

Engenharia de Software



Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

Material Teórico



Introdução à Engenharia de Software

Responsável pelo Conteúdo:

Prof.^a Dr.^a Ana Paula do Carmo Marchetti Ferraz

Revisão Textual:

Prof.^a Me. Luciene Oliveira da Costa Santos



- Introdução
- Engenharia de Software (ES) e produto de software
- A diversidade na ES, sua integração com a internet e a ética relacionada ao desenvolvimento de produto de software
- Reengenharia e Engenharia Reversa



- Apresentar os conceitos gerais sobre Engenharia de Software.
- Entender os fatores que influenciam e dificultam a construção do software.
- Conhecer os reais objetivos da Engenharia de Software.

Primeiramente, é necessária a conscientização de que a exigência – tanto de estudo, quanto de avaliação – da disciplina *online* é, no mínimo, a mesma que ocorre na disciplina presencial.

Assim, é necessário organizar seu tempo para ler atentamente o material teórico e assistir aos vídeos, se houver, inclusive do material complementar.

Organize-se também de forma a não deixar para o último dia a realização das atividades (AS e AP). Podem ocorrer imprevistos e, quando encerrada a Unidade, encerra-se a possibilidade de obter a nota relativa a cada atividade.

Para ampliar seu conhecimento, bem como aprofundar os assuntos discutidos, pesquise, leia e consulte os livros indicados nas Referências e/ou na Bibliografia.

As Referências estão indicadas ao final dos textos de conteúdo de cada Unidade.

A Bibliografia Fundamental para esta Disciplina é: SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011. A Bibliografia Complementar está indicada em item específico, em cada unidade.

Caso ocorram dúvidas, contate o professor tutor por meio do Fórum de Dúvidas, local ideal, pois assim a explicação poderá ser compartilhada por todos.

Existe, ainda, a possibilidade de contatar o professor pelo link Mensagens, caso seja algum assunto relativo somente a você, e não uma dúvida sobre a matéria, que pode ser também dúvida de outros alunos.

Fique atento(a) quanto a possíveis dúvidas e problemas, lembrando-se de que o professor tutor está à sua disposição para resolver assuntos pedagógicos, isto é, dúvidas quanto ao conteúdo da matéria.

Se suas dúvidas ou problemas se referirem ao sistema, aos programas etc., contate o Suporte, ou procure auxílio de um monitor no webclass de seu campus.

Contextualização

O mundo atual não seria o que é sem que tivéssemos software, estruturas de redes e toda tecnologia que integra esses sistemas. Serviços nacionais e internacionais são controlados por software. Produtos são desenvolvidos num determinado país, fabricados em outro e comercializados num terceiro. Todo esse processo está informatizado, da produção de matéria-prima ao controle de satisfação de um produto finalizado, sem contar a indústria da música, jogos, cinema e televisão.

Nesta unidade da disciplina **Engenharia de Software (ES)**, apresentaremos alguns conceitos gerais sobre ES e como, aos poucos, ela foi dando forma a uma atividade (desenvolvimento de software) que era considerada caótica, sem muito controle e passível de não cumprimento de prazo em grandes proporções.

Aqui, você entrará em contato com os assuntos principais deste conteúdo de forma breve e geral e terá a oportunidade de aprofundar essas questões no estudo de cada unidade. No entanto, esta unidade inicial visa fornecer-lhe o conhecimento básico necessário para que você possa construir um referencial teórico com base sólida – científica e cultural – e o exerça, no futuro exercício de sua profissão, com competência cognitiva, ética e responsabilidade social.

Vamos começar nossa aventura pela apresentação das ideias e dos princípios básicos que fundamentam esta disciplina.

Como forma de despertá-lo(a) para a matéria e de direcionamento, vamos retornar à seguinte pergunta:

Somente o domínio de uma linguagem é suficiente para uma pessoa ou equipe produzir softwares de qualidade?

1. Introdução



Para iniciar o estudo desta unidade, é importante que você tenha em mente que desenvolver software é muito mais do que a atividade de codificação, ou seja, é muito mais do que utilizarmos adequadamente uma linguagem de programação.

Antes, desenvolver sistemas era apenas “programá-los”, mas à medida que a tecnologia evoluiu os sistemas baseados em computador ficaram mais complexos. O que antes era trabalho de um profissional ou uma equipe fisicamente próximo, começou a ser desenvolvido por inúmeras pessoas fisicamente distantes. E como garantir a qualidade do produto final? Como garantir prazos? Como mensurar prazo de entrega, tempo de desenvolvimento, teste e valor do produto?

Foi nesse contexto que a Engenharia de Software (ES) surgiu.

Antes de abordarmos os conceitos de ES vamos apresentar um conceito importante e bastante utilizado na área de informática, assim como nesta disciplina: o de **sistemas** e suas aplicações – sistemas computacionais.

É interessante, porém, que você entenda o conceito dessa palavra, que é muito empregada não só na informática, mas em muitas outras áreas. Certamente, você já deve ter utilizado tal expressão para se referir, por exemplo, ao sistema solar, ao sistema educacional, ao sistema respiratório etc.

No nosso contexto, **sistema computacional é um conjunto de programas – softwares – integrados**.

Segundo Pressman (2006), se quisermos definir **software**, numa forma clássica, podemos até dizer que ele faz parte de um **conjunto de instruções** que, quando executadas, produzem a função e o desempenho desejados, possui **estruturas de dados** que permitem que as informações relativas ao problema a resolver sejam manipuladas adequadamente e documentação que é necessária para um melhor entendimento da sua operação e uso.

Pressman (1995, p. 179) inclui como elementos de um sistema computacional:

1. **Software:** programas de computador, estruturas de dados e documentação correlata que servem para efetivar o método, processo ou controle lógico necessário.
2. **Hardware:** dispositivos eletrônicos que fornecem a capacidade ao computador e dispositivos eletromecânicos que oferecem funções ao mundo externo.
3. **Pessoas:** usuários e operadores de hardware e software.
4. **Banco de dados:** uma grande e organizada coleção de informações a que se tem acesso pelo software e é parte integrante da função do sistema.
5. **Documentação:** manuais, formulários e outras informações descritivas que retratam o uso do sistema.
6. **Procedimentos:** os passos que definem o uso específico de cada elemento do sistema ou o contexto processual em que o sistema reside.

Esses elementos interagem uns com os outros para transformar informações. Assim, para que um sistema computacional cumpra realmente seus objetivos, é importante não apenas se preocupar com a construção de seus elementos, mas também é fundamental que todos os elementos estejam integrados entre si de forma efetiva.

Entretanto, se pensarmos no contexto da Engenharia de Software, o software deve ser visto como um sistema que se transforma em **produto** a ser desenvolvido para ser vendido.

Antes de abordarmos esse conceito de produto, devemos entender que, segundo Sommerville (2011, p. 2):

[...] os sistemas de software são abstratos e intangíveis. Eles não são restringidos pelas propriedades dos materiais, nem governados pelas leis da física ou pelos processos de manufatura. Isso simplifica a ES, porque não há limites para o potencial de software. Existem vários tipos de sistemas de software, desde os mais simples sistemas embutidos até os sistemas de informações complexos de alcance mundial.

É nesse contexto que iniciaremos a nossa discussão sobre ES, ou seja, existem diferentes tipos de sistemas computacionais e não faz sentido procurarmos por métodos, notações, técnicas comuns e universais, porque diferentes softwares precisam de diferentes abordagens de desenvolvimento. Desenvolver e comercializar um software corporativo é completamente diferente de atuar no controle de instrumentos científicos, que é diferente também de atuar na telefonia e ainda é completamente distinto de desenvolver um *game*.

Todas essas aplicações precisam dos conceitos de ES no software no desenvolvimento de um produto que é denominado **produto de software**.

Os conceitos abordados nesta disciplina estão relacionados a produtos de software de acesso **não restrito**, ou seja, para clientes maiores, que possuem características de integração com outros programas/sistemas e bases de dados internas e externas.

Para softwares menores, restritos, com um único usuário/cliente, os conceitos de Engenharia de Software não se aplicam em toda sua extensão. No caso desses programas, documentação, testes, manutenção, técnicas de modelagem, portabilidade, flexibilidade etc., a associada é pequena ou, na maioria das vezes, inexistente, pois, segundo Sommerville (2011, p. 3), a ES “[...] tem por objetivo apoiar o desenvolvimento profissional de software, mais do que a programação individual”.

2. Engenharia de software e produto de software



Engenharia de Software é uma disciplina de Engenharia cujo foco está em todos os aspectos da produção de software.

Engenharia se refere a obter resultados de qualidade requeridos dentro do cronograma e do orçamento. Isso frequentemente envolve ter compromissos (SOMMERVILLE, 2011, p. 5).

Considerando o mundo de tecnologia integrada em que vivemos, a ES é importante por dois motivos: cada vez mais pessoas dependem dos sistemas informatizados e temos que ser capazes de desenvolvê-los de forma confiável, rápida e econômica. Além disso, é mais barato quando utilizamos técnicas, métodos e processos ao invés de desenvolvê-los de forma aleatória, como se fossem um projeto pessoal sem controle de prazos e custos.

Considerando o apresentado, os engenheiros de software se preocupam em desenvolver **produtos de software**, os quais podem ser vendidos para clientes (SOMMERVILLE, 2011).

Um **produto de software** propriamente dito é sistematicamente destinado ao uso por pessoas com formações e experiências diferentes, o que significa que uma preocupação não apenas nas características de desenvolvimento, como também de interface, documentação, seja ela de sistemas ou de usuário. Sem contar que, nesse caso, deve-se testar o software exaustivamente antes de entregá-lo ao cliente para detectar e corrigir os eventuais erros de execução.

Um programa desenvolvido para resolver um dado problema (usuário restrito e único) e um produto de software destinado à resolução do mesmo problema (para vários clientes e com a meta de comercialização) são duas coisas diferentes. É óbvio que o esforço e o consequente custo associado ao desenvolvimento de um produto serão superiores aos de acesso restrito até mesmo devido à sua concepção de comercialização futura.

Existem, segundo Sommerville (2011), dois tipos de produtos de software:

1. **Produtos genéricos:** sistemas stand-alone, produzidos por uma organização de desenvolvimento e vendidos no mercado para qualquer cliente que deseja utilizá-los. Ex. ferramentas de banco de dados, processadores de texto etc.
2. **Produtos sob encomendas:** sistemas desenvolvidos para um cliente em particular. Ex. sistemas de controle de tráfego aéreo.

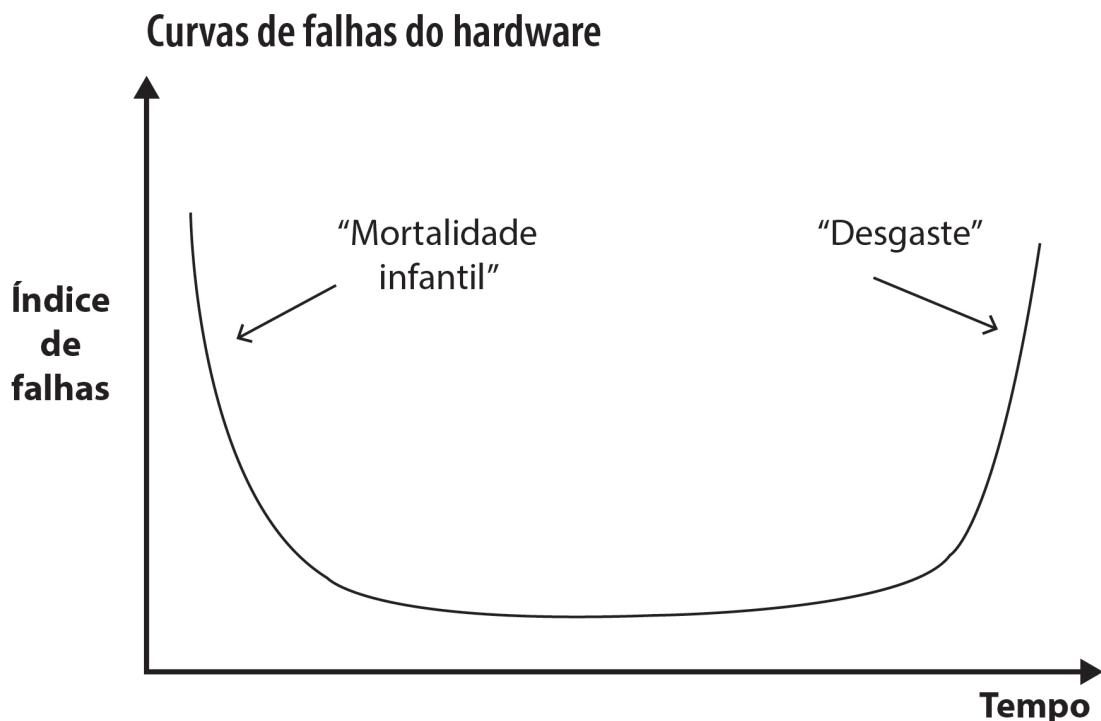
A diferença primordial nesses dois sistemas é que, no primeiro, a organização que o desenvolve controla suas especificações, no segundo é o cliente que define as especificações.

Entretanto, é importante paramos um momento para levantar algumas comparações entre produto de software que é composto por partes lógicas e físicas e outros tipos de produtos:

- **O produto de software é concebido e desenvolvido como resultado de um trabalho de engenharia** e não manufaturado no sentido clássico. Neste sentido, é interessante que você reflita novamente sobre o conceito de engenharia. Pense em uma atividade de engenharia que lhe seja familiar (engenharia civil, mecânica, produção etc.). Imagine-a como Ciência de construção, na qual, por meio de técnicas, ela pode desenvolver partes que pertencerão a um produto específico e chegue às suas próprias conclusões, fundamentado(a) no fato de que o software não é um produto manufaturado, embora possua algumas características conceituais relacionadas à manufatura.
- **O produto de software não se desgasta**, ou seja, ao contrário da maioria dos produtos, o software não se caracteriza por um aumento na possibilidade de falhas, à medida que o tempo passa, devido ao desgaste físico e à ação ambiental (como, por exemplo, a poeira e o calor). O que pode acontecer é ele se tornar obsoleto

e suas funcionalidades não mais satisfazerem à necessidade do usuário. Portanto, pode-se dizer que o software não se desgasta, mas, sim, deteriora-se. Você poderá compreender melhor as diferenças entre o desgaste do hardware e a deterioração do software observando o gráfico das curvas de falhas, proposto por Pressman (2006). O gráfico 1, a seguir, exemplifica isto:

Gráfico 1



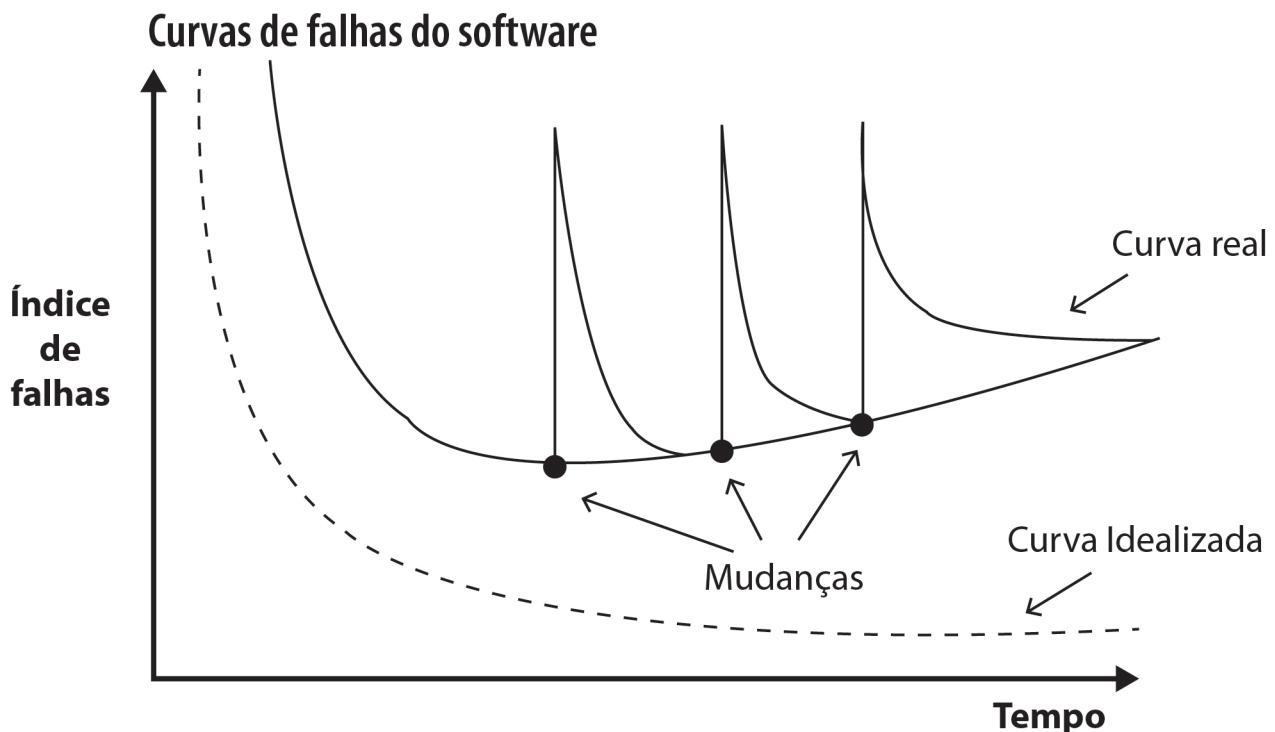
Fonte: Adaptado de Pressman (2006, p. 14).

