Mantis, além de oferecer um relatório com o resumo consolidado de todos os defeitos relatados (**Figura 10**), também permite a visualização de gráficos com as principais métricas utilizadas na gestão de defeitos (**Figura 11**). Na listagem a seguir, são apresentadas diversas sugestões de indicadores e métricas de um processo de gestão de defeitos eficaz, afinal, você não pode gerenciar o que você não pode medir:

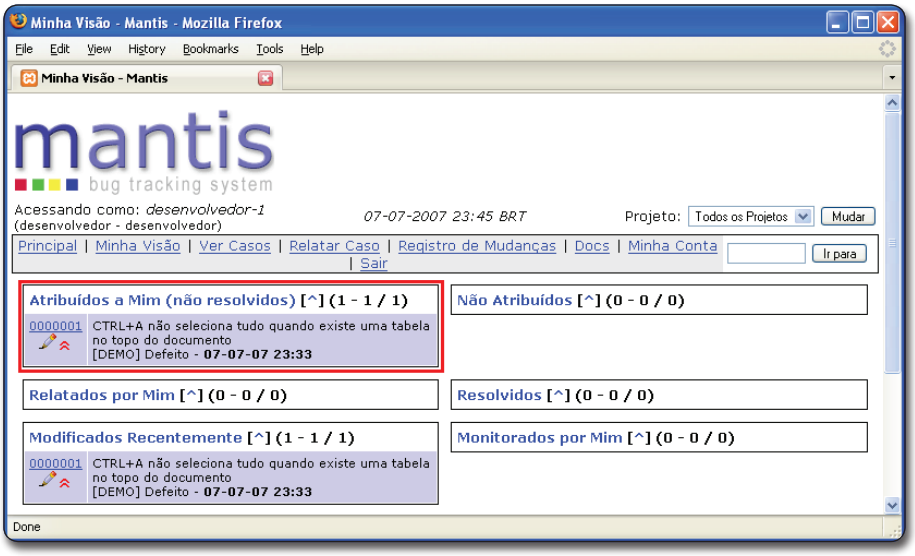


Figura 7. Consolidação dos defeitos associados ao usuário logado.

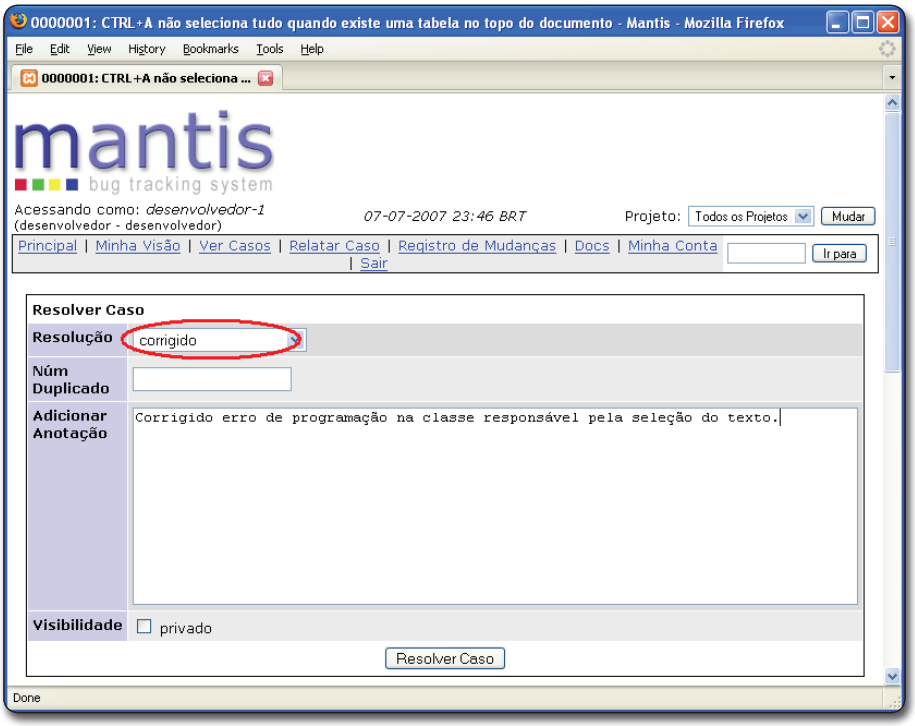


Figura 8. Reporte da correção de um defeito.

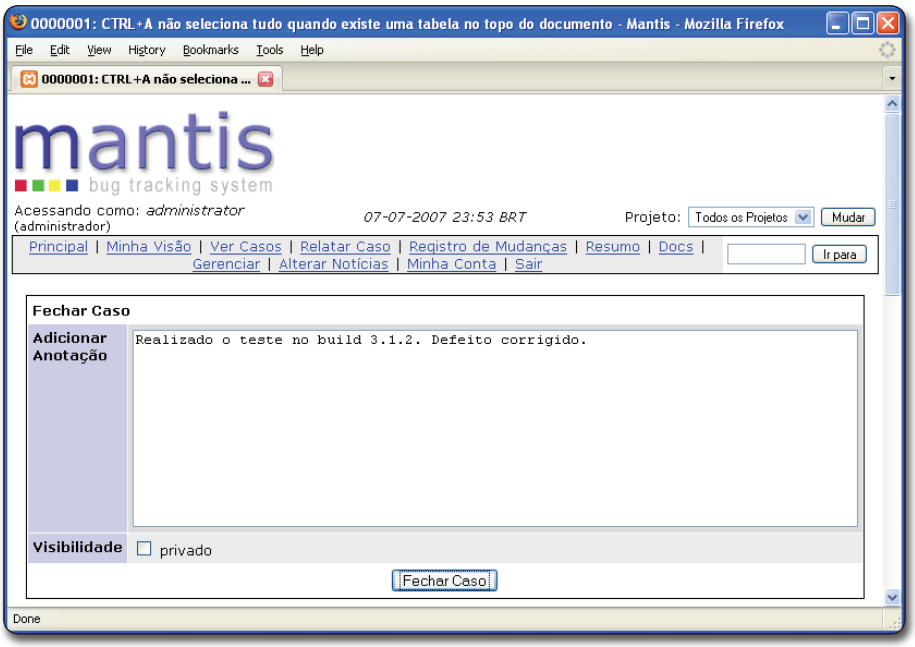


Figura 9. Fechamento de um defeito.

