ENGENHARIA DE SOFTWARE I

Editora da ULBRA

Dados Internacionais de Catalogação na Publicação (CIP)

G216e Garcia, Luís Fernando Fortes.
Engenharia de software I / Luís Fernando Fortes Garcia. – Canoas: Ed. ULBRA, 2013.
104p.

1. Computação. 2. Engenharia de software. 3. Projeto e arquitetura de software. I. Título.

CDU: 681.3.41

Setor de Processamento Técnico da Biblioteca Martinho Lutero - ULBRA/Canoas

ISBN 978-85-7528-490-2

Editoração: Roseli Menzen

Dados técnicos do livro
Fontes: Palatino Linotype, Franklin Gothic Demi Cond
Papel: offset 75g (miolo) e supremo 240g (capa)
Medidas: 15x22cm

APRESENTAÇÃO

Prezados(a) alunos(a),

Sejam bem vindos(as) a disciplina de Engenharia de Software I.

Esta disciplina apresenta a proposta da área de Engenharia de Software para estruturação e organização do processo de desenvolvimento de software, bem como a sua área coirmã de Qualidade de Software.

Visa abordar e analisar os principais tópicos relacionados a desenvolvimento profissional de software, contribuindo para a discussão dos pontos positivos e negativos de cada conceito.

O objetivo principal da disciplina é disseminar o conceito de "respeito" ao software, isto é, software é atualmente importante demais na sociedade para ser desenvolvido de forma artesanal, sem cuidados e projetos. As consequências de um desenvolvimento desleixado são terríveis em termos de perdas em vidas humanas e perdas financeiras!

São abordados, na parte de Engenharia de Software, conceitos de Processos de Software, de Engenharia de Requisitos, de Projeto e Arquitetura de Software, Testes de Software e evolução/Sistemas Legados.

Na segunda parte do texto, em Qualidade de Software, são apresentados conceitos relacionados a Qualidade, Qualidade de Software em seus dois aspectos – Qualidade de Produtos de Software e Qualidade de Processos de Software, bem como discutidos os modelos de Maturidade em Qualidade de Softwares, representados pelo CMMI e pelo MPS.BR.

Todos os tópicos aqui discutidos têm aplicabilidade na vida prática de um profissional de desenvolvimento de software, seja ele um programador, um analista, um arquiteto ou um testador. O sucesso do processo é dado pela integração correta e junção de esforços de todos estes profissionais, em conjunto com os clientes/usuários.

A Engenharia de Software está em constante atualização e redesenho – os Métodos Ágeis são um dos exemplos mais claros disso. O estudo continuado da área é imprescindível para o seu sucesso profissional!

Um bom trabalho a todos!

Prof. Dr. Luís Fernando Fortes Garcia

Professor/pesquisador na ULBRA/Canoas e Faculdade Dom Bosco de Porto Alegre

Consultor em Carreiras de TI na ofiTIo Consultoria Ltda.

SOBRE O AUTOR

Luís Fernando Fortes Garcia, prof. Dr.

Doutor e mestre em Ciências da Computação pelo Instituto de Informática da UFRGS – Universidade Federal do Rio Grande do Sul. Professor universitário e pesquisador nas áreas de Engenharia de Software, Qualidade de de Software e Governança de TI na ULBRA e Faculdade Dom Bosco de Porto Alegre.

Sócio-diretor e consultor em carreiras de TI na ofoTIo consultoria Ltda.

SUMÁRIO

1 INTRODUÇÃO À ENGENHARIA DE SOFTWARE

Luís Fernando Fortes Garcia, prof. Dr.

Este capítulo apresenta a grande área da Engenharia de Software, seu histórico junto aos desenvolvedores de software ao longo do tempo, alguns questões básicas que devem ser consideradas em todos os momentos pelos profissionais, suas definições, etapas, focos (pirâmide da Engenharia de Software), bem como seus princípios e qualidades desejáveis.

O principal objetivo deste capítulo é apresentar a Engenharia de Software como uma alternativa prática e adequada ao correto desenvolvimento de produtos de software, com o devido **respeito** que estes produtos de software merecem.

1.1 Computadores e Software

Atualmente tem-se a total e completa onipresença dos computadores – e consequentemente do software – em nossas vidas. Eles estão em forma de computadores de mesa, servidores, notebooks, smartphones, tablets e mesmo embarcados/embutidos nos mais diferentes aparelhos, desde geladeiras até em um automóvel. Sem eles, não podemos fazer uma ligação telefônica, acender uma simples lâmpada, fazer transações financeiras, viajar de avião, entre outras coisas do cotidiano.

Esses mesmos softwares podem ser considerados abstratos, intangíveis, complicados, diferentes, complexos e sem limitações. Visto que não são regidos pelas leis da física, a princípio não há limites em seu desenvolvimento. A ausência de limites, entretanto, é fator complicador extremo para seu desenvolvimento.

É importante ressaltar os inúmeros aspectos positivos dos softwares em nossa sociedade, como aumento de produtividade de uma empresa, velocidade de processamento de dados e economia de tempo, entre outros. Entretanto – quando há falhas e bugs – existem problemas ou aspectos negativos que não podem ser ignorados:

- Quedas de aviões;
- Desvios de mísseis;
- Colapsos em centrais telefônicas;
- Desvios bancários.

Para tornar, então, o software mais confiável, robusto e eficaz, propõe-se o estudo e a aplicação da Engenharia de Software.

1.2 Evolução Histórica do Software

Para fins de avaliação histórica do software, de seus tipos e de seus processos de desenvolvimento, pode-se classificá-los em 4 grandes gerações. Cada avanço de geração agrega complexidade e aumento de demanda por software.

- 1ª. Geração – 1950 a 1960 – Software para sistemas computacionais de grande porte (Mainframes) com orientação de processamento em lotes (batch), com distribuição limitada apenas a grandes empresas;
- 2ª. Geração – 1960 a 1975 – Software para sistemas de médio porte, com suporte a acessos multiusuários, de tempo real, e bancos de dados. Nesta geração, inicia-se o acesso a softwares por empresas de menor porte;
- 3ª. Geração – 1975 a 1990 – Software para computadores de pequeno porte – microcomputadores – ligados em redes locais. Início da utilização por pessoas físicas e pequenas empresas, o que aumentou grandemente a demanda;
- 4ª. Geração – 1990 a atual – Softwares distribuídos em todas as áreas da sociedade, extremamente poderosos e complexos (sistemas inteligentes, distribuídos, paralelos), rodando em todos os tipos de dispositivos. Chega-se ao pico da complexidade e da demanda.

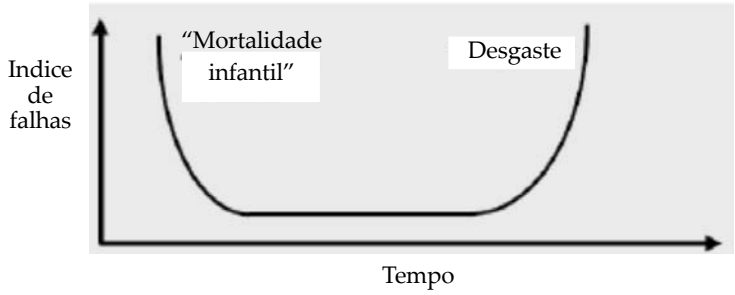
É importante ressaltar que a evolução não para nesta 4ª. Geração – novas pesquisas e desenvolvimentos estão sendo realizados em empresas e centros de pesquisa espalhados por todo o mundo. Estes novos softwares estão cada vez mais complexos e úteis, e por isso precisam ser desenvolvidos com processos produtivos e controlados.

1.3 Questões sobre Desenvolvimento de Software

O processo de desenvolvimento de software suscita algumas importantes questões – e sua discussão é necessária:

- Software é desenvolvido, não fabricado – Software, diferentemente de outros produtos, como celulares e sapatos, normalmente é desenvolvido sob encomenda para clientes e aplicações específicas. Cada software tende a ser, em menor ou maior grau, diferente dos demais. Isso impede que técnicas de fabricação em escala (industriais) possam ser utilizadas em seu desenvolvimento, que poderiam diminuir custos e prazos;
- Software não desgasta – Software, ao contrário do hardware – em que se pode facilmente perceber o desgaste de peças e engrenagens – não desgasta no sentido literal da palavra. Entretanto, instalações de software "desgastam" ao longo do tempo, e, portanto, necessitam de manutenção e de reparos. Um exemplo clássico é a instalação de um sistema operacional em um microcomputador; No início, tem-se velocidade e estabilidade, mas, com o passar do tempo (e pela instalação e desinstalação de programas, jogos, etc) a velocidade do sistema cai e os problemas aparecem, como apresentado na figura 1.1.

Figura 1.1 – Desgaste em Software.



- Natureza mutável do software – Esta é uma questão central no desenvolvimento de software. Ao contrário de outros produtos, em que as mudanças são raras e, quando existem, são implementadas somente após exaustivas discussões e projetos/simulações – como em prédios, estradas e automóveis – o software apresenta demandas de alterações constantes, muitas vezes diárias. Estas alterações partem tanto dos clientes, que precisam acompanhar o mercado, quanto dos governos e órgãos de regulação, em formas de novas leis, novos cálculos de impostos e taxas. Portanto raramente pode-se considerar o desenvolvimento de um software concluído.

Uma conferência da OTAN (Organização do Tratado do Atlântico Norte) entre 1968 e 1969 cunhou o termo "Crise de Software", ao discutir e apresentar questões, na época, relacionadas ao desenvolvimento de softwares:

- Cronogramas não observados – atrasos constantes;
- Projetos abandonados;
- Módulos que não operam corretamente quando combinados;
- Programas que não fazem exatamente o que era esperado;
- Sistemas tão difíceis de usar e que são descartados;
- Sistemas que simplesmente param de funcionar.

É importante refletir que – passados mais de 40 anos – muitos (a maioria) desses problemas apresentados continuam a existir.

1.4 Enfoques do Desenvolvimento de Software

O desenvolvimento de software pode ser dividido em dois enfoques:

- Artesanal;
- Engenharia de Software.

O desenvolvimento Artesanal (também conhecido por Gambiarra, P.O.G e outros) baseia-se exclusivamente em habilidades pessoais do desenvolvedor para a construção do software. Apresenta um processo bastante simplificado, em que normalmente, após uma breve conversa do programador com o cliente para a obtenção dos requisitos, é iniciada a programação, sem etapas importantes, como análise, projeto e testes. Implementa uma técnica conhecida como "tentativa e erro".

É um processo que, apesar de funcionar em alguns casos, é muito arriscado para os padrões atuais da indústria de software. Tom de Marco, um dos precursores da Engenharia de Software, afirma que "na falta de padrões expressivos, uma nova indústria, como a de software, passa a depender de folclore".

O desenvolvimento baseado em Engenharia de Software, por sua vez, apresenta uma abordagem mais sistemática, produtiva, econômica e organizada do processo de desenvolvimento, com a aplicação de etapas bem definidas, com documentação e acompanhamento por profissionais habilitados e especializados em suas tarefas, sendo estas relacionadas a requisitos, análise, projeto, programação, testes e implantação.

1.5 Engenharia de Software

A primeira definição da Engenharia de Software descreve-a como: "Estabelecimento e uso de sólidos princípios de engenharia para que se possa obter economicamente um software que seja confiável e que funcione eficientemente em máquinas reais", por Fritz Bauer, em 1969.

Esta definição é muito relevante, porque destaca claramente os aspectos mais importantes da Engenharia de Software:

- "Estabelecimento e uso" - Ou seja, não adianta definir e estabelecer regras e processos, mas sim, é necessário seu efetivo uso para a obtenção das vantagens da Engenharia de Software;
- "Sólidos princípios de engenharia" - desenvolvimento de software baseado nos mesmos princípios das engenharias tradicionais – civil, elétrica – que incluem projetos, cálculos, simulações e testes;
- "Se possa obter economicamente" – Apresenta o maior fator motivador da implantação da Engenharia de Software – o econômico. O objetivo final da Engenharia de Software é produzir produtos com mais qualidade, com mais produtividade, visando o lucro do desenvolvedor;
- "Que seja confiável" – No contexto atual da sociedade, este é fator indispensável para a sobrevivência de um software no mercado. Produtos não confiáveis são rapidamente descartados pelos usuários.

O SEI (Instituto de Engenharia de Software da Universidade Carnegie Mellon, USA) define que "Engenharia é a aplicação sistemática de conhecimentos científicos na criação e construção de soluções com um bom custo-benefício para a resolução de problemas práticos da sociedade".

O IEEE (Instituto de Engenheiros Eletricistas e Eletrônicos, dos Estados Unidos) define a Engenharia de Software como: "Aplicação de uma abordagem sistemática, disciplinada e quantificável para o desenvolvimento, operação e manutenção do software".

O IEEE edita o SWEBOK (Guide to the Software Engineering Body of Knowledge), que é o corpo de conhecimento que apresenta de forma completa a Engenharia de Software, suas áreas de conhecimento e as suas disciplinas relacionadas.

As áreas da Engenharia de Software, segundo o SWEBOK, são:

- Requisitos de software;
- Projeto de software;
- Construção de software;

- Teste de software;
- Manutenção de software;
- Gerenciamento de configuração de software;
- Gerenciamento de engenharia de software;
- Processo de engenharia de software;
- Ferramentas e métodos da engenharia de software;
- Qualidade de software.

A área de Requisitos de Software aborda a elicitação, análise, especificação e validação de requisitos de software.

Projeto de Software é definido como o processo de definição da arquitetura, componentes, interfaces e outras características de um sistema ou componente, assim como o resultado desse processo.

Construção de software se refere à criação detalhada de software relevante e funcional a partir de uma combinação de codificação, verificação, teste unitário, teste integrado e debugging.

Teste de software consiste numa verificação dinâmica do comportamento de um programa em um conjunto finito de casos de teste contra o comportamento esperado.

Manutenção de software aborda defeitos e novos requisitos durante a operação do software.

Gerência de Configuração de Software é um processo que envolve a gestão de projetos, as atividades de desenvolvimento, manutenção e garantia.

A Gerência de Engenharia de Software pode ser definida como a aplicação de atividades de gestão - planejamento, coordenação, medição, monitoramento, controle e divulgação – para garantir que o desenvolvimento e a manutenção de software seja sistemática, disciplinada e quantificada.

O processo de engenharia de software inclui atividades técnicas e de gestão dentro dos processos do ciclo de vida de software. Além disso, está preocupado com a definição, implementação, avaliação, gerenciamento da mudança e melhorias nos próprios processos do ciclo de vida de software.

Ferramentas de desenvolvimento de software são ferramentas baseadas em computador que apoiam os processos de ciclo de vida de software. Os métodos impõe uma estrutura na atividade de engenharia de software.

A área de Qualidade de Software enfoca questões relacionadas à qualidade do software, tanto do produto quanto do processo de desenvolvimento do software.

As disciplinas relacionadas com a Engenharia de Software, segundo o SWEBOK, incluem, são:

- Engenharia da computação;
- Ciência da computação;
- Administração;
- Matemática;
- Gestão de projetos;
- Gestão da qualidade;
- Ergonomia de software;
- Engenharia de sistemas.

1.6 Pirâmide da Engenharia de Software

A Engenharia de Software pode ser representada em forma de uma pirâmide (figura 1.2), em que os conceitos mais importantes ficam em sua base e os mais voláteis (e menos importantes) em seu topo.

Figura 1.2 – Pirâmide da Engenharia de Software

O conceito mais importante no estudo e prática da Engenharia de Software é seu foco na qualidade, ou seja, o atendimento e satisfação dos desejos e necessidades do usuário. Esta é a razão de ser da Engenharia de Software.

A qualidade é garantida pelo uso de processos de desenvolvimento de software corretos e robustos, com produtividade e eficiência. Estes processos podem ser ditos tradicionais como modernamente denominados de ágeis.

Os processos, por sua vez, são implementados através de métodos, que descrevem como o software deve ser desenvolvido.

E, finalmente, as ferramentas são facilitadoras de todo esse processo. Nesta categoria entram as linguagens de programação, ferramentas de modelagem, ferramentas de teste, entre outras. São voláteis, pois facilmente são superadas e substituídas por novas e que apresentam melhor desempenho.

1.7 Princípios da Engenharia de Software

A Engenharia de Software, com seu foco em qualidade, baseia-se em uma série de princípios, que são declarações gerais e abstratas, e que descrevem as propriedades desejadas dos processos de desenvolvimento e de seus produtos relacionados.

Os principais princípios abordam tópicos como:

- Rigor e formalismo – O processo de desenvolvimento de software é uma atividade baseada fortemente em criatividade e inspiração. O rigor e o formalismo devem coexistir como forma de complemento a esta criatividade. As próprias linguagens de programação já aplicam este princípio, visto que desvios e falhas de codificação não são toleradas pelos compiladores.
- Separação de preocupações – Este princípio visa tratar individualmente de diferentes aspectos de um problema, de forma a concentrar esforços separadamente. Segundo ele, a única forma de dominar a complexidade do projeto é separar as preocupações e decisões do projeto. Primeiramente, alguém deve tentar isolar problemas que estão menos relacionados com outros. Como exemplos desta separação, pode-se considerar regras de negócio e interface.
- Modularização – O princípio da modularização visa primordialmente dividir a complexidade. Softwares que não são divididos de alguma forma são considerados "monolíticos", enquanto softwares baseados em módulos são chamados de "modulares". O principal benefício da modularização é que ela permite que o princípio da separação de preocupações seja aplicado em duas fases. Primeiramente quando tratarmos dos detalhes de cada módulo isoladamente (ignorando detalhes dos outros módulos) e, posteriormente, quando tratarmos das características globais de todos os módulos, incluindo seus relacionamentos, o que possibilita interligá-los para formar um sistema íntegro e coeso. Em software modulares, busca-se bastante coesão (todas as informações relacionadas ao módulo devem estar contidas no mesmo) e pouca inter-relação (necessidade de ligação entre módulos distintos).
- Abstração – Processo pelo qual se identifica os aspectos importantes de um fenômeno, ignorando seus detalhes.
- Antecipação de mudanças – Princípio que prega a habilidade de evolução do software, pela preparação prévia para mudanças. Atualmente, pela

disseminação dos processos ágeis de desenvolvimento de software, este princípio é bastante controverso.