Como você pode observar, no Gráfico 1, inicialmente, o hardware apresenta um elevado índice de falhas, as quais, normalmente, são atribuídas a problemas de projeto ou de fabricação. Com o passar do tempo, aferições são realizadas no hardware e essas falhas são corrigidas. Em seguida, são realizados ajustes e a curva se estabiliza, indicando que todos os defeitos foram eliminados, ou que o nível de falhas se encontra em um limite aceitável para o produto. No entanto, com o passar do tempo, o índice de falhas aumenta devido ao desgaste do produto, levando à sua substituição.

Observe, agora, o Gráfico 2, em que são apresentadas as duas curvas de falhas de *software*: **a curva real e a curva ideal**.

Na curva idealizada, representada pela linha tracejada no Gráfico 2, o software, sendo um elemento lógico, não sofre os desgastes físicos e ambientais comuns ao hardware. Assim, teoricamente, uma vez eliminadas as falhas iniciais inseridas durante o projeto e a construção do software (semelhante à construção do hardware), a curva se manterá estabilizada sem falhas ou com o menor índice possível de falhas. Portanto, o software não se deterioraria.

Gráfico 2



Fonte: Adaptado de Pressman (2006, p. 15).

No entanto, a curva ideal apresenta uma utopia do desenvolvimento de software: uma vez construído o software, todos os problemas acabaram. A realidade, porém, é bem diferente, e é mostrada na curva real de falhas do software, também no Gráfico 2. Durante a vida do software, ele passará por várias e inevitáveis mudanças. À medida que as mudanças são realizadas, novas falhas podem ser inseridas, representando na curva real os picos de aumento de índice de falhas. Antes mesmo de se eliminarem as falhas inseridas na última modificação, nova necessidade de mudança pode ocorrer. Com o passar do tempo e as frequentes modificações, o índice mínimo de falhas vai aumentando, isto é, o software deixa de atender aos requisitos para os quais foi construído. Portanto, o software não se desgasta, mas se deteriora.

- **A maioria dos produtos de software é concebida inteiramente sob medida**, sem a utilização de componentes pré-existentes.

Em função dessas características diferenciais, o **processo de desenvolvimento** de produto de software gera um conjunto de dificuldades, com influência direta na **qualidade final do produto**.

Embora hoje conceitos de componentes e reuso sejam bastante utilizados no desenvolvimento de software, eles fazem parte de um conceito maior que é o de melhorar as técnicas de desenvolvimento de software

3. A diversidade na ES, sua integração com a internet e a ética relacionada ao desenvolvimento de produto de software



Engenharia de Software é uma abordagem sistemática para produção de software; ela analisa questões práticas de custo, prazo e confiança, assim como as necessidades dos clientes e produtos do software (SOMMERVILLE, 2011, p. 6).

Existem, no mercado, segundo Sommerville (2011) e Pfleeger (2004), diferentes tipos de software comercializados:

- *Stand-alone*: aplicações executadas em um computador local. Exemplo: processador de texto.
- Aplicações interativas baseadas em transações: aplicações executadas num computador remoto. Nesta categoria temos os softwares corporativos integrados à web. Exemplo: software de e-mail.
- Sistemas de controle embutido: aplicações que controlam e gerenciam dispositivos de hardware. Exemplo: softwares de um micro-ondas, de telefone celular etc.
- Sistemas de processamento em lotes: aplicações corporativas desenvolvidas para processar grandes quantidades de informação, em lote. Exemplo: sistemas de cobranças telefônicas.
- Sistemas de entretenimento: aplicações relacionadas à diversão. Exemplo: jogos.
- Sistemas para modelagem e simulação: desenvolvidos por cientistas, compostos por vários objetos que podem ser combinados entre si.

Para cada um desses sistemas, e para outros não mencionados, não existem barreiras físicas sobre onde um tipo começa e outro termina. Por exemplo, você pode desenvolver um jogo para telefone celular no qual deverá ser cobrado um valor mensal/anual para todos os usuários do sistema de uma telefonia específica. Assim, para cada tipo de software, temos que utilizar diferentes técnicas de desenvolvimento, como veremos na próxima unidade.

Mesmo com tantas características individuais, segundo Pfleeger (2004) e Sommerville (2011), existem alguns conceitos fundamentais comuns:

- Todo produto de software deve ser desenvolvido em um processo gerenciado e compreendido. A empresa que o desenvolve deve possuir mecanismos de planejamento e controle do próprio processo de desenvolvimento.
- Todos os produtos devem possuir características como confiança e desempenho adequado, ou seja, devem se comportar conforme o esperado, sem falhas.
- É fundamental entender e gerenciar todos os requisitos do sistema a fim de garantir que o produto final seja o esperado.
- Você deve fazer o melhor uso possível dos recursos existentes e isso, hoje, significa reutilizar objetos e softwares desenvolvidos ao invés de iniciar o processo do zero.

Além dessas características, ainda temos que considerar a integração com a internet.

Se voltarmos no tempo e tentarmos montar uma linha cronológica sobre desenvolvimento de software, encontraremos acontecimentos que foram fundamentais para definir e estruturar o processo e a tecnologia existentes, principalmente em relação à integração com a internet.

Foi na década de 1940 que se iniciou a evolução dos sistemas computadorizados. Grande parte dos esforços e consequentes custos era concentrada no desenvolvimento do hardware, em razão, principalmente, de limitações e dificuldades encontradas na época. O foco era desenvolvimento de hardware, uma vez que o software tinha uma característica secundária.

À medida que a tecnologia de hardware foi dominada, as preocupações se voltaram ao software e, no início da década de 1950, o desenvolvimento dos sistemas operacionais começou a entrar em foco. Foi nessa época que as linguagens de programação de alto nível, como *Fortran* e *Cobol*, e respectivos compiladores, começaram a ser o centro da atenção.

A tendência da época foi de poupar cada vez mais o usuário de um computador de conhecer profundamente as questões relacionadas ao funcionamento interno da máquina, permitindo que ele pudesse concentrar seus esforços na resolução dos problemas computacionais, em lugar de preocupar-se com os problemas relacionados ao funcionamento da máquina (PRESSMAN, 1996).

Já no início da década de 1960, com o surgimento dos sistemas operacionais com características de multiprogramação, a eficiência e a utilidade dos sistemas computacionais tiveram um considerável crescimento. Esse fato contribuiu, de forma significativa, para a queda de preço do hardware e um aumento significativo de vendas de máquinas e software.

Uma característica desse período foi a necessidade, cada vez maior, de desenvolver grandes sistemas de software em substituição aos pequenos programas aplicativos que eram utilizados até então – a empresa conseguia comprar as máquinas e queria softwares “que funcionassem nelas”.

Disso surgiu um problema nada trivial devido à falta de experiência e a não adequação dos métodos de desenvolvimento existentes para pequenos programas. Esse problema, entre o que era necessidade de mercado e o que os desenvolvedores podiam suprir, foi caracterizado, ainda na década de 1960, como a “crise do software”.



Explore:

- **Pesquise mais sobre a crise do software da década de 1960.**

Com o passar do tempo, o preço de hardware foi, significativamente, baixando enquanto o de software não obedeceu a essa mesma tendência.

Por volta de 2000, a Internet começou a evoluir, e mais e mais recursos passaram a ser adicionados aos navegadores. Isso significou uma portabilidade diferente, ou seja, foi possível desenvolver um sistema e utilizar a interface do navegador para acessá-lo. Isso levou ao desenvolvimento de um número significativo de sistemas que puderam ser acessados via web. Outra característica é que não apenas foi possível acessá-lo a distância, mas também dar manutenção, atualizá-lo, excluí-lo. Isso também reduziu custos de produção, de manutenção e de testes.

Considerando essa evolução, é possível perceber que junto com essa conexão de sistemas também temos conceitos de ética relacionados.

Existem, na área de desenvolvimento de software, relações de confiabilidade, competência, uso inadequado de sistemas etc. Enfim, existe um código ético social além daqueles formalmente definidos, como o de direito de propriedade intelectual, que deve ser respeitado dentro dos parâmetros éticos da profissão.

Sociedades e instituições profissionais têm um papel importante a desempenhar na definição de padrões éticos. Organizações como ACM, IEEE (*Institute of Electrical em Eletronic Engineers*) e British Computer Society publicam códigos de conduta profissional, ou códigos de ética (SOMMERVILLE, 2011, p. 9).

Pessoas que se associam a essas instituições se comprometem a seguir tais códigos. A filiação a elas é comum a todos os profissionais de ES, uma vez que isso traz credibilidade mercadológica à empresa e ao produto desenvolvido.



Explore

IEEE, ACM, British Computer Society

- <http://www.computer.org/portal/web/about/sistersocieties>.

Hoje, o software corresponde a uma percentagem cada vez maior no custo global de um sistema informatizado; principalmente porque a tecnologia de desenvolvimento de software implica, ainda, em grande carga de trabalho, geralmente a um grande número de pessoas trabalhando juntas e num prazo relativamente pequeno de desenvolvimento. A elaboração desses sistemas é realizada, na maior parte das vezes, de forma *ad hoc*, o que pode conduzir a uma extração do tempo definido inicialmente para o desenvolvimento, o que acarreta aumento no custo final do sistema.

4. Reengenharia e Engenharia Reversa



Além do conceito de Engenharia de Software, há também dois outros conceitos relacionados: reengenharia e engenharia reversa.

O conceito de reengenharia está bastante relacionado à manutenção do software.

Os sistemas computacionais desenvolvidos para apoiar as atividades empresariais, chamados de sistemas de informação, devem se adaptar às mudanças ocorridas na empresa, para que continuem a atender às necessidades dos usuários. Esses sistemas são os que normalmente mais sofrem mudanças depois de sua implantação.

Um sistema de informação pode ser definido tecnicamente como um conjunto de componentes inter-relacionados que coleta (ou recupera), processa, armazena e distribui informações destinadas a apoiar a tomada de decisões, a coordenação e o controle de uma organização. Além de dar suporte à tomada de decisões, à coordenação e ao controle, esses sistemas também auxiliam os gerentes e trabalhadores a analisar problemas, visualizar assuntos complexos e criar novos produtos. (LAUDON; LAUDON, 2006, p. 7)

Segundo Laudon e Laudon (1999), o ambiente empresarial está se alterando e é fluido, isto é, novas tecnologias, tendências econômicas, desenvolvimentos políticos e regulamentações que afetam os negócios estão constantemente emergindo. Geralmente, quando as empresas falham, é porque negligenciaram a resposta ao ambiente mutável.

A empresa que não se adaptar às mudanças ambientais deixará de ser competitiva. Da mesma forma, os sistemas de informação dessas empresas devem também se adaptar às suas novas necessidades, para que continuem úteis e cumpram seu papel de apoiar as decisões.

Porém, muitos dos sistemas de informação fundamentais aos negócios estão se tornando de difíceis alterações. Para Pressman (1996), remendos são feitos sobre remendos, resultando em softwares que funcionam de forma ineficiente e falham com frequência, não correspondendo às necessidades dos usuários. Portanto, a manutenção de sistemas em fase de envelhecimento tornou-se proibitivamente dispendiosa para muitos sistemas de informação.

Quando, ao longo das manutenções, as alterações são feitas sem administração, sem atualização da documentação, na verdade, quando são feitos “remendos”, à medida que o tempo passa, novas alterações ficam cada vez mais difíceis de serem feitas. Além disso, sistemas elaborados com tecnologia muito arcaica, muitas vezes, não acomodam as necessidades atuais. É exatamente nesses casos que é feita a reengenharia do software. Reengenharia, em poucas palavras, significa reconstruir o software utilizando uma nova tecnologia, para melhorar a facilidade de manutenção.

De acordo com Pressman (1996), a reengenharia não somente recupera as informações de projeto de um software existente, mas usa essas informações para alterar ou reconstruir o sistema existente, num esforço para melhorar sua qualidade. Logo, um sistema que tenha sofrido reengenharia reimplementa a função do sistema existente, acrescido de novas funções.

A reengenharia pode envolver redocumentar, organizar e reestruturar o sistema, traduzir o sistema para uma linguagem de programação mais moderna e atualizar a estrutura e os valores dos dados do sistema. (SOMMERVILLE, 2005, p. 533)

A engenharia reversa tem sua origem no mundo do hardware. Uma empresa desmonta o produto de um concorrente para entender os segredos do projeto e da fabricação do mesmo. Esses segredos poderiam ser obtidos através das especificações do projeto e fabricação do produto, mas tais documentos não são disponíveis à empresa que está fazendo a engenharia reversa. A engenharia reversa de software é bastante semelhante. No entanto, na maioria das vezes, o software que passa pela engenharia reversa não é o de um concorrente, mas é trabalho desenvolvido pela própria empresa há muitos anos. Os segredos a serem descobertos são obscuros porque não houve, em casos assim, documentação (PRESSMAN, 2006).

Logo,

[...] a engenharia reversa é o processo de análise de um programa, em um esforço para representá-lo em uma abstração mais alta do que o código-fonte. A engenharia reversa é um processo de recuperação de projeto. (PRESSMAN, 2006, p. 687)

Para Sommerville (2005, p. 534), na “[...] engenharia reversa o programa é analisado e as informações são extraídas dele, a fim de ajudar a documentar sua organização e funcionalidade”.

A engenharia reversa não é o mesmo que reengenharia. O objetivo da engenharia reversa é derivar o projeto ou a documentação de um sistema a partir de seu código-fonte. Já o objetivo da reengenharia é produzir um novo sistema com manutenção mais fácil. Assim, a engenharia reversa para desenvolver uma melhor compreensão de um sistema é com frequência parte do processo de reengenharia (SOMMERVILLE, 2005).

Em outras palavras, a engenharia reversa pode anteceder um processo de reengenharia, num esforço de compreender os requisitos que levaram à construção do software a ser reconstruído.

Sendo assim, podemos concluir que a Engenharia de Software é o processo de desenvolver o software a partir de um estudo das necessidades do cliente e futuros usuários.

O processo de reengenharia de software tem como partida o conhecimento das características do sistema antigo a ser reconstruído. As características do sistema antigo podem ser levantadas com o auxílio da engenharia reversa.

Na nossa próxima unidade, apresentaremos os **modelos de processo de desenvolvimento de software**.

Material Complementar

O objetivo do material complementar é lhe ajudar a entender, sob uma ótica diferente daquela da professora conteudista, assuntos abordados nas unidades teóricas.

É fundamental a leitura deste material para o melhor entendimento sobre o assunto.



Explore

Como nesta unidade abordamos os conceitos gerais da Engenharia de Software, nossa sugestão de material complementar é a unidade introdutória da obra:

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2003. p. 4-16.

Referências

Bibliografia Fundamental

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011.

Bibliografia Básica

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação**. 4. ed. Rio de Janeiro: LTC, 1999.

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação Gerenciais. Administrando a empresa digital**. 5. ed. São Paulo: Pearson Education do Brasil, 2006.

PFLEEGER, S. L. **Engenharia de Software: teoria e prática**. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 2006.

SOMMERVILLE, I. **Engenharia de Software**. 6. ed. São Paulo: Pearson Addison Wesley, 2005.

Bibliografia Complementar

ALCADE, E.; GARCIA, M.; PENUELAS, S. **Informática Básica**. São Paulo: Makron Books, 1991.

FAIRLEY, R. E. **Software engineering concepts**. New York: McGraw-Hill, 1987.

IEEE **Software Engineering Standards (2013)**. Disponível em: <http://www.ieee.org/portal/innovate/products/standard/ieee_soft_eng.html>. Acesso em: 10 dez. 2013.

LUKOSEVICIUS, A. P.; CAMPOS FILHO, A. N.; COSTA, H. G. **Maturidade em Gerenciamento de Projetos e Desempenho dos Projetos**. Disponível em: <www.producao.uff.br/conteudo/rpep/.../RelPesq_V7_2007_07.doc>. Acesso em 12 nov. 2013.

MAFFEO, B. **Engenharia de software e especialização de sistemas**. Rio de Janeiro: Campus, 1992.

MICHAELIS. **Moderno dicionário da língua portuguesa**. São Paulo: Cia. Melhoramentos, 1998.

PARREIRA JÚNIOR, W. M. O. P. **Apostila de Engenharia de software**. Disponível em: <http://www.waltenomartins.com.br/ap_es_v1.pdf>. Acesso em: 13 nov. 2013.

PAULA FILHO, W. P. **Engenharia de Software: fundamentos, métodos e padrões**. 2. ed. Rio de Janeiro: LTC, 2001.

Revista Engenharia de Software. Disponível em: <<http://www.devmedia.com.br/revista-engenharia-de-software-magazine>>. Acesso em: 12 nov. 2013.