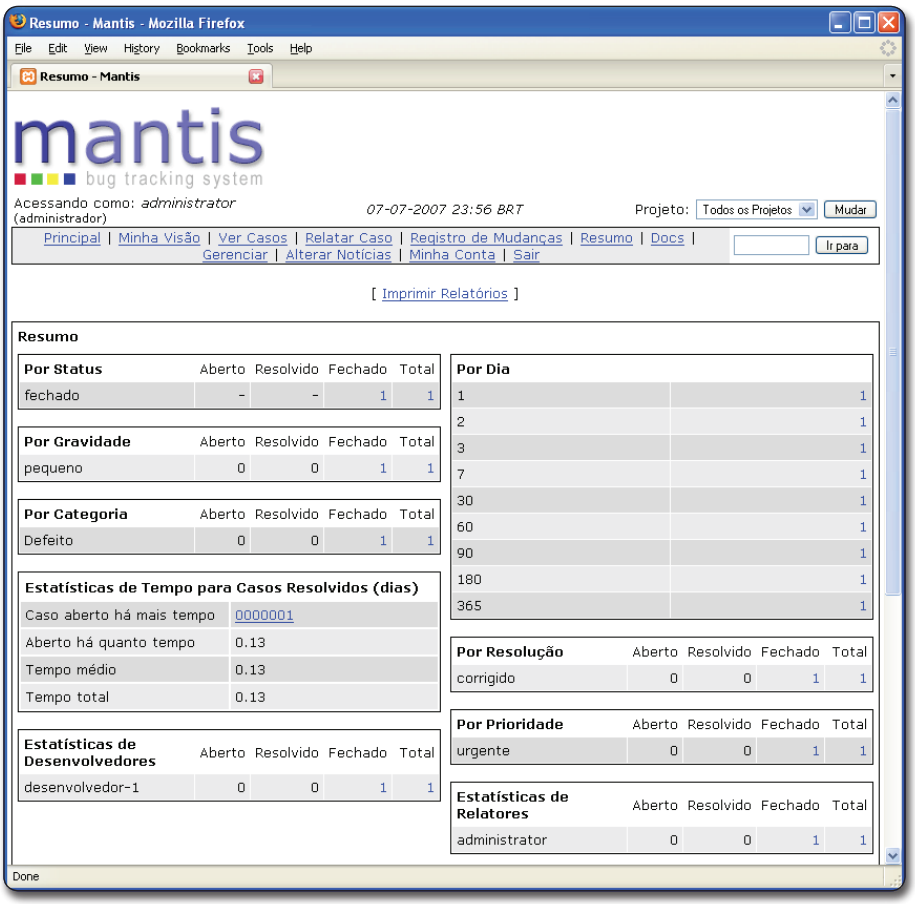


Figura 10. Resumo consolidado de todos os defeitos relatados.

- Densidade dos defeitos: Indica o número de defeitos por uma unidade. É normalmente utilizado para identificar a quantidade de defeitos por cada mil linhas de código (KLOC);
- Taxa de abertura dos defeitos: Utilizado para acompanhar o progresso da execução dos testes e o nível da qualidade do sistema em teste;
- Taxa de correção dos defeitos: Utilizado para acompanhar e medir a capacidade do time de desenvolvimento de corrigir os defeitos encontrados;
- Taxa de defeitos re-abertos: Utilizado para acompanhar a qualidade da correção dos defeitos. Pode identificar baixa qualidade no trabalho dos desenvolvedores ou baixa qualidade nos relatos dos defeitos (o que leva a uma correção incorreta);
- Taxa de defeitos abertos agrupados por testador: Identifica a taxa de abertura de defeitos por testador. Deve ser analisada em conjunto com outros dados, caso contrário, vai privilegiar testadores que encontram dezenas de defeitos bobos em detrimento dos testadores que encontram poucos defeitos (porém são defeitos complexos e difíceis de reproduzir);
- Taxa de defeitos corrigidos agrupados por desenvolvedor: Identifica a taxa de correção dos defeitos por desenvolvedor. Deve ser analisada em conjunto com outros dados, caso contrário, vai privilegiar desenvolvedores que corrigem dezenas de defeitos bobos em detrimento dos desenvolvedores que corrigem poucos defeitos (porém são defeitos de difícil resolução);
- Severidade por módulo: Utilizado para identificar os módulos que contém muitos defeitos com severidade alta. Com base nesses dados, podemos tomar alguma decisão para identificar a causa raiz dos problemas e promover alguma modificação no processo para diminuir a quantidade de defeitos com severidade alta no módulo;

Links

Comparação entre ferramentas de gestão de defeitos
http://en.wikipedia.org/wiki/Comparison_of_issue_tracking_systems

Website do Mantis
http://www.mantisbt.org

Gauging Software Readiness With Defect Tracking
http://www.stevemcconnell.com/ieeesoftware/bp09.htm

Microsoft Solutions Framework (Zero Bug Bouce e Bug Convergence)
http://www.microsoft.com/technet/solutionaccelerators/msf/default.mspx

• Prioridade por módulo: Identifica módulos com alta taxa de defeitos com prioridade alta. Com base nesses dados, podemos realocar os recursos a fim de atender a demanda. Normalmente é analisada em conjunto com a métrica de Severidade por módulo;

• Vazamento de defeitos: Utilizada para medir a efetividade da abordagem de testes em cada fase de teste (Unidade, Integração, Sistema e Aceitação). Dessa forma, podemos identificar que a abordagem e cobertura de testes utilizada na fase de testes de Sistema não foram muito eficientes em virtude de que muitos defeitos "vazaram" e foram encontrados durante a fase de testes de Aceitação;

• Bug Convergence: Identifica o ponto de declínio da taxa de abertura dos defeitos. Este é o ponto onde a quantidade de defeitos abertos tende a ser menor do que a quantidade de defeitos corrigidos. Dessa forma, espera-se que a partir desse ponto a quantidade de defeitos abertos diminua até chegar a zero;

• Zero Bug Bounce: Utilizada para acompanhar os picos e vales do gráfico de defeitos encontrados a partir do momento do declínio da taxa de abertura dos defeitos (Bug Convergence). Os vales desse gráfico representam os pontos no tempo onde não existem defeitos conhecidos com status igual a aberto, ou seja, estes são os melhores momentos para liberar o sistema para a próxima fase ou para o cliente;

Conclusão

Neste artigo foram apresentados os conceitos e melhores práticas na gestão de defeitos. O objetivo principal do artigo era destacar a importância da gestão de defeitos como um processo de apoio ao desenvolvimento e teste de software. Para facilitar o entendimento do leitor, foram demonstrados na prática os conceitos discutidos por meio da utilização do Mantis (ferramenta Open Source para gestão de defeitos).

Se o leitor quiser se aprofundar no assunto, são apresentadas na **Tabela 3** algumas das principais ferramentas comerciais e Open Source para a gestão de defeitos. Não foram esgotadas todas as opções disponíveis, mas já é um bom ponto de partida para auxiliar o leitor a encontrar a solução ideal para a sua necessidade. ●

Open Source	Comercial
Bugzilla http://www.bugzilla.org/	FogBugz http://www.fogcreek.com/FogBugz/
Scarab http://scarab.tigris.org/	Jira http://www.atlassian.com/software/jira/
BugNET http://www.bugnetproject.com/	yKAP http://www.ykap.com/
TRAC http://trac.edgewall.org/	Rational ClearQuest www.ibm.com/software/awdtools/clearquest/

Tabela 3. Ferramentas de gestão de defeitos alternativas.