- Generalização – Princípio que orienta o foco do desenvolvimento no problema mais geral e abrangente do negócio do cliente, partindo posteriormente para os problemas mais específicos e detalhados.
- Incrementabilidade - é o princípio que busca a perfeição ou a obtenção dos objetivos através de passos que evoluem (ou são incrementados) ao longo do tempo.

Estes princípios apresentados devem ser considerados como guias de boas práticas para os processos de desenvolvimento de software, independentemente se tradicionais ou ágeis.

1.8 Qualidades Desejáveis em um Software

Tanto os processos de desenvolvimento quanto os produtos de software, para terem sucesso e aceitação no mercado, devem apresentar qualidades básicas, como:

- Corretude – O software deve ser correto, isto é, funcionar de acordo com sua especificação formal.
- Confiabilidade – Um software é considerado confiável quando os seus usuários podem depender dele. Este conceito, porém, é bastante relativo e depende dos tipos de usuários envolvidos. O próprio sucesso de mercado é um indicador de confiabilidade: produtos não confiáveis não sobrevivem por muito tempo.
- Robustez – Um software, para ser robusto, deve apresentar capacidade de recuperar-se de erros e problemas não previstos, como quedas de energia, falhas de hardware etc. O nível de robustez requerido depende do tipo de aplicação – jogos e aplicações simples podem ter robustez menor que aplicações de missão criticam, como controles de usinas nucleares.
- Performance – A qualidade de performance diz respeito ao desempenho do software em determinado hardware. Problemas de performance afetam principalmente sua usabilidade e, assim como a robustez, dependem do propósito da aplicação.
- Amigabilidade – Qualidade de um software com facilidade de utilização por parte de usuários. Esta qualidade é relativa e depende do nível e perfil do usuário.
- Manutenabilidade – Qualidade de todos os softwares com facilidade de alterações após sua liberação. Pode ser classificada como corretiva, que possibilita a correção de defeitos e problemas; perfectiva, que possibilita a

melhoria continua do produto ao longo do tempo de vida útil; e adaptativa, que aborda a adaptação do software a diferentes plataformas computacionais.

- Reusabilidade – Qualidade de processo de desenvolvimento que prega a reutilização não somente de código-fonte, mas de aspectos de projeto e teste.
- Portabilidade – Qualidade do software com capacidade de execução em diferentes plataformas computacionais. Esta qualidade atualmente é importante devido ao grande número de plataformas computacionais a disposição do usuário – computadores de mesa, notebooks, smarthphones e tablets, por exemplo.
- Interoperabilidade – Qualidade referente ao software que permite interação com demais software para fins de compartilhamento de dados e informações. Um exemplo bastante conhecido de software com interoperabilidade é do Microsoft Office, que permite a troca de dados entre seus aplicativos de edição de texto, planilha eletrônica e de apresentações, sem necessidade de redundância.

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	Engenharia de Software se caracteriza pelo estabelecimento de sólidos princípios para se obter software confiável e que funcione.
	O objetivo final da Engenharia de Software é o lucro e a confiabilidade.
	Definir a linguagem de desenvolvimento é mais importante do que definir o processo de desenvolvimento.
	Rigor e formalismo anulam o fato do desenvolvimento de software ser baseado em criatividade e inspiração.
	O objetivo da modularização é produzir software com baixa coesão e muito inter-relacionamento.

Bibliografia

ANDRADE, Vítor. *Introdução ao Guia SWEBOK*. Disponível em http://www.cin.ufpe.br/~processos/TAES3/slides-2012.2/Introducao_SWEBOK.pdf.

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre: McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo: Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

V V F F F

2 PROCESSOS DE SOFTWARE

Luís Fernando Fortes Garcia, prof. Dr.

Este capítulo apresenta os Processos de Desenvolvimento de Software e os Modelos de Desenvolvimento de Software Tradicionais.

Estes modelos são importantes como guias de boas práticas em ambientes tradicionais de desenvolvimento de software. São discutidos aspectos positivos e negativos de cada modelo, bem como sua aplicabilidade.

2.1 Processos e Software

Processo de software, segundo Sommerville (2011) é um conjunto de atividades relacionadas que levam à produção de software, e um conjunto de etapas ou níveis que formam o que se convencionou chamar de "ciclo de vida" de um software.

Independentemente do modelo de processo de software, se tradicional ou ágil, pode-se destacar cinco atividades ou etapas fundamentais para o processo de desenvolvimento de software:

- Especificação – Requisitos e funcionalidades que devem ser implementadas no software – "o que" deve ser feito;
- Projeto – Etapa onde define-se como o software deverá ser desenvolvido – "como" o software será feito;
- Implementação – Etapa relacionada a codificação (programação) do software;
- Validação – Etapa onde o software deve ser validado (testado) frente aos requisitos;
- Evolução – Etapa posterior à entrega/homologação que visa manter/corrigir e evoluir o software.

2.2 Modelos de Processos de Software

Um modelo de processo de software é uma representação abstrata do processo. Ele apresenta a descrição de um processo a partir de uma perspectiva particular, mostrando as atividades e sua sequência (Sommerville, 2011). São usados para expressar abordagens de desenvolvimento de software e, geralmente, podem e devem ser adaptados e personalizados para atendimento de demandas específicas.

Os modelos podem, então, ser classificados em Artesanais, Tradicionais ou Ágeis.

- Artesanais – Modelos de desenvolvimento sem qualquer preocupação formal com a Engenharia de Software. Não há etapas, documentos ou papéis (atores) formalmente definidos. É popularmente conhecido como "gambiarra".
- Tradicionais – Modelos clássicos, que foram criados a partir da década de 1970 e implantam a maioria dos conceitos e práticas da Engenharia de Software;
- Ágeis – Modelos modernos, criados a partir de dissidências entre Engenheiros de Software e grupos de desenvolvedores experientes, que questionam e se opõem a vários princípios da Engenharia de Software. Sugerem novas práticas e abordagens para o desenvolvimento de software.

Os modelos ágeis – especialmente a Extreme Programming (XP) e o SCRUM serão detalhadamente abordados no próximo capítulo deste material.

2.3 Modelo Cascata

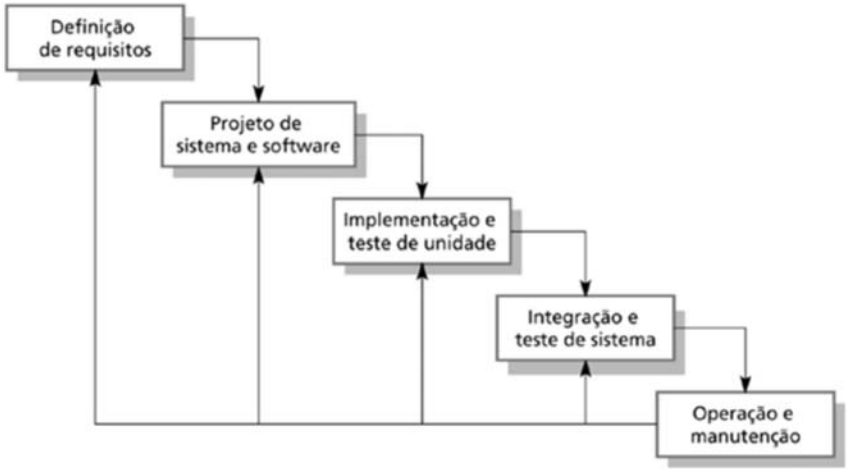
O modelo Cascata (ou *waterfall*) é o modelo de desenvolvimento de software mais antigo e mais clássico. Ainda é bastante utilizado em empresas de desenvolvimento de software ao redor do mundo, entretanto, em versões adaptadas/atualizadas.

A versão original do modelo – extremamente rígida e linear - não poderia ser utilizada atualmente frente ao novo contexto mundial, que apresenta mudanças frequentes de requisitos, equipes multidisciplinares com trabalho em paralelo, entre outras limitações ao modelo original. O fato de exigir particionamento inflexível do projetos em etapas estanques dificulta a resposta aos requisitos de mudança dos clientes.

O modelo cascata é particularmente adequado a grandes projetos de software, como sistemas de ERP, e a equipes de desenvolvimento geograficamente distribuídas, em que a divisão de etapas e a documentação são fatores positivos para a gerência do projeto.

As etapas do modelo cascata podem ser observadas na figura 2.1:

Figura 2.1 - Modelo Cascata. Fonte: (Sommerville, 2011)

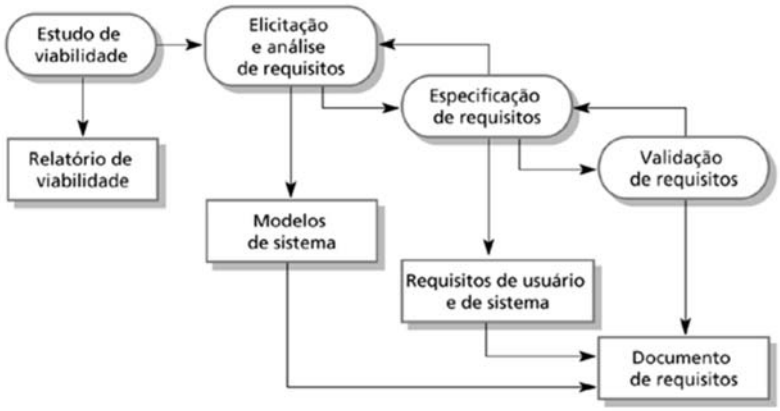


Etapa de Engenharia de Requisitos – Esta etapa inclui o processo de engenharia de requisitos ("O QUE" deve ser feito), com fases de **elicitação** (obtenção através de entrevistas, questionários, protótipos), **especificação** (usando técnicas de modelagem como UML, através de diagramas de casos de uso, classes e sequência) e **validação e verificação** (verificar se tanto os requisitos estão descritos e modelados tecnicamente corretos - em UML correta, por exemplo - quanto se representam os anseios corretos dos clientes/usuários).

Os profissionais responsáveis por esta etapa incluem os Analistas, tanto Analistas de Negócios e quanto Analistas de Sistemas. Os Analistas de Negócios – profissionais com perfil mais especializado em regras de negócios e gestão de pessoas – normalmente ficam com as primeiras etapas, relacionadas à abordagem e obtenção dos requisitos, e os Analistas de Sistemas com as etapas seguintes, relacionadas ao registros/especificação e validação/verificação dos requisitos.

Um subprocesso da Engenharia de Requisitos pode ser observado na figura 2.2:

2.2 – Engenharia de Requisitos. Fonte: (SOMMERVILLE, 2011)



Destaca-se a subetapa de **Estudo de Viabilidade**, onde são realizados estudos de viabilidade técnica e econômica acerca do possível desenvolvimento do software.

Os fatores que devem ser levados em consideração nesta subetapa incluem viabilidade comercial/econômica (o mercado para sua comercialização) e técnica (os conhecimentos técnicos requeridos para seu desenvolvimento, em plataforma computacional - desktop, móvel, distribuída -, linguagens de programação, sistemas de banco de dados e testes).

Neste ponto, costuma-se dividir o software em dois tipos: Software de Prateleira e Software sob Encomenda. Software de Prateleira é aquele desenvolvido pela empresa ainda sem nenhum cliente específico, para posterior venda em empresas especializadas, em lojas e sites de comércio eletrônico. Software sob Encomenda, por sua vez, é aquele desenvolvido pela empresa sob encomenda particular e direta de determinado cliente.

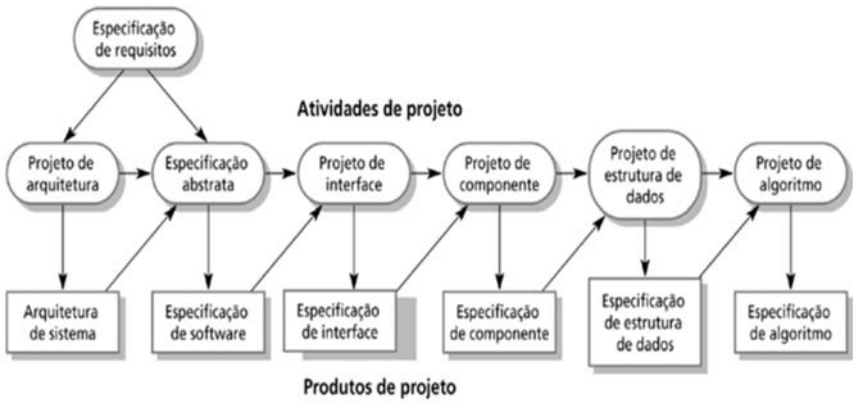
Ao contrário que se pode pensar, mesmo Softwares sob Encomenda devem passar por esta subetapa, visto que mesmo tendo garantia a comercialização podem apresentar problemas técnicos e/ou de projetos, relacionados a prazos e custos. Insistir no desenvolvimento de um software com viabilidade incerta é ter problemas no futuros, como perdas de mercado, perdas financeiras e atrasos, bem como problemas relacionados à imagem da empresa e dos profissionais envolvidos.

Projeto – Esta etapa inclui atividades relacionadas à definição da melhor solução técnica a ser aplicada no desenvolvimento do software. Aqui define-se "COMO" o software será feito sem, necessariamente, fazê-lo/implementá-lo.

O profissional responsável por esta etapa é conhecido por "arquiteto de software", que tradicionalmente foi, antes, um programador experiente, com boa visão tanto técnica quanto relacionada à gerência de projetos – prazos, riscos e custos.

Um subprocesso desta etapa pode ser observado na figura 2.3:

2.3 – Projeto de Software. Fonte: (SOMMERVILLE, 2011)



A partir da especificação dos requisitos (produto da fase anterior do modelo cascata) o arquiteto inicia a definição da solução técnica, incluindo a arquitetura do sistema (divisão do sistema em módulos e suas interfaces (ligações), especificação de partes de código, chegando à definição de estruturas de dados (como matrizes e listas encadeadas) e, ainda, se necessário, à exemplos de algoritmos que possam guiar o programador na maneira mais adequada na implementação do software. Nesta fase, na prática, são definidos tanto o ambiente de desenvolvimento, as linguagens de programação, os bancos de dados e os frameworks a ser utilizados.

É considerada fase vital para o sucesso do desenvolvimento do software, visto que uma implementação incorreta (ou não adequada) pode levar a baixo desempenho (principalmente relacionado a integração com bases de dados), duplicidade desnecessárias de dados, entre outros problemas.

Implementação – Nesta fase, ocorre a codificação/programação dos requisitos de software, baseado nas definições técnicas da fase de projeto. A implementação atualmente utiliza linguagens de programação visuais e orientadas a objeto, com ambientes de desenvolvimento fáceis e amigáveis.

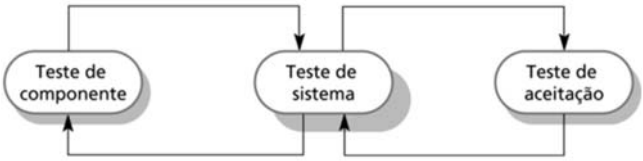
O profissional responsável por esta etapa é conhecido por Programador ou Desenvolvedor, cujo perfil apresenta excelentes capacidades lógicas e analíticas.

Teste de Software – Esta etapa visa aplicar testes de software que buscam descobrir erros e falhas que porventura tenham ficado desde as etapas anteriores. O processo de teste nunca é 100% efetivo, e muitos erros certamente ainda restarão para serem descobertos em ambiente de produção, ou seja, pelos próprios clientes e usuários.

O profissional desta etapa do modelo de desenvolvimento é conhecido genericamente como Testador, com variações entre Gerente de Testes, Projetista de Testes, Testador, Automatizador de Testes, entre outros.

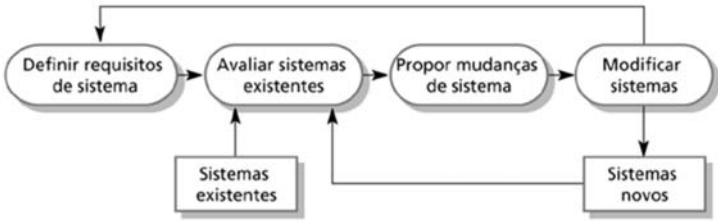
O subprocesso (figura 2.4) de teste envolve deste o teste unitário e específico de cada componente/módulo do software (teste de componente), passando por um teste de integração destes componentes/módulos (teste de sistema) e chegando ao teste de aceitação por parte do cliente/usuário, também conhecido por teste de homologação.

2.4 – Teste de Software. Fonte: (SOMMERVILLE, 2011)



Evolução – Esta fase inicia após a entrega do produto de software e inclui atividades de manutenção e evolução do software (figura 2.5).

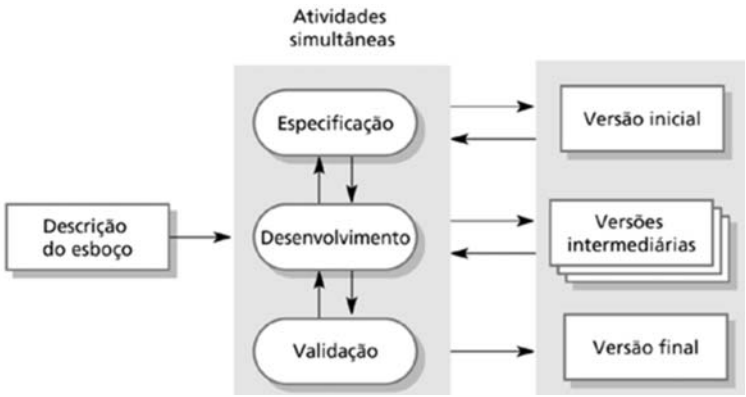
2.5 – Evolução de Software. Fonte: (SOMMERVILLE, 2011)



2.4 Modelo Evolutivo ou Evolucionário

O Modelo Evolutivo baseia-se no levantamento rápido de requisitos, projeto rápido e na construção evolutiva de protótipos de software, que serão avaliados e refinados até a liberação do mesmo. O Modelo Evolutivo pode ser observado na figura a seguir:

2.6 – Modelo Evolutivo. Fonte: (SOMMERVILLE, 2011)



Seu objetivo principal é a solidificação (certeza) nos requisitos juntos aos clientes/usuários, sendo a direção do desenvolvimento determinada pela experiência operacional.