VONSTA, A. **Engenharia de programas.** Rio de Janeiro: LTC, 1983.

WIENNER, R.; SINCOVEC, R. **Software engineering with Modula 2 and ADA.** New York: Wiley, 1984.

WIKIPEDIA (2007a). **ISO 9000.** Disponível em: <http://pt.wikipedia.org/wiki/ISO_9000>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007b). **Melhoria de processo de software brasileiro.** Disponível em: <<http://pt.wikipedia.org/wiki/MPS.BR>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007c). **CMMI.** Disponível em: <<http://pt.wikipedia.org/wiki/Cmmi>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007d). **Engenharia de Software.** Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software>. Acesso em: 1 fev. 2007.

Anotações





Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

www.cruzeirodosulvirtual.com.br
Campus Liberdade
Rua Galvão Bueno, 868
CEP 01506-000
São Paulo SP Brasil
Tel: (55 11) 3385-3000



Universidade
Cruzeiro do Sul



UNICID
Universidade
Cidade de S. Paulo



UNIFRAN
Universidade
de Franca



UDF
Centro
Universitário



Módulo
Centro
Universitário

Engenharia de Software



Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

Material Teórico



Processos de software

Responsável pelo Conteúdo:

Prof.^a Dr.^a Ana Paula do Carmo Marchetti Ferraz

Revisão Textual:

Prof.^a Me. Luciene Oliveira da Costa Santos



- Introdução

- Modelos de desenvolvimento de software, chamados também de modelos de processo de software ou paradigmas de software



Objetivo de APRENDIZADO

- Compreender o que é um processo de software.
- Entender a importância do processo para o sucesso do produto desenvolvido.
- Conhecer e aplicar os principais paradigmas de desenvolvimento de software.

Organize-se de forma a não deixar para o último dia a realização das atividades (AS e AP), pois podem ocorrer imprevistos. Lembre-se de que, após o encerramento da Unidade, encerra-se a possibilidade de obter a nota relativa a cada atividade.

Para ampliar seu conhecimento, bem como aprofundar os assuntos discutidos, pesquise, leia e consulte os livros indicados nas Referências e/ou na Bibliografia.

As Referências estão indicadas ao final dos textos de conteúdo de cada Unidade.

A Bibliografia Fundamental para esta Disciplina é: SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011. A Bibliografia Complementar está indicada em item específico, em cada unidade.

Caso ocorram dúvidas, contate o professor tutor por meio do Fórum de Dúvidas, local ideal para esclarecê-las, pois assim a explicação poderá ser compartilhada por todos.

Existe, ainda, a possibilidade de contatar o professor pelo link mensagens, caso seja algum assunto relativo somente a você, e não uma dúvida sobre a matéria, que pode ser também dúvida de outros alunos.

Fique atento(a) quanto a possíveis dúvidas e problemas, lembrando-se de que o professor tutor está à sua disposição para resolver assuntos pedagógicos, isto é, dúvidas quanto ao conteúdo da matéria.

Se suas dúvidas ou problemas se referirem ao sistema, aos programas etc., contate o Suporte ou procure auxílio de um monitor no webclass de seu campus.

Contextualização

A partir das dificuldades envolvidas na produção de software, iniciaremos a contextualização sobre desenvolvimento de software, métodos e técnicas envolvidos, além de conhecer alguns dos principais paradigmas para o desenvolvimento de software.

Durante esta unidade, você conhecerá os principais modelos de processo de software. Afinal, são muitos os modelos propostos pela Engenharia de Software. Portanto, não limite seu aprendizado apenas aos modelos aqui apresentados, estude, pesquise, busque novos conhecimentos!

Adquira o hábito da pesquisa, visite sites, pesquise em livros e periódicos. Sugerimos que você dê pequenas pausas na leitura e pesquise sobre os conceitos abordados nas unidades. E lembre-se de compartilhar suas descobertas com seus colegas de turma.

1. Introdução



Um processo de software é um conjunto de atividades relacionadas que levam à produção de um produto de software (SOMMERVILLE, 2011, p. 33).

Para Pfleeger (2004), é um conjunto de atividades cujo objetivo é o desenvolvimento ou evolução do software.

Existem muitos processos diferentes de desenvolvimento de software, entretanto, algumas atividades são genéricas a todos eles:

- **Especificação:** a funcionalidade do software e as restrições em sua operação devem ser definidas.
- **Desenvolvimento:** produção do software de modo que atenda a suas especificações.
- **Validação:** o software deve ser validado para garantir que ele faça o que o cliente deseja.
- **Evolução:** o software deve evoluir para atender às necessidades mutáveis do cliente.

Glossário



Cliente: “[...] é a empresa, organização ou pessoa que está pagando para o sistema de software ser desenvolvido”. (PFLEEGER, 2004, p. 11)

Não existe, segundo Pressman (2006, p. 27), uma única abordagem ou método de produção de software mágico, que funcione para todos os tipos, o que existe é uma combinação de métodos abrangentes a todas as etapas relacionadas.

Além disso, é importante e desejável que esses métodos sejam suportados por um conjunto de ferramentas que permita automatizar o desenrolar das etapas, juntamente com uma definição clara de critérios de qualidade e produtividade de software. São esses aspectos que caracterizam de maneira mais influente a disciplina de **Engenharia de Software**.

Assim, podemos relembrar com propriedade que Engenharia de Software é uma disciplina que reúne **procedimentos** (metodologias), **métodos e ferramentas** a serem utilizados, desde a percepção do problema até o momento em que o sistema desenvolvido deixa de ser operacional (existir), visando resolver problemas inerentes ao processo de desenvolvimento e ao produto de software (PFLEEGER, 2004).

Segundo Sommerville (2009) e Pfleeger (2004):

- **Métodos:** proporcionam os detalhes de como fazer para construir o software (gerenciamento do projeto, análise de sistemas, análise de requisito, projeto do software, geração do código, teste, manutenção etc.). Muitos desses veremos nas próximas unidades.
- **Ferramentas:** dão suporte automatizado aos métodos. Existem ferramentas para sustentar cada um dos métodos. Quando as ferramentas são integradas, é estabelecido um sistema de suporte ao desenvolvimento de software chamado *Case - Computer Aided Software Engineering*.
- **Procedimentos:** constituem o elo entre os métodos e ferramentas. Estão relacionados à sequência em que os métodos serão aplicados aos produtos de software. Por meio dos procedimentos, ocorrem controles que asseguram a qualidade e coordenam as alterações, além de permitirem marcos de referência que possibilitam a administração do progresso do produto de software.

Nesse contexto, a Engenharia de Software apoia-se na tecnologia para produzir software de alta qualidade, a um baixo custo e num tempo menor. Em contrapartida, os engenheiros de software a utilizam para enfrentar os desafios do desenvolvimento de software.

Existem vários modelos de processo de software também chamados de **paradigmas de engenharia de software**. Cada modelo representa uma perspectiva particular de um processo, portanto, não fornece todas as informações. Por exemplo, um determinado modelo de atividade de processo pode mostrar as atividades envolvidas e suas sequências, mas não mostra os papéis dos agentes envolvidos.

Glossário

Processo de Software: “[...] um arcabouço para as tarefas que são necessárias para construir softwares de alta qualidade”. (PRESSMAN, 2006, p. 16)



2. Modelos de desenvolvimento de software, chamados também de modelos de processo de software ou paradigmas de software



Glossário



Paradigma da Engenharia de Software: “[...] representa a abordagem ou filosofia em particular para a construção de software”. (PFLEEGER, 2004, p. 3) Segundo Pressman (1996, p. 33): “[...] a engenharia de software compreende um conjunto de métodos, ferramentas e procedimentos [...] estas etapas são muitas vezes citadas como paradigmas de engenharia de software”.

O esforço de desenvolvimento deve resultar em um **produto**.

O **modelo de processo de desenvolvimento** corresponde ao conjunto e ao ordenamento de atividades de modo a que o produto desejado seja obtido (SOMMERVILLE, 1994).

Um **modelo de desenvolvimento** corresponde a uma representação abstrata do processo de desenvolvimento que vai, em geral, definir como as etapas relativas à criação do software serão conduzidas e inter-relacionadas para atingir o objetivo – que é a obtenção de um produto de software de alta qualidade a um custo relativamente baixo.

Vários são os modelos existentes:

- **Sequencial Linear**

- Modelo Cascata ou Ciclo de Vida Clássico *
- Prototipação *
- O Modelo RAD (*Rapid Application Development*)

- **Modelos Evolutivos de Processo de Software**

- Incremental *
- Espiral *
- De Montagem de Componentes *
- De Desenvolvimento Concorrente

- **Modelos de Métodos Formais**

- **Técnicas de Quarta Geração**

- **Métodos Ágeis***

- *Extreme Programming (XP)*

Na nossa disciplina, por uma questão de tempo, não veremos todos os modelos, mas apenas os que estão marcados com asteriscos. Por isso é importante que você faça uma pequena pesquisa sobre outros modelos para conhecê-los.

2.1 Modelos Lineares

2.1.1 O modelo cascata ou ciclo de vida clássico

Este é o modelo mais simples de desenvolvimento de software, estabelecendo uma ordenação linear no que diz respeito à realização das diferentes etapas.

Ele é o modelo mais antigo e o mais amplamente usado da Engenharia de Software por ter sido modelado em função do ciclo da Engenharia convencional.

Requer uma abordagem sistemática, sequencial ao desenvolvimento de software e o resultado de uma fase se constitui na entrada da outra. (Figura 1)

Existem inúmeras variações desse modelo, dependendo da natureza das atividades e do fluxo de controle entre elas.

Os estágios principais do modelo estão relacionados às atividades fundamentais de desenvolvimento.

O ponto de partida do modelo é uma etapa de **Engenharia de Sistemas**, onde o objetivo é ter uma visão global do sistema (incluindo hardware, software, equipamentos e as pessoas envolvidas). Em seguida, a etapa de **Análise de Requisitos** vai permitir uma clara definição dos requisitos de software, na qual o resultado será utilizado como referência para as etapas posteriores de **Projeto, Codificação, Teste e Manutenção**.

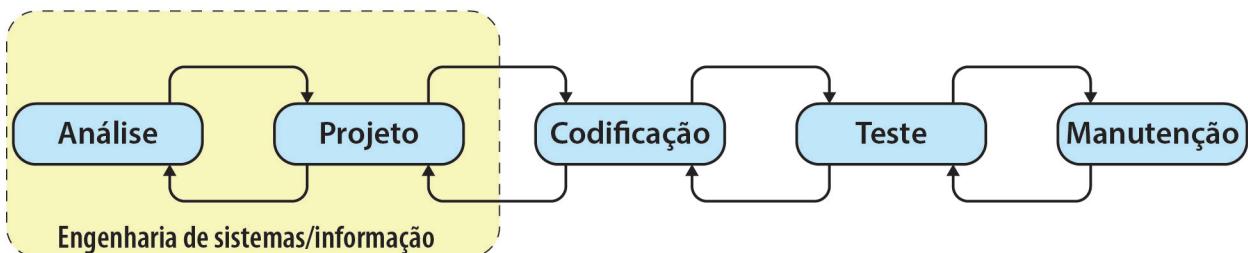


Figura 1 - Ilustração do modelo Cascata adaptado de Pressman (2006).

O modelo Cascata, por ter uma ordenação linear nas etapas de desenvolvimento, apresenta características úteis no processo de desenvolvimento de software, particularmente em razão da definição de um ordenamento linear das etapas de desenvolvimento.

Como forma de identificar o fim de uma etapa e o início da próxima, existem mecanismos implementados ao final de cada etapa que serve como balizador para início da próxima. Isso é feito normalmente através da aplicação de algum método de validação ou verificação, cujo objetivo é garantir que a saída de uma dada etapa seja coerente com a sua entrada (a qual já é a saída da etapa precedente).

Isso significa que, ao final de cada etapa realizada, deve existir um resultado (ou saída) a qual possa ser submetida à atividade de certificação.

Segundo Pressman (2006, p. 67), duas diretivas que norteiam o desenvolvimento, segundo o modelo Cascata, são:

1. Todas as etapas definidas no modelo devem ser realizadas, isto porque, em projetos de grande complexidade, a realização formal delas vai determinar o sucesso ou não do desenvolvimento.

A realização informal e implícita de algumas dessas etapas poderia ser feita apenas no caso de projetos de pequeno porte.

2. A ordenação das etapas na forma como foi apresentada deve ser rigorosamente respeitada.

Temos sempre que lembrar que os resultados de um processo de desenvolvimento de produto de software não é apenas o programa executável, mas também toda documentação associada.

Cada fase mostrada na Figura 1 pode gerar resultados que devem ser documentados (processo e resultados). Alguns desses documentos são: de especificação de requisitos, de projeto do sistema, de plano e relatório de testes, de codificação ou implementação, de utilização, relatórios de revisões etc.

Todo modelo tem suas vantagens e desvantagens na sua utilização, e no modelo Cascata não seria diferente. Vejamos algumas limitações desse modelo:

- Nenhum modelo segue rotinas tão sequenciais quanto almeja.
- No início do processo, não temos todos os requisitos mapeados ou definidos.
- No começo dos projetos, sempre existe uma incerteza. O cliente deve ter paciência, pois uma versão final do produto só fica pronta e disponível no final do processo.

Algumas contribuições do modelo:

- Esse modelo requer que o desenvolvedor e todos os envolvidos tenham disciplina, e realizem o planejamento e gerenciamento do processo.
- O desenvolvimento do produto deve ser adiado até que os objetivos tenham sido completamente compreendidos.

2.1.2 Modelo de prototipação

A Prototipação é um modelo de processo de desenvolvimento que busca contornar algumas das limitações existentes no modelo Cascata, inclusive apresentadas anteriormente. Isso é possível devido ao fato de que o foco principal é o desenvolvimento de um protótipo, com base no conhecimento dos requisitos iniciais para o sistema. Dessa forma, o cliente poderá ter uma ideia de como ficará seu produto, antes da etapa final.

Para desenvolver o protótipo, etapas devem ser seguidas (a análise de requisitos, o projeto, a codificação e os testes). Entretanto, elas não precisam ser realizadas com o rigor necessário (final) que o modelo Cascata define. Podemos criar o protótipo com apenas alguns requisitos definidos e, ao longo do processo, o protótipo vai se completando conforme a definição de outros.

Esse modelo tem como objetivo entender os requisitos do usuário para, assim, obter uma melhor definição dos requisitos do sistema e possibilitar que o desenvolvedor crie um modelo (protótipo) do software que deve ser construído.

É apropriado para quando o cliente já definiu um conjunto de objetivos gerais para o software, mas não identificou detalhadamente esses requisitos.

Após criar o protótipo, ele é colocado à disposição do cliente, que vai ajudar na melhor compreensão do que será o sistema desenvolvido. Além disso, através da manipulação desse protótipo, é possível validar ou reformular os requisitos para as etapas seguintes do sistema.

Segundo Pressman (2006), o modelo ilustrado na figura 1 apresenta algumas características interessantes, tais como:

- * Através da construção de um protótipo do sistema, é possível demonstrar a reusabilidade do mesmo.
- * É possível obter uma versão, mesmo que simplificada, do que será o sistema, com um pequeno investimento inicial.

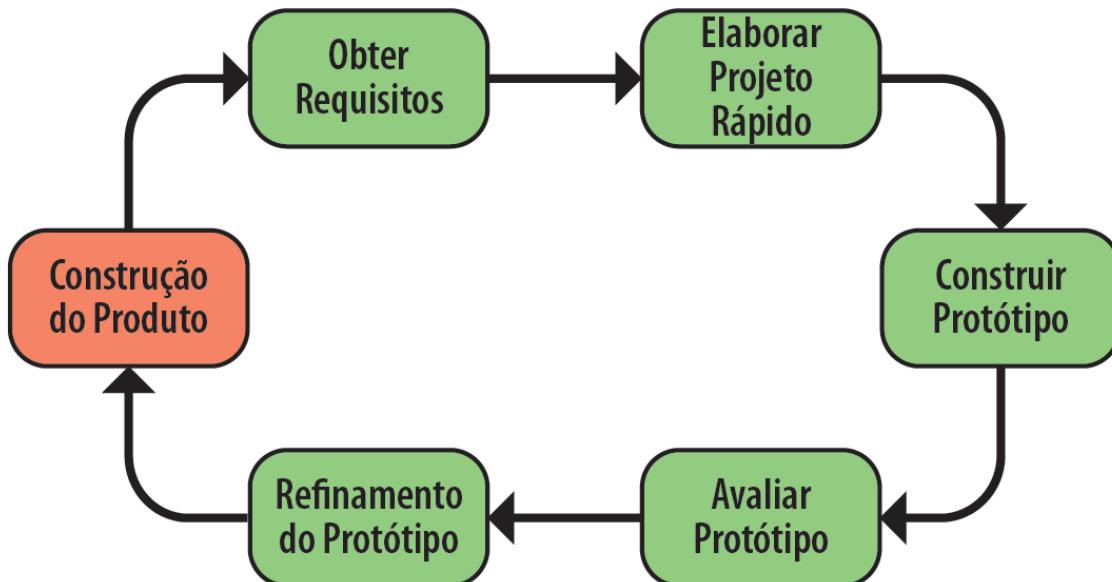


Figura 2 - Esquema de evolução da prototipação.