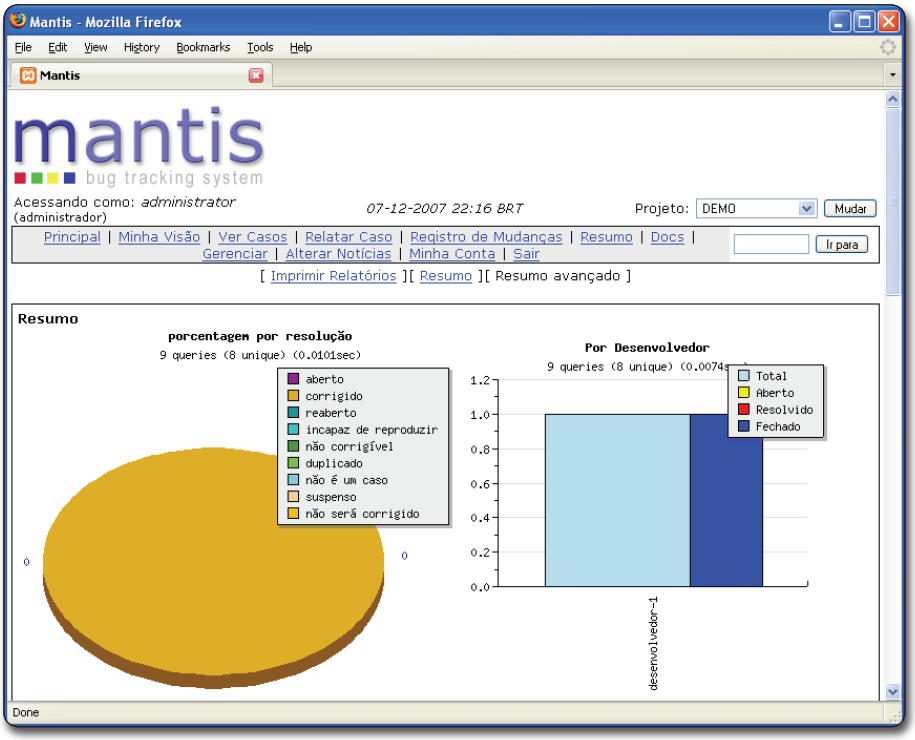


Figura 11. Principais métricas utilizadas na gestão de defeitos.

Introdução à Inspeção de Software

Aumento da qualidade através de verificações intermediárias

Marcos Kalinowski
mk@kalisoftware.com
Doutorando em Engenharia de Sistemas e Computação (COPPE/UFRJ). Mestre em Engenharia de Software (COPPE/UFRJ, 2004). Bacharel em Ciências da Computação (UFRJ, 2001). Diretor executivo e instrutor da Kali Software (www.kalisoftware.com), empresa de consultoria e treinamento em Engenharia de Software. Professor e pesquisador do curso de Ciência da Computação do Centro Universitário Metodista Bennett, ministrando disciplinas de Engenharia de Software. Consultor para implementação e avaliação do MPS.BR tendo colaborado na elaboração do guia geral e do guia de implementação deste modelo.

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 em projetos de consultoria na COPPE/UFRJ. É colaborador da Engenharia de Software Magazine.

Na engenharia de software, assim como em outras disciplinas de engenharia, é necessário considerar variáveis como esforço, produtividade, tempo e custo de desenvolvimento. Essas variáveis são afetadas negativamente quando artefatos defeituosos são produzidos, devido ao retrabalho para corrigir defeitos. Sabe-se, ainda, que o custo do retrabalho para correção de defeitos aumenta na medida em que o processo de desenvolvimento progride. Desta forma, iniciativas devem ser realizadas no sentido de encontrar e corrigir defeitos tão logo sejam introduzidos. Uma abordagem que tem se mostrado eficiente e de baixo custo para encontrar defeitos, reduzindo o retrabalho e melhorando a qualidade dos produtos é a revisão dos artefatos produzidos ao longo do processo de desenvolvimento de software.

Inspeção de software é um tipo particular de revisão que pode ser aplicado a todos os artefatos de software e possui um processo de detecção de defeitos rigoroso e bem definido. A **Figura 1** ilustra a possibilidade de se realizar inspeções nos diferentes artefatos de software.

De forma resumida, o processo tradicional de inspeção envolve o planejamento da inspeção, indivíduos revisando um determinado artefato, um encontro em equipe para discutir e registrar os defeitos, a passagem dos defeitos para o autor do artefato para que possam ser corrigidos e uma avaliação final sobre a necessidade de uma nova inspeção.

A importância de inspeções na redução do retrabalho e na garantia da qualidade de software está bem documentada na literatura e é discutida em maiores neste artigo. A seção seguinte apresenta a definição de alguns conceitos utilizados ao longo deste trabalho. Veremos ainda neste artigo alguns benefícios de se realizar inspeções em artefatos produzidos ao longo do processo de desenvolvimento de software são descritos; conceitos sobre técnicas de leitura para detecção de defeitos em artefatos de software;

o processo de inspeção de software e suas características, e; o estado atual da utilização de inspeções na prática e do suporte ferramental existente.

Definição dos Conceitos

O termo defeito muitas vezes é utilizado de forma genérica. No entanto, é importante ter em mente que sua interpretação dependerá do contexto em que ele for utilizado. Defeitos encontrados através de revisões estarão relacionados às faltas no artefato sendo revisado. Quando um defeito se manifesta através de atividades de teste, por sua vez, estaremos lidando com uma falha no software. Estas definições seguem a terminologia padrão para Engenharia de Software do IEEE (IEEE 610.12, 1990):

- **Erro:** É um defeito cometido por um indivíduo ao tentar entender uma determinada informação, resolver um problema ou utilizar um método ou uma ferramenta.
- **Defeito (ou Falta):** É uma manifestação concreta de um erro num artefato de software. Um erro pode resultar em diversos defeitos.
- **Falha:** É o comportamento operacional do software diferente do esperado pelo usuário. Uma falha pode ter sido causada por diversas faltas e algumas faltas podem nunca causar uma falha.

Em alguns momentos o termo discrepância será utilizado. Este termo refere-se a um suposto defeito encontrado. No nosso contexto, uma discrepância poderá ser considerada um defeito de fato ou um chamado falso positivo.

Para obter uma classificação para os defeitos encontrados nas revisões (as faltas), partimos do fato de que todos os artefatos gerados durante o desenvolvimento de software utilizam como base o documento de requisitos ou artefatos gerados a partir deste. Desta forma, as classes de defeito seriam os tipos de defeito presentes em documentos de requisitos acrescidos dos tipos de defeitos introduzidos pela transformação de artefatos ao longo do desenvolvimento de software.

Um padrão IEEE (IEEE 830, 1998), que recomenda práticas para especificação de requisitos de software, define atributos de qualidade que um documento de requisitos deve possuir. Foi considerado que a falta de qualquer um destes atributos constituiria um tipo de defeito. Assim, a seguinte taxonomia foi definida:

- **Omissão:** (1) Algum requisito importante relacionado à funcionalidade, ao desempenho, às restrições de projeto, ao atributo, ou à interface externa não foi incluído; (2) não está definida a resposta do software para todas as possíveis situações de entrada de dados; (3) faltam seções na especificação de requisitos; (4) faltam referências de figuras, tabelas, e diagramas; (5) falta definição de termos e unidades de medidas.
- **Ambigüidade:** Um requisito tem várias interpretações devido a diferentes termos utilizados para uma mesma característica ou vários significados de um termo para um contexto em particular.
- **Inconsistência:** Dois ou mais requisitos são conflitantes.
- **Fato Incorreto:** Um requisito descreve um fato que não é verdadeiro, considerando as condições solicitas para o sistema.
- **Informação Estranha:** As informações fornecidas no requisito não são necessárias ou mesmo usadas.
- **Outros**: Outros defeitos como a inclusão de um requisito numa seção errada do documento.

É importante ressaltar que estas classes genéricas de defeitos podem ser divididas em classes mais específicas, dependendo da necessidade. Além disto, esta classificação não pretende ser definitiva e cada organização pode acrescentar mais tipos de defeito de acordo com suas necessidades.