A aplicabilidade do Modelo Evolutivo dá-se em sistemas interativos de pequeno e médio portes e sistemas com ciclo de vida curto, para aplicações específicas e datadas. Um software de gerência de um sorteio de uma emissora de rádio pode ser considerado um exemplo de aplicabilidade do modelo, visto que é simples, agrega funcionalidades aos poucos (cadastros de participantes, cadastros de brindes e, finalmente, mecanismos de sorteio). Além da simplicidade, este software terá uma vida útil bastante curta, em paralelo com o período de vigência da promoção.

Como problemas, pode-se citar a falta de visibilidade geral do processo, a má estruturação técnica do sistema (pelos projetos rápidos e sem cuidados) e a necessidade de profissionais especializados em linguagens de prototipação rápida. Outro problema a ser destacado é a necessidade constante de iteração por parte do cliente/usuário final nas etapas de validação das versões intermediárias, que pode levar a um grande descontentamento e stress em ambas as partes.

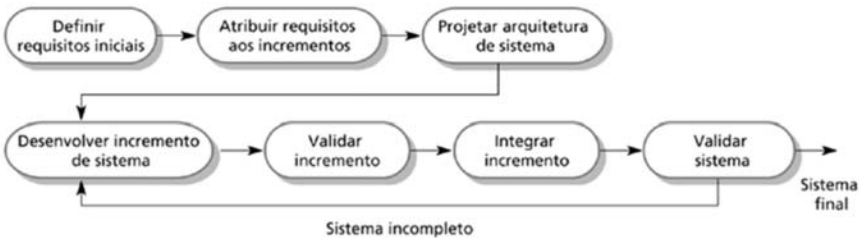
2.5 Modelo Iterativo – Incremental

O modelo Iterativo-Incremental baseia-se na premissa da constante evolução e alteração dos requisitos, de modo que não seria interessante tanto aguardar por uma improvável estabilização quanto desenvolver completamente o sistema para depois adequá-lo na manutenção.

O processo do modelo Iterativo-Incremental (figura 2.7) parte de um conjunto de requisitos iniciais e estáveis, e com base nestes, projeta a arquitetura completa do

sistema (principal diferença em relação ao modelo Evolutivo visto anteriormente). Com base na arquitetura e na agregação dos demais requisitos, o sistema vai sendo construído, passo a passo, com atividades tradicionais de análise-projeto-codificação e testes.

2.7 - Modelo Iterativo-Incremental. Fonte: (Sommerville, 2011)



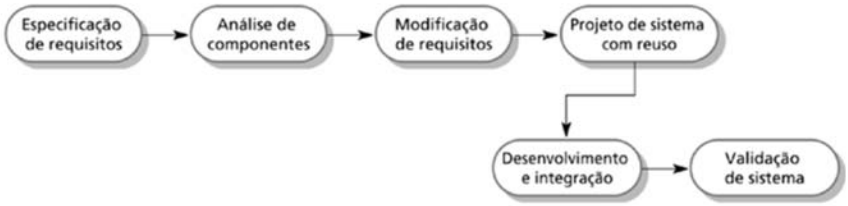
Como possível principal problema deste modelo destaca-se a dependência extrema da definição dos requisitos por parte dos clientes, o que pode afetar criticamente o projeto em relação a prazos e principalmente seus custos.

2.6 Modelo Baseado em Componentes

O Modelo Baseado em Componentes prevê o desenvolvimento do software através da integração de componentes pré-prontos disponíveis no mercado. Atualmente, existem várias empresas no mercado atuando exclusivamente na fabricação de componentes genéricos a serem utilizados em projetos específicos de desenvolvimento de aplicações de software.

O processo deste modelo inicia com a tradicional especificação de requisitos, mas, posteriormente, ao invés de fases como projeto e codificação, parte para uma pesquisa e análise de componentes disponíveis e, caso seja necessário, para sua modificação antes da integração no produto de software final (Figura 2.8).

2.8 - Modelo Baseado em Componentes. Fonte: (Sommerville, 2011)



Certamente nem todos os requisitos particulares podem ser implementados somente com componentes pré-prontos, levando a uma necessidade de desenvolvimento própria dos restantes.

Como principal problema do Modelo Baseado em Componentes destaca-se a necessidade de confiança na qualidade do componente adquirido junto a terceiros, bem como a possível necessidade de modificação em softwares que podem não apresentar documentação adequada.

A aplicabilidade deste modelo dá-se em aplicações de software tradicionais/clássicos, ou seja, com poucos requisitos específicos, como softwares de cadastros e emissão de relatórios.

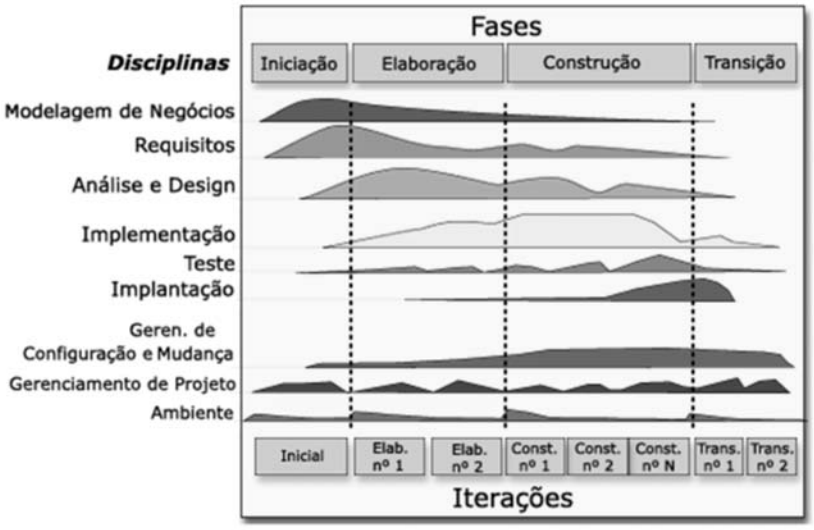
2.7 Modelo RUP – Rational Unified Process

O modelo RUP (Rational Unified Process) é um modelo de desenvolvimento de software derivado da UML e do respectivo processo unificado de desenvolvimento de software, que separa um conjunto de atividades a partir de um conjunto pré-definido e bem conhecido de fases.

Aborda três perspectivas:

- Fases do projeto;
- Atividades do projeto;
- Boas práticas de desenvolvimento.

2.9 - Modelo RUP.



As fases do modelo RUP (figura XX) incluem:

- Iniciação - define o escopo do projeto;
- Elaboração – define os requisitos e a arquitetura do sistema;
- Construção – desenvolvimento do software;
- Transição – implantação do software.

As boas práticas do RUP versam sobre:

- Desenvolvimento iterativo do software;
- Gerência de requisitos;
- Utilização de arquiteturas baseadas em componentização;
- Modelagem visual
- Verificação da qualidade do software;
- Foco em Controle de mudanças do software.

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	O modelo cascata, em sua versão original, pode ser usado por equipes globais que trabalhem em paralelo ...
	Softwares desenvolvidos sob encomenda não precisam passar pela subetapa de estudo de viabilidade, pois já tem cliente garantido ...
	O subprocesso de projeto de arquitetura é considerado muito importante pois pode evitar problemas de integração com bases de dados ...
	O modelo evolutivo se aplica quando os projetos são longos e terão ciclo de vida completo...
	O modelo incremental apresenta forte dependência do relacionamento com o cliente ...
	O modelo baseado em componentes favorece a reusabilidade do software ...

Bibliografia

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre: McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. **Engenharia de Software**. 9ª. Edição. São Paulo: Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

F V V F V

3 PROCESSO ÁGEIS DE SOFTWARE

Luís Fernando Fortes Garcia, prof. Dr.

Este capítulo apresenta os Processos Ágeis de desenvolvimento de software, discutindo os conceitos de agilidade, GAP (Gerenciamento Ágil de Projetos), XP (Extreme Programming) e SCRUM.

Os processos ágeis tornaram-se excelentes alternativas aos processos tradicionais de desenvolvimento de software, visto que implementam/abordam de forma atual/moderna várias questões fundamentais do processo de software, como o gerenciamento de requisitos mutáveis.

3.1 Processos Ágeis de Software

O principal conceito envolvido neste capítulo é o conceito de **Agilidade**:

> *"Agilidade é a habilidade de criar e responder a mudanças como forma de manter a lucratividade num ambiente turbulento de negócios" – Highsmith, 2002;*

Os chamados métodos ágeis surgem em paralelo – fevereiro de 2001 - a um novo contexto da indústria de desenvolvimento de software, onde, nestes novos tempos, temos uma grande demanda de software e, adicionalmente, de um software cada vez mais complexo – sistemas inteligentes, distribuídos, computação em nuvem etc. Além disso, junta-se a este panorama o aumento da questão dos requisitos mutantes, devido a mudanças de mercado, de governo (leis, taxas, impostos etc.).

Entretanto, apesar do movimento ter sido formalizado em 2001, os principais conceitos envolvidos são mais antigos, como citados em (RICO, 2005):

- Times auto-organizáveis – em pesquisa desde 1951, com Bachuk e Goode;
- Prototipação (em Engenharia de Software) – 1982, com McCracken e Jackson;

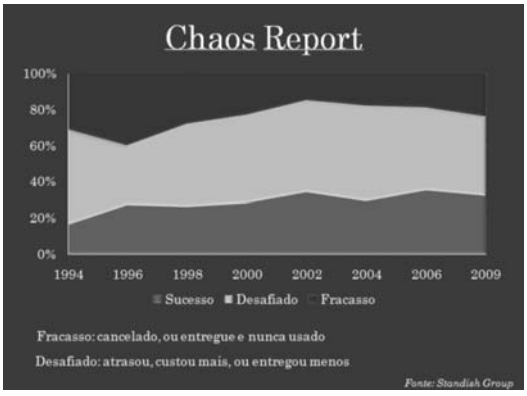
- Releases e iterações – desde 1975, com pesquisas de Basili e Turner;
- Envolvimento do cliente – desde 1978, em artigos de Von Hippel.

Os Métodos Ágeis são uma tentativa – patrocinada originalmente por um grupo de desenvolvedores/programadores experientes e consultores (Kent Beck, notadamente) que questionam e se opõem às práticas de até então da Engenharia de Software – de abordagem e minimização de uma série de problemas da área de desenvolvimento de software clássica/tradicional:

- Suposição de que seja possível prever o futuro;
- Baixa interação com clientes;
- Ênfase em documentos e processos.

Uma famosa pesquisa, o "Chaos Report", do instituto de pesquisas Standish Group, monitora há décadas o mercado de projetos de desenvolvimento de software, classificando-os em "sucesso", "problema" e "fracasso".

Figura 3.1 – Chaos Report. Fonte. Standish Group.



Outro fator analisado pelo Standish Group, dentro do contexto do desenvolvimento de software envolve a questão do percentual das funções utilizadas em um sistema típico de software. A pesquisa descobriu que:

- Partes sempre usadas: 07%
- Partes raramente usadas: 19%
- Partes nunca usadas: 45%
- Partes utilizadas as vezes: 29%

Como se pode ver, gasta-se muito tempo e dinheiro desenvolvendo softwares que provavelmente nunca serão usados.

O movimento, então, surge com a publicação de um manifesto com quatro questões principais (www.agilemanifesto.org/):

1. Indivíduos e interações mais do que processos e ferramentas;
2. Software em funcionamento mais do que documentação abrangente;
3. Colaboração com o cliente mais do que negociação de contratos;
4. Responder a mudanças mais do que seguir um plano.

O principal objetivo deste manifesto é "Satisfazer o cliente, entregando, rapidamente e com frequência, sistemas com algum valor", ou seja, com entregas funcionais e frequentes, com preparação e tratamento de requisitos mutáveis, com pessoal de desenvolvimento e negócios trabalhando juntos e com troca direta de informações, sem burocracias.

Os princípios dos Métodos Ágeis incluem:

- Satisfação do cliente através de entrega contínua e adiantada de software com valor agregado;
- Mudanças nos requisitos são bem vindas, mesmo tardiamente;
- Entregar frequentemente software funcionando;
- Pessoal de desenvolvimento e negócios trabalhando junto;
- Indivíduos motivados;
- Conversas presenciais;
- Andamento do progresso medido pelo funcionamento do software;
- Desenvolvimento sustentável;
- Busca da excelência técnica e do bom design;
- Simplicidade;
- Equipes auto-organizáveis;
- Reflexões periódicas.

A grande abordagem dos Métodos Ágeis surge da manipulação correta das conhecidas variáveis de um projeto de software em busca do sucesso e da qualidade. Enquanto os processos Tradicionais manipulam (tanto positivamente quanto negativamente) a **qualidade**, mantendo fixos o tempo, escopo e custo, os projetos ágeis focam-se no **escopo**, mantendo fixos, por sua vez, tempo, qualidade e custo.

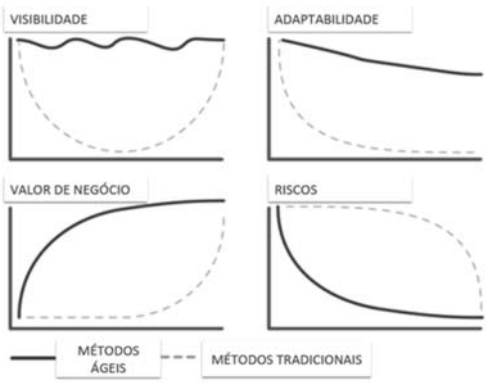
Uma possível comparação entre os modelos Tradicionais e Ágeis de desenvolvimento de software pode ser observada na tabela 3.1:

Tabela 3.1 – Comparação entre Enfoques.

Item	Enfoque Clássico	Enfoque Ágil
Gerenciamento	Controle absoluto	Colaboração e Liderança
Papéis/atores	Individual	Equipes (auto-organizadas)
Cliente	Importante (no início)	Crítico (sempre)
Desenvolvimento	Tradicional (cascata e outros)	Ágil (XP e outros).

Esta comparação, mais especificamente relacionada à visibilidade, adaptabilidade, valor de negócio e ao risco pode ser analisada nos gráficos abaixo, onde o eixo vertical representa o item analisado e o eixo horizonte representa o tempo (Figura 3.2):

Figura 3.2 – Comparação Ágil x Tradicional. Fonte: www.versionone.com



As principais metodologias ágeis incluem:

- FDD – Feature Driven Development;
- Extreme Programming;
- Lean Development;
- Crystal;
- Scrum;

As características comuns a todos os métodos ágeis citados acima incluem focar nas pessoas (desenvolvedores e clientes) e não focar em processos e documentações rígidas e detalhadas.

Visam, ao final, mesmo contrapondo várias premissas da Engenharia de Software, ser considerados como um conjunto coeso e consistente de **Boas Práticas** para projetos de desenvolvimento de software.

3.2 XP – Extreme Programming

A Extreme Programming é a principal metodologia de desenvolvimento ágil, criada por Kent Beck, Ron Jeffries e Ward Cunningham.

Surge como uma alternativa aos modelos tradicionais de desenvolvimento de software, representados principalmente pelo Modelo Cascata. É focada primordialmente na figura do programador, sem contudo apresentar níveis hierárquicos no processo de desenvolvimento. Normalmente é indicada para programadores experientes, que com XP ganham liberdade para codificar.

As premissas ou princípios do modelo XP baseiam-se numa mudança de proposta de atitude dos desenvolvedores, no sentido de:

- Tomar decisões importantes o mais tarde possível ;
- Implementar somente o necessário neste momento;
- Não implementar código desnecessário (ao contrário do princípio de Antecipação de Mudanças da Engenharia de Software);
- Feedback rápido e constante;
- Simplicidade;
- Alta qualidade de código-fonte;

Destes, pode-se apontar valores a serem destacados:

- Respeito;
- Comunicação;
- Coragem;
- Simplicidade;
- Feedback.

A XP destina-se, ainda, a um cenário bem definido dentro do âmbito global de desenvolvimento de software, em:

- Grupos pequenos – de 2 a 10 desenvolvedores;
- Projetos pequenos – de 1 a 12 meses previstos;
- Softwares pequenos – até 250.000 linhas de código.

Os papéis profissionais em uma equipe XP incluem:

- Programadores;
- Coach (treinadores ou técnicos) – normalmente o programador mais experiente do grupo, responsável pelo controle da equipe nos instantes iniciais do projeto;
- Tracker (acompanhador) – programador responsável pela gerência do projeto, em relação às estimativas e cobrança de tempos e custos;
- Cliente - a participação do cliente – real ou representado por um desenvolvedor – é fator chave no sucesso de um projeto de desenvolvimento XP. Ao cliente cabe a tarefa de escrever histórias (requisitos).

O XP é representado por um conjunto de 12 práticas originais/oficiais. O atendimento das práticas garante o ambiente necessário ao desenvolvimento XP.