Problemas com a prototipação:

- O cliente não sabe que o software que ele vê não considerou, durante o desenvolvimento, a qualidade global e a manutenibilidade em longo prazo.
- O desenvolvedor frequentemente faz uma implementação comprometida (utilizando o que está disponível), com o objetivo de produzir rapidamente um protótipo.
- Os protótipos não são sistemas completos e deixam, normalmente, a desejar em alguns aspectos. Um desses aspectos é normalmente a interface com o usuário.
- Os esforços de desenvolvimento são concentrados principalmente nos algoritmos que implementam as principais funções associadas aos requisitos apresentados, a interface, sendo, nesse nível, parte supérflua do desenvolvimento, o que permite caracterizar esta etapa por um custo relativamente baixo.

Contribuições:

- Ainda que possam ocorrer problemas, a prototipação é um ciclo de vida eficiente.
- A chave é definir todas as regras (requisitos, processos, métodos, técnicas etc.) logo no começo do desenvolvimento.
- O cliente e o desenvolvedor devem concordar que o protótipo seja construído para servir como um mecanismo a fim de definir os requisitos.
- A experiência adquirida no desenvolvimento do protótipo vai ser de extrema utilidade nas etapas posteriores do desenvolvimento do sistema real, permitindo reduzir certamente o seu custo, resultando também num sistema melhor concebido.

Segundo Sommerville (2011), às vezes, os desenvolvedores são pressionados a entregar protótipos descartáveis, principalmente quando há atraso no prazo de entrega. Entretanto, isso não é aconselhável, pois, dentre outras coisas, pode ser impossível ajustar o protótipo para atender aos requisitos não funcionais como desempenho, robustez, segurança, confiabilidade etc. durante o desenvolvimento do protótipo.

2.2 Modelos Evolutivos de Processo de Software

Existem situações em que a Engenharia de Software necessita de um modelo de processo que possa acomodar um produto que evolui com o tempo.

São eles, segundo Pressman (2006, p. 87):

- Quando os requisitos de produto e de negócio mudam conforme o desenvolvimento procede.
- Quando um prazo de entrega é estreito (mercado), há impossibilidade de conclusão de um produto completo.
- Quando um conjunto de requisitos importantes é bem conhecido, mas os detalhes ainda devem ser definidos.

Os modelos evolutivos são iterativos e possibilitam o desenvolvimento de versões cada vez mais completas do software.

2.2.1 Modelo Incremental

Como no modelo de Prototipação, o modelo Incremental também foi concebido a partir da exploração das limitações do modelo Cascata, ao mesmo tempo em que combina as vantagens do modelo Cascata com as do de Prototipação.

A ideia principal desse modelo, ilustrada na Figura 3, é a de que um sistema deve ser desenvolvido de forma incremental, sendo que cada incremento vai adicionando ao sistema novas capacidades funcionais, até a obtenção do sistema final, ao passo que, a cada etapa realizada, modificações podem ser introduzidas.

Uma das vantagens dessa abordagem é a facilidade em testar o sistema durante cada uma das fases de desenvolvimento.

Um aspecto interessante desse modelo é a criação de uma lista de controle de projeto. Ela deve conter todas as etapas a serem realizadas, da concepção à obtenção do sistema final. Ela vai servir também para medir, num dado nível, o quanto distante está da última versão aquela que será disponibilizada ao cliente.

Esta lista de controle de projeto gerencia todo o desenvolvimento, definindo quais tarefas devem ser realizadas a cada iteração. Na lista de tarefas, podemos, inclusive, inserir as redefinições de componentes já implementados, em razão de erros ou problemas detectados numa eventual etapa de análise.

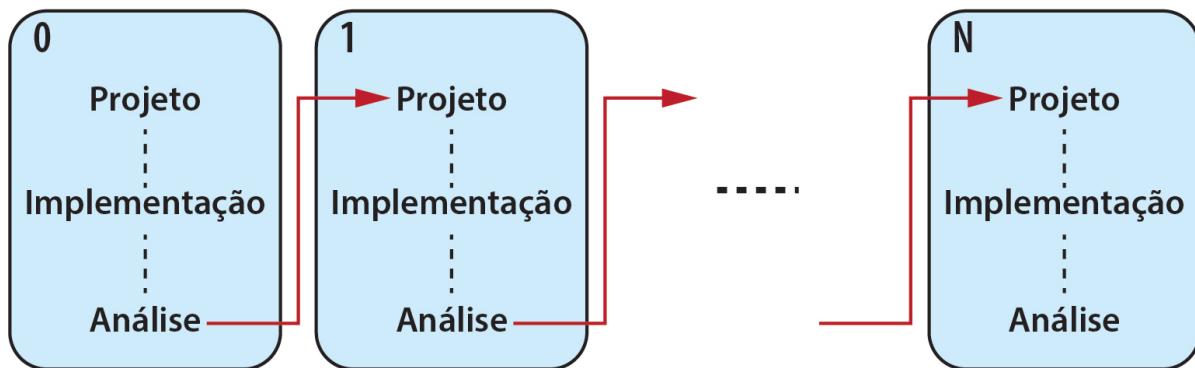


Figura 3 - O modelo de desenvolvimento incremental.

Considerações positivas sobre o modelo:

- A versão inicial é frequentemente o núcleo do produto (a parte mais importante).
 - O desenvolvimento começa com as partes do produto que são mais bem entendidas.
 - A evolução acontece quando novas características são adicionadas à medida que são sugeridas pelo usuário.
- O desenvolvimento incremental é importante quando é difícil, ou mesmo impossível, estabelecer *a priori* uma especificação detalhada dos requisitos.
- As primeiras versões podem ser implementadas com poucas pessoas se o núcleo do produto for bem recebido, novas pessoas podem participar do desenvolvimento da nova versão.
- As novas versões podem ser planejadas de modo que os riscos técnicos possam ser administrados.

Exemplo:

- O sistema pode exigir a disponibilidade de um hardware que está em desenvolvimento e cuja data de liberação é incerta.
- Podem ser planejadas pequenas versões de modo a evitar o uso desse hardware.
- O software com funcionalidade parcial pode ser liberado para o cliente.

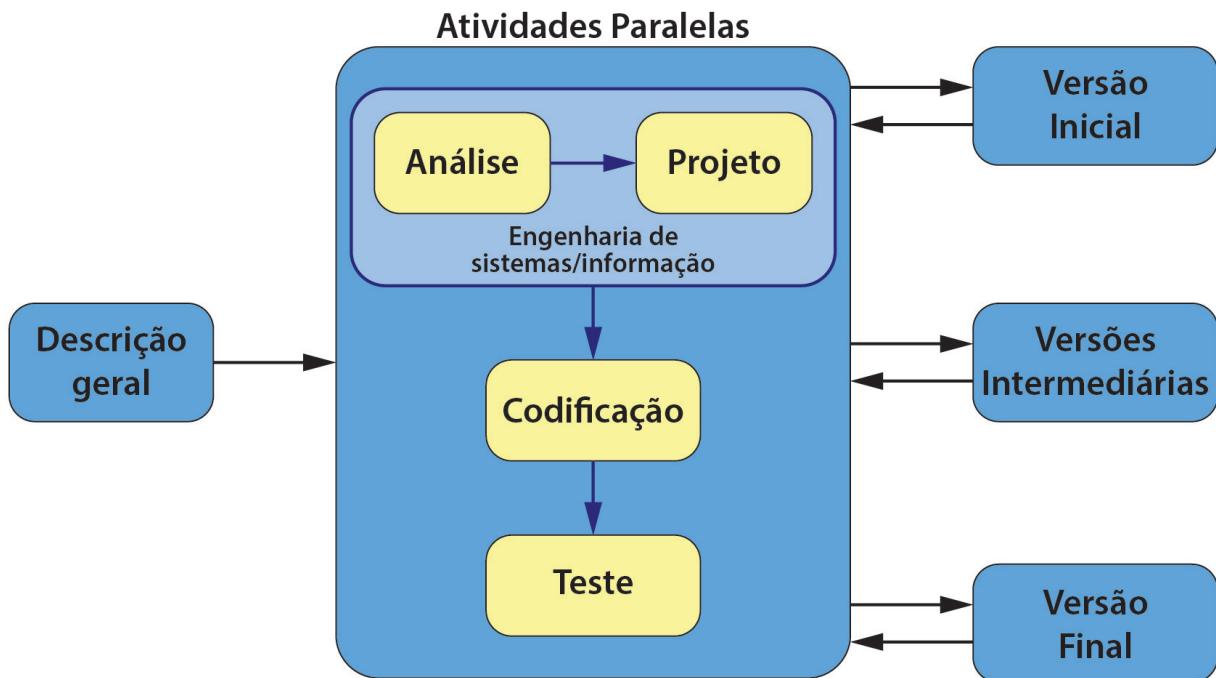


Figura 4 - Modelo Incremental, com a exemplificação das atividades paralelas.

2.2.2 O Modelo Espiral

Esse modelo foi proposto em 1988 e sugere uma organização de desenvolvimento em espiral, o que deu origem ao nome Modelo de Desenvolvimento Espiral.

Ele reúne a natureza iterativa da prototipação com os aspectos controlados e sistemáticos do modelo Cascata.



Explore

Faça uma pesquisa em algum dicionário sobre a diferença entre interação e iteração. O conhecimento e reconhecimento dessa diferença, para você que está estudando sobre Engenharia de Software, é muito importante.

O modelo Espiral é dividido em uma série de atividades de trabalho ou regiões de tarefa.

Existem tipicamente de 3 a 6 regiões ou setores de tarefa. Na Figura 5 temos um exemplo de 4 regiões ou setores definidos pelos quadrantes.

Segundo Sommerville (2011), em cada uma delas, temos:

Definição dos objetivos: os objetivos específicos para esta fase são definidos, restrições ao processo e ao produto são identificadas e um plano de gerenciamento é detalhado e elaborado; **riscos são identificados.**

Avaliação e redução de riscos: para cada um dos riscos identificados é feita uma análise detalhada e ações para eliminá-los ou reduzi-los são tomadas. A continuidade do processo de desenvolvimento é definida como função dos riscos que ainda não foram resolvidos, inclusive

em ordem de prioridade. Um bom exemplo disso são os riscos relacionados ao desempenho ou à interface que são considerados mais importantes do que aqueles relacionados ao desenvolvimento do programa. A partir das decisões tomadas na avaliação e redução de riscos, um próximo passo pode ser o desenvolvimento de um protótipo que elimine os riscos considerados para novamente avaliar o produto, o processo e identificarmos os riscos.

Por outro lado, se os riscos de desenvolvimento de programa forem considerados os mais importantes, e se o protótipo construído já resolver alguns dos riscos ligados ao desempenho e à interface, podemos seguir à implementação utilizando o modelo Cascata na nossa fase do projeto.

Desenvolvimento e Avaliação: após a avaliação dos riscos, é selecionado um modelo para o aprimoramento do sistema, que pode ser Prototipação, Incremental, Cascata etc.

Planejamento: o projeto é revisado e uma decisão é tomada a respeito da continuidade do modelo com mais de uma volta da espiral. Caso se decida pela continuidade, novos planos são elaborados para a próxima fase do projeto.

Como se pode ver, o elemento que conduz esse processo é a consideração sobre os riscos existentes ou iminentes, o que permite, em alguns casos, adequação e escolha de qualquer política ou modelo de desenvolvimento, como a baseada em especificação, baseada em simulação, baseada em protótipo, dentre outras.

Esse modelo, por ter sido desenvolvido há pouco tempo, não é muito utilizado, embora seja muito adequado ao desenvolvimento de sistemas complexos.

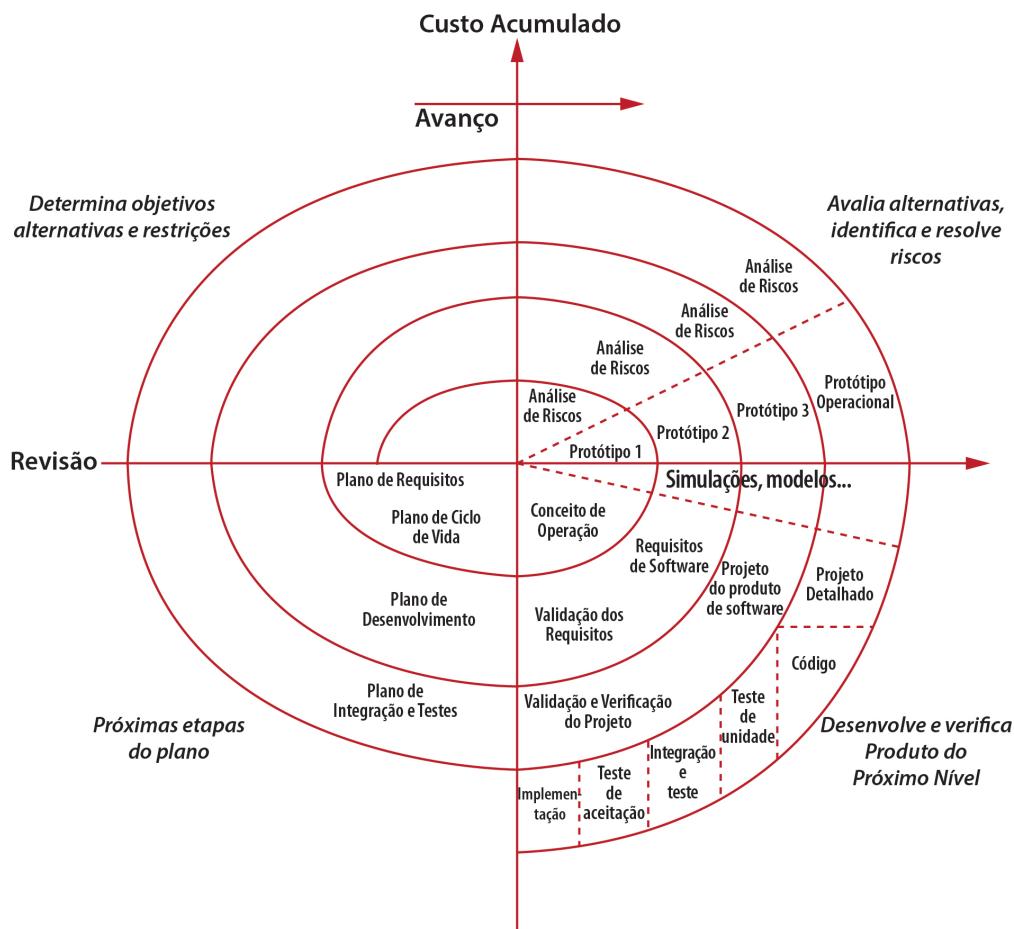


Figura 5 - O modelo Espiral. (SOMMERVILLE, 2011, p. 33).

Observações sobre esse modelo, segundo Sommerville (2006, 2001) e Pfleeger (2004):

- Cada loop no espiral representa uma fase do processo de software. O mais interno está concentrado nas possibilidades do sistema.
- O próximo loop está concentrado na definição dos requisitos do sistema.
- O loop um pouco mais externo está concentrado no projeto do sistema.
- Engloba as melhores características do ciclo de vida Clássico e da Prototipação, adicionando um novo elemento: a Análise de Risco.
- Esse modelo segue a abordagem de passos sistemáticos do Ciclo de Vida Clássico, incorporando-os numa estrutura iterativa que reflete mais claramente o mundo real.
- Usa a Prototipação, em qualquer etapa da evolução do produto, como mecanismo de redução de riscos.
- É, atualmente, a abordagem mais realística para o desenvolvimento de software em grande escala.
- Usa uma abordagem que capacita o desenvolvedor e o cliente a entender, perceber e resolver os riscos em cada etapa evolutiva.
- Pode ser difícil convencer os clientes de que uma abordagem evolutiva é controlável.
- Exige considerável experiência na determinação de riscos e depende dessa experiência para ter sucesso.

Uma característica importante desse modelo é o fato de que cada ciclo é encerrado por uma atividade de revisão, na qual todos os produtos do ciclo são avaliados, incluindo o plano para o próximo passo (ou ciclo).

2.2.3 O Modelo de Montagem de Componentes

As tecnologias de objeto fornecem o arcabouço técnico para o modelo de processo baseado em componentes. Esse modelo é orientado a objeto e enfatiza a criação de classes que encapsulam tanto os dados quanto os algoritmos que são usados para manipular os dados.

Quando projetadas e implementadas de forma adequada, todas as classes orientadas a objeto são reutilizáveis em diferentes aplicações e arquiteturas de sistema.

Esse modelo incorpora características de tecnologias orientadas a objetos no modelo Espiral, é de natureza evolutiva e demanda uma abordagem iterativa para a criação do software.

2.3 Métodos Ágeis

Entre as décadas de 1980 e 1990, havia uma visão generalizada de que a melhor forma de produzir um software era por meio de planejamento cuidadoso do projeto, qualidade de segurança formalizada do uso de métodos de Engenharia de Software e projeto apoiado por ferramentas CASE e de processo de desenvolvimento de software rigorosamente controlado.