Benefícios da Aplicação de Inspeções de Software

Esforço, Produtividade, Tempo e Custo

O esforço gasto por organizações de software com retrabalho pode variar em média entre 40% e 50% do esforço total do desenvolvimento de um projeto. Uma estimativa da distribuição do retrabalho pelas atividades de desenvolvimento de software está ilustrada na **Figura 2**.

Analisando a **Figura 2**, é possível verificar que o retrabalho tende a aumentar na medida em que o desenvolvimento progride. Uma das razões para isto é o aumento no esforço para corrigir defeitos nas atividades finais do processo de desenvolvimento. Através da análise de 63 projetos, BOEHM (1981) apresenta o custo relativo da correção de defeitos encontrados em cada uma das atividades de desenvolvimento (**Figura 3**).

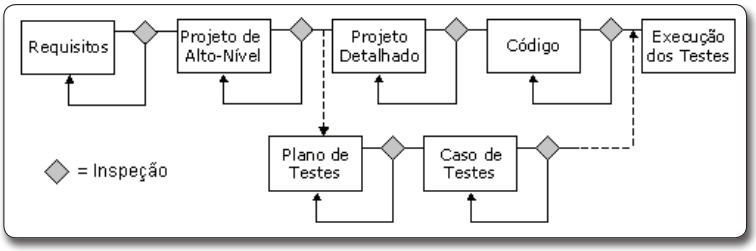


Figura 1. Inspeções de Software nos Diferentes Artefatos. Adaptado de (ACKERMAN et al., 1989)

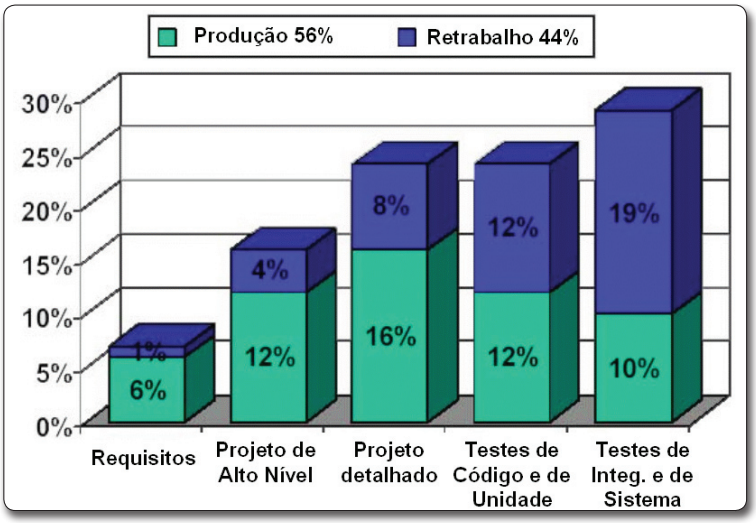


Figura 2. Distribuição do retrabalho pelas atividades de desenvolvimento de software. Adaptado de (WHEELER et al., 1996).

Assim, um dos maiores benefícios de se utilizar inspeções de software é a detecção de defeitos nas fases iniciais do processo de desenvolvimento de software, facilitando a correção destes defeitos com menor esforço e custo. Desta forma, de acordo com (JONES, 1991), o esforço com retrabalho é reduzido em média para 10% a 20% do esforço total de desenvolvimento. Esta redução no retrabalho pode implicar em melhorias significativas para a produtividade de software. De acordo com (BOEHM et al., 2000) a maior redução de esforço é gerada pela melhoria de (1) maturidade de processos de software, (2) arquiteturas de software e (3) gerência de riscos é proveniente da redução do retrabalho. Resultados experimentais mostram como este benefício pode afetar as variáveis esforço, produtividade, tempo e custo, mencionadas na seção anterior:

- Esforço. O departamento de desenvolvimento da Ericsson em Oslo, Noruega, calculou uma redução bruta do esforço total de desenvolvimento em 20% aplicando inspeções. Além disto, resultados de estudos mostram que a introdução de inspeções de projeto pode reduzir o esforço com retrabalho em 44%.
- Produtividade. De acordo com (GILB e GRAHAM, 1993), inspeções aumentam a produtividade de 30% a 50%;
- Tempo. De acordo com (GILB e GRAHAM, 1993), inspeções reduzem o tempo de desenvolvimento de 10% a 30%;
- Custo. Resultados de estudos mostram que a introdução de inspeções de código pode reduzir os custos de implementação de projetos em 39%.

Uma estimativa de (WHEELER et al., 1996) sobre o custo de desenvolvimento quando inspeções são aplicadas, quando comparado ao desenvolvimento sem utilizar inspeções, se encontra na **Figura 4**. Esta estimativa representa bem os benefícios apresentados nesta seção. É possível observar que utilizando inspeções se obtém um aumento na produtividade, já que projetos são concluídos em menos tempo e envolvem menos gastos.

Qualidade de Software

De acordo com Pressman (2001), qualidade de software é a conformidade a: (1) requisitos funcionais e não funcionais que têm sido explicitamente declarados, (2) padrões de desenvolvimento que tenham sido claramente documentados e (3) características implicitamente esperadas de todo software a ser desenvolvido. De forma resumida, qualidade consiste de um conjunto de requisitos e de um produto ou serviço que esteja em conformidade com estes requisitos e, por esta razão, atenda completamente às necessidades dos clientes. Entre as atividades que podem ser utilizadas para verificar a qualidade de software se encontram:

- Revisões de software;
- Testes;
- Padrões e procedimentos formais;
- Controle de mudanças;
- Métricas de software;
- Procedimentos para coleção e disseminação de informações.

A importância das revisões na garantia da qualidade é destacada pelo modelo CMMI, que exige a realização de revisões como uma prática específica do processo de verificação. Sabe-se ainda que inspeções de software, em particular, capturam em torno de 60% dos defeitos de artefatos (BOEHM e BASILI, 2001) o que deixa explícita a sua contribuição para a melhoria da qualidade.

Uma outra maneira de detectar defeitos em artefatos é através da aplicação de testes. No entanto, a aplicação de testes descobre apenas sintomas de problemas e, desta forma, pode ocasionar um trabalho refinado e custoso para detectar o defeito que causou o sintoma. BOEHM e BASILI (2001) ressaltam ainda que inspeções e testes capturam diferentes tipos de defeito e em diferentes momentos do processo de desenvolvimento de software.

Portanto, é interessante aplicar tanto inspeções quanto testes para detectar defeitos em artefatos de software.

Além disto, precedendo os testes com as inspeções, defeitos podem ser removidos nas fases iniciais do processo de desenvolvimento e os desenvolvedores terão uma visão mais ampla da complexidade do sistema. Esta visão os deixa mais bem preparados para o momento em que se confrontam com esta complexidade e com os defeitos que podem possivelmente surgir durante os testes.