- Planejamento
- Fases pequenas
- Metáforas
- Design simples
- Testes
- Refatoração
- Programação pareada
- Propriedade coletiva do código
- Integração contínua no projeto
- Semana de trabalho de 40horas
- Permanência do cliente junto ao time de desenvolvimento
- Padronização do código

O processo de um projeto XP tipicamente apresenta as seguintes etapas:

- Seleção de histórias (story cards) contendo os requisitos do cliente;
- Seleção de um par livre para programação pareada – Atividade não aleatória, visto que o objetivo da programação pareada é o desenvolvimento das habilidades de todos os elementos da equipe;
- Seleção, dentro da história, de um "cartão" contendo uma necessidade específica;
- Discussão, em par, das alterações recentes no software;

- Testes – o teste é considerado um dos principais – senão o principal – pilar de sucesso e segurança no modelo XP. Os desenvolvedores costumam executar testes antes, durante e depois da codificação, visando garantir que o código adicionado ao projeto está correto e funciona adequadamente; Engloba tanto testes unitários quanto testes funcionais;
- Codificação – No XP, na programação pareada, somente um desenvolvedor programa/codifica. O outro desenvolvedor deve ficar ao lado, analisando o código e propondo contraexemplos e planos de testes; Este código sempre poderá ser refatorado, ou seja, melhorado em busca de excelência técnica e adequação a um padrão oficial de codificação da equipe. Visto também que a documentação em um projeto XP é bastante reduzida, na etapa de codificação é enfatizada a necessidade de inclusão de comentários padronizados junto às linhas de código;
- Integração – Integração final do código ao repositório do projeto do software.

XP normalmente funciona muito bem, desde que respeitadas suas características básicas (equipes pequenas e experientes, projetos curtos e dinâmicos, entre outras). Entretanto, quando o cenário do desenvolvimento não é adequado (cliente não está disposto a participar/colaborar, projetos grandes - e estáveis -, é necessária uma especificação completa do software antes do processo iniciar, os programadores não são suficientemente experientes e capazes de autogerenciamento) XP não deve ser imposta à força, já que certamente irá gerar mais problemas do que soluções.

3.3 SCRUM

SCRUM é o principal framework do Gerenciamento ágil de projetos (GAP) que, segundo Highsmith (Highsmith, 2004) é um conjunto de valores, princípios e práticas que auxiliam equipes de projetos a entregar produtos ou serviços de valor em um ambiente complexo, instável e desafiador.

Os principais objetivos do GAP incluem:

- Inovação contínua;
- Adaptabilidade de pessoas, produtos e processos;
- Redução dos tempos de entrega – planejamento de curto prazo;
- Resultados confiáveis – análise do progresso real do projeto;
- Simplicidade;
- Foco na solução, ao invés de foco nos problemas;
- Remoção de barreiras ao orgulho da execução;

- Engajamento e poder às pessoas;
- Para tanto, são sugeridas boas práticas no Gerenciamento Ágil de Projetos, especificamente em projetos de desenvolvimento de software:
- Foco e envolvimento de pessoas – clientes e desenvolvedores;
- Foco em iterações – entregas parciais;
- Plano de projeto de alto nível e alta qualidade;

Segundo Chin (2004), a aplicabilidade ideal de GAP em projetos de desenvolvimento de software dá-se em:

- Projetos de desenvolvimento de novos produtos – Único cliente;
- Projetos de desenvolvimento de novas plataformas/tecnologias – Único cliente.

GAP, ainda segundo Chin, não é adequado para desenvolvimento de projetos meramente operacionais, independente do tipo de cliente (único ou em forma de conglomerado). Entretanto, este conceito mais recentemente está sendo bastante questionado, principalmente pela comunidade de desenvolvimento de projetos de software ágeis.

SCRUM foi criado por Jeff Sutherland e Ken Schwaber, baseados no artigo "The new product development Game", de Hirotaka Takeuchi e Ikujiro Nonaka, de 1986.

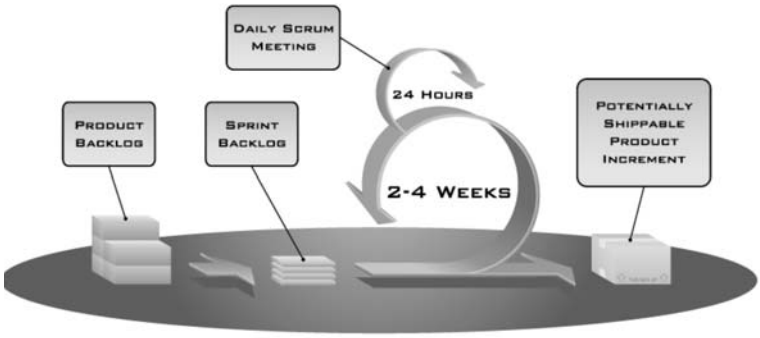
SCRUM pode ser definido como um framework para tratamento e resolução de problemas complexos e adaptativos, enquanto produtiva e criativamente entregam produtos com o mais alto valor possível, sendo então leve e simples de entender, porém, difícil de dominar. É importante frisar que SCRUM não é apenas um processo nem uma técnica, mas sim um framework dentro do qual são possíveis empregar vários processos e diversas técnicas.

Scrum é, então, um processo iterativo e incremental para desenvolvimento de projetos com foco em comunicação e trabalho em equipes, preocupado, primordialmente, com os fatores tempo e valor de negócio. O tempo (*time boxes*) é destacado no SCRUM por ser limitado e aplicado a tudo, incluindo deste reuniões a *sprints*.

O cenário ideal de aplicação do framework SCRUM são projetos com aspectos de tecnologia (plataformas) complexos, com requisitos adaptáveis e dinâmicos (igualmente complexos).

O processo do SCRUM dá-se através da figura 3.3.

Figura 3.3 – SCRUM. Fonte: http://epf.eclipse.org/wikis/scrumpt/Scrum/guidances/supportingmaterials/resources/ScrumLargeLabelled.png



Os eventos ou cerimônias do SCRUM incluem tradicionalmente:

- Reunião de planejamento de release (requisitos completos) – mais de 8 horas de duração;
- Reunião de planejamento da *Sprint* (requisitos selecionados) – 8 horas de duração;
- *Sprint* – de 2 a 4 semanas de duração;
- Reunião diária de acompanhamento – 15 minutos de duração, em pé, com as questões:
  - O que eu fiz ontem;
  - O que vou fazer hoje;
  - Encontrei algum problema/impedimento?
- Reunião de revisão da *Sprint* – 4 horas de duração;
- Reunião de retrospectiva da *Sprint* – 4 horas de duração.

Os papéis do SCRUM incluem:

- PO (*Product Owner*) – Responsável pelos requisitos/conhecimento do negócio;
- SCRUM Master – Responsável pelo processo SCRUM;
- Time SCRUM – Implementadores do SCRUM.

Os artefatos (documentos, relatórios e gráficos) do SCRUM são normalmente representados por:

- Product Backlog – Lista geral/completa de requisitos priorizados pelo PO;
- Spint Backlog – Subconjunto de requisitos selecionados pelo Time Scrum;
- Gráfico de Burndown – Gráfico de acompanhamento do SCRUM.

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	Em se tratando de desenvolvimento ágil, o teste de software não apresenta importância, uma vez que é realizado em paralelo ao desenvolvimento.
	Métodos ágeis são fortemente dependentes de interação com o cliente.
	Os princípios nos quais se baseiam os métodos ágeis são facilmente aplicáveis a equipes globais.
	O conceito de agilidade está fortemente associado a manter a lucratividade frente a turbulência dos mercados.
	No SCRUM, o PO (product owner) não apresenta relevância considerável no time de desenvolvimento.

Bibliografia

CHIN, G. *Agile Project Management: how to succeed in the face of changing project requirements*. 2004. NY: Amacon.

GUSTAFSON, David. *Engenharia de Software*. 2003. Porto Alegre. Editora Bookman.

HIGHSMITH, J., *Agile Project Management, Creating innovative products*. 2004. AddisonWesley.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. 2011. Porto Alegre. Editora McGraw-Hill/Bookman.

RICO, David. *Agile Methods for Developing Internet Products, Customer Satisfaction, and Firm Performance*. 2005. Disponível em http://davidfrico.com/rico05b.pdf,

SBROCCO, José. *Metodologias Ágeis – Engenharia de Software sob medida*. 1ª. Edição. São Paulo: Érica, 2012.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SHORE, James. WARDEN, Shane. *A Arte do Desenvolvimento Ágil*. Rio de Janeiro: Alta Books, 2008.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo: Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

F V F V F

4 ENGENHARIA DE REQUISITOS

Luís Fernando Fortes Garcia, prof. Dr.

Este capítulo apresenta a Engenharia de Requisitos, a etapa que pode ser considerada como a mais importante em todo processo de desenvolvimento de software.

Nesta etapa é definido o que o software deve fazer e, caso isto seja definido errado ou incompleto, estes erros serão replicados nas etapas subsequentes.

4.1 Engenharia de Requisitos

Requisitos estabelecem o que o software deve fazer, serviços que o software deve fornecer e definem as restrições sobre suas operações e implementação (SOMMERVILLE, 2011).

A etapa de Engenharia de Requisitos, de qualquer modelo de desenvolvimento de software, tradicional ou ágil, certamente pode ser considerada a mais importante de todas (entre requisitos, projeto, codificação, teste, implantação, manutenção), visto que todo processo inicia por aqui e, sendo assim, quaisquer erros nesta fase são replicados nas demais, alguns inclusive impossibilitando sua detecção e correção.

Outro complicador na Engenharia de Requisitos vem do fato desta etapa ser a etapa com maior interação entre pessoas, isto é, interação entre Analistas (de Negócio e Sistemas), Desenvolvedores e Clientes/Usuários. Infelizmente esta relação e/ou gestão de pessoas (abordagem, comunicação, confiança etc.) não faz parte do perfil da maioria dos profissionais de desenvolvimento de software, o que induz a erros, inconsistências, falhas e lacunas em todo processo.

O cenário atual de desenvolvimento de software, exibido na figura 4.1, mostra claramente que a área de Requisitos merece ainda pouco mais de 5% dos investimentos em desenvolvimento de software, mesmo que sabidamente seja crítica e responsável por mais da metade (55%) dos erros do software. Estes erros, quando descobertos e tratados no início, ainda na fase de requisitos, têm um custo relativo de correção bastante baixo (fator 1). Entretanto, quando não descobertos a tempo, na fase de requisitos, são replicados nas etapas de projeto e codificação e seu custo e esforço de correção passa a ser multiplicado até por 100 vezes (fator 10 a 100) o custo inicial.

Figura 4.1 – Custos de manutenção – Fonte: (Ávila e Spinola, 2007)

	% do custo de desenvolvimento	% dos erros introduzidos	% dos erros encontrados	Custo relativo de correção
Análise de requisitos	5	55		1
Projeto	25	30		1 - 1.5
Codificação e teste de unidade	50			
Teste	10	10		1 - 5
Validação e documentação	10			
Manutenção		5	22	10 - 100

Na figura 4.2, também disponibilizada no artigo de Ávila e Spinola (2007), tem-se os fatores críticos de sucesso no processo de desenvolvimento de software. Esta tabela confirma que, de oito fatores críticos, ao menos metade (fatores 1, 2, 4 e 6) são diretamente relacionados a Requisitos de Software.

Figura 4.2 – Fatores Críticos. Fonte: (Ávila e Spinola, 2007)

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%

Requisitos são importantes no processo de desenvolvimento de software para:

- Estabelecer base de concordância entre desenvolvedor/cliente;
- Estabelecer uma referencia para a validação do produto final;
- Diminuir retrabalho no processo de desenvolvimento.

4.2 Definições de Requisito

Requisitos podem ser definidos, em forma crescente de complexidade e abrangência, como em Ávila e Spinola (2007):

- Descrições das funções e restrições do software;
- Propriedade que o software deve exibir para resolver algum problema do mundo real;
- Condição ou capacidade que deva ser alcançada ou estar presente em um sistema para satisfazer um contrato, padrão, especificação ou outro documento formalmente imposto.

4.3 Tipos de Requisitos

Requisitos tradicionalmente são classificados, primariamente, como:

- Requisitos de Usuário – Documentos e diagramas para clientes, contendo, em linguagem acessível (às vezes mesmo linguagem natural), as principais funções e serviços a serem disponibilizados no software;
- Requisitos de Sistema – Documentos e diagramas mais estruturados, com detalhes técnicos e operacionais do sistema, para todos os envolvidos no processo de desenvolvimento do software, desde analistas, arquitetos, programados e testadores. Escritos em linguagem técnica de alto nível, normalmente utilizando UML.

Outra classificação possível dos requisitos aborda:

- Requisitos Funcionais;
- Requisitos Não Funcionais;
- Requisitos de Domínio.

Os Requisitos Funcionais descrevem as funcionalidades e funções do software, como o sistema deve reagir a entradas de dados, bem como o seu comportamento esperado.

Exemplos de Requisitos Funcionais:

- O software deve gerar relatório de todos os clientes cadastrados;
- O módulo de contas a receber deve listar todos os clientes com contas vencidas a mais de 30 dias;
- O cliente deve ser identificado no sistema pelo seu número de CPF.

Os Requisitos Não Funcionais expressam condições e restrições que devem ser atendidas pelo software, como restrições de tempo, padrões de desenvolvimento e de utilização, padrões de usabilidades, entre outros. Podem ser divididos entre vários subtipos, como:

- Requisitos de produto;
  - Requisitos de eficiência;
    - Requisitos de desempenho;
    - Requisitos de espaço;
  - Requisitos de confiabilidade;
  - Requisitos de portabilidade;
- Requisitos organizacionais;
  - Requisitos de entrega;
  - Requisitos de implementação;
  - Requisitos de padrões;
- Requisitos externos;
  - Requisitos de interoperabilidade;
  - Requisitos éticos;
  - Requisitos legais;
    - Requisitos de privacidade;
    - Requisitos de segurança.

Como exemplos de Requisitos Não Funcionais, pode-se citar:

- O software deve ser compatível com os navegadores web Internet Explorer e Firefox;
- O software deve exibir o resultado das consultas em até 3 segundos em operações que envolvam banco de dados;

- Os dados dos clientes devem ser mantidos criptografados dentro do banco de dados.

Os Requisitos de Domínio são herdados do domínio da aplicação (financeiro, educacional, industrial, hospitalar etc.) e refletem diretamente as suas características. São considerados superiores aos Requisitos Funcionais e geralmente descrevem características gerais a toda aplicação. Como exemplos destes requisitos, tem-se:

- O desconto máximo em operações de venda no sistema será de 10%;
- O cálculo da média final nas disciplinas presenciais dar-se-á pela média aritmética entre todas as notas.

4.4 Processo da Engenharia de Requisitos

A Engenharia de Requisitos aborda etapas e atividades relacionadas a toda vida útil dos requisitos no software, desde sua obtenção até sua manutenção e atualização após o desenvolvimento do software.

Pode ser dividido em dois grandes processos: Produção e Gerência de Requisitos, onde ambos apresentam atividades essenciais para o sucesso do software.

O processo de Produção de Requisitos envolve atividades de:

- Levantamento/Elicitação/Obtenção;
- Análise e negociação;
- Registro/Documentação;
- Validação;
- Verificação.

A etapa de Levantamento/Elicitação/Obtenção é responsável pela obtenção dos requisitos do software junto ao *stakeholders* (clientes e usuários diretamente envolvidos com o software). Para tanto, são desenvolvidas técnicas e utilizadas ferramentas como:

- Entrevistas – conversas pessoais ou mediadas por computador entre analistas e *stakeholders*;
- Questionários – documentos enviados por email previamente com questões envolvendo o que deve ser feito, como e por que?;
- Prototipação – construção conjunta de softwares provisórios junto aos *stakeholders*;

Como abordado anteriormente esta etapa, dentre todas do processo de Engenharia de Requisitos, certamente é a mais complicada e arriscada dada a exigência de

comunicação e interação entre pessoas. Normalmente existem problemas clássicos a serem abordados nesta fase, como:

- *Stakeholders* não sabem o que querem ou como deve funcionar;
- Desenvolvedores com baixo conhecimento do domínio de negócios do software;
- Problemas de comunicação entre as partes envolvidas – conflitos, ambiguidades e "obviedades" não formalmente definidas;

A etapa de Análise e Negociação é crucial no processo de Produção de Requisitos, visto que é responsável, em um primeiro momento, pela análise de viabilidade – técnica e comercial – em relação ao desenvolvimento dos requisitos (é possível desenvolver com as técnicas, ferramentas e conhecimentos atuais? Tem sentido para o conjunto do sistema ser desenvolvido? O requisito não é duplicado ou contraditório com algum outro requisito?) e, posteriormente, caso seja necessário, novos contatos com os *stakeholders* para negociação/readequação de ajustes nos requisitos anteriormente levantadas. Os mesmos problemas da etapa anterior são enfrentados aqui, multiplicados pela necessidade de negociação, que pode levar a desgostos e insatisfações por partes dos envolvidos.

A etapa de Registro/Documentação é responsável pelo registro formal dos requisitos em documentos oficiais de requisitos da empresa. Este registro é necessário para posterior consulta, controle e evolução dos requisitos. Estes documentos podem ser tanto utilizando linguagem natural (o que é desaconselhável, já que textos são propícios a problemas de interpretação) ou usando linguagens de modelagem como a UML (Unified Modeling Language), com seus diagramas de:

- Casos de uso;
- Classe;
- Sequência;
- Atividades.

A etapa de Verificação é responsável pela análise da documentação dos requisitos (produzida na etapa anterior) em busca de erros, falhas, ambiguidades, omissões e inconsistências internas do processo de obtenção e registro. Erros de modelagem UML, por exemplo, devem ser corrigidos nesta etapa.

Já na etapa de Validação, busca-se o ciente e aprovado dos *Stakeholders* em relação aos requisitos. Esta "assinatura" de aprovação não deve ser considerada como definitiva visto que, como já foi comentado, alterações nos requisitos são comuns e muitas vezes inevitáveis.