Glossário



Métodos de engenharia de software: Segundo Pressman (1996, p. 31): “[...] proporcionam detalhes de ‘como fazer’ para construir o software. Os métodos envolvem um grande número de tarefas que incluem: planejamento e estimativa de projeto, análise de requisitos de software e de sistema, projeto da estrutura de dados, arquitetura de programa e algoritmo de processamento, codificação, teste e manutenção.”

Entretanto, foi percebido que essa abordagem pesada de desenvolvimento, num ambiente dinâmico e conectado não era adequada. Muito tempo era gasto no processo de planejamento e análise de requisitos e isso prejudicava algumas etapas, sem contar o atraso no prazo de entrega.

A insatisfação com essa abordagem na década de 1990 foi tão grande que um grupo de engenheiros de software idealizou uma nova abordagem, a qual foi chamada de Métodos Ágeis. Esses métodos permitiram que a equipe de desenvolvimento focasse no software em si e não na documentação do processo. Assim, esse modelo mostrou-se adequado para aqueles cujos requisitos mudam constantemente durante o processo de desenvolvimento.

Com esse método foi possível executar a entrega de sistemas complexos em tempos menores. Além disso, quaisquer novos requisitos solicitados têm a possibilidade de inclusão como alterações de sistema, o que significa a entrega de uma nova versão com tais adequações.

Segundo Sommerville (2011), práticas ágeis enfatizam a importância de se escrever códigos bem estruturados e investir na sua melhoria. A falta de documentação não deve ser um problema na manutenção do sistema por meio de uma abordagem ágil.

Entretanto, talvez esse seja o principal problema dessa abordagem, pois manutenção, testes e alterações de sistemas se tornam extremamente complexos quando não existem documentações disponíveis.

À medida que as características dos softwares, dos desenvolvedores e dos clientes mudam, a Engenharia de Software procura adaptar seus modelos à nova realidade.

A metodologia Extreme Programming, também conhecida como XP, ou Programação Extrema, proposta por Kent Beck, em 1998 (TELES, 2004), é um dos mais novos métodos da Engenharia de Software e é um exemplo de metodologia ágil para desenvolvimento de software. Por ser um modelo novo e bastante aceito, vejamos a seguir alguns de seus detalhes:

O XP é um processo de desenvolvimento que busca assegurar que o cliente receba o máximo de valor de cada dia de trabalho da equipe de desenvolvimento. Ele é organizado em torno de conjunto de valores e práticas que atuam de forma harmônica e coesa para assegurar que o cliente sempre receba um alto retorno do investimento em software. (TELES, 2004, p. 21)

Essa abordagem foi desenvolvida para impulsionar práticas reconhecidamente boas, como desenvolvimento interativo, em níveis extremos. Por exemplo, em um XP, várias novas versões de um sistema podem ser desenvolvidas, integradas e testadas em um único dia por programadores diferentes (SOMMERVILLE, 2011).

O ciclo de vida do XP tem quatro atividades básicas: codificar, testar, escutar e modelar, demonstradas através de quatro valores: **a comunicação, a simplicidade, o feedback e a coragem** (SOMMERVILLE, 2006).

A prática do XP tem **comunicação** contínua entre o cliente e a equipe, com o objetivo de criar o melhor relacionamento entre o cliente e desenvolvedor, dando preferência às conversas pessoais em vez de manter contato através de outros meios. É incentivada também a comunicação intensa entre desenvolvedores e gerente do projeto. Essa comunicação entre o cliente e a equipe permite que todos os detalhes do projeto sejam tratados com a atenção e a agilidade que merecem, ou seja, a equipe pode codificar uma funcionalidade preocupando-se apenas com os problemas de hoje e deixar os problemas do futuro para o futuro.

A **simplicidade** foca sempre no mais simples que possa funcionar, visando minimizar o código, desprezando funções consideradas necessárias. O importante é ter em mente que os requisitos são mutáveis. Sendo assim, o interessante no XP é implementar apenas o que é estritamente necessário.

O contato incessante com o cliente a respeito do projeto é o que se pode chamar de **feedback constante**. As informações sobre o código são dadas por teste periódico, os quais indicam erros tanto individuais quanto do software integrado. O cliente terá sempre uma parte do software funcional para avaliar.

A grande maioria dos princípios do XP corresponde a práticas de senso comum e que fazem parte de qualquer processo disciplinado. Por exemplo, simplicidade significa foco nas partes do sistema que são de alta prioridade e, somente depois, pensar em soluções para problemas que, até então, não são relevantes (e talvez nunca sejam devido às grandes modificações no sistema), pois o cliente aprende com o sistema que utiliza e reavalia as suas necessidades, gerando o feedback para a equipe de desenvolvimento.

As equipes XP acreditam que errar é natural e que falhas no que estava funcionando acontecem cedo ou tarde. É necessário ter coragem para lidar com esse risco, o que em XP se traduz em confiança nos seus mecanismos de proteção. A **coragem** encaixa-se na implantação dos três valores anteriores. São necessários profissionais comunicativos e com facilidade de se relacionar. A coragem auxilia a simplicidade, quando a possibilidade de simplificar o software é detectada. Desse modo, a coragem é necessária para garantir que o feedback do cliente ocorra com frequência, pois essa abordagem implica a possibilidade de mudanças constantes do produto.

Enfim, no sistema desenvolvido de forma incremental, a equipe está continuamente fazendo a manutenção do software e criando novas funcionalidades. Por isso a equipe precisa ser corajosa e acreditar que, utilizando práticas e valores do XP, será capaz de fazer o software evoluir com segurança e agilidade.

A abordagem tradicional é mais arriscada que a do XP, pois os usuários não têm a chance de avaliar o software sempre, e o projeto só começa a gerar receita após a conclusão.

Segundo Brooks (1995), é comum o cliente fazer um grande investimento na construção de um software e este não atender às suas necessidades, levando a um novo projeto para a construção de outro software que atenda aos anseios dos usuários.

Por todas essas razões, os projetos em XP trabalham com releases de, no máximo, dois meses. Isso permite incorporar uma boa quantidade de funcionalidade em cada release e permite que os usuários aprendam com o software com bastante frequência.

O XP é aplicado em larga escala em vários projetos no mundo todo, mas ainda há muito a evoluir em sua compreensão e aplicação. Notamos isso principalmente em pontos polêmicos como testes unitários, programação em dupla, rodízio de pessoas, propriedade coletiva do código e otimização de jornadas, práticas que, se mal utilizadas, podem realmente trazer aumentos no custo e no prazo de projetos. Ou seja, é de extrema importância que entendamos bem a essência em XP e, principalmente, que tenhamos disciplina e criatividade, duas qualidades básicas em quem pretende desenvolver projetos. Somente a partir de uma visão criativa sobre a metodologia e disciplina para cumprimento de tarefas é que todos poderão usar e ter benefícios em XP.

2.4 As fases genéricas da Engenharia de Software e suas composições

Agora podemos definir e agrupar alguns dos assuntos abordados até o momento com o intuito de dar uma visão geral do papel do Engenheiro de Software.

Sabemos que a Engenharia engloba métodos, técnicas e procedimentos com o objetivo de desenvolver um bom produto de software.

Entretanto, é importante lembrarmos que, independentemente da área de aplicação, tamanho ou complexidade do projeto/produto, o trabalho associado à Engenharia de Software pode ser categorizado em 3 fases genéricas: fase de definição de projeto, fase de desenvolvimento e fase de manutenção, sendo que todas são complementadas por atividades de apoio.

A fase de **definição de processo** focaliza **o que** será desenvolvido: que informação vai ser processada, que função e desempenho são desejados, que comportamento pode ser esperado do sistema, que interfaces vão ser estabelecidas, que restrições de projeto existem, que critérios de validação são exigidos para definir um sistema bem-sucedido. Nessa fase, acontece a Engenharia de Sistemas, planejamento do projeto de software e análise de requisitos.

A fase de **desenvolvimento de processos** de software focaliza como o software será desenvolvido: como os dados vão ser estruturados, como a função vai ser implementada como uma arquitetura de software, como os detalhes procedimentais vão ser implementados, como as interfaces vão ser caracterizadas, como o projeto será traduzido em uma linguagem de programação, como os testes serão efetuados. Nessa fase, acontece o projeto do software, geração do código e o teste do software.

A última fase é a da **manutenção**, que focaliza as **mudanças** que ocorrerão depois que o software for liberado para uso operacional. Essa etapa reaplica os passos das fases de definição e desenvolvimento, mas faz isso no contexto de um software existente.

As fases do processo de software são complementadas por uma série de atividades de apoio. Essas atividades são aplicadas durante toda a Engenharia do Software. São elas: Métricas, Gerenciamento de Riscos, Acompanhamento e Controle do Projeto de Software, Garantia de Qualidade de Software, Gerenciamento de Configuração de Software.

Nas nossas próximas unidades, abordaremos vários dos itens existentes em cada uma das fases.



Explore

Dica: visite sempre o site: <http://www.devmedia.com.br/revista-engenharia-de-software-magazine>.

Essa revista sobre Engenharia de Software apresenta relatos, casos e aplicações práticas dos assuntos abordados nesta disciplina.

Material Complementar

O objetivo do material complementar é lhe ajudar a entender, sob uma ótica diferente daquela da professora conteudista, assuntos abordados nas unidades teóricas.

É fundamental a leitura deste material para o melhor entendimento sobre o assunto.



Explore

Como nesta unidade abordamos os conceitos gerais da Engenharia de Software, nossa sugestão de material complementar é o capítulo intitulado Processo de software, no seguinte livro:

SOMMERRVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011. p. 18-37.

Referências

Bibliografia Fundamental

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011.

Bibliografia Básica

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação**. 4. ed. Rio de Janeiro: LTC, 1999.

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação Gerenciais. Administrando a empresa digital**. 5. ed. São Paulo: Pearson Education do Brasil, 2006.

PFLEEGER, S. L. **Engenharia de Software: teoria e prática**. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 2006.

SOMMERVILLE, I. **Engenharia de Software**. 6. ed. São Paulo: Pearson Addison Wesley, 2005.

Bibliografia Complementar

ALCADE, E.; GARCIA, M.; PENUELAS, S. **Informática Básica**. São Paulo: Makron Books, 1991.

FAIRLEY, R. E. **Software engineering concepts**. New York: McGraw-Hill, 1987.

IEEE Software Engineering Standards (2013) Disponível em: <http://www.ieee.org/portal/innovate/products/standard/ieee_soft_eng.html>. Acesso em: 10 dez. 2013.

LUKOSEVICIUS, A. P.; CAMPOS FILHO, A. N.; COSTA, H. G.. **Maturidade Em Gerenciamento De Projetos E Desempenho Dos Projetos**. Disponível em: <[www.producao.uff.br/conteudo/rpep/.../RelPesq_V7_2007_07.doc](http://producao.uff.br/conteudo/rpep/.../RelPesq_V7_2007_07.doc)>. Acesso em 12 nov. 2013.

MAFFEO, B. **Engenharia de software e especialização de sistemas**. Rio de Janeiro: Campus, 1992.

MICHAELIS. **Moderno dicionário da língua portuguesa**. São Paulo: Cia. Melhoramentos, 1998.

PARREIRA JÚNIOR, W. M. O. P. **Apostila de Engenharia de software**. Disponível em: <http://www.waltenomartins.com.br/ap_es_v1.pdf>. Acesso em: 13 nov. 2013.

PAULA FILHO, W. P. **Engenharia de Software: fundamentos, métodos e padrões**. 2. ed. Rio de Janeiro: LTC, 2001.

Revista Engenharia de Software. Disponível em: <<http://www.devmedia.com.br/revista-engenharia-de-software-magazine>>. Acesso em 12 nov. 2013.

VONSTA, A. **Engenharia de programas.** Rio de Janeiro: LTC, 1983.

WIENNER, R.; SINCOVEC, R. **Software engineering with Modula 2 and ADA.** New York: Wiley, 1984.

WIKIPEDIA (2007a). **ISO 9000.** Disponível em: <http://pt.wikipedia.org/wiki/ISO_9000>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007b). **Melhoria de processo de software brasileiro.** Disponível em: <<http://pt.wikipedia.org/wiki/MPS.BR>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007c). **CMMI.** Disponível em: <<http://pt.wikipedia.org/wiki/Cmmi>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007d). **Engenharia de Software.** Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software>. Acesso em: 1 fev. 2007.

Anotações





Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

www.cruzeirodosulvirtual.com.br
Campus Liberdade
Rua Galvão Bueno, 868
CEP 01506-000
São Paulo SP Brasil
Tel: (55 11) 3385-3000



Universidade
Cruzeiro do Sul



UNICID
Universidade
Cidade de S. Paulo



UNIFRAN
Universidade
de Franca



UDF
Centro
Universitário



Módulo
Centro
Universitário

Engenharia de *Software*



Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

Material Teórico



Planejamento e gerenciamento de projetos

Responsável pelo Conteúdo:

Prof.^a Dr.^a Ana Paula do Carmo Marchetti Ferraz

Revisão Textual:

Prof.^a Me. Luciene Oliveira da Costa Santos



- Introdução
- Acompanhamento de Processo



Objetivo de APRENDIZADO

Nesta unidade apresentaremos a importância do planejamento e do gerenciamento para que um projeto de software seja bem-sucedido. Você conhecerá tarefas e o esforço necessário para a criação de um plano realístico e administrável, além da importância das atividades de levantamento e gerenciamento de requisitos.

Você também compreenderá quais atividades devem ser executadas para a melhor análise das necessidades do cliente. Além disso, conhecerá a problemática da comunicação entre as pessoas envolvidas nessas atividades.



Atenção

Para um bom aproveitamento da Disciplina, leia o material teórico atentamente antes de realizar as atividades. É importante também respeitar os prazos estabelecidos no cronograma.

Contextualização

Durante as duas unidades anteriores, você percebeu que o desenvolvimento de *software* envolve inúmeras tarefas. Essas tarefas precisam ser executadas para haver a produção de um *software* que atenda às necessidades do cliente.

À medida que se expandiram as aplicações para *software*, ele passou a ser utilizado nas mais variadas áreas. Um dos desafios da Engenharia de *Software* é aumentar a produtividade dos processos de desenvolvimento para que a indústria de *software* consiga suprir a crescente demanda por novos *softwares*. Porém, além de focar na velocidade da criação dos processos de *software*, é importante que os produtos entregues tenham a qualidade esperada.

Para que um projeto seja bem-sucedido, é importante que ele seja planejado e gerenciado. Isso vale para todos os tipos de projetos, inclusive para os de *software*.

Nesta unidade 3, apresentaremos alguns conceitos relacionados ao planejamento e gerenciamento de projetos.

Não existe projeto bem-sucedido, sem planejamento e gerenciamento e não existe planejamento sem informações. Sugerimos que assista aos seguintes vídeos, sugeridos para entender um pouco mais sobre a importância do planejamento e do gerenciamento e seu inter-relacionamento com diversas áreas do conhecimento:



Explore

- http://www.youtube.com/watch?v=_xustQAKn6I
- <http://www.youtube.com/watch?v=fVwk9WUNYuU>
- <http://www.youtube.com/watch?v=z9hqZPgktO8>

Nesta unidade, especificamente sugerimos pesquisas adicionais sobre métodos de estimativa de tempo para execução de determinadas atividades. Dentre eles, podemos citar o Método do Caminho Crítico (COM – *Critical Path Method*), que é bastante utilizado. No entanto, devido à carga horária de nossa disciplina, não teremos como abordá-lo.

Outro assunto que sugerimos pesquisas adicionais são *softwares* que possuem funcionalidades relacionadas à gestão de projetos. Alguns deles são *Excel* da Microsoft, *NetOffice*, *MS-Project* da Microsoft, *Planner*, *DotProject*, *GranttProject* e, se considerar necessário, faça o download (alguns *softwares* são free). É importante que você realize a pesquisa para conhecer a funcionalidade de ferramentas como essas e como se relacionam na gestão de projetos de *software*, principalmente na relação das ferramentas que utilizam o “Diagrama de Gantt” ou “Gráfico de barras de atividades” como recursos acoplados ou independentes e verifique as principais características.

Introdução



Sempre quando começamos qualquer coisa na vida, ou melhor, quando pensamos em executar qualquer atividade, seja uma viagem, arrumar o guarda-roupa, levar o carro para lavar, sair de casa etc., a primeira coisa a fazer é **Planejar** e, posteriormente, **Gerenciar** e avaliar se o que foi planejado foi executado dentro do esperado.

Construir um software não é diferente.

Já estamos familiarizados com os modelos de desenvolvimento de software e optar por cada um deles é considerar características específicas de funcionalidades e dos requisitos e escolher a forma como planejar o desenvolvimento – que inclui custo, tempo, investimento, pessoas etc.

Nesta unidade, entenderemos um pouco desse processo e conheceremos algumas técnicas que nos ajudarão.

Não importa o tamanho ou a funcionalidade do seu software – atividades de análise, planejamento e gerência nunca deverão ser ignoradas ou menosprezadas. Delas pode depender o sucesso do seu projeto.

O tempo é o bem mais valioso que está disponível a um Engenheiro de Software.