Outros Benefícios

A aplicação de inspeções de forma bem planejada pode trazer diversos outros benefícios:

- Aprendizado. Inspetores experientes podem tentar detectar padrões de como os defeitos ocorrem e definir diretrizes que ajudem na detecção destes. Além disto, bons padrões de desenvolvimento podem ser observados durante a inspeção, sendo possível sua descrição como melhores práticas para a organização.
- Integração entre processos de detecção e de prevenção de defeitos. Saber onde e quando os defeitos ocorrem pode ajudar a estabelecer planos de contingência que

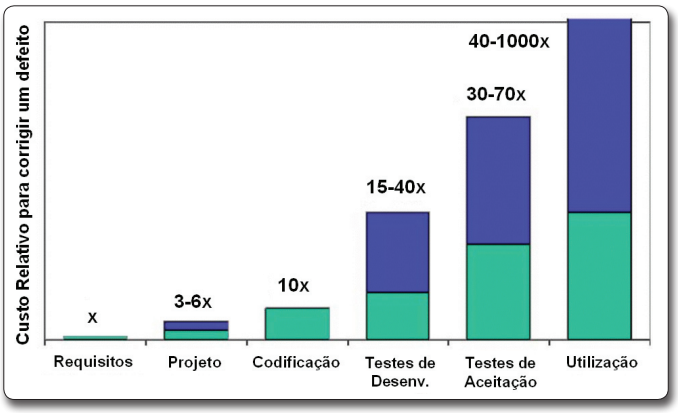


Figura 3. Custo relativo para corrigir um defeito. Adaptado de (BOEHM, 1981).

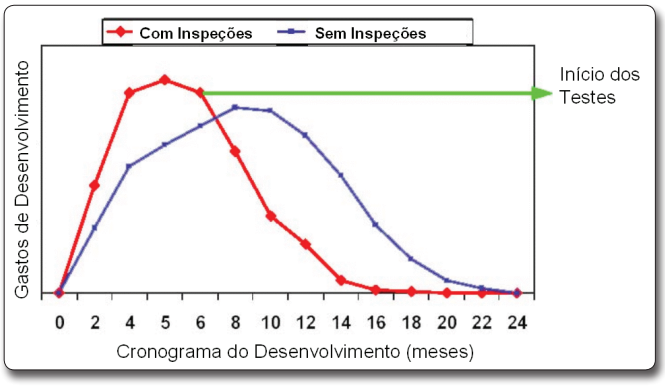


Figura 4. Estimativa dos gastos de desenvolvimento utilizando e não utilizando inspeções. Adaptado de (WHEELER et al., 1996).

evitem a sua ocorrência.

• Produtos mais inteligíveis. Os autores dos diversos artefatos, sabendo que estes serão inspecionados, passarão a produzir artefatos de uma forma que sua compreensão seja facilitada. A produção de artefatos mais inteligíveis, além de facilitar a inspeção, trará benefícios para as fases seguintes do processo de desenvolvimento, incluindo principalmente a fase de manutenção.

• Dados defeituosos ajudam a melhorar o processo de software do projeto. Analisando a causa dos defeitos encontrados é possível fazer ajustes no processo para evitar a ocorrência de defeitos deste mesmo tipo.

O Processo de Inspeção de Software

FAGAN (1976) desenvolveu o processo tradicional de inspeção de software, uma forma detalhada de se realizar uma revisão. Neste processo, existem seis atividades principais:

• Planejamento. Um usuário, desempenhando o papel de moderador da inspeção, define o contexto da inspeção (descrição da inspeção, técnica a ser utilizada na detecção de defeitos, documento a ser inspecionado, autor do documento, entre outros), seleciona os inspetores e distribui o material a ser inspecionado.

• Apresentação. Os autores dos artefatos a serem inspecionados apresentam as características destes. Esta fase pode ser omitida se os inspetores possuem conhecimento sobre o projeto e os artefatos que devem ser inspecionados.

• Preparação. Os inspetores estudam os artefatos individualmente, e eventualmente fazem anotações sobre estes produzindo uma lista de discrepâncias. O fornecimento de técnicas de leitura pode facilitar a execução desta tarefa.

• Reunião. Uma reunião em equipe ocorre, envolvendo o moderador, os inspetores e os autores do documento. Discrepâncias são discutidas, e classificadas como defeito ou falso positivos. A decisão final sobre a classificação de uma discrepância sendo discutida é do moderador. A solução dos defeitos não é discutida durante a reunião, que não deve exceder duas horas, uma vez que após este tempo a concentração e a capacidade de análise dos inspetores costuma reduzir drasticamente. No caso em que uma reunião precisar de mais de duas horas, é sugerido que o trabalho de inspeção continue no próximo dia.

• Retrabalho. O autor corrige os defeitos encontrados pelos inspetores e confirmados pelo moderador.

• Continuação. O material corrigido pelos autores é repassado para o moderador, que faz uma análise da inspeção como um todo e re-avalia a qualidade do artefato inspecionado. Ele tem a liberdade de decidir se uma nova inspeção deve ocorrer ou não.

A forma como estas atividades se relacionam no processo de inspeção está ilustrada na **Figura 5**. Note que a atividade de apresentação, opcional, não está representada na figura.

Entre as características deste processo, temos que ele pode ser aplicado a todos os artefatos produzidos ao longo do processo de desenvolvimento, permitindo a utilização de técnicas de leitura de artefatos específicos na atividade de preparação individual. Além disto, ele possui uma estrutura rígida, com aspectos colaborativos, onde papéis, atividades e os relacionamentos entre atividades estão bem definidos.

A Reorganização do Processo de Inspeção

Baseados em diversos estudos experimentais sobre inspeções de software SAUER et al., (2000) propuseram uma reorganização do processo tradicional de inspeção. Essa reorganização visa a adequação do processo tradicional a inspeções com reuniões assíncronas e equipes geograficamente distribuídas. Assim, mudanças para reduzir o custo e o tempo total para a realização deste tipo de inspeção foram introduzidas. Este projeto alternativo para inspeções de software mantém a estrutura rígida e os aspectos colaborativos do processo tradicional e consiste basicamente em substituir as atividades de preparação e de reunião do processo tradicional por três novas atividades seqüenciais: detecção de defeitos, coleção de defeitos e discriminação de defeitos. Estas atividades podem ser descritas da seguinte forma:

• Detecção de Defeitos. Cada um dos inspetores selecionados pelo moderador no planejamento realizará uma atividade de detecção de defeitos. A principal tarefa desta atividade consiste em buscar defeitos no documento a ser inspecionado e assim produzir uma lista de discrepâncias.

• Coleção de Defeitos. O moderador agrupa as listas de discrepâncias dos inspetores. Esta atividade envolve eliminar discrepâncias repetidas (encontradas por mais de um inspetor), mantendo só um registro para cada discrepância.

• Discriminação de Defeitos. O modera-

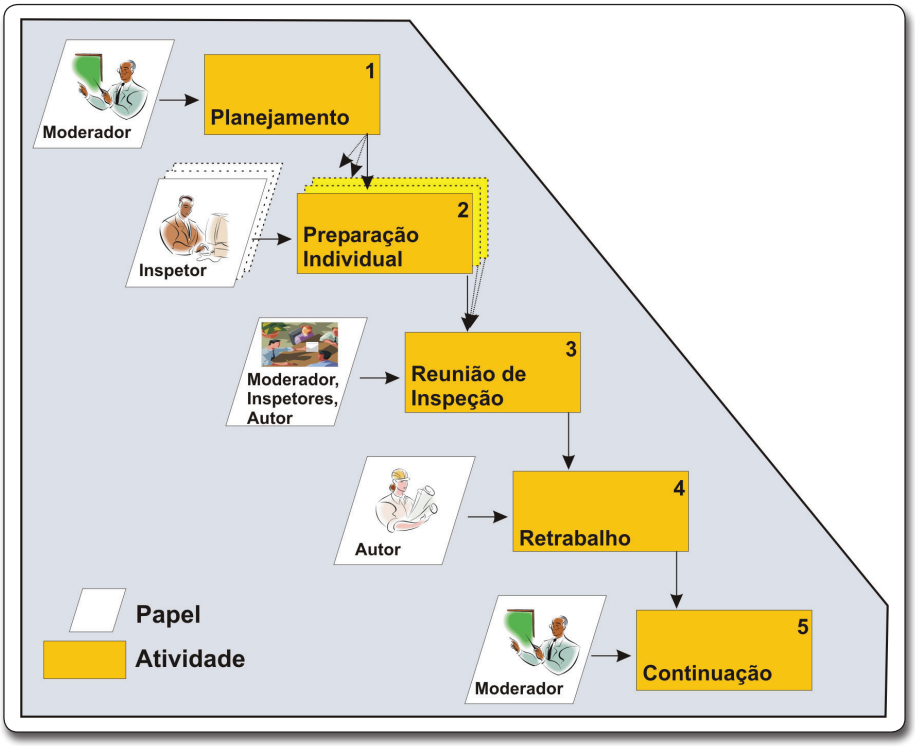


Figura 5. Processo de inspeção de software, conforme definido em (adaptado de FAGAN, 1976).