O processo de Gerência de Requisitos aborda as atividades seguintes ao processo de Produção de Requisitos, com etapas de:

- Controle de mudanças;
- Gerência de configuração;
- Rastreabilidade;
- Qualidade de Requisitos.

Este processo de Gerência de Requisitos ainda é incomum junto a desenvolvedores de software dada sua complexidade e custo elevados. Entretanto, sua utilização vem crescido ao longo do tempo junto com o crescimento das implementações de modelos de maturidades de qualidade de software, como CMMI e MPS.BR, que formalmente os exigem desde os níveis mais iniciais. A constatação vigente é de que não basta apenas levantar e registrar os requisitos para uso imediato pelos desenvolvedores, mas, principalmente, são necessárias etapas posteriores, que possibilitem sua correta gerência, armazenamento e controle ao longo do tempo, senão, perde-se completamente seu valor em momentos de manutenção/evolução do software.

A etapa de Controle de Mudanças é encarregadas do controle dos requisitos em relação ao longo do tempo, controlando as mudanças ocorridas desde a sua criação. Um histórico de mudanças é importante para poder identificar os autores das mudanças, seus motivos e o que efetivamente mudou nos requisitos. Esta é uma tarefa bastante complicada para ser gerenciada manualmente, sendo indicado o uso de ferramentas computacionais específicas. O subprocesso do Controle de Mudanças inclui:

- Verificação da validade da solicitação da mudança;
- Identificação de todos os requisitos afetados na mudança e suas dependências;
- Estimar custos e prazos da mudança;
- Obter aceite e aprovação da mudança.

A Gerência de Configuração aborda os critérios de integridade no processo de manutenção dos requisitos, desde sua criação até seu descarte (somente após o término da vida útil do software). Define que seja desenvolvido um plano formalizado que inclua atores (pessoas), artefatos (documentos) e ferramentas.

A etapa de Rastreabilidade visa acompanhar todas mudanças sofridas por um requisito, mas não somente por ele, e sim por todos os demais requisitos envolvidos/relacionados. Normalmente utiliza-se uma matriz de rastreabilidade, em que os relacionamentos entre requisitos são modelados. Um exemplo clássico

da necessidade de controle de rastreabilidade em um sistema comercial dá-se no controle da fórmula de juros: não é possível alterar a fórmula em apenas um módulo (aquele que solicitou a alteração), mas sim em todos os demais que a utilizam, senão, provavelmente teremos, em diferentes partes do sistema, juros calculados de forma diferente, refletindo em relatórios e fechamentos que jamais estarão corretos, para completa desordem dos contadores.

A Qualidade dos requisitos é descrita na norma IEEE 830, que aborda características como:

- Correção;
- Não Ambiguidade;
- Completude;
- Consistência;
- Verificabilidade;
- Modificabilidade.

A norma IEEE 830 também descreve uma estrutura genérica para um documento de requisitos, composto de:

- Prefácio;
- Introdução;
- Glossário;
- Requisitos de usuário;
- Arquitetura do sistema;
- Requisitos de sistema;
- Modelos de sistema;
- Evolução de sistema;
- Apêndices;
- Índice.

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	Requisitos incompletos e falta de envolvimento do usuário são fatores críticos de sucesso no desenvolvimento de software porque 55% dos investimentos em desenvolvimento de software se concentram nesta área.
	Requisitos são importantes, porém, não estabelecem referência para a validação do produto final.
	Requisitos apresentam condição ou capacidade que deve ser alcançada para atender a uma especificação formal.
	Requisitos de domínio são mais facilmente identificáveis que requisitos funcionais e não funcionais.
	A etapa de análise e negociação de requisitos está diretamente relacionada com a análise de viabilidade.

Bibliografia

AVILA, Ana. SPINOLA, Rodrigo. *Introdução à Engenharia de Requisitos*. Revista Engenharia de Software edição especial. 2007. Editora DevMidia. Disponível em: http://www.devmedia.com.br/artigo-engenharia-de-software-introducao-a-engenharia-de-requisitos/8034

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre: McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo: Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

F F V F V

5 PROJETO E ARQUITETURA DE SOFTWARE

Luís Fernando Fortes Garcia, prof. Dr.

Este capítulo apresenta a etapa de Projeto e Arquitetura de Software, onde define-se "como" o software será construído em termos de solução técnica. Outro conceito importante discutido neste capítulo é o de Reuso, que prega a ampla reutilização de todos os conceitos e artefatos possível no processo de desenvolvimento de software.

5.1 Projeto e Arquitetura de Software

Se na etapa de Engenharia de Requisitos resumidamente definimos "O QUE" deve ser feito no processo de desenvolvimento de software, na etapa de Projeto e Arquitetura de Software define-se "COMO" o software deve ser construído, sem, entretanto, chegarmos a codificá-lo de fato.

Nesta fase, devem ser estudadas e definidas as melhores soluções técnicas, tanto em nível de modelo/arquitetura, quanto em termos de Linguagens de Programação, Sistemas Gerenciadores de Banco de Dados etc.

O sucesso da fase seguinte – codificação (programação) – depende das indicações e regras oriundas desta fase de projeto.

5.2 Arquitetura de Sistema

A definição da arquitetura de um sistema visa identificar os subsistemas (módulos) que o constituem e o seu *framework* de controle e comunicação (SOMMERVILLE, 2011). Constitui-se da identificação dos componentes e suas comunicações.

A definição formal da arquitetura de um sistema visa:

- Possibilitar a comunicação entre os envolvidos no desenvolvimento do sistema;
- Possibilitar a discussão e análise do sistema, tanto em termos de requisitos quanto em termos de soluções técnicas;
- Possibilitar futuro reuso das soluções propostas.

Toda arquitetura de um sistema deve incluir ou preocupar-se com as seguintes características:

- Desempenho;
- Proteção;
- Segurança;
- Disponibilidade;
- Facilidade de manutenção.

Um dos principais tópicos de discussão nesta fase do desenvolvimento de software está relacionado à granularidade dos módulos, isto é, ao tamanho dos módulos. Se tivermos poucos módulos, e portanto módulos de grande tamanho, a manutenção do sistema pode ser facilitada, mas certamente aspectos de reuso serão prejudicados. Já numa situação inversa (muitos módulo de pequeno tamanho) o reuso pode ser facilitado ao preço de uma gerência bem mais complicada.

Decisões a serem tomadas em projetos de arquiteturas incluem também (SOMMERVILLE, 2011):

- Existe uma arquitetura genérica de aplicação que possa ser usada?
- Como o sistema será distribuído?
- Quais estilos de arquitetura são apropriados?
- Qual será a abordagem fundamental usada para estruturar o sistema?
- Como o sistema será decomposto em módulos?
- Qual estratégia deve ser usada?
- Como o projeto de arquitetura será avaliado?
- Como a arquitetura do sistema deve ser documentada?

Modelos de arquitetura mostram como o sistema é estruturado em subsistemas e como os dados são compartilhados através das interfaces (SOMMERVILLE, 2011).

São considerados parte da documentação do sistema e normalmente são compostos por submodelos (documentos) de:

- Processos;
- Interface;
- Relacionamentos;
- Distribuição.

Os três modelos de arquiteturas mais comuns em sistemas computacionais tradicionais (não incluindo aqui sistemas inteligentes, sistemas distribuídos ou mesmo computação em nuvem) são:

- Modelo de repositório – Baseado em um repositório central, que pode ser um computador de grande porte (mainframe) ou um SGBD (Sistema Gerenciador de Banco de Dados). Indicado para aplicações que manipulam grandes quantidades de dados. Como desvantagem principal, pode-se destacar a dependência total do ponto central. Caso ele estrague, toda aplicação é comprometida irremediavelmente.
- Modelo Cliente-Servidor – Baseado em uma rede de computadores, em que partes do sistema são distribuídas nos computadores da rede, que podem atuar tanto como servidores quanto como clientes em momentos específicos. Neste modelo, cada computador é autônomo e fornece apenas seus serviços, sendo que em caso de problema somente seus serviços são comprometidos. Como os computadores de uma rede de computadores são mais comuns e baratos que os modelos de grande porte é uma solução indicada para ambientes com restrições orçamentárias. Igualmente torna fácil a ampliação da solução apenas pela inclusão de novas máquinas na rede.
- Modelo em Camadas – Baseado em empilhamento de camadas de software, tornando um modelo mais conceitual que prático/implementável. Um exemplo de arquitetura em camadas dá-se no modelo OSI de redes de computadores.

5.3 Solução Técnica

Além da definição e estruturação do sistema em subsistemas e módulos, é função do projeto a definição da melhor solução técnica para o sistema em questão.

A solução técnica deve ser analisada frente a vários critérios de decisão:

- Tipo de aplicação;
- Tamanho da aplicação;

- Plataforma computacional – Desktop, móvel, web, etc.
- Quantidade prevista de dados a serem armazenados e manipulados pela solução;
- Capacitação técnica da equipe de desenvolvimento;
- Disponibilidade de fornecedores e assistência técnica/suporte.

Por exemplo, para um sistema pequeno, com poucos dados e aplicação/acesso via web pode-se indicar uma solução envolvendo linguagem de programação PHP com um banco de dados tipo MySQL. Já para uma solução de grande porte, com grande volume de dados e de transações, incluindo distribuição de serviços, uma escolha mais adequada envolveria linguagem de programação JAVA e banco de dados ORACLE.

O erro neste momento pode ser bastante grave no futuro, na etapa de manutenção do sistema (mesmo ainda quando este não seja considerado legado), visto que mudanças estruturais, de linguagens de programação e mesmo de banco de dados não são mais possíveis (e caso sejam incluem um volume de trabalho muito acima do aceitável).

5.4 Reuso em Projeto e Arquitetura de Software

Partindo das constatações de que o processo de desenvolvimento de software pode ser considerado lento, difícil, arriscado e consequentemente caro, uma das melhores soluções envolve o reuso (reutilização), tanto de códigos (abordagem mais comum) quanto com enfoque mais sistêmico, que propõe reuso desde os requisitos, envolvendo as arquiteturas de projeto, passado pelos códigos e chegando a planos de teste.

O reuso de software traz uma série de benefícios, como:

- Aumento de confiança, visto que o software baseia-se em algo que já foi usado e testado;
- Redução de risco, visto que os maiores custos no processo de desenvolvimento estão relacionados à concepção original do software;
- Uso eficiente de pessoas, visto que um módulo reusado carrega o conhecimento do profissional que o desenvolveu;
- Conformidade com padrões, visto que a homogeneização de interfaces e dados fica facilitada pelo reuso de módulos sabidamente aderentes ao padrão;
- Desenvolvimento acelerado.

Entretanto o reuso não é fácil de ser obtido, visto que não é automático, apresentando uma curva de aprendizagem difícil, com *overhead* e falta de motivação nos primeiros momentos.

Como fatores negativos ou problemas relacionados ao reuso, tem-se, por exemplo (SOMMERVILLE, 2011):

- Custos de manutenção aumentados;
- Problemas de ferramentas;
- Problemas de compreensão dos módulos existentes;
- Problemas relacionados à biblioteca de módulos.

Portanto, o reuso deve ser encarado como um aspecto cultural na empresa de desenvolvimento de software, baseado em programas de consultoria para ações de convencimento e treinamento e, então, considerados como patrimônio intelectual e até mesmo financeiro da empresa.

O Reuso antigamente era colocado em prática apenas baseado em ações de *copy-paste*, no uso de rotinas, funções e procedimentos e uso de bibliotecas de códigos prontos. Atualmente o reuso dá-se pelo uso de padrões de projeto (*design patterns)*, *frameworks* e componentização (bibliotecas de componentes).

Padrões de projeto podem ser considerados soluções já testadas para problemas de modelagem que ocorrem com frequências em situações específicas. Cada Padrão de Projeto descreve um problema específico que ocorre frequentemente nos ambientes de desenvolvimento de software e, então, descreve a/uma solução de forma a poder reusá-la inúmeras vezes em novas situações. Seria um reuso amplo de ideias, não somente de código-fonte.

Já *frameworks* são considerados aplicações "quase" prontas, faltando apenas prover as partes faltantes, que normalmente são específicas para cada aplicação. Uma boa analogia em relação a *frameworks* é o conceito de pré-pizza, onde temos a base de pão, o molho de tomate e até mesmo o queijo, tudo previamente colocado; Caso queiramos uma pizza específica, calabresa por exemplo, basta completa-la com os ingredientes faltantes.

O conceito de *framework* funciona melhor em domínio de negócios definidos, isto é, uma empresa de desenvolvimento de software para a área de saúde pode ter um *framework* básico desenvolvido e, sob demanda, completá-lo para aplicações de vários portes, como sistemas de controle hospitalar (grande), clínicas médicas (médio) e consultórios (pequeno porte), sendo que a base é a mesma (clientes, convênios médicos, processos do Ministério da Saúde etc.).

Um *framework*, para ser realmente útil e fazer a diferença do processo de desenvolvimento de software, deve ser:

- Reusável;
- Amplamente documentado;
- Fácil de usar;
- Extensível;
- Seguro;
- Eficiente;
- Completo o quanto possível.

Gera, com isso, economia de tempo e dinheiro, aumento da qualidade, compatibilidade entre softwares do mesmo domínio de aplicação e armazenamento do conhecimento dos especialistas. É importante frisar, porém, que estes grandes benefícios aparecem somente a médio e longo prazo.

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	O tamanho dos módulos é diretamente proporcional a sua capacidade de reuso.
	Reuso de software reduz riscos porque os maiores custos estão relacionados a concepção original do software.
	O uso de frameworks não promovem/facilitam o reuso de software.
	Software pode ser considerado patrimônio intelectual e financeiro da empresa quando reuso é promovido.
	O desenvolvimento de um framework de uma aplicação gera benefícios a curto prazo, ou seja, já em seu primeiro uso.

Bibliografia

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre: McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo: Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

F V F V F

6 TESTE DE SOFTWARE

Luís Fernando Fortes Garcia, prof. Dr.

Este capítulo discute o processo de Teste de Software e sua importância tanto para o processo de desenvolvimento de software quanto para a sociedade.

É apresentado o contexto atual da área, as definições de Teste de Software, os prováveis defeitos e falhas que podem ser encontrados em produtos de software, bem como o processo formal e as técnicas de teste.

6.1 Contexto atual do Teste de Software

Atualmente tem-se uma completa dependência dos softwares no mundo moderno, sendo que quase tudo que conhecemos possui um software embutido ou é controlado por um, como em celulares, carros, aviões, televisores, redes de comunicação telefônica, redes de transmissão de energia, redes de fornecimento de água potável, sistema financeiro e bancário, entre vários outros produtos e processos do cotidiano.

Pesquisas antigas, do ano 2000, já comprovavam perdas financeiras da ordem de 60 bilhões de dólares com erros de software, que hoje, certamente – pela maior quantidade e complexidade dos softwares mais modernos - estão com cifras de três ou quatro dígitos em bilhões de dólares. Essa mesma pesquisa mostrou que, caso fossem feitos mais testes e estes testes fossem mais abrangentes e eficientes, ao menos 50% destas perdas poderiam ser economizadas e vidas poupadas.

Testes de Software tornam-se, então, fatores cruciais para a Qualidade de Software, sem, entretanto, ser sinônimo de Qualidade de Software. A satisfação de um cliente/usuário (qualidade) passa pelo correto funcionamento do software (sem erros/falhas), mas este fator não é o único e nem, em muitos casos, o principal fator. Tal

constatação não diminui a importância desta etapa do processo de desenvolvimento de software, muito pelo contrário.

Alguns erros clássicos da área de Teste de Software são bem conhecidos e apresentaram consequências desastrosas, tanto em perdas financeiras quanto em perdas de vidas humanas:

- Choque da Estação Climática enviada para pesquisa em Marte – Perda de US$ 165 milhões;
- Avião comercial Airbus A320 abatido por engano – 290 vítimas fatais;
- Máquinas de radiação em tratamento de câncer – dezenas de vítimas fatais;
- Sistema de ambulância (SAMU) de Londres em 1992 - várias vítimas fatais;
- Queda de avião comercial A300 em 1994 – 264 vítimas fatais;
- Míssil Scud na Guerra do Golfo – 30 vítimas fatais;
- Vários colapsos de telefonia – Milhões de pessoas atingidas.

Estes exemplos reforçam tanto a necessidade de desenvolvimento de software com mais **respeito** e cuidado quanto a necessidade de processos de testes de software mais abrangentes e eficazes.

Este respeito (pregado em amplitude pela Engenharia de Software como um todo) advêm de fatos cientificamente comprovados como o que diz que, mesmo em softwares construídos com bastante cuidados, são esperadas (consideradas normais) 5 (cinco) a 10 (dez) falhas a cada mil linhas de código.

6.2 Definição de Teste de Software

O Teste de Software pode ser definido, de forma mais simples, como o "processo de executar um programa com o objetivo de revelar a presença de erros".

A partir desta definição, pode-se verificar que o processo de teste é sempre incompleto e não garante a inexistência de erros no programa (nenhum processo de teste de software garante 100% de inexistência de erros).

Em uma definição mais completa/complexa tem-se que "Teste de Software consiste na verificação dinâmica do funcionamento de um programa em um conjunto finito de casos de testes, cuidadosamente selecionado dentro de um domínio infinito de entradas, contra seu funcionamento esperado".

Analisando parte a parte da definição completa acima tem-se:

- Verificação dinâmica – Necessidade de execução do programa (apesar de existirem métodos de teste que podem ser executados sem a execução do programa);

- Conjunto finito – Os casos de teste são infinitos conceitualmente, sendo que então é necessário selecionar alguns mais apropriados;
- Seu funcionamento esperado – O teste basicamente consiste na comparação entre o resultado obtido na execução contra o resultado esperado conceitualmente/oficialmente (normalmente a partir dos requisitos de software).