Se houver tempo disponível, um software pode ser adequadamente analisado, uma solução pode ser comprehensivamente planejada, o código-fonte cuidadosamente implementado e testado. O problema é que **nunca há tempo suficiente.**(Autor desconhecido)

Acompanhamento do Processo



A atividade de gerenciamento e planejamento de um sistema pode ser vista como uma *atividade guarda-chuva*, pois abrange todo o **processo** de desenvolvimento. Ela possibilita **compreender** o escopo do trabalho, riscos, recursos exigidos, tarefas a executar, programação (cronograma) a ser seguido, esforço despendido, dentre outras características.

Segundo Pfleeger (2004, p. 63), “[...] **um software somente é útil se realizar uma função desejada ou fornecer um serviço necessário**”. Por isso um projeto geralmente começa quando um cliente faz um pedido de um sistema, ou quando, a partir da percepção visionária de alguém, começa a ser idealizado, pensado, projetado.

Imagine que alguém lhe solicite a construção de um software para efetuar a matrícula de alunos de uma escola que vai do Maternal ao Ensino Médio e que também os pais, por meio da web, possam ter acesso a algumas informações específicas dos seus filhos (matrículas, boletins, boletos etc.).

Você provavelmente, ao final da reunião sobre essa proposta, teria que responder a algumas questões para o cliente, dentre elas:

- » Você entendeu o que eu quero?
- » Você pode projetar esse sistema?
- » Quanto tempo você levaria para projetá-lo?
- » Quanto vai custar o desenvolvimento disso?

Responder às duas últimas perguntas pode ser a coisa mais complicada do sistema e requer um cronograma de projeto bem planejado. Muitas vezes, devido à ausência de respostas claras e objetivas sobre esses dois itens é que acontecem falhas como, por exemplo, sistemas não entregues no prazo previsto porque sua estimativa de tempo de desenvolvimento e esforço foi subdimensionada.

Um cronograma de projeto descreve o ciclo de desenvolvimento de software para um projeto específico, enumerando as etapas ou estágios e dividindo cada um deles em tarefas ou atividades a serem realizadas. O cronograma também retrata as interações entre essas atividades e estima o tempo necessário para a realização de cada tarefa ou atividade. Portanto, o cronograma é uma linha de tempo que mostra quando as atividades começarão e terminarão, e quando estarão prontos os produtos relacionados ao desenvolvimento. (PFLEINGER, 2004, p. 66)

Se até as tarefas mais simples – rotina diária – precisam de planejamento, podemos dizer que tudo nasce de um projeto (formal ou não) e, a partir da definição do que esperamos como resultados, são previstos tempo e valores.

Segundo Sommerville (2011), os critérios do sucesso de um sistema alteram-se de um para outro. Entretanto, questões como fornecer o software ao cliente no prazo estabelecido, manter os custos gerais dentro do orçamento, entregar o produto que atenda à expectativa do cliente e manter a equipe de desenvolvimento satisfeita, bem e feliz são pontos comuns do reconhecimento positivo de qualquer software.

Se você analisar com cuidado o parágrafo anterior, perceberá que esses itens não são exclusivos da Engenharia de Software (ES), mas de todo projeto de Engenharia. Entretanto, o que diferencia o gerenciamento de um produto de software de outros é o fato de que o produto de software é intangível. Os grandes projetos de software geralmente são únicos, pois todos eles, de uma forma ou de outra, possuirão especificações diferentes para clientes diferentes. Mesmo com questões relacionadas a padrões existentes na área de ES, os produtos de software são mudados de uma organização para outra.

Sendo assim, todo produto precisa levar em conta as fases de análise dos recursos e objetivos e, a partir deles, considerar a definição de atividades que serão executadas e de marcos.

Mas o que é realmente **atividade de marco** na ES?

Você já ouviu a expressão “isso foi um marco na minha vida”? Consegue perceber, quando alguém fala isso, o que significa?

Para entendermos, temos que definir primeiro o que é **atividade**. Ela é uma parte do projeto que acontece ao longo de determinado período, enquanto **marco** é a finalização de uma atividade – um momento específico no tempo que pode ou não marcar o início de outra atividade. Por exemplo, temos a atividade de entrevista do cliente; após esta atividade, podemos finalizá-la (marco) e iniciamos o desenvolvimento.

Ao pensarmos num projeto como um todo, podemos (e devemos) definir que seu desenvolvimento seja o resultado de uma sucessão de fases, todas com etapas e com uma ou várias atividades inseridas, como mostra a Figura 1:

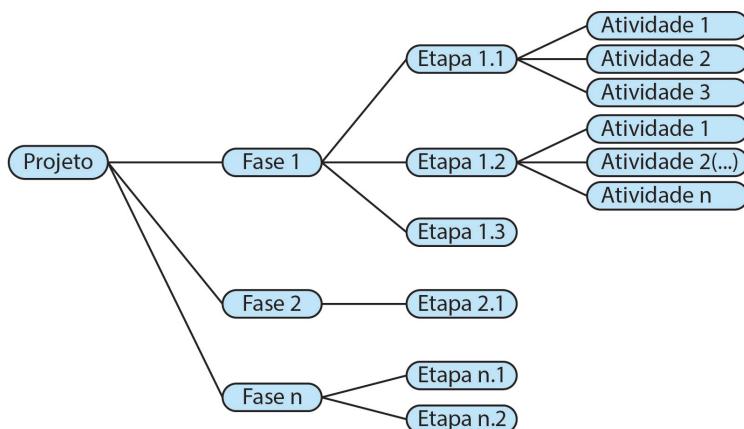


Figura 1 – Esquema de um projeto dividido em fases e etapas.

Para ilustrarmos esta Figura, pense no desenvolvimento do projeto solicitado no início deste Material teórico – um sistema escolar. Ele deverá ser dividido em grandes fases como, por exemplo, análise, codificação e teste.

Na análise, podemos definir algumas etapas – falar com o cliente, analisar os dados e requisitos coletados, modelar o sistema, dentre outras.

Se pensarmos na etapa falar com o cliente, podemos definir algumas atividades como: marcar a reunião, definir as perguntas que nortearão ações, tomar nota das solicitações dos clientes etc.

Essas atividades e os marcos gerados a partir delas funcionarão como referências para acompanhar o desenvolvimento ou a manutenção do projeto.

Em cada atividade, quatro parâmetros devem ser definidos: o precursor, a duração, a data prevista e o encerramento. Podemos dizer que um **precursor** é um evento ou um conjunto de eventos que deve ocorrer antes da atividade começar. Ele descreve uma série de condições necessárias para que a atividade tenha início. A **duração** é o tempo necessário para completar a atividade. A **data prevista** é aquela em que a atividade deve ser concluída, em geral determinadas por prazos contratuais. O **encerramento** é um marco ou um requisito que foi concluído.

A partir disso, é possível começarmos a definir uma **estrutura de divisão do trabalho**. É nesse momento em que são montadas as equipes de análise, de programação, do setor financeiro, as equipes de teste e até de manutenção **para determinado projeto**.

Cada projeto possuirá equipes diferentes, de acordo com as especificidades requeridas e competências de cada pessoa que trabalha na empresa.

Essas equipes podem ser alocadas para participar durante todo o projeto ou apenas em algumas fases, por exemplo, a equipe de análise pode participar apenas da fase inicial do projeto na qual são definidos os requisitos e modelado o sistema, depois, ao final dessas funções, talvez possa se desligar do projeto “A” e ser realocada para o projeto “B”.

Os itens precursor, data, divisão do trabalho etc., ao serem planejados, darão origem à documentação do projeto que deverá servir de “mapa” para o gerenciamento do desenvolvimento.



É importante conhecermos as técnicas de determinação de estimativa de tempo para execução de determinadas atividades. Dentre elas, podemos citar Método do Caminho Crítico (COM – *Critical Path Method*) que é bastante usado. Entretanto, devido ao conteúdo delimitado e nosso prazo, não teremos como abordá-lo.

Gerenciar significa cuidar de:

- » **Pessoa:** Quem participa do Processo;
- » **Produto:** O que será desenvolvido;
- » **Processo:** Como será feito;
- » **Projeto:** Das fases, etapas e atividades.



Pense

O gerente que esquece que o trabalho da ES está intensamente relacionado com as pessoas nunca terá sucesso na gerência de projetos.



Atenção

O gerente de sistemas ou engenheiro de software que não incentivar uma compreensiva comunicação com o cliente, logo no início do projeto, corre o risco de equívocos na tentativa de solucionar problemas.

Ferramenta para Acompanhar o Processo

Existem inúmeras ferramentas que podemos utilizar para acompanhar o projeto, algumas mais simples como *Excel* e outras mais complexas, como o *MS-Project*, da *Microsoft*.

Para verificar qual delas pode ser utilizada, uma boa dica é analisar a estrutura do trabalho, dividir tarefas e trabalhos e, daí, optar por uma ferramenta ou uma combinação de diferentes ferramentas.

Para ilustrarmos essa ideia de ferramenta, vamos utilizar um exemplo de Pfleeger (2004, p. 71-72).

Vamos considerar a seguinte estrutura de divisão de trabalho:

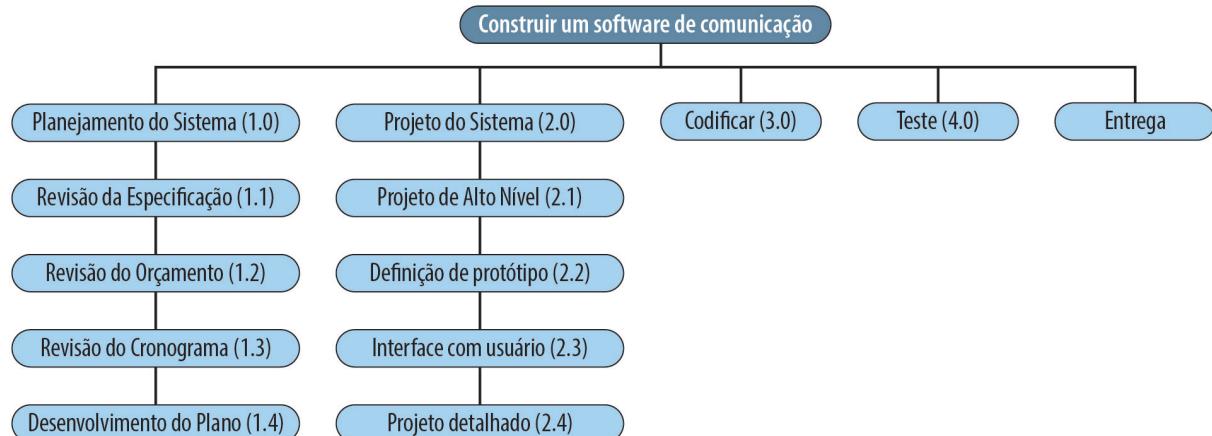


Figura 2 – Exemplo de uma divisão do trabalho, adaptado de Pfleeger (2004).

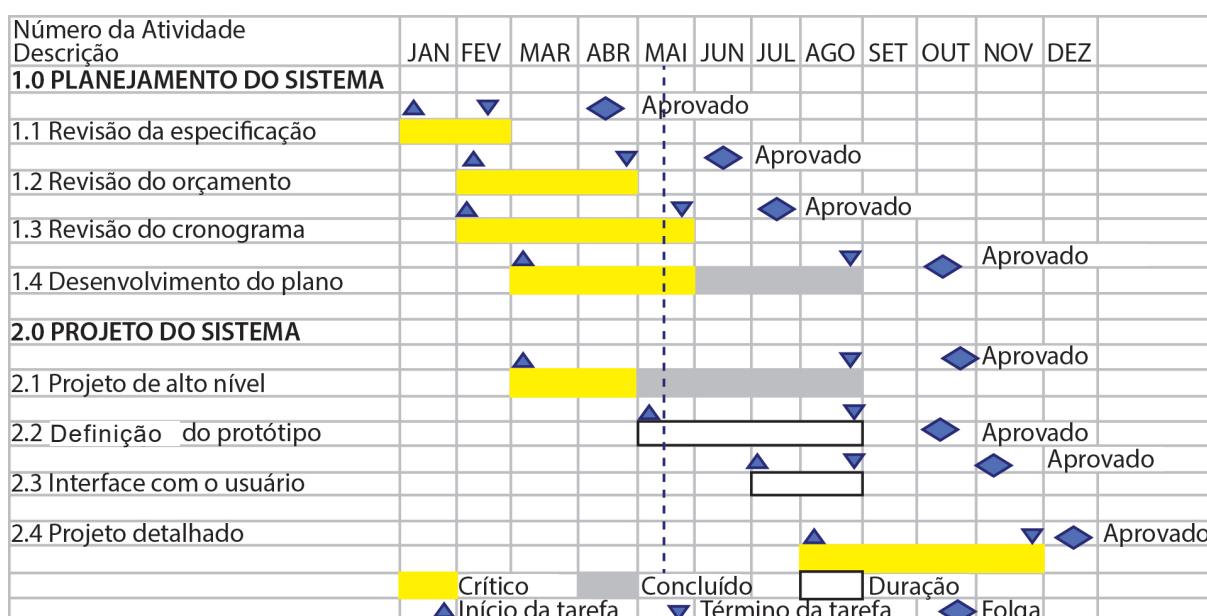


Figura 3 – Diagrama de Gantt para o exemplo da Figura 2; adaptada de Pfleeger (2004)

A partir da divisão do trabalho, podemos discutir, com mais precisão, tempo e gerenciamento do desenvolvimento do projeto.

Algumas ferramentas conduzem o gerente por meio de um **Diagrama de Barra de Atividades**. Nele, as atividades são mostradas em paralelo, com grau de conclusão indicado por uma cor ou um ícone. Esse diagrama ajuda o gerente a perceber quais atividades podem ser executadas simultaneamente e quais são críticas para o cumprimento das etapas.

A utilização correta de uma ferramenta que produza um diagrama de barra de atividade pode ser fundamental para o cumprimento de etapas, fases e atividades dentro do prazo e custo estipulados.

Na Figura 2 você pode ver o diagrama de barras de atividades, a partir da divisão do trabalho.

Fazer uma pesquisa sobre as ferramentas que utilizam o diagrama de barra de atividades e verificar as características de seus recursos.

Estimativas

Estimar um cronograma é uma das tarefas mais difíceis de realizar na etapa de planejamento do software.

São vários os aspectos a serem considerados: a disponibilidade de recursos no momento da execução de uma dada tarefa, as interdependências das diferentes tarefas, a ocorrência de possíveis estrangulamentos do processo de desenvolvimento e as operações necessárias para agilizar o processo, identificação das principais atividades, revisões e indicadores de progresso do processo de desenvolvimento etc.

Uma das formas de se garantir que os cronogramas sejam cumpridos é pensar no projeto como um todo, as interdependências que existem e como o documento pode trabalhar com o Diagrama de Barra de Atividades, pois a partir dele é possível verificar quais atividades são dependentes, simultâneas e o tempo total para a execução.

O cronograma deve apresentar as atividades importantes do desenvolvimento e os indicadores de progresso, ou seja, deve prever os marcos do projeto. É importante que os indicadores de progresso sejam percebidos por resultados concretos que devem estar documentados.

A disponibilidade de recursos deve ser representada ao longo do cronograma. O impacto da indisponibilidade dos recursos no momento em que eles são necessários deve também ser representado, se possível, no cronograma.

Fixar prazos do projeto não é tarefa fácil, entretanto, quando realizadas, algumas questões devem ser respondidas:

- » Como relacionar o tempo cronológico com o esforço humano? Quantas pessoas serão necessárias para realização do projeto?
- » Que tarefas e que grau de paralelismo podem ser obtidos?
- » Como podemos medir o progresso do processo de desenvolvimento, ou seja, quais os indicadores de progresso que devemos monitorar?
- » Como o esforço pode ser distribuído ao longo do processo?
- » Que métodos estão disponíveis para a determinação de prazos?
- » Como representar fisicamente o cronograma e como acompanhar o progresso a partir do início do projeto?

Outro ponto crucial no planejamento e gerenciamento de um projeto é a **Estimativa de Custo**.

Um custo mal dimensionado acarreta problemas: se superestimado, pode provocar uma rejeição do cliente na hora de fechar o negócio; uma estimativa de preço abaixo do real pode levar a equipe a trabalhar muito além do tempo necessário sem o devido retorno financeiro.

Uma boa estimativa de custo faz com que o projeto se desenvolva de forma adequada, auxilia na definição das etapas, da quantidade de pessoas no desenvolvimento do projeto, além da competência e do perfil necessário de cada integrante da equipe antecipadamente.

Existem inúmeras razões para estimativas imprecisas e Pfeleger (2004, p. 81) define algumas:

- » Frequentes solicitações de mudança pelos usuários.
- » Tarefas negligenciadas.
- » Falta de entendimento do usuário sobre suas próprias exigências.
- » Análise insuficiente no desenvolvimento de uma estimativa.
- » Falta de coordenação do desenvolvimento do sistema, dos serviços técnicos, das operações, do gerenciamento de dados e de outras funções durante o desenvolvimento.
- » Complexidade do sistema proposto.
- » Necessidade de integração com os sistemas existentes.
- » Complexidade dos programas no sistema.
- » Capacidade dos membros da equipe de projeto.
- » Experiência da equipe do projeto com a aplicação.
- » Quantidade de padrões de programas e documentação.
- » Disponibilidade de ferramentas, tais como geradores de aplicação.
- » Experiência da equipe com o hardware.