dor, o autor do documento e os inspetores discutem as discrepâncias de forma assíncrona. Durante esta discussão, algumas discrepâncias serão classificadas como falso positivo e outras como defeito. Os falso positivos serão descartados e os defeitos serão registrados em uma lista de defeitos, que então será passada para o autor para que a correção possa ocorrer.

A forma como as atividades se relacionam na reorganização do processo de inspeção de SAUER et al. (2000) está ilustrada na **Figura 6**.

Este processo permite a utilização de um número grande de inspetores em paralelo para a detecção de defeitos e, assim, aumentar a probabilidade de se encontrar defeitos difíceis de serem encontrados. Um grande número de inspetores tem um efeito no custo, mas não no tempo de detecção de defeitos e não implicará em problemas de coordenação, afinal discrepâncias encontradas por mais de um inspetor são encaminhadas diretamente para a atividade de retrabalho e as discrepâncias encontradas por apenas um inspetor não precisam ser discutidas necessariamente por todos os inspetores.

Estado Atual: Prática de Revisões em Organizações de Software e Suporte Ferramental Existente

Ao longo dos anos, muito conhecimento tem sido produzido na área de inspeções de software. Incluindo variantes do processo tradicional de inspeção, técnicas de estimativa do número de defeitos de documentos e da cobertura de inspeções, técnicas de leitura de documentos visando aumentar o número de defeitos encontrados por inspetores, diretrizes para pontos de tomada de decisão do processo de inspeção, dentre outras. Parte deste conhecimento tem sido avaliado em estudos experimentais e se mostrado adequado.

No entanto, muito deste conhecimento não tem sido utilizado na prática por organizações de software ao realizarem suas inspeções, e é negligenciado na maioria das propostas de apoio ferramental ao processo de inspeção de software.

Esta seção apresenta o estado da prática de revisões de software e o ferramental de apoio atualmente existente.

Prática de Revisões em Organizações de Software

Em um artigo comparando as diversas abordagens sobre inspeções de software encontradas na literatura, LAITENBERGER e DEBAUD (2000) mostraram que, conforme representado na **Figura 7**, inspeções em código são as mais comuns. No entanto, sabe-se que os benefícios de inspeções são maiores para os artefatos produzidos no início do ciclo de desenvolvimento. É importante destacar aqui que estas informações podem não refletir o estado atual de desenvolvimento de software, estamos em 2008.

Um survey foi realizado em 2002 visando avaliar o estado da prática de revisões de software (CIOLKOWSKI et al., 2003). De acordo com seus resultados, embora muitas organizações de software realizem revisões, a forma como as revisões são realizadas ainda é pouco sistematizada, pouco conhecimento da área de inspeções de software é utilizado e assim o verdadeiro potencial das revisões raramente é explorado. Este argumento é baseado em três observações sobre as organizações participantes do survey:

• (1) Revisões raramente cobrem sistematicamente as fases de desenvolvimento de software. Cerca de 40% das organizações responderam realizar inspeções em

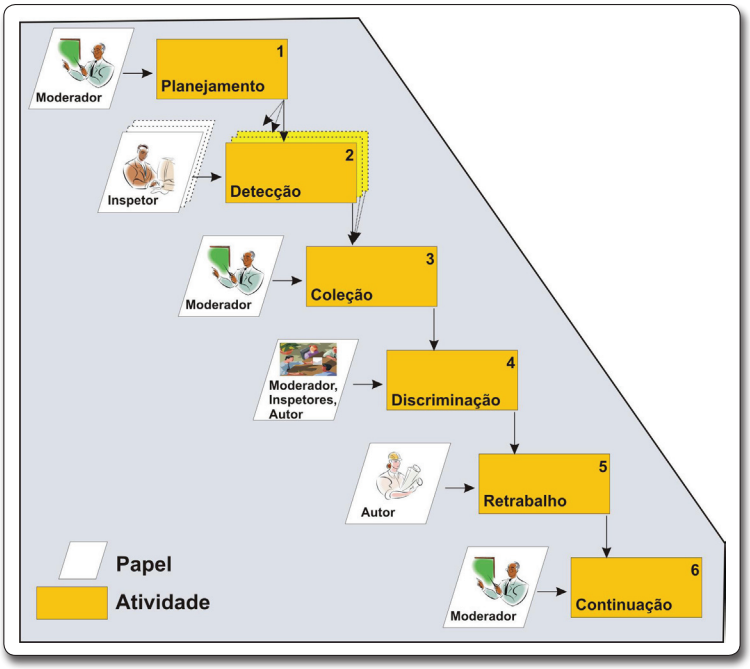


Figura 6. Reorganização do processo de inspeção, adaptada de (SAUER et al., 2000).

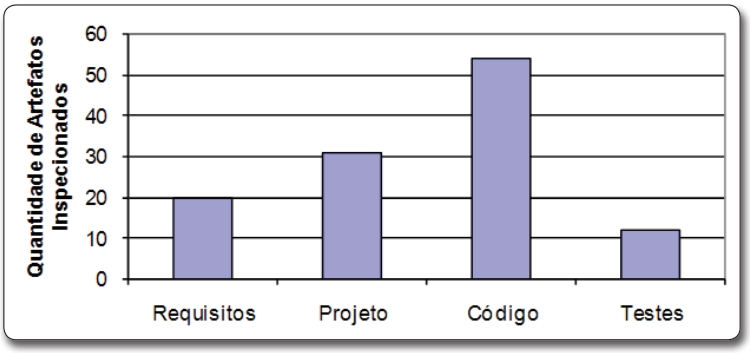


Figura 7. Distribuição do Uso de Inspeção nos Diferentes Artefatos. Adaptado de (LAITENBERGER e DEBAUD, 2000).

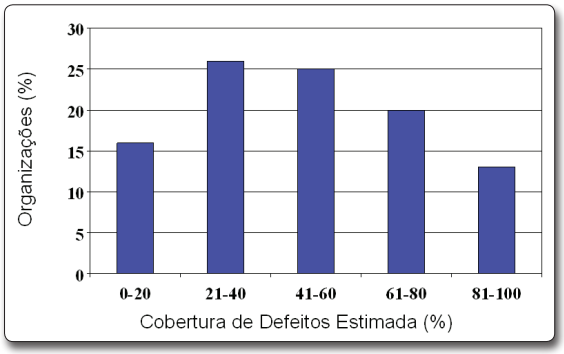


Figura 8. Cobertura de defeitos estimada para as organizações participantes do survey. Adaptado de (CIOLKOWSKI et al., 2003).

requisitos e 30% inspeções em código.