A terminologia da área de Teste de Software inclui termos que precisam ser bem definidos:

- Falta – Causa de um mau funcionamento;
- Falha – Efeito observável não desejável;
- Erro – Diferença entre o valor obtido e o valor teoricamente correto;
- Exceção – Evento causador da suspensão da execução do programa;
- Anomalia – Qualquer coisa observada no programa que não está de acordo com as expectativas (requisitos);
- Verificação – "Estamos construindo certo o produto?"
- Validação – "Estamos construindo o produto certo?"
- Oráculo – Entidade responsável pela determinação do valor teoricamente correto.

Os Testes de Software são regidos por uma série de princípios:

- Não planejar o teste assumindo que o programa está correto;
- Caso de teste bem sucedido é aquele que tem alta probabilidade de detectar erros ainda não descobertos;
- Testes devem ser executados por outras pessoas que não o autor do programa;
- Dados de teste devem ser sempre inválidos e não esperados;
- Sempre é necessário previamente determinar os resultados esperados;
- Casos de testes devem ser mantidos por toda vida útil do sistema.

6.3 Prováveis defeitos e tipos de falhas

Os prováveis defeitos em um produto de software normalmente são gerados por seres humanos e:

- Podem ocorrer desde a especificação de requisitos;
- São gerados na comunicação entre *stakeholders*;
- Continuam presentes nos programas mesmo após processos de manutenção;

- Encontram-se em partes do código-fonte raramente usadas;
- Quanto antes revelados, mais barato fica sua correção.

Os tipos de falhas mais comuns em software incluem:

- Algoritmo – Erros de lógica de programação;
- Sintaxe – Erros de escrita de códigos-fonte;
- Precisão – Erros relacionados a tipos de dados em fórmulas matemáticas;
- Sobrecarga – Erros relacionados a estouros de capacidade em estruturas de dados;
- Coordenação – Erros relacionados à coordenação de sistemas distribuídos;
- Recuperação – Erros relacionados à recuperação de eventos inesperados, como falhas de energia;
- Hardware – Erros (raros) de hardware.

6.4 Processo de Teste de Software

O Teste de Software deve ser implementado através de um processo documentado e bem conhecido. Ele não deve ser considerado apenas uma das etapas do processo de desenvolvimento de software – aquela após a codificação – , mas sim deve ser feito em paralelo a todas as etapas do processo de desenvolvimento, testando desde requisitos até mesmo testes de software.

O processo de teste deve incluir etapas de:

- Planejamento – Definir os objetivo do processo de tese;
- Projeto – Projeto detalhado do plano de testes (conjunto de testes a serem executados);
- Execução – Execução do programa com os casos de teste;
- Análise dos resultados – Confronto entre os dados obtidos na execução e os dados teoricamente corretos.

6.5 Classificação dos Teste de Software

Os Testes de Software podem ser, para fins de estudo e análise de aplicabilidade, considerados como:

- Estáticos – Inspeção do código-fonte sem executá-lo;
- Dinâmicos – Os mais tradicionais, que executam o programa para realização dos testes;

- Simbólicos – Processo de execução do programa com dados simbólicos – utilização de grafos;
- Formais – Teste de programa baseado em provador de teoremas matemáticos.

6.6 Técnicas de Teste de Software

As técnicas mais conhecidas de Teste de Software são:

- Teste Funcional (ou Caixa Preta) – Testa os requisitos funcionais do software, sem acesso a detalhes de codificação;
- Teste Estrutural (ou Caixa Branca) – Teste da estrutura interna (código-fonte) do programa.

Além desta classificação clássica, pode-se classificá-los em:

- Teste de Unidade;
  - Testa cada módulo do programa separadamente;
- Teste de Integração;
  - Testa a integração entre os módulos do sistema – testa a interface/ comunicação entre módulos;
- Teste de Validação;
  - Testa a conformidade do software com os requisitos originais, normalmente dividido em teste Alfa (teste no ambiente do desenvolvedor) e teste Beta (teste no ambiente do usuário);
- Teste de Sistema;
  - Testes de segurança (acessos), estresse (condições anormais de volumes de dados e acessos) e desempenho (tempo de resposta).

Muitas das técnicas acima podem ser automatizadas, isto é, implantadas em processos automáticos de testes, baseados em scripts programados pelos testadores com base nos requisitos do software. A automação é forte aliada em testes de grandes volumes de dados, entretanto exigem profissionais de teste com habilidades de programação.

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	Teste de Software pode ser usado como sinônimo de Qualidade de Software.
	O processo de Teste de Software, quando executado de forma correta, visa garantir a inexistência de erros no programa.
	Teste de Software deve ser realizado tendo em vista que o software, por definição de construção, contém erros.
	Falhas de sobrecarga estão associadas ao mau projeto de estrutura de dados.
	Testes funcionais e estruturais são complementares.

Bibliografia

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre: McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo: Pearson: 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

F F V V V

7 EVOLUÇÃO DE SOFTWARE

Luís Fernando Fortes Garcia, prof. Dr.

O objetivo deste capítulo é permitir uma discussão inevitável em relação a produtos de software: O que fazer quando o software começa a chegar perto do final de seu ciclo de vida? Qual abordagem aplicar? Mantê-lo? Substituí-lo? Descartá-lo? Atualizá-lo?

Essa discussão é importantíssima, visto que normalmente somente pensa-se no processo de desenvolvimento de um software, sua concepção, e não na sua manutenção e posterior aposentadoria.

7.1 Evolução de Software

Softwares são produtos inerentemente mutáveis, isto é, sofrem diversas mudanças e correções ao longo de sua vida útil. A evolução do software é necessária também para manter sua utilidade/aplicabilidade, pois caso o cenário de utilização/contexto mude, ele precisa acompanhar/ser atualizado, caso contrário se tornará inútil e até mesmo perigoso para a organização.

Outro fator decisivo para a análise da evolução do software é o ROI (retorno do investimento), sendo que o investimento no desenvolvimento de um software é bastante alto e é um processo de risco, portanto, busca-se a utilização do sistema por vários anos.

7.2 Sistemas Legados

Sistemas legados são softwares que foram desenvolvidos há um bom tempo – muitas vezes mais de uma década – e que continuam desempenhando um papel estratégico decisivo na organização.

Entretanto, depois de tanto tempo, os sistemas não são mais os originalmente desenvolvidos, tendo sofrido diversas alterações devido a:

- Mudanças Internas;
    - o Mudanças na organização da empresa;
    - o Mudanças de gestão na empresa;
    - o Mudanças de pessoas e cargos na empresa;
    - o Mudança de ramo de atividade por parte da empresa;
    - o Erros no sistema que devem ser reparados;
    - o Inclusão de novas tecnologias na infraestrutura da empresa;
- Mudanças Externas;
    - o Mudanças na economia regional, nacional e internacional;
    - o Alterações de leis e regras;
    - o Mudanças de sistema monetário e troca de moedas;

Em relação às pessoas – fator crítico no desenvolvimento de software – os legados sofrem consideravelmente, visto que provavelmente nenhuma pessoa envolvida na sua criação ainda permaneça nos quadros da empresa, gerando o fato de que ninguém mais o conhece integralmente. Este fato coloca importância extra na documentação de um sistema de software.

Pode ser considerado fácil o processo de atualização ou migração de plataformas de *hardware* ou redes de computadores, entretanto, a mudança de softwares certamente não é. É uma estratégia de negócios extremamente arriscada, visto que:

- Processos corporativos e de sistemas altamente entrelaçados;
    - o Alteração no sistema, levando à alterações de processos, e vice-versa;
- Regras de negócios e restrições corporativas inseridas no sistema e não documentadas em nenhum outro lugar.

Tecnicamente também existem vários problemas relacionados aos sistemas legados, como:

- Não há estilo ou padrão de programação, já que diferentes pessoas ao longo do tempo alteraram o código;

- Linguagem de programação obsoleta - Dependendo do tempo de vida do legado, este pode ter sido escrito em linguagens antigas, como COBOL, o que dificulta encontrar mão-de-obra, suporte e documentação;
- Raramente existe especificação/documentação completa do sistema legado;
    - o Documentação original perdida;
    - o Documentação original presente, porém desatualizada;
- Em alguns casos não se tem acesso ao código-fonte, somente ao executável, o que leva à necessidade de engenharia reversa ou construção de módulos externos que contornem este problema;
- Estrutura do sistema corrompida, que leva a problemas de performance e segurança;
- Dados espalhados (replicados) em vários arquivos.

7.3 Estratégias em Sistemas Legados

O tratamento de Sistemas Legados normalmente resume-se a quatro estratégias/abordagens, sendo que a decisão entre elas depende de vários fatores, técnicos e de negócios:

- Descartar o sistema;
- Manter o sistema;
- Transformar o sistema;
- Substituir o sistema.

A decisão entre selecionar uma ou outra estratégia deve considerar aspectos como:

- Perspectiva de negócios – Valor do sistema para a empresa;
- Perspectiva técnica – avaliação de qualidade técnica remanescente no sistema;
- Ambiente
    - o Estabilidade do fornecedor;
    - o Taxa de falhas do sistema;
    - o Desempenho atual;
    - o Custos de manutenção;
    - o Capacidade de interoperabilidade;

- Software legado
    - Facilidade de utilização e compreensão;
    - Documentação;
    - Dados;
    - Desempenho;
    - Linguagem de programação

A estratégia de "Descartar o sistema" é considerada uma estratégia radical e definitiva, visto que prega simplesmente que o sistema seja descartado, junto com seus dados e regras. Tal ação somente pode ser considerada em casos onde, por exemplo, a empresa está abandonando uma área de negócios específica e, portanto, este sistema e seus dados não são mais importantes.

Na estratégia de "Manter o sistema" tem-se o foco na manutenção – tanto reparativa quanto adaptativa – do software, visando a extensão da sua vida útil. Normalmente requer um grande investimento, visto que todos os fatores negativos acima descritos encontram-se presentes. Normalmente neste foco não são acrescidas novas funcionalidades ao sistema, apenas mantidas as atuais, o que leva cada vez mais à inevitável obsolescência da aplicação.

Para a manutenção do sistema são considerados fatores como:

- Equipe de profissionais - Que devem ser fluentes tanto na linguagem de programação legada quanto na área de negócios do sistema;
- Fatores técnicos do programa – Idade, linguagem, banco de dados etc.

Na manutenção é importante a definição e real utilização de um processo de manutenção, que tem etapas de:

- Solicitação de alteração;
- Análise de impacto – Relevante para a aprovação ou não da alteração;
- Planejamento da mudança – Com muito cuidado para não danificar partes boas do sistema;
- Implementação da mudança;
- Release/entrega do sistema.

O planejamento da mudança/manutenção é crítico também no fator custo, pois os custos de mudança podem chegar a ser 100 vezes maiores do que o custo de desenvolvimento.

Na estratégia de "Transformar o sistema" é abordada a alteração (modernização) da estrutura do sistema, isto é, sua conversão para soluções técnicas mais modernas e seguras. Um exemplo de analogia de transformação de sistema seria no caso de transformação de uma casa com paredes de madeira para paredes de alvenaria. Neste caso, assim como no sistema, nenhuma nova funcionalidade é acrescida na casa, apenas as paredes existentes são modernizadas.

Em sistemas, esta transformação dá-se normalmente pela substituição de uma linguagem de programação legada (como COBOL) por uma linguagem mais atual, como JAVA, ou mesmo a substituição de bancos de dados antigos e restritos como CLIPPER para modernos sistemas gerenciadores de bancos de dados como ORACLE.

Uma das vantagens mais evidentes desta estratégia é a redução de custos e de riscos envolvidos, entretanto, como desvantagem, temos a manutenção das funcionalidades originais do sistema.

Na estratégia de "Substituir o sistema" aplica-se um processo de migração – que pode ser bem longo e complexo – do sistema legado para um novo sistema, desenvolvido do zero. Normalmente neste momento considera-se a implantação de sistemas de ERP (*Enterprise Resource Planning*), que, entretanto, não podem ser considerados como solução mágica para todos os problemas de TI da empresa.

Uma possível solução generalizada para sistemas legados, em relação as quatro estratégias, pode se dar considerando fatores como qualidade do software legado e seu valor de mercado:

- Baixa qualidade e baixo valor de mercado → Descartar o sistema;
- Baixa qualidade e alto valor de mercado → Transformar o sistema;
- Alta qualidade e baixo valor de mercado → Substituir o sistema;
- Alta qualidade e alto valor de mercado → Manutenção do sistema.

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	Sistemas Legados podem apresentar problemas por causa de dados replicados e estrutura corrompidas.
	Dentro da perspectiva técnica, a estratégia de descartar o sistema normalmente é abordagem mais adequada se o sistema apresenta baixa qualidade.
	Substituir o sistema é uma opção quando o valor de mercado é alto.
	O processo de migração traz embutido o risco de se considerar a implantação de um ERP uma solução mágica.
	Migrar plataformas de software é mais fácil que atualizar infraestrutura.

Bibliografia

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre: McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo: Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

V V F V F

8 QUALIDADE

Luís Fernando Fortes Garcia, prof. Dr.

O objetivo do capítulo é apresentar a área de Qualidade, as motivações para seu estudo, seu histórico (desde 4000 AC), suas definições e seus princípios, definidos por Deming.

Adicionalmente, abordamos as ferramentas de qualidade, os órgãos certificadores e os fatores humanos envolvidos na qualidade, bem como uma introdução ao 5s.

8.1 Motivações no estudo da Qualidade

Podemos destacar inúmeros aspectos importantes para motivar o estudo da qualidade na área de TI, especialmente da chamada Qualidade de Software:

- Exportações – Crescimento das exportações de software e serviços, onde os clientes externos exigem certificações de qualidade como CMMI;
- Licitações – Crescimento da exigência de certificações de qualidade em licitações do considerado maior cliente de TI do mercado, o governo (em todas suas esferas);
- Melhoria de Processos – A melhoria dos processos de desenvolvimento de software leva a uma maior produtividade e consequentemente a uma maior lucratividade;
- Crescimento Profissional – Os conceitos de qualidade impactam e guiam mudanças de comportamento e atitudes por parte dos profissionais de desenvolvimento de software;
- Globalização;
- Diferencial competitivo;
- Padrões internacionais.

Outro forte motivador pode ser visto na figura 8.1, que infelizmente ainda representa várias áreas dos setores produtivos. Acredita-se que somente com qualidade é possível mudar este cenário.

Figura 8.1 – Cenário mundial da Qualidade.

	média mundial	**média japonesa**	**Brasil**
peças rejeitadas	200/milhão	10/milhão	23000 a 28000/milhão
retrabalho	2%	0,001%	30%
prazo de entrega	3 dias	2 dias	35 dias
treinamento	XX	10% das horas de trabalho	1% das horas de trabalho

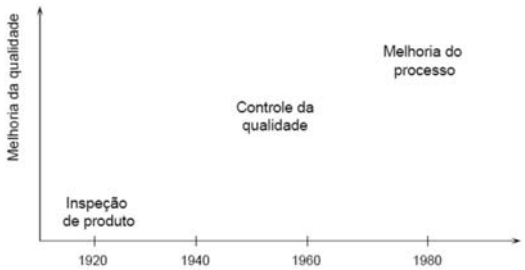
Entretanto, o maior e mais forte fator motivador no estudo da qualidade é o **lucro**, que vem pela redução de custos, estreitamento de laços com clientes e fornecedores, correta delegação de competências, com o consequente aumento da lucratividade.

8.2 Histórico da Qualidade

A Qualidade é um conceito antigo – os primeiros registros remontam a 4000 AC, na construção das pirâmides do Egito - que entrou no centro das atenções depois da Segunda Guerra Mundial, notadamente no Japão, pela necessidade de reconstrução das fábricas e reconquista do mercado.

Um contexto histórico da Qualidade pode ser dividido em três grandes momentos:

Figura 8.2 – Contexto Histórico da Qualidade



No primeiro momento, até a década de 1920, tem a dependência da inspeção em massa dos produtos – no Taylorismo, um por um, pela não confiança nos processos produtivos. Esse modelo não seria mais possível no contexto atual da indústria pelo grande volume de produção e consumo. Um dos mais notáveis exemplos desta época pode ser visto no filme *Tempos Modernos*, com Charles Chaplin.

Em um segundo momento, no pós-guerra, tem-se uma evolução para o chamado "Controle da Qualidade", normalmente uma seleção estatísticas de produtos a serem retirados das linhas de produção para serem avaliados. Ao invés da análise da totalidade dos produtos apenas 2 ou 3% dos produtos são avaliados, no conceito de lote. Caso estejam com problemas todo o lote é condenado.

Já no cenário atual, desde a década de 1980, tem-se o foco da Qualidade na melhoria preventiva dos processos de produção, ou seja, ao invés de se focar na detecção e correção de produtos defeituosos, é mais inteligente pesquisar e melhorar os processos produtivos. É neste cenário que se insere a Qualidade de Software e seus processos, ferramentas e certificações.

8.3 Definições da Qualidade

Qualidade é difícil de ser formalmente conceituada, visto que apresenta um conceito relativo, isto é, seu conceito é diferente para diferentes envolvidos em diferentes momentos e aspectos. O conceito mais aceito define qualidade como "Satisfação do usuário", isto é, um produto tem qualidade quando satisfaz o seu usuário/cliente.

Vários autores também apresentam definições acerca do termo "Qualidade":

- Deming - Aperfeiçoamento contínuo e firmeza de propósitos. Compreender o que acontece, construir e interpretar estatísticas e agir aperfeiçoando. Não há respostas corretas, apenas respostas geradas pelos métodos usados para gerá-las. O objetivo deve ser as necessidades do usuário presentes e futuras.
- Juran - Adequado ao uso.
- Crosby - Conformidade com os requisitos, fazer certo da primeira vez.
- Feigenbaum - O total das características de um produto e de um serviço referentes a marketing, engenharia, manufatura e manutenção, pelas quais o produto ou serviço, quando em uso, atende às expectativas do cliente.
- Oakland - Atendimento às exigências do cliente.
- Ishikawa - Fabricar produtos mais econômicos, mais úteis e sempre satisfatórios para o consumidor.