Claro que essas não são as únicas razões que podem provocar uma estimativa mal feita. Porém, ao observarmos esses pontos, estaremos conduzindo o projeto em direção a uma estimativa de custo adequada.

Quando falamos em custo ou orçamento de projeto, estamos falando de vários tipos: instalações, pessoal, métodos e ferramentas.

É importante pensarmos em todos os custos diretos e indiretos na hora de dimensionarmos o valor. Custo de instalações, por exemplo, englobam hardware, espaço, mobília, telefones, modem, sistemas de aquecimento ou ar-condicionado, cabos, discos, papel, caneta, copiadores e todos os outros itens que fazem parte do ambiente no qual será desenvolvido o projeto.

É preciso pensarmos também em custos ocultos, que não são visíveis aos gerentes e desenvolvedores como, por exemplo, espaço físico para trabalhar, silêncio do ambiente etc.



Pense

Qual tipo de custo oculto você poderia citar?

Outro tipo de estimativa que deve ser considerada é a **Estimativa de Esforço**.

Um dos itens importantes no desenvolvimento de um projeto é o esforço a ser despendido.

Essa estimativa está relacionada ao número de pessoas/dia para desenvolver o projeto dentro do cronograma estipulado.

O esforço é o componente de custo de maior grau de incerteza, pois capacitação, experiência, interesse e treinamento podem influir no tempo de execução de uma tarefa (PLEEDGER, 2004).

Estimativas de custo, de cronograma e de esforços devem ser feitas o quanto antes, durante o ciclo de vida do projeto, pois elas afetam a distribuição de recursos e a viabilidade do projeto.

Análise de Riscos

A análise dos riscos é uma das atividades essenciais para o bom encaminhamento de um projeto de software e, consequentemente, da sua gerência.

Essa atividade está baseada na realização de quatro tarefas, conduzidas de forma sequencial:

- a) Identificação dos riscos.
- b) Projeção dos riscos.
- c) Avaliação dos riscos.
- d) Administração e monitoramento dos riscos.

Identificação dos Riscos

Na primeira etapa, o objetivo é que sejam levantados, pelos responsáveis gerentes e analistas, os eventuais riscos aos quais o projeto poderá ser submetido.

É importante salientar que é difícil identificar todos os riscos, mas é possível, a partir da experiência existente, identificar uma quantidade significativa e trabalhar num projeto mais coeso.

Podemos dividi-los em 3 categorias, de acordo com a sua natureza:

- **Riscos de projeto:** estão associados a problemas relacionados ao próprio processo de desenvolvimento (orçamento, cronograma, pessoal).
- **Riscos técnicos:** aqueles relacionados ao projeto nas fases de implementação, manutenção, interfaces, plataformas de implementação.
- **Riscos de produto:** estão mais relacionados aos problemas que surgirão a partir e para a inclusão do software como produto no mercado como, por exemplo, oferecer um produto que ninguém está interessado, produto ultrapassado ou inadequado à venda.

A categorização dos riscos, apesar de interessante, não garante a obtenção de resultados satisfatórios, uma vez que nem todos os riscos podem ser identificados facilmente. Uma boa técnica para conduzir a identificação dos riscos de forma sistemática é o estabelecimento de um conjunto de questões (checklist) relacionado a algum fator de risco.

Projeção dos Riscos

A projeção ou estimativa de riscos permite definir basicamente duas questões:

Qual a probabilidade de que o risco ocorra durante o projeto? Quais as consequências dos problemas associados ao risco no caso de ocorrência do mesmo?

As respostas a essas duas questões podem ser obtidas basicamente a partir de quatro atividades:

- » Estabelecimento de uma escala que reflita a probabilidade estimada de ocorrência de um risco.
- » Estabelecimento das consequências do risco.
- » Estimativa do impacto do risco sobre o projeto e sobre o software (produtividade e qualidade).
- » Anotação da precisão global da projeção de riscos.

A natureza do risco permite indicar os problemas prováveis, como casos de risco técnico – má definição de interface entre o software e o hardware –, que conduzirá certamente a problemas de teste e integração.

Avaliação dos Riscos

O objetivo da atividade de avaliação dos riscos é processar as informações sobre o fator de risco, o impacto do risco e a probabilidade de ocorrência.

Nesta avaliação, serão verificadas todas as informações coletadas na fase de projeção de riscos, a fim de priorizá-las e definir quais as melhores formas de controlá-los ou evitar sua ocorrência.

Para tornar a avaliação eficiente, deve ser definido um **nível de risco referente**. Exemplos de níveis referentes típicos são: custo, prazo e desempenho. Isso significa analisar se poderá haver excesso de custo, ultrapassagem de prazo de entrega e degradação do desempenho, ou qualquer combinação dos três em quaisquer níveis (profundidade).

Dessa forma, se os problemas observados ou a combinação dos riscos provocarem a superação de alguns dos níveis definidos, o projeto poderá ser suspenso. Normalmente, é possível estabelecer um limite, denominado ponto referente (*breakpoint*) onde tanto a decisão de continuar o projeto ou de abandoná-lo pode ser avaliada.

Administração e Monitoração dos Riscos

Uma vez avaliados os riscos de desenvolvimento, é importante que medidas sejam tomadas para evitar que eles realmente ocorram, ou planejar ações para realizá-las caso não tenhamos como evitá-los. Esse é o objetivo da tarefa de administração e monitoração dos riscos. Para isso, as informações mais importantes são aquelas obtidas na tarefa anterior, avaliação dos riscos.

Por exemplo, vamos imaginar que, numa determinada empresa, os fatores de riscos são dimensionados de zero a um. Especificamente no tocante à função de desenvolvedores, existe alta rotatividade de pessoal. Com base em dados de projetos passados, obtém-se que a probabilidade de ocorrência desse risco é de 0,70 (muito elevada) e que a sua ocorrência pode aumentar o prazo do projeto em 15% e o seu custo global em 12% (PRESSMAN, 2006, p. 234).

Sendo assim, podemos propor as seguintes ações para administrar os fatores de riscos:

- » Reuniões com os membros da equipe para determinar as causas mais importantes da rotatividade de pessoal.
- » Perceber questões relacionadas às condições de trabalho, salários, oportunidade de mercado de trabalho.
- » Criar planos de ações/providências para eliminar ou reduzir as causas “controláveis” desse processo antes de iniciar o projeto.
- » Prever, desde o início, quais trocas de pessoal poderão ocorrer. Prever a possibilidade de substituição dessas pessoas quando elas deixarem a equipe, criar equipe suplente a partir das competências exigidas para a função, principalmente para as funções mais críticas.
- » Organizar equipes de projeto de forma que as informações sobre cada atividade sejam compartilhadas, gerando um maior senso de equipe, principalmente no tocante a revisões dos trabalhos executados para que todos tenham ciência do processo.
- » Definir padrões para que a documentação seja construída de forma adequada.

É importante observar que a implementação dessas ações pode afetar também os prazos e o custo do projeto, por isso é fundamental avaliar o quanto tais ações poderão influenciar no tempo e valor final.

Todas essas ações não são e nem devem ser feitas empiricamente. Existem softwares que dimensionam e montam equipes (podem ser baseadas em experiências prévias, disponibilidade, habilidade técnica etc.), esforços e riscos. Calculam, matematicamente, esses valores e fornecem apoio às decisões.

O resultado do trabalho realizado, nesse momento, deverá ser registrado num documento denominado **Plano de Administração e Monitoração de Riscos**, o qual será utilizado posteriormente pelo gerente de projetos para a definição do Plano de Projeto, que é gerado ao final da etapa de planejamento que ajudará no gerenciamento.

Antes de tratarmos de Plano de Projeto, devemos conhecer o documento que engloba todos os assuntos vistos até o momento: **Plano de Software**.

Plano de software

Ao final da etapa de estimativa, de discussão sobre riscos e equipe está prevista a criação de um documento chamado de **Plano de Software (PS)**, o qual deverá ser revisto para servir de referência às etapas posteriores.

É importante mencionar que ele servirá como plano de ação para as próximas etapas e poderá ser atualizado sempre que necessário.

O PS apresentará as informações iniciais de custo e cronograma que vão direcionar todo o processo de desenvolvimento do produto de software.

PS não deve ser um documento longo, pois deverá ser enviado a várias pessoas envolvidas no projeto.

Todo PS deve, dentre outras, conter as seguintes informações:

- O contexto e a descrição dos recursos necessários para a efetiva gestão do projeto.
- A definição de custos e cronograma que serão acompanhados na gestão do projeto.
- A visão global do processo de desenvolvimento do software.

O Plano de Software não deve ser um documento extenso e complexo. O seu objetivo é auxiliar a análise de viabilidade dos esforços de desenvolvimento. Esse documento está associado aos conceitos de “o que”, “quanto” e “quão longo” ligados ao desenvolvimento. As etapas mais à frente vão estar associadas ao conceito de “como” (PRESSMAN, 1995).

Plano do projeto

Para comunicarmos a análise e o gerenciamento de riscos, a estimativa de custo, tempo e esforço, assim como a equipe e as diretrizes de teste, manutenção, garantia de qualidade do sistema etc., é necessário criarmos um **Plano de Projeto (PP)**.

O plano descreve as necessidades do cliente, assim como está sendo idealizada a execução dessas atividades.

Quando o sistema ainda está em análise, a documentação é chamada de PLANO; ao final do projeto, esse documento deverá ser atualizado e se tornará a documentação oficial do projeto que servirá como referência para novas atualizações, manual de usuário etc.

Um bom plano de projeto inclui os seguintes itens:

1. **Escopo do projeto:** descrição geral do projeto.
2. **Plano de Software:** informações que servirão de documentação para as etapas posteriores.
3. **Descrição técnica, ferramentas e procedimentos propostos para o projeto:** definição de requisitos e modelagem.
4. **Plano de garantia da qualidade:** definição das estratégias para garantia da qualidade do software.
5. **Plano de gerência de configurações:** definição das estratégias para gerenciar as alterações de software.
6. **Plano de documentação.**
7. **Plano de gerência de dados:** plano de gerenciamento de dados e integração de Bancos de Dados (BD).
8. **Plano de gerência de recursos:** plano de gerenciamento de recursos de desenvolvimento (pessoal, técnico etc.).
9. **Plano de testes.**
10. **Plano de treinamento.**
11. **Plano de segurança.**
12. **Plano de gerência de riscos.**
13. **Plano de manutenção.**



Explore

Plano de projeto. Disponível em: <<http://www.devmmedia.com.br/artigo-engenharia-de-software-3-plano-de-projeto/9527>>. Acesso em: 15 nov. de 2013.

MASIEIRO, P. Plano de Projeto. Disponível em: <http://www.univasf.edu.br/~ricardo.aramos/disciplinas/ESI2009_2/PlanejamentoProjeto_Masiero.pdf>. Acesso em: 15 nov. de 2013.

A partir de agora, podemos abordar os assuntos relacionados à definição de requisitos que direcionará todo o desenvolvimento.

Definir requisitos significa definir o que é necessário para o projeto atingir seus objetivos e por isso é uma fase de crucial importância na Engenharia de Software e dará origem à documentação de requisitos que servirá de Norte para os desenvolvedores.

Material Complementar



Explore

O objetivo do material complementar é lhe ajudar a entender, sob uma ótica diferente daquela da professora conteudista, assuntos abordados nas unidades teóricas.

É fundamental a leitura deste material para o melhor entendimento sobre o assunto.

Como nesta unidade abordamos os conceitos gerais da Engenharia de Software, nossa sugestão de material complementar é o capítulo 23, intitulado **Planejamento de projeto**, no seguinte livro:

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011. (p. 431-442).

Referências

Bibliografia Fundamental

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011.

Bibliografia Básica

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação**. 4. ed. Rio de Janeiro: LTC, 1999.

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação Gerenciais**. Administrando a empresa digital. 5. ed. São Paulo: Pearson Education do Brasil, 2006.

PFLEEGER, S. L. **Engenharia de Software**: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 2006.

SOMMERVILLE, I. **Engenharia de Software**. 6. ed. São Paulo: Pearson Addison Wesley, 2005.

Bibliografia Complementar

ALCADE, E.; GARCIA, M.; PENUELAS, S. **Informática Básica**. São Paulo: Makron Books, 1991.

FAIRLEY, R. E. **Software engineering concepts**. New York: McGraw-Hill, 1987.

IEEE. **Software Engineering Standards**. 2013. Disponível em: <http://www.ieee.org/portal/innovate/products/standard/ieee_soft_eng.html>. Acesso em: 10 dez. 2013.

LUKOSEVICIUS, A. P.; CAMPOS FILHO, A. N.; COSTA, H. G. **Maturidade em gerenciamento de projetos e desempenho dos projetos**. Disponível em: <www.producao.uff.br/conteudo/rpep/.../RelPesq_V7_2007_07.doc>. Acesso em: 12 nov. 2013.

MAFFEO, B. **Engenharia de Software e especialização de sistemas**. Rio de Janeiro: Campus, 1992.

MICHAELIS. **Moderno dicionário da língua portuguesa**. São Paulo: Cia. Melhoramentos, 1998.

PARREIRA JÚNIOR, W. M. **Apostila de Engenharia de Software**. Disponível em: <http://www.waltenomartins.com.br/ap_es_v1.pdf>. Acesso em: 13 nov. 2013.

PAULA FILHO, W. P. **Engenharia de Software**: fundamentos, métodos e padrões. 2. ed. Rio de Janeiro: LTC, 2001.

Revista Engenharia de Software. Disponível em: <<http://www.devmedia.com.br/revista-engenharia-de-software-magazine>>. Acesso em: 12 nov. 2013.

VONSTA., A. **Engenharia de Programas.** Rio de Janeiro: LTC, 1983.

WIENNER, R.; SINCOVEC, R. **Software engineering with Modula 2 and ADA.** New York: Wiley, 1984.

WIKIPEDIA (2007a). **ISO 9000.** Disponível em: <http://pt.wikipedia.org/wiki/ISO_9000>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007b). **Melhoria de processo de software brasileiro.** Disponível em: <<http://pt.wikipedia.org/wiki/MPS.BR>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007c). **CMMI.** Disponível em: <<http://pt.wikipedia.org/wiki/Cmmi>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007d). **Engenharia de Software.** Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software>. Acesso em: 1 fev. 2007.

Anotações



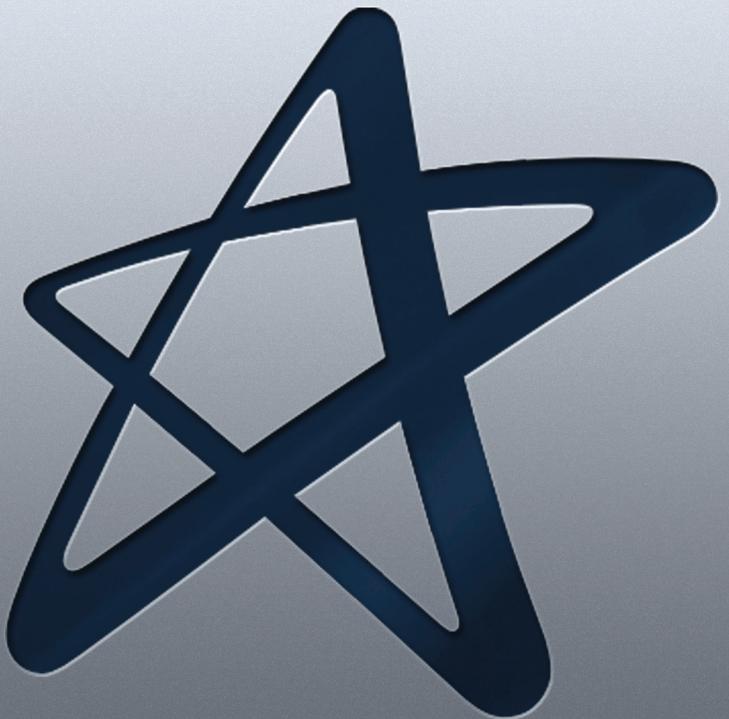


Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

www.cruzeirodosulvirtual.com.br
Campus Liberdade
Rua Galvão Bueno, 868
CEP 01506-000
São Paulo SP Brasil
Tel: (55 11) 3385-3000



Engenharia de *Software*



Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

Material Teórico



Engenharia de requisitos e gerenciamento de projetos

Responsável pelo Conteúdo:

Prof.^a Dr.^a Ana Paula do Carmo Marchetti Ferraz

Revisão Textual:

Prof.^a Me. Luciene Oliveira da Costa Santos

UNIDADE

Engenharia de requisitos e gerenciamento de projetos



- Introdução
- Requisitos
- Tipos de requisitos
- O processo de gerenciamento de configuração de software



Objetivo de APRENDIZADO

Apresentaremos algumas questões relacionadas ao planejamento e gerenciamento de projetos e à engenharia de requisitos.