• (2) Muitas vezes a atividade de detecção de defeitos não é realizada sistematicamente. Cerca de 60% das organizações responderam não conduzir uma atividade de detecção de defeitos regularmente.

• (3) Organizações raramente utilizam revisões de software no contexto de um programa sistemático de avaliação e melhoria do processo. Mais da metade das empresas participantes do survey respondeu não coletar dados (40%) ou coletar dados e não os utilizar para realizar uma análise (18%).

A utilização pouco sistematizada de revisões na prática tem sido um fator de confusão entre os resultados esperados e os obtidos através de sua aplicação. A **Figura 8** ilustra a cobertura de defeitos estimada das revisões realizadas pelas organizações participantes do survey.

Visando apoiar a realização de revisões na prática de forma mais sistemática foram elaboradas propostas de apoio ferramental. Algumas destas propostas se encontram descritas na sessão seguinte.

Ferramentas de Apoio ao Processo de Inspeção

Entre as ferramentas que apóiam a coordenação e as diversas atividades do processo de inspeção que tem sido efetivamente avaliadas e utilizadas recentemente na indústria de software destacamos as seguintes: GRIP (GRoupware supported Inspection Process), IBIS (Internet Based Inspection System) e ISPIS (Infra-estrutura de Suporte ao Processo de Inspeção de Software). Esta última sendo tecnologia nacional desenvolvida na COPPE/UFRJ, em processo de registro junto ao INPI. Uma descrição destas ferramentas se encontra a seguir:

• GRIP (Grünbacher et al., 2003). Nasceu da iniciativa de se adaptar um sistema COTS (Comercial Off The Shelf) de apoio a trabalho colaborativo (GSS) (Groupware Support System) para realizar inspeções em requisitos de software. Assim, GRIP fornece um framework e ferramentas colaborativas para a realização de inspeções assíncronas com equipes geograficamente distribuídas. GRIP fornece suporte a um processo próprio de inspeção que envolve quatro atividades: planejamento, inspeção individual, reunião assíncrona, e avaliação da inspeção. A ferramenta não fornece apoio a técnicas específicas de detecção de defeitos, apenas ad-hoc. Uma taxonomia é fornecida junto com o artefato a ser inspecionado. Os elementos desta taxonomia representam, de forma hierárquica, os tópicos abordados no artefato a ser inspecionado. As discrepâncias dos inspetores são então agrupadas de acordo com os elementos da taxonomia. Assim, uma discrepância na descrição do caso de uso Mantendo Usuário poderia ser adicionada utilizando a seguinte navegação pela taxonomia: Requisitos Funcionais > Casos de Uso > Mantendo Usuário > Descrição. Os artefatos em si são lidos na ferramenta mais conveniente ao inspetor. Desta forma, GRIP é capaz de lidar com diferentes tipos de artefato, bastando que uma taxonomia para o artefato seja criada descrevendo sua estrutura. Na reunião GRIP fornece apoio para a classificação das discrepâncias que de acordo com um estudo experimental, mostrou reduzir o esforço da reunião e apoiar a classificação correta de discrepâncias;

• IBIS (LANUBILE et al., 2003). Utiliza a web em conjunto com notificações por e-mail para apoiar inspeções assíncronas com equipes geograficamente distribuídas. IBIS visa explicitamente apoiar a reorganização do processo de inspeção proposta em SAUER et al. (2000) e permitir a sua realização de forma sistematizada. IBIS não limita o tipo de artefato a ser inspecionado e na detecção de defeitos atualmente permite apenas a utilização das técnicas ad-hoc e de checklists. Na reunião de inspeção desta proposta as discrepâncias encontradas na detecção de defeitos são tratadas como tópicos de discussão, onde mensagens podem ser acrescentadas e o moderador pode realizar a classificação das discrepâncias como defeito ou falso positivo. IBIS não fornece apoio aos pontos de tomada de decisão do processo de inspeção, sendo as atividades de planejamento e de continuação do processo tratadas como simples cadastros, sem apoio para a realização de suas tarefas. IBIS tem sido utilizada em estudos experimentais recentes para obter conhecimento na área de inspeções de software e para avaliar aspectos da reorganização do processo de inspeção. A **Figura 9** ilustra o apoio de IBIS ao registro de discrepâncias na atividade de detecção de defeitos utilizando um checklist.

• ISPIS (KALINOWSKI e TRAVASSOS, 2004). Esta ferramenta também visa apoiar a reorganização do processo de inspeção proposta em SAUER et al. (2000). Entretanto, ISPIS foi desenvolvida utilizando uma metodologia experimental, levando em consideração conhecimento a respeito de inspeções de software selecionado criteriosamente, dando preferência a conhecimento efetivamente avaliado. ISPIS é a única ferramenta que possibilita o uso automatizado deste conhecimento sem que a equipe de inspeção precise adquiri-lo através de revisões bibliográficas e aplicá-lo manualmente. Assim, os

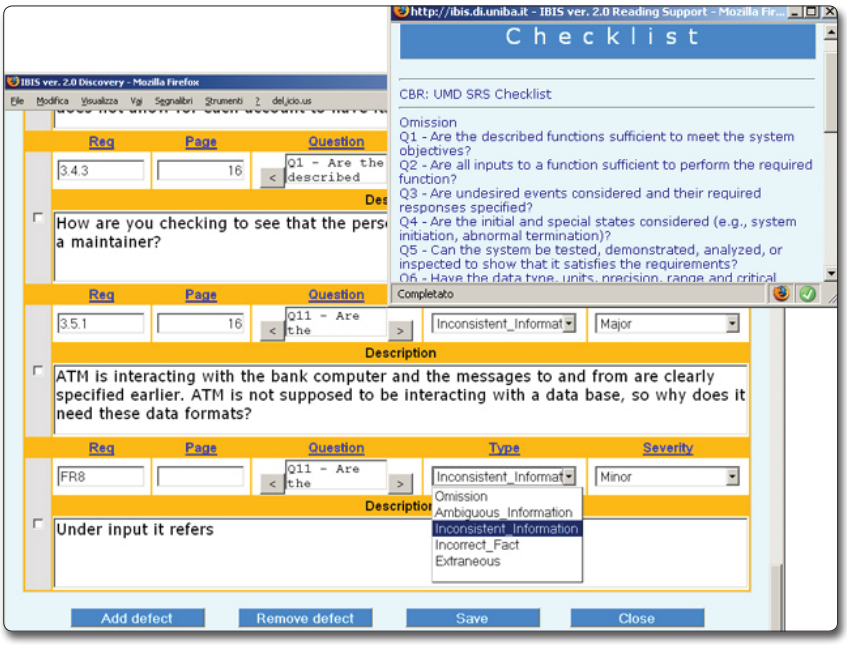


Figura 9. Apoio de IBIS ao registro de discrepâncias na atividade de detecção de defeitos (LANUBILE et al., 2003).

pontos de tomada de decisão do processo são apoiados por resultados de técnicas de estimativa e dados quantitativos representando a realidade da própria organização. ISPIS pode ser utilizada para coordenar e apoiar a inspeção de qualquer tipo de artefato de software (documentos de requisitos, de projeto, código, etc) e, opcionalmente, faz uso de um mecanismo de integração para se comunicar com ferramentas de detecção de defeitos existentes para artefatos específicos. Outra característica da ferramenta é permitir reuniões de inspeção tanto síncronas quanto assíncronas. As inspeções com reuniões assíncronas se adequam ao contexto de desenvolvimento global de software. Assim, ISPIS pôde, por exemplo, ser utilizada com sucesso em inspeções com participantes distribuídos entre Rio de Janeiro, Nova Iorque e Londres. A ferramenta tem ainda sido utilizada para a realização de projetos na indústria. Por representar o estado da arte a respeito de inspeções de software, ISPIS resultou em artigos científicos em conferências renomadas. Sua aplicabilidade na indústria permitiu ainda publicações sob forma de relatos de experiência. A **Figura 10** ilustra algumas telas do apoio de ISPIS à atividade de planejamento.