- Falconi - produto ou serviço de qualidade é aquele que atende perfeitamente, de forma confiável, de forma acessível, de forma segura e no tempo certo às necessidades dos clientes. O verdadeiro critério da boa qualidade é a preferência do consumidor.
- Dicionário Aurélio - Qualidade como "propriedade, atributo ou condição das coisas ou das pessoas capaz de distingui-las das outras e de lhes determinar a natureza".
- NBR ISO 8402 - Qualidade é a totalidade das características de uma entidade que lhe confere a capacidade de satisfazer às necessidades implícitas e explícitas".

As chamadas "dimensões da qualidade" ajudam na sua definição e são descritas como:

- Desempenho – características primárias de operação de um produto;
- Características secundárias/*features* – Características secundárias ou opcionais (não básicas) de um produto;
- Confiabilidade – Probabilidade de falha de um produto;
- Conformidade – Grau de atendimento a padrões por parte de um produto
- Durabilidade – Durabilidade prevista de um produto;
- Capacidade de receber assistência técnica – Incluindo velocidade, custo e facilidade de conserto;
- Estética – Qualidade subjetiva (julgamento pessoal) relacionada à aparência, gosto, cheiro;
- Qualidade percebida – Influência da origem (local) e fabricante do produto, por exemplo, relacionado a produtos chineses;
- Prontidão de atendimento – Capacidade de atendimento imediato ao produto.

8.4 Princípios de Qualidade de Deming

W. Edwards Deming (1900-1993), um dos maiores autores da área de Qualidade definiu um conjunto de princípios relacionados à qualidade de um produto:

- Propósitos - Criar constância de propósitos para melhoria de produtos e serviços;
- Filosofia – Encarar Qualidade como uma nova filosofia e adotá-la fortemente;

- Dependência da inspeção em massa – Suspendê-la pela melhoria dos processos;
- Negociação baseada apenas no preço – Acabar com esta prática nociva;
- Sistema de produção e serviços - Melhorar sempre e constantemente;
- Treinamento - Instituir o treinamento e um programa educacional;
- Liderança – Adotar, instituir e incentivar a plena liderança;
- Medo - Afastar o medo dos empregados quando da execução de suas tarefas;
- Áreas de apoio - Derrubar as barreiras entre as áreas de apoio/ departamentos;
- Slogans, exortações e metas – Eliminá-las entre os empregados;
- Cotas numéricas - Eliminar as cotas numéricas como fator de qualidade;
- Orgulho - Remover as barreiras ao orgulho da execução;

Deming também considera Qualidade como um processo cíclico, como no ciclo PDCA:

Qualidade → Diminuição do custo → Melhoria da Produtividade → Ganho de Mercado → Crescimento dos negócios → Mais Competitividade → Qualidade

Figura 8.3 - Ciclo PDCA - Fonte: http://www.sobreadministracao.com/wp-content/uploads/2011/06/ciclo-pdca.jpg

8.5 Ferramentas da Qualidade

A Qualidade pode ser trabalhada e materializada em diversas ferramentas, algumas bem simples e outras mais complexas. Alguns exemplos de ferramentas de qualidade são:

- *Check-lists* (folhas de verificação) – Como utilizados em decolagens de aviões, até que todos os itens a serem verificados estejam analisados e corretos, não é possível continuar com a decolagem;
- Diagramas de Pareto – Ferramenta gráfica para priorização de problemas, pela ordenação da frequência de ocorrência;
- Diagramas de Causa-Efeito – Também conhecidos como Espinha de Peixe ou Diagrama de Ishikawa, visa identificar as causas de um problema pela relação entre o efeito e suas causas causadoras;
- Histogramas;
- Gráficos de dispersão – Exibem a mudança dinâmica entre variáveis relacionadas;
- Fluxogramas – Gráficos simples de ordenação e encadeamento de etapas;
- Sistemas de informação – Sistemas computacionais, desde os mais simples controles até sistemas complexos como sistemas ERP;
- *Brainstorm* – Técnica de geração de ideias que envolve a contribuição espontânea de cada participante;
- Relatórios de auditoria.

8.6 Órgãos de Certificação da Qualidade

Os principais órgãos – públicos ou privados – de certificação da Qualidade incluem:

- ISO - International Organization for Standartization;
- IEC - International Electrotechnical Commission;
- IEEE - Institute of Electrical and Eletronics Engineering;
- ANSI - American National Standards Institute;
- ABNT -Associação Brasileira de Normas Técnicas;

Também pode-se incluir aqui entidades independentes especializadas em determinadas áreas de produção, que medem, analisam e atestam através de certificações de reconhecimento de Qualidade, como:

- ABIC – Associação Brasileira da Indústria de Café;
- ABCP – Associação Brasileira de Cimento Portland;
- ABICAB - Associação Brasileira da Indústria de Chocolate, Cacau, Amendoim, Balas e Derivados.

8.7 Fatores Humanos na Qualidade

A área de Qualidade é fortemente ligada à cultura organizacional da empresa, e normalmente apresenta forte resistência a mudanças, entre todos os envolvidos de todas as áreas (alta administração, gerentes intermediários e pessoal de produção).

O tratamento deste problema normalmente deve basear-se em um projeto de mudanças que deve iniciar e ser apoiado pelos níveis superiores, através de um projeto piloto. O CMMI, na qualidade de software, por exemplo, baseia-se fortemente nesta ideia quando em seu nível 2 aponta que a qualidade deva iniciar pela definição de um (ou mais) projeto-piloto.

8.8 5S na Qualidade

O conceito de 5S (Cinco Sensos) na Qualidade surge no período pós-guerra industrial japonês, onde o sucesso da reconstrução passa diretamente pela capacidade de produção de suas fábricas e pela qualidade dos seus produtos.

- Senso 1 – Seiri – Utilização
- Senso 2 – Seiton - Ordenação
- Senso 3 – Seisou - Limpeza
- Senso 4 – Seiketsu - Asseio
- Senso 5 – Shitsuke – Autodisciplina

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	Qualidade é um conceito associado somente a processos, dissociado da satisfação do usuário.
	Qualidade é um processo com fim em si mesmo, sem a característica de ser cíclico.
	Diagramas de Pareto são úteis para definir a relação causa-efeito de um problema.
	Gráficos de dispersão e diagramas de Pareto são ferramentas de quantificação de problemas.
	Qualidade está associada às dimensões de desempenho e durabilidade.

Bibliografia

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

KOSCIANSKI, A. *Qualidade de Software*. Novatec, 2006..

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre: McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo: Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

F F F V V

9 QUALIDADE DE SOFTWARE

Luís Fernando Fortes Garcia, prof. Dr.

Este capítulo apresenta a especialização da Qualidade em produtos de software, sua definição, os fatores e aspectos da qualidade, bem como suas normas.

Os enfoques da qualidade de software – produto de software e processo de software – têm especial destaque no texto.

9.1 Software

Software é, certamente, um produto diferente dos demais que estamos acostumados, como televisores, celulares, automóveis e outros, visto que é um produto:

- Intangível;
- Complicado;
- Diferente;

Entretanto, é um produto, e como tal pode (e deve) ser discutido em relação à sua qualidade.

O processo de desenvolvimento de software é discutido desde seus primórdios, mas foi efetivamente analisado em 1968, na Conferência das Nações Unidas sobre Software, que apresentou problemas relacionados a:

- Cronogramas não observados;
- Projetos abandonados;
- Módulos que não operam corretamente quando combinados;
- Programas que não fazem exatamente o que era esperado;

- Sistemas tão difíceis de usar que são descartados;
- Sistemas que simplesmente param de funcionar.

Esses problemas, adicionados aos problemas do aspecto não repetitivo do processo de desenvolvimento (tornando a tarefa difícil e imprevisível), do problema da delimitação do seu escopo e do problema da volatidade dos requisitos contribuem fortemente para a necessidade de uma pesquisa e aplicação dos conceitos de Qualidade na área de Software.

9.2 Qualidade de Software

A Qualidade de Software pode ser definida como:

- "A qualidade de software é um conjunto de características ou fatores de software, que determinam o nível de eficiência do software em uso, em relação ao atendimento das expectativas dos clientes" (IEEE).
- "Conformidade a requisitos funcionais e de desempenho explicitamente declarados, a padrões de desenvolvimento claramente documentados e a características implícitas que são esperadas de todo software profissionalmente desenvolvido" (PRESSMAN)

A motivação para a busca da Qualidade de Software vem de:

- Aumento da qualidade do produto;
- Diminuição do retrabalho;
- Maior produtividade;
- Redução do tempo para atender o mercado (*time to market*);
- Maior competitividade;
- Maior precisão nas estimativas.

Entretanto, a maior motivação para a implantação de um processo de Qualidade vem dos clientes e usuários do software – "O cliente, o REI". Estes clientes/usuários querem/exigem:

- Atendimento aos requisitos especificados;
- Defeito zero;
- Alto desempenho;
- Baixo custo;
- Desenvolvimento rápido;

- Facilidade de uso;
- Eficiência nos serviços associados;
- Inovação.

Portanto, para que um software tenha efetivamente qualidade ele deve:

- Preencher as expectativas do cliente;
- Ser obtido dentro de um prazo previsto;
- Ser produzido dentro de custos pré-estabelecidos;
- Conformar com as especificações de requisitos previamente estabelecidas;
- Definir claramente o seu objetivo, a sua finalidade e o seu propósito;
- Especificar seus requisitos para atender às necessidades do usuário;
- Produzi-lo e utilizá-lo dentro de processos bem estabelecidos.

9.3 Fatores da Qualidade de Software

Os fatores da Qualidade de Software podem ser classificados em:

- Implícitos – visíveis somente para os desenvolvedores/aspectos técnicos;
    - Flexibilidade – Facilidade de modificação;
    - Manutenabilidade – Esforço necessário para remover defeitos;
    - Testabilidade – Facilidade de execução de testes;
    - Eficiência – Quantidade de recursos para cumprir determinada tarefa;
    - Interoperabilidade – Integração das partes de um sistema;
    - Reusabilidade – Possibilidade de reaproveitamento de software/partes;
    - Portabilidade – Capacidade de usar diferentes plataformas;
    - Estimativas – Exatidão nas estimativas de custo/prazo/esforço;
    - Estabilidade – Extensão do ciclo de vida onde ele mantém a qualidade.

- Explícitos – Visíveis para os usuários/clientes do software.
    - Usabilidade – Expressa a facilidade de uso;
    - Confiabilidade – Capacidade de dependência do software, por determinado período de tempo;
    - Integridade – Controle de acesso ao sistema;
    - Prazo – Prazo estimado de entrega;
    - Informações sobre o progresso – Relatórios descrevendo o progresso;
    - Tempo de atendimento – Tempo gasto para as manutenções;
    - Retorno do Investimento – Retorno em forma de benefícios.

9.4 Aspectos da Qualidade de Software

Ao contrário do senso comum, que Qualidade de Software está relacionada somente ao seu processo de desenvolvimento, Qualidade de Software permeia software em pelo menos quatro aspectos:

- **No Processo de Desenvolvimento**
    - Definir um processo adequado para o ciclo de desenvolvimento;
    - Selecionar e aplicar métodos adequados de análise, projeto e implementação;
    - Definir processos adequados de verificação e validação (testes);
    - Sistematizar os testes por meio de planos, procedimentos e documentos de teste;
    - Utilizar ferramentas adequadas;
    - Aplicar normas e padrões pertinentes;
    - Gerenciar a configuração do software;
    - Acompanhar e avaliar a evolução das especificações de requisitos;
- **No Processo de Aquisição**
    - Buscar o produto mais adequado para a solução do problema;
    - Comprovar o bom funcionamento do produto;
    - Garantir a existência de bons fornecedores por meio de existência de treinamento e manuais de documentação.

- **No Processo de Integração**
  - Especificar de forma precisa os componentes a serem integrados;
  - Definir uma estratégia de integração;
  - Sistematizar as fases de desenvolvimento do software
- **No Processo de Utilização**
  - Definir o processo de utilização;
  - Definir os procedimentos de utilização;
  - Fornecer treinamento aos usuários;
  - Definir os responsáveis pelo software;
  - Manter os equipamentos hospedeiros;
  - Receber, em tempo, informações precisas e corretas.

9.5 Normas da Qualidade de Software

- ISO 9126 – qualidade de produto
- ISO 14598 – qualidade de produto
- ISO 12119 – pacotes de software
- ISO 12207 – Processo/ciclo de vida
- ISO 9000-3 – ISO 9001 para software
- CMM e CMMi
- MPS.BR
- PSP
- SPICE
- Entre outros.

9.6 Enfoques da Qualidade de Software

A Qualidade de Software pode ser encarada em dois enfoques:

- Qualidade de Produto de Software;
- Qualidade de Processo de Software.

A Qualidade de Produto de Software é tratada inicialmente pela norma ISO 9126 (NBR 13596), de 1991 (em processo de atualização para a norma ISO 25000) que define Qualidade de Produto como Um conjunto de atributos que têm impacto na capacidade do software de manter o seu nível de desempenho dentro de condições estabelecidas por um dado período de tempo".

A norma ISO 9126 é dividida em quarto partes (livros):

- 9126-1 – Modelo de qualidade de software;
- 9126-2 – Métricas externas;
- 9126-3 – Métricas internas;
- 9126-4 – Métricas para qualidade em uso.

A aplicação da norma ISO 9126 dá-se em:

- Definição dos requisitos de qualidade de um produto de software;
- Avaliação das especificações do software durante o desenvolvimento para verificar se os requisitos de qualidade estão sendo atendidos;
- Descrição das características e atributos do software implementado, por exemplo nos manuais de usuário;
- Avaliação do software desenvolvido antes da entrega ao cliente;
- Avaliação do software desenvolvido antes da aceitação pelo cliente.

A norma ISO 9126 apresenta um conjunto da **características e sub-características** a serem consideradas no processo de avaliação da qualidade do software:

- **Funcionalidade** - Satisfaz às necessidades;
  - Adequação;
  - Acurácia;
  - Interoperabilidade;
  - Conformidade;
  - Segurança de acesso;
- **Confiabilidade** – Tolerante a falhas;
  - Maturidade;
  - Tolerância a falhas;
  - Recuperabilidade;

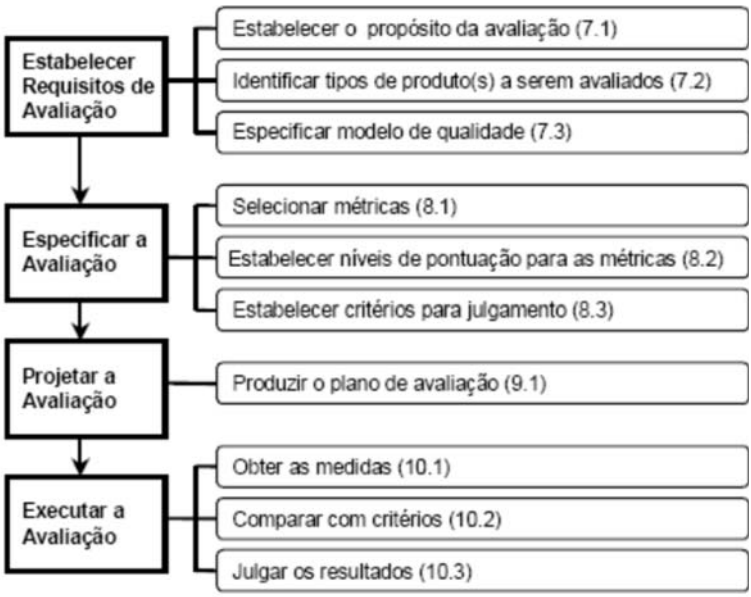
- **Usabilidade** - Fácil de usar;
  - Intelegibilidade;
  - Aprensibilidade;
  - Operacionalidade;
- **Eficiência** – Rápido, enxuto;
  - Tempo;
  - Recursos;
- **Manutenabilidade** - Fácil de modificar;
  - Analisabilidade;
  - Modificabilidade;
  - Estabilidade;
  - Testabilidade;
- **Portabilidade** – Fácil de portar para diferentes plataformas computacionais
  - Adaptabilidade
  - Capacidade de instalação
  - Conformidade
  - Capacidade de substituição

Em adição à norma ISO 9126, que traz as características pelas quais o produto software deve ser avaliado, é necessário a definição de um processo formal de avaliação, que é dado na norma ISO 14598. A norma 14598 descreve, então, um processo ideal de avaliação de produtos de software e é composta de seis partes/ livros:

- ISO-IEC 14598-1: Visão Geral;
- ISO-IEC 14598-2: Planejamento e Gestão;
- ISO-IEC 14598-3: Processo para desenvolvedores;
- ISO-IEC 14598-4: Processo para adquirentes;
- ISO-IEC 14598-5: Processo para avaliadores;
- ISO-IEC 14598-6: Documentação de módulos de avaliação.

O processo definido na norma ISO 14598 é composto de uma série de atividades:

Figura 9.1 - ISO 14598 – Processo de Avaliação de Produto de Software.



A Qualidade de Processo de Software baseia-se na melhoria dos processos de desenvolvimento de software – tradicionais ou ágeis.

Um processo de desenvolvimento de software considerado imaturo traz vários problemas tanto ao processo em si quanto ao produto resultante:

- Improvisado;
- Dependente de profissionais;
- Indisciplinado;
- Baixa produtividade;
- Baixo sucesso em previsões e estimativas;
- Altos custos de manutenção;
- Problemas frequentes.