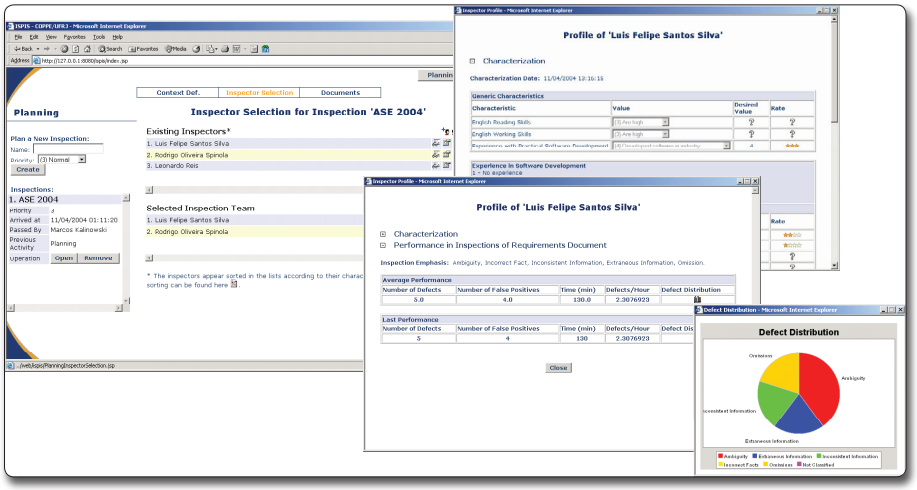


Figura 10. Telas do apoio de ISPIS ao planejamento de uma inspeção de software (KALINOWSKI et al., 2004).

Conclusão

O objetivo de inspeções de software é melhorar a qualidade de artefatos de software através de sua análise, detectando e removendo defeitos antes que o artefato seja passado para a próxima fase do processo de desenvolvimento de software. Conforme visto neste artigo, a aplicação de inspeções entre as atividades do ciclo de vida de software pode trazer diversos benefícios para organizações de software.

O processo de inspeção foi inicialmente elaborado por FAGAN (1976). Argumentos da literatura e estudos experimentais, visando avaliar este processo, vêm indicando reuniões assíncronas para inspeções de software. Tendo em vista as inspeções com reuniões assíncronas SAUER et al., (2000), baseados em estudos experimentais, apresentam uma reorganização com mudanças para reduzir o custo e o tempo total da realização deste tipo particular de inspeção.

Atualmente muitas organizações realizam revisões, mas a forma como as revisões são realizadas ainda é pouco sistematizada e pouco conhecimento da área de inspeções de software é utilizado. Assim o verdadeiro potencial das revisões raramente é explorado, introduzindo um fator de confusão entre os resultados esperados pelas revisões e os obtidos na prática. Visando endereçar este problema e permitir a realização de inspeções mais sistemáticas algumas propostas de apoio ferramental surgiram. Entre estas propostas destacamos a infra-estrutura ISPIS, desenvolvida na COPPE/UFRJ. no contexto da dissertação de mestrado do Marcos Kalinowski (um dos autores deste artigo).

Reconhecimento

Aos pesquisadores Forrest Shull, Guilherme Horta Travassos e Jeffrey Carver por suas contribuições acadêmicas para a área de inspeções de software, fundamentais para a elaboração da tese de mestrado que, entre outros, reúne os conhecimentos discorridos neste artigo. ●

Referências

- ACKERMAN, A., BUCHWALD, L., LEWSKI, F., 1989, "Software Inspections: An Effective Verification Process", IEEE Software, vol. 6, no. 3, pp. 31-37.
- KALINOWSKI, M., SPÍNOLA, R. O., TRAVASSOS, G. H., Infra-Estrutura Computacional para Apoio ao Processo de Inspeção de Software. No: Simpósio Brasileiro de Qualidade de Software, 2004, Brasília.
- BOEHM, B. W., BASILI, V. R., 2001, "Software Defect Reduction Top 10 List.", IEEE Computer 34 (1): 135-137.
- BOEHM, B. W., ABTS, C., BROWN, A. W., CHULANI, S., CLARK, B. K., HOROWITZ, E., MADACHY, R., REIFER, D., STEECE, B., 2000, Software Cost Estimation with COCOMO II, Prentice Hall.
- BOEHM, B. W., 1981, Software Engineering Economics, Prentice Hall.
- CIOLKOWSKI, M., LAITENBERGER, O., BIFFL, S., 2003, "Software Reviews: The State of the Practice", IEEE Software 20 (6): 46-51.
- FAGAN, M. E., 1976, "Design and Code Inspection to Reduce Errors in Program Development", IBM Systems Journal, vol. 15, no. 3, pp. 182-211.
- GILB, T., GRAHAM, D., 1993, "Software Inspection", Addison-Wesley.
- Grünbacher, P., Halling, M., Biffl, S., "An Empirical Study on Groupware Support for Software Inspection Meetings", Proceedings of the 18th IEEE International Conference on Automated Software Engineering, 2003.
- JONES, C., 1991, "Applied Software Measurement", McGraw Hill.
- Kalinowski, M., Travassos, G. H., "A Computational Framework for Supporting Software Inspections", In: 19th IEEE International Conference on Automated Software Engineering - ASE'04, pp. 46-55, Linz, Austria, 2004.
- LAITENBERGER, O., ATKINSON, C., 1999, "Generalized Perspective Based Inspection to handle Object Oriented Development Artifacts", Proceedings of ICSE 99, p. 494-503, Los Angeles, CA, USA.
- Lanubile, F., Mallardo, T., CALEFATO, F., 2003, "Tool support for Geographically Dispersed Inspection Teams", Software Process Improvement and Practice, 2003, 8: 217-231 (DOI: 10.1002/spip.184).
- PRESSMAN, R. S., 2001, Software Engineering: A Practitioner´s Approach, Fifth Edition, McGraw Hill.
- SAUER, C., JEFFERY, D. R., LAND, L., YETTON, P., 2000, "The Effectiveness of Software Development Technical Review: A Behaviorally Motivated Program of Research", IEEE Transactions on Software Engineering, 26 (1): 1-14, January.
- WHEELER, D. A., BRYKEZYNSKI, B., MEESON, R. N., 1996, Software Inspections: An Industry Best Practice, IEEE Computer Society.

Chegou o Sistema de Créditos DevMedia

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

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

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

Todas as vídeos que você sempre quis ver a partir de **R$2,50!**

Saiba mais sobre o Sistema de Créditos!

DevMedia GROUP

www.devmedia.com.br/creditos