Os modelos e normas da Qualidade de Processo de Software incluem:

- ISO 9000-3 – Adaptação da norma ISO 9000 para produtos de software;
- ISO 12207 – Definição de processo, atividades, tarefas e artefatos de desenvolvimento de software (tradicional);
    - o Aquisição;
    - o Fornecimento;
    - o Desenvolvimento;
    - o Operação;
    - o Manutenção;
- CMMI
- MPS.BR

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	Fatores de qualidade de software explícitos são mais percebidos pelos clientes do que fatores implícitos.
	Qualidade de software é um conceito associado unicamente ao processo de desenvolvimento, sendo quantificada no processo de testes.
	Qualidade no processo de software se ocupa da minimização de problemas no produto final de software.
	É possível analisar a qualidade de produtos de software somente com a ISO 9126.
	Consultoria é um dos aspectos da qualidade de software.

Bibliografia

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

KOSCIANSKI, A. *Qualidade de Software*. Novatec, 2006.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre. McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo. Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

V F F F F

10 MATURIDADE EM QUALIDADE DE SOFTWARE

Luís Fernando Fortes Garcia, prof. Dr.

O capítulo apresenta o conceito de Maturidade em Qualidade de Software, bem como os modelos mais famosos atualmente: O internacional CMMI e o nacional MPS.BR.

O objetivo principal deste capítulo, além da apresentação dos modelos e seus conceitos e níveis, é capacitar o leitor tanto na importância de sua implantação como na correta seleção de modelo e de nível de maturidade para cada realidade de organização.

10.1 Maturidade em Qualidade de Software

A qualidade de processos de desenvolvimento de software baseia-se em um tripé:

- Processos;
- Ferramentas;
- Pessoas.

Todos estes aspectos devem ser igualmente abordados e melhorados para que se tenha um processo de desenvolvimento "maduro", que possa ser analisado e classificado/ranqueado. Infelizmente, na área de TI, apesar de possuirmos grande domínio da área de ferramentas – sistemas operacionais, linguagens de programação, sistemas gerenciadores de banco de dados, redes de computadores etc. – tem-se ainda sérios problemas relacionados à parte de processos (regras de negócios em diferentes domínios de aplicação) quanto pessoas (especialmente relacionados a comunicação e confiança).

Define-se maturidade o quanto um processo está:

- Definido;
- Gerenciado;
- Medido;
- Controlado;
- Efetivo.

10.2 CMMI

O CMMI (Capability Maturity Model for Software Integration) é uma estrutura de modelo de maturidade de processos de desenvolvimento de software internacional, definido inicialmente em 1986, no SEI (Instituto de Engenharia de Software) da Universidade Carnegie Mellon, nos Estados Unidos da América.

O CMMI foi desenvolvido sob encomenda do Departamento de Defesa Americano, que procurava uma forma tanto de avaliar a capacidade de seus fornecedores de software quanto de melhorar-lhes seus processos de desenvolvimento.

O seu foco foi definido em dois aspectos que até hoje garantem a efetividade e o sucesso do modelo:

- Foco em PROJETOS (projetos de curto prazo);
- Foco em PEQUENOS PASSOS (níveis).

Adicionalmente, sua proposta baseia-se na experiência prática das empresas, refletir o melhor estado da prática, ser documentado e ser público.

Atualmente, está na versão 1.3, e todos os materiais relacionados estão disponíveis no site oficial http://cmmiinstitute.com/. O CMMI é indicado para grandes e médias empresas de desenvolvimento de software, com capacidade de investimento na faixa de R$ 500.000 ou mais.

Figura 10.1 - Página oficial do CMMI – Fonte: http://cmmiinstitute.com/.

Os objetivos do CMMI incluem:

- Guiar organizações a conhecerem e melhorarem seus processos de software;
- Identificar práticas para um processo de software maduro, definindo as características de um processo de software efetivo;
- Descrever como as práticas de engenharia de software evoluem sob certas condições;
- Organizar os estágios de evolução da melhoria dos processos em cinco níveis de maturidade;

10.3 Níveis do CMMI

O CMMI é organizado em cinco níveis de maturidade:

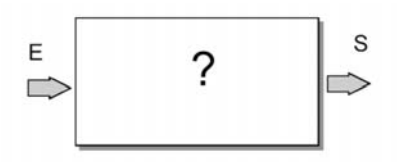
Figura 10.2 - Níveis do CMMI

No CMMI nível 1 (nível inicial) estão todas as empresas de desenvolvimento de software que ainda não iniciaram seus processos de melhoria, e tem-se as seguintes características (problemas):

- Não há repetibilidade dos processos;
- Em crise há abandono de procedimentos;
- As chances de sucesso baseiam-se em habilidades pessoais/GURUS/ HERÓIS;
- Sucesso, qdo existe, em projetos com experiência anterior;
- Tentativas isoladas de manutenção de procedimentos do processo;
- As qualidades pertencem as pessoas, não aos processos;
- Estimativas/cronogramas não realistas;
- Mesmo o planejado não é seguido (falta de costume).

A visibilidade (ou falta de) dos processos em nível 1 é nula, onde não há detalhes do processo:

Figura 10.3 - Visibilidade do CMMI nível 1



Para avançar ao CMMI nível 2 é comum:

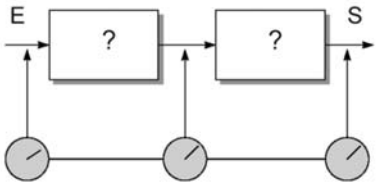
- Necessidade de mudança cultural;
- Resistência a mudanças;
- Reações intransigentes;
- Falta de credibilidade de que dá/dará certo;
- Introdução gradativa de KPAs (áreas chave de processos).

O CMMI nível 2 (repetível) é considerado o primeiro nível com qualidade no processo de desenvolvimento de software e apresenta as seguintes características:

- Políticas de gerência de desenvolvimento de software definidas e seguidas;
- Utilização de experiências anteriores, de maneira formalizada e não intuitiva;
- Projetos usam processos definidos, documentados, usados, disseminados, medidos, fiscalizados e com rotinas de melhoria;
- Inclusão de gerência de projetos;
- Compromissos assumidos com bases realistas;
- Compromissos assumidos com base em requisitos documentados;
- Desenvolvimento é acompanhado e revisado (custos, prazos, etc...);
- Mecanismos formais de correção de desvios;
- Inclusão de gerência de requisitos;
- A definição de processos é feita por projeto – pode não haver padronização na organização;
- Disciplina ao executar projetos;
- Processos repetíveis com resultados esperados;
- As qualidades pertencem aos projetos, não mais às pessoas.

No CMMI nível 2, a visibilidade do processo já é maior, com mais possibilidades de verificações e correções:

Figura 10.4 - Visibilidade do CMMI nível 2



No CMMI nível 3 (definido), a qualidade evolui de projetos específicos para toda a organização, baseado principalmente em fortes programas de treinamento. Apresenta as seguintes características:

- Processos estabelecidos e padronizados na organização – não somente repetição de sucessos de projetos anteriores;
- Estabelecimento de infraestrutura de processos adaptáveis a mudanças;
- Aderência ao processo mesmo em crise;
- Processos ainda em nível qualitativo;
- Foco em documentação;
- Processos de engenharia de software e gerenciais aplicados;
- Oportunidade de escolha das melhores práticas;
- Treinamento (técnico e gerencial);
- Possibilidade de adaptação dos processos às necessidades dos clientes;
- Os processos pertencem à organização e não aos projetos.

No CMMI nível 4 (gerenciado) o foco principal aborda métricas, tanto qualitativas quanto quantitativas. Apresenta as seguintes características:

- Estabelecimento de metas quantitativas para processos e produtos;
- Avaliação e análise contínua do desempenho;
- Melhoria no controle de processos e produtos;
- Gestão baseada quantitativamente;
- Proficiente em métricas/análise destas;
- Base de dados de processo;
- Gerenciamento de riscos.

E no nível máximo do CMMI (nível 5 – em otimização), procura-se manter e expandir o modelo de maturidade em todos os processos de todos os projetos. Apresenta as seguintes características:

- Melhoria contínua de processos;
- Identificação de pontos fracos e defeitos;
- Ação preventiva;
- Mudanças de tecnologia com base em análises de custo/benefício;
- Ações visando reduzir retrabalho e desperdício;
- Melhoria da produtividade.

10.4 Áreas Chave do CMMI

As áreas chaves (ou áreas de processos) são usadas pelo CMMI para a avaliação dos processos. Correspondem a um grupo de práticas relacionadas a uma área do desenvolvimento de software que, quando executadas, satisfazem um conjunto de objetivos de melhoria de qualidade.

Prática → Objetivo → Área de Processo

A tabela abaixo (figura 10.5) apresenta as Áreas de Processo dos cinco níveis do CMMI:

Figura 10.5 – Áreas Chaves do CMMI (todos os níveis)

Nível CMMI	Área Chave
2	CM – Gerência de configuração
2	MA – Medição e análise
2	PP – Planejamento de projetos
2	PMC – Acompanhamento de projetos
2	PPQA – Garantia da qualidade do processo e produto
2	REQM – Gerenciamento de requisitos
2	SAM – Gerenciamento de acordos com fornecedores
3	PI – Integração de produto
3	RD – Desenvolvimento de requisitos
3	TS – Solução técnica
3	VER – Verificação

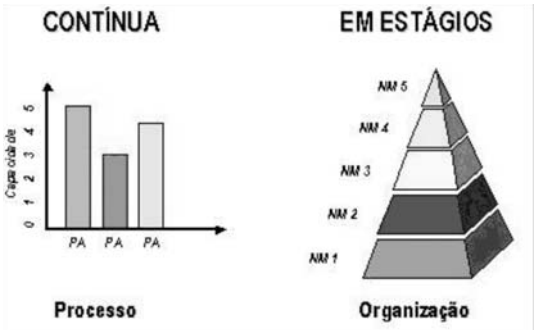
Nível CMMI	Área Chave
3	OPD – Definição do processo organizacional
3	OPF – Foco no processo organizacional
3	OT – Treinamento organizacional
3	IPM – Gerenciamento integrado de projeto
3	DAR – Decisão formal
3	RSKM – Gerenciamento de riscos
4	QPM – Gerenciamento quantitativo de projetos
4	OPP – Desempenho do processo organizacional
5	CAR – Análise de causa
5	OID – Inovação organizacional

10.5 Representações do CMMI

O CMMI permite a representação da maturidade do processo de desenvolvimento de software de duas maneiras:

- Contínua – Permite diferentes níveis de maturidade para diferentes áreas chave. Uma empresa de teste de software pode certificar somente as áreas chaves mais relevantes ao processo da empresa – teste de software;
- Estagiada – Cada nível representa um conjunto fixo de áreas chaves.

Figura 10.6 – Representações de níveis do CMMI. Fonte: http://3.bp.blogspot.com/-m1T0Tf-0K15A/T5MHd1eRbeI/AAAAAAAADFM/08MdfyK7niU/s400/CMMI.jpg



10.6 Processo de implantação do CMMI

O processo de implantação de CMMI, independente do nível buscado, tipicamente é implementado com a ajuda de uma consultoria especializada e apresenta as seguintes etapas:

- Treinamento CMMI;
- *GAP analysis* – Análise de lacunas no processo atual;
- Melhoria do processo atual;
- Elaboração de artefatos – Processos, documentos, guias, templates etc.;
- Implantação do CMMI em um projeto;
- SCAMPI B – Avaliação extraoficial do CMMI;
- Correção de desvios;
- SCAMPI A – Avaliação oficial do CMMI.

10.7 SCAMPI – avaliação do CMMI

O SCAMPI (*Standard CMMI Appraisal Method for Process Improvement* ou Método Padrão de Avaliação CMMI para Melhoria de Processos) é o método oficial utilizado em implantações CMMI.

Normalmente é dividido em dois momentos:

- SCAMPI B – Avaliação extraoficial do processo, normalmente realizado pela mesma consultoria responsável pela implantação. Visa detectar possíveis falhas e desvios remanescentes no processo de melhoria. É importante para dar rumo e segurança em direção à avaliação final.
- SCAMPI A – Avaliação oficial do CMMI, conduzida por uma auditoria oficial. Após a análise de documentos e processos, bem como de reuniões com todos os envolvidos, delibera-se e, caso não haja não conformidades, emite-se o certificado oficial, que apresenta validade definida.

10.8 MPS.BR

O MPS.BR, acrônimo de Melhoria de Processo do Software Brasileiro, é uma adaptação ou versão do CMMI adequada a nossa realidade. É indicado para empresas de desenvolvimento de software de micro, pequeno e médio porte, cujo custo é um fator crítico e projetos são pensados somente em pequeno prazo.

Está em desenvolvimento contínuo desde 2003 e envolve os três ramos brasileiros da área de TI (Tecnologia da Informação):

- Mercado – Representado pela SOFTEX e RioSoft;
- Academia – Representado pela Coppe/UFRJ e CESAR;
- Governo – Representado pela CenPRA e Celepar.

Este foco triplo é certamente um dos fatores de sucesso do modelo, visto que tudo foi criado e implementado segundo os anseios de todas as áreas envolvidas. Atualmente está amplamente difundido para outros países do Mercosul, como Argentina, Peru etc. Hoje, conta-se com 400 empresas certificadas nos diversos níveis MPS.BR.

É composto por guias oficiais, que incluem:

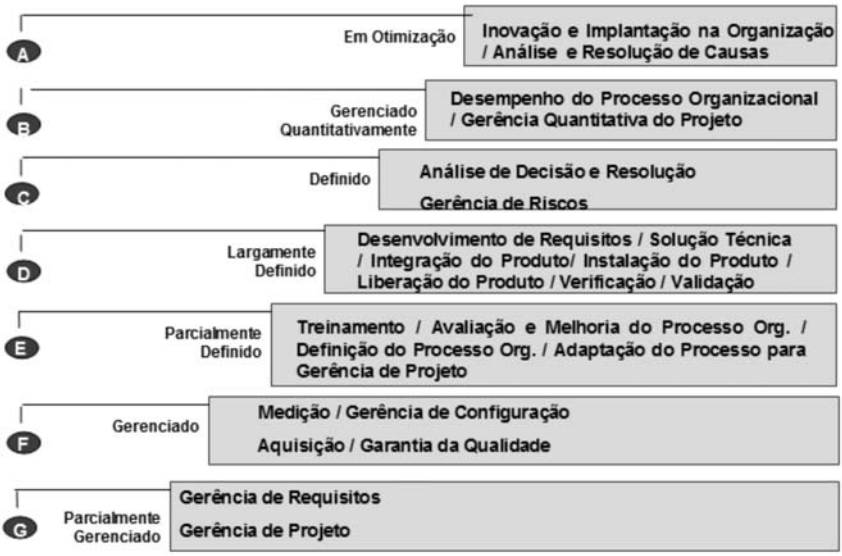
- Guia geral – descrição geral do MPS.BR, detalhando o modelo de referência (MR-MPS), seus componentes e as definições comuns necessárias para seu entendimento e aplicação;
- Guia de aquisição – Recomendação para a condução de compras de software e serviços correlatos. Elaborado para guiar as instituições que irão adquirir produtos de software;
- Guia da avaliação – Descrição do processo de avaliação, os requisitos para o avaliador e para a avaliação, o método e os formulários para guiar a avaliação.

A estrutura do MPS.BR é dada em três modelos oficiais:

- Modelo de Referência (MR-MPS)
    - Contém os requisitos a serem cumpridos pelas empresas que desejam estar em conformidade com o MPS.BR;
    - Definições dos níveis de maturidade da capacitação de processos;
- Método de Avaliação (MA-MPS)
    - Processo de avaliação, os requisitos para averiguação da conformidade;
    - Descrito de forma detalhada no guia de avaliação;
- Modelo de Negócio (MN-MPS)
    - Descrição das regras para a implementação do MPS.BR pelas empresas de consultoria, software e de avaliação.

O principal diferencial em relação ao CMMI é a implantação em sete níveis (de nível A até nível G), que possibilita, com isso, uma implantação mais gradual e adequada a pequenas empresas. O custo de implantação também é bem mais reduzido e pode ser negociado com órgãos de fomento estatal.

Figura 10.7 - Modelo MPS.BR – Sete níveis de maturidade. Fonte: www.softex.br/mpsbr/☐



Uma grande contribuição do MPS.BR frente ao CMMI são as capacidades, isto é, atributos de cada processo, indicando o quanto um processo ou área chave está madura, frente o nível de maturidade MPS.BR de A a G:

- 1.1 – Processo é executado
- 2.1 – Processo é gerenciado
- 2.2 – Resultado do processo (produtos) são gerenciados
- 3.1 – Processo é definido
- 3.2 – Processo está implementado

Atividades

Complete com V (verdadeiro) ou F (falso) para as seguintes afirmações:

	CMMI é um modelo adequado a empresas com baixa capacidade de investimento.
	O objetivo do CMMI incluir guiar as organizações a conhecerem e melhorarem seus processos de software, identificando práticas para um processo de software maduro.
	O nível 1 do CMMI trata do detalhamento dos processos, com visibilidade máxima.
	O nível G do MPS.BR apresenta requisitos e projetos como gerenciados.
	O modelo MPS.BR tende a ser mais indicado para empresas de pequeno porte.

Bibliografia

GUSTAFSON, David. *Engenharia de Software*. Porto Alegre: Bookman, 2003.

KOSCIANSKI, A. *Qualidade de Software*. Novatec, 2006.

PRESSMAN, Roger. *Engenharia de Software, uma abordagem profissional*. 7ª. Edição. Porto Alegre: McGraw-Hill/Bookman, 2011.

SCHACH, Stephen. *Engenharia de Software: Os Paradigmas Clássico e Orientado a Objetos*. 7ª. Edição. São Paulo: McGraw-Hill, 2009.

SOMMERVILLE, Ian. *Engenharia de Software*. 9ª. Edição. São Paulo: Pearson, 2011.

TONSIG, Sérgio Luiz. *Engenharia de Software – Análise e Projeto de Sistemas*. São Paulo: Futura, 2003.

Gabarito:

F V F F V