Esses assuntos nos ajudarão a entender um pouco melhor os conceitos de planejamento de desenvolvimento de produto de software, por meio dos requisitos e recursos existentes para nos ajudar na gestão do desenvolvimento destes.

Organize-se de forma a não deixar para o último dia a realização das atividades (AS e AP), pois podem ocorrer imprevistos. Encerrada a unidade, encerra-se a possibilidade de obter a nota relativa a cada atividade.

Para ampliar seu conhecimento, bem como aprofundar os assuntos discutidos, pesquise, leia e consulte os livros indicados nas Referências e/ou na Bibliografia.

As Referências estão indicadas ao final dos textos de conteúdo de cada unidade.

A Bibliografia Fundamental para esta disciplina é: SOMMERVILLE, Ian. **Engenharia de Software**. 9.ed. São Paulo: Pearson, 2011. A Bibliografia Complementar está indicada em item específico, em cada unidade.

Contextualização

Na unidade anterior, você teve a oportunidade de aprender sobre gerência e planejamento de projeto e conhecer um pouco mais sobre a importância dessas atividades. Até na nossa vida diária planejar nossas ações e gerenciá-las são fundamentais para que tenhamos bons resultados.

Nesta unidade, você terá a oportunidade de aprender os conceitos e as definições de requisitos de software, assim como a importância do gerenciamento das configurações de um software.

Entender o que são requisitos de software é fundamental para o sucesso de um projeto de software. Outro item fundamental neste projeto são as ferramentas que são utilizadas para gerenciar as configurações de software.

Definir requisitos e gerenciar configuração não são tarefas de simples execução, mas são fundamentais para manter a qualidade do software durante seu desenvolvimento.



Explore

Sugerimos que assista aos vídeos disponíveis em: <http://www.youtube.com/watch?v=UZfpUdYLsao> e <http://www.youtube.com/watch?v=cQiQN4NsBZ4> para entender um pouco melhor sobre a importância da definição clara dos requisitos e o gerenciamento de configurações.

Se você tiver dúvidas durante o estudo desta unidade, interaja com seus colegas, discuta sobre o tema, peça auxílio ao seu tutor.

Tenha disciplina em seus estudos, programe-se, tenha metas a serem alcançadas semanalmente e procure atingi-las.

Seu aprendizado só depende de você. Dedique-se, interaja com seus colegas e não acumule dúvidas.

Adquira o hábito da pesquisa, visite sites, pesquise em livros e periódicos. Dê pequenas pausas na leitura e pesquise sobre os conceitos abordados nas unidades.

Lembre-se de compartilhar suas descobertas com seus colegas de turma.

Introdução



Toda vez que falamos sobre analistas e projetistas chegamos a um impasse: saber até onde a atividade de um vai e quando começa a do outro.

O analista decide o que fazer e o projetista decide como fazer. Essas duas funções andam juntas e “o que” e o “como” fazem parte dos requisitos.

Segundo Sommerville (2011), os requisitos de um sistema são as descrições do que o sistema deve fazer, os serviços que oferecerá e seu funcionamento, ou seja, requisitos: é um conjunto de necessidades a serem atendidas, condição.

Entender o que são requisitos de software é um ponto fundamental para o sucesso de um projeto de software.

A Análise de Requisitos é uma tarefa que envolve, antes de tudo um trabalho de descoberta, refinamento, modelagem e especificação das necessidades e desejos relativos ao software que deverá ser desenvolvido.

Nessa tarefa, tanto o cliente como o desenvolvedor vão desempenhar um papel de grande importância, uma vez que caberá ao primeiro a formulação (de modo concreto) das necessidades em termos de funções e desempenho, enquanto o segundo atua como indagador, consultor e solucionador de problemas.

Essa etapa permite que o engenheiro de sistemas especifique as necessidades do software em termos de funções e de desempenho, estabeleça as interfaces com os demais elementos do sistema e especifique as restrições de projeto.

Ao engenheiro de software (ou analista), a análise de requisitos permite uma alocação mais precisa de recursos e a construção de modelos do processo, dos dados e dos aspectos comportamentais que serão tratados pelo sistema.

Ao projetista, essa etapa proporciona a obtenção de uma representação da informação e das funções, podendo ser traduzida em projeto procedural, arquitetônico e de dados.

Requisitos



Por meio dos requisitos é possível definir critérios de avaliação da qualidade do software a serem verificados uma vez que o software esteja concluído.

O termo *requisito* não é usado de forma consistente pela indústria de software. Em alguns casos, o requisito é apenas uma declaração abstrata em alto nível de um serviço que o sistema deve oferecer ou uma restrição a um sistema.

No outro extremo, uma definição detalhada e formal de uma função de sistema. Davis (1993) explica porque estas diferenças existem: Se uma empresa pretende fechar um contrato para um projeto de desenvolvimento de software de grande porte, deve definir as necessidades de forma abstrata o suficiente para que a solução para essas necessidades não seja predefinida. Os requisitos precisam ser descritos de modo que várias contratantes possam concorrer pelo contrato e oferecer diferentes maneiras de atender às necessidades da organização do cliente. Uma vez que o contrato tenha sido adjudicado, o contratante deve escrever para o cliente uma definição mais detalhada do sistema, para que este o entenda e possa validar o que o software fará. Ambos os documentos podem ser chamados de documentos de requisitos para o sistema (SOMMERVILLE, 2011, p. 57).

As atividades da análise de requisitos

Todo processo de análise relacionada nessa etapa define dois tipos de requisitos: Funcionais e Não Funcionais.



Informação

Segundo Pressman (1996):

Requisito: é a condição necessária para a obtenção de certo objetivo, ou para o preenchimento de certo objetivo.

Especificação: é a descrição minuciosa das características que um material, uma obra, ou um serviço deverá apresentar.

Portanto, Especificação é diferente de Requisito e as duas atividades juntas nos ajudam a melhorar a descrição dos objetivos do projeto.

Às vezes, na literatura podemos encontrar:

- Especificação de Requisitos
- Especificação de Projeto
- Os termos são sinônimos.



Trocando Ideias

No começo do projeto, o analista estuda os documentos de especificação do sistema e o plano do software, como forma de entender o sistema e revisar o escopo utilizado para definir as estimativas de projeto. Quando um cliente pede para construirmos um novo sistema, ele tem uma noção do que o sistema fará, se substituirá um antigo, ou se apenas criará novos recursos ao antigo. Não importa se é um sistema novo, ou adequação de um antigo, cada um tem um propósito, uma funcionalidade.

Para iniciarmos o processo de definição de requisitos, precisamos, antes de qualquer procedimento, conversar com nosso cliente, fazer perguntas, demonstrar sistemas similares, desenvolver protótipos. Em seguida, é necessário registrarmos essas informações em um documento ou base de dados.

Geralmente, os requisitos são escritos, num primeiro momento, numa linguagem que o cliente possa entender para aprovará-los; posteriormente, eles são reescritos em uma representação mais matemática ou visual para que os projetistas possam transformar os requisitos em um projeto de sistema. Esse processo de reescrever os requisitos numa linguagem diferenciada é o processo de modelagem.

Na seguinte Figura temos a descrição do processo de requisitos.

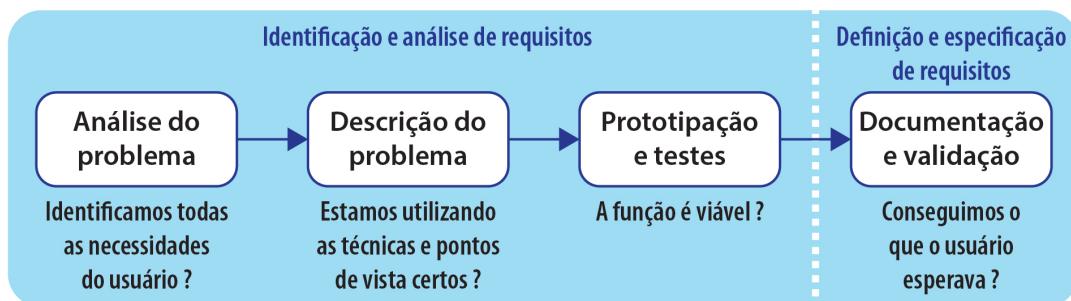


Figura 1 – Processo de definição de requisitos. Fonte: Pfleeger (2004, p. 112).

Existem inúmeras maneiras de identificação de requisitos de acordo com a experiência e preferência pessoal; entretanto, uma forma adequada é dividir o projeto em parte e definir quem teria as informações adequadas de forma a ajudar na definição dos requisitos para aquela fase do sistema.

Conversando, por meio de entrevista, é possível definir a amplitude do nosso sistema ou da parte que estamos responsáveis.

Durante a conversa, é necessário não deixar passar nenhuma dúvida. Ao final, durante o processo de documentação dos requisitos, eles devem ser categorizados em 3:

- 1 - Requisitos que devem ser totalmente satisfeitos.
- 2 - Requisitos que são altamente desejáveis, mas não são necessários.
- 3 - Requisitos que são possíveis, mas poderiam ser eliminados.

É importante salientarmos que nenhum requisito especifica “como o sistema deverá ser implementado”. O requisito informa o propósito do sistema.

Tipos de documentos de requisitos

Como o enfoque é no problema do nosso cliente, a identificação e a análise dos requisitos servem para propósitos distintos, porém relacionados.

A identificação dos requisitos nos habilita a escrever um documento denominado definição de requisitos. Esse documento deve ser escrito em linguagem natural de forma que o usuário possa lê-lo, entendê-lo e interpretá-lo.

Ele representa um consenso entre o que o cliente deseja e o que é possível e será realizado.

Posteriormente, outro documento é gerado para conter as especificações dos requisitos.

Esse documento redefine o anterior para uma linguagem mais técnica, apropriada para o desenvolvimento do projeto.

Em algumas vezes, um único documento é gerado e serve para os dois propósitos: comunicação com cliente e projetista.

Uma coisa importante a ser descrita é que deve haver uma correspondência direta entre cada requisito do documento de definição e de especificação de requisitos.

É nesse momento que se inicia o processo de Gerência de Configurações utilizadas durante o ciclo de vida e que veremos com mais detalhes na próxima unidade. Segundo Pfleeger (2004), Gerência de Configurações é um conjunto de procedimentos que controlam:

- Os requisitos que definem o que o sistema fará.
- Os módulos de projetos que serão gerados a partir dos requisitos.
- O código do programa que implementa o projeto.
- Os testes que verificam a funcionalidade do sistema.
- Os documentos que descrevem o sistema.

Para Pfleeger (2004, p. 114), “[...] a gerência de configuração fornece as ‘linhas’ que unem as partes do sistema entre si, unificando os componentes que tenham sido desenvolvidos separadamente. Essas linhas nos permitem coordenar as atividades de desenvolvimento [...]”.

Especificamente, durante a identificação e análise de requisitos, o gerenciamento de configurações detalha a correspondência entre os elementos da definição de requisitos e a especificação de requisitos de tal modo que a visão do cliente esteja associada à visão do desenvolvedor, de uma maneira que possa ser acompanhada.

Requisitos funcionais e NÃO funcionais



Diálogo com o Autor

Segundo Pfleeger (2004, p. 115):

“Os requisitos descrevem o comportamento de um sistema. À medida que o sistema atua nos dados ou nas instruções, objetos ou entidades se ‘movem’ de um estado para outro, por exemplo: de vazio para cheio; de ocupado para desocupado; de envio para recebimento. Isto é, em um determinado estado, o sistema satisfaz um conjunto de condições; quando o sistema atua, ele pode mudar seu estado como um todo, modificando-o de um objeto.

Os requisitos expressam os estados e as transições do sistema e do objeto, de um para outro. Em particular, os requisitos descrevem atividades do sistema, tais como uma reação à inserção de dados, e o estado de cada entidade antes e depois de a atividade ocorrer. Por exemplo, em um sistema de folha de pagamento, os funcionários podem existir em pelo menos dois estados: funcionários que ainda não foram pagos e funcionários que já foram pagos. Os requisitos descrevem, como, na emissão de contracheques, um funcionário pode se ‘mover’ do primeiro estado para o segundo”.

Para descrevermos os requisitos podemos classificá-los em funcionais e não funcionais.

A. Requisitos Funcionais

Os requisitos funcionais descrevem uma interação entre o sistema e o ambiente, descrevem como o sistema deve funcionar, considerando determinado estado, declara quais serviços que o sistema deve fornecer, como o sistema deve reagir a entradas específicas e como deve se comportar em determinadas situações.

Tomemos como exemplo o sistema de emissão de contracheque (holerite de pagamento mensal): se esse sistema for utilizado semanalmente, os requisitos funcionais deverão responder às questões relacionadas aos contracheques emitidos. Para isso podemos pensar em algumas perguntas como: quais informações são necessárias para que um contracheque seja emitido? Sob quais condições o valor poderá ser alterado? O que causa a exclusão de um funcionário da folha de pagamento?

É importante lembrarmos que as respostas a essas perguntas independem da implementação da solução para o cliente. Estamos descrevendo o que o sistema fará e quais os requisitos adicionais necessários para que ele funcione.

Imagine um sistema de matrícula escolar: Sabemos que ele deve permitir inclusão, alteração, exclusão e consulta, mas quais as perguntas que seriam necessárias para definição dos requisitos? Quais perguntas você faria para o cliente que contratou seus serviços?



Pense

Neste momento, é importante que você pare, compare o parágrafo anterior com o que veio antes e tente traçar um paralelo entre eles. Tente ir além do exemplo do contracheque, transponha o conteúdo abordado até aqui para um novo sistema e defina os requisitos.

B. Requisitos Não Funcionais

Em vez de informar o que o sistema fará, requisitos não funcionais colocam restrições no sistema, por isso podem ser chamados de restrições.

Informações como: o sistema deverá rodar numa plataforma XYZ, as consultas devem ser respondidas em X intervalo de tempo, o sistema deve permitir um compartilhamento de dados entre diferentes escolas do mesmo grupo etc.

Esses requisitos limitam nossa seleção com relação à linguagem a ser utilizada, modelagem a ser utilizada (estruturada, essencial, orientada a objeto etc.).

Os requisitos não funcionais podem afetar a estrutura do sistema assim como podem gerar inúmeros outros requisitos funcionais, como é o caso de um requisito de proteção que definirá, por exemplo, os serviços necessários ao sistema (SOMMERVILLE, 2011).

Os requisitos funcionais e não funcionais devem ser definidos e identificados com os clientes de uma maneira cuidadosa, pois definirão os próximos passos de implementação, testes e manutenção.

Tipos de requisitos



O documento de especificação de requisitos descreve tudo sobre o sistema e a integração com o ambiente. Eles incluem os seguintes itens, segundo Pfleeger (2004) e Pressman (1996):

Tabela 1 – Questões e tipos relacionados aos requisitos.

TIPOS	QUESTÕES
Ambiente físico	Onde o equipamento funcionará? Esse funcionamento se dará em um ou vários locais? Existe alguma restrição ambiental, tal como temperatura, umidade ou interferência magnética? (Softwares da área médica, de controladores de voos etc. devem considerar essas interferências de forma diferenciada).
Interface	A entrada tem origem em outros sistemas? A saída vai para outro sistema? Existe uma maneira pré-estabelecida pela qual os dados devem ser formatados? (CEP – sempre numérico) Existe alguma mídia definida que os dados devem utilizar?
Usuários e fatores humanos	Quem utilizará o sistema? Haverá diversos tipos de usuários? Qual o nível de habilidade de cada tipo de usuário? Que tipo de treinamento será necessário para cada tipo de usuário? Que facilidade o usuário terá para entender e utilizar o sistema? Qual será a dificuldade para que o usuário utilize adequadamente o sistema?
Funcionalidade	O que o sistema fará? Quando o sistema fará? Existem diversos modos de operação? Como e quando o sistema pode ser modificado ou aprimorado? Existem limitações quanto à velocidade de execução, ao tempo de resposta, ou à saída?
Documentação	Que documentação é necessária? Essa documentação deve ser on-line, no formato de livro, ou ambos? A que público se destina cada tipo de informação?
Dados	Qual deverá ser o formato dos dados de entrada e saída? Com que frequência eles serão enviados e recebidos? Que precisão devem ter os dados? Com que grau de precisão os cálculos deverão ser feitos? Existem dados que devem ser mantidos por determinado tempo?

Recursos	Que materiais, pessoal ou outros recursos são necessários para construir, utilizar e manter o sistema? Que habilidade os desenvolvedores devem ter? Quanto espaço físico será ocupado pelo sistema? Quais os requisitos quanto à energia, ao aquecimento ou condicionamento de ar? Existe um cronograma já definido para o desenvolvimento? Existe um limite de custo para o desenvolvimento ou para a aquisição de hardware ou de software?
Segurança	O acesso ao sistema ou às informações deve ser controlado? Como os dados de um usuário serão isolados dos outros usuários? Como os programas dos usuários serão isolados de outros programas e do sistema operacional? Com que frequência será feito backup? Onde serão armazenadas as cópias de backup? Devem ser tomadas precauções contra fogo, danos provocados pela água, ocorrência de roubo etc.?
Garantia de qualidade	Quais os requisitos quanto à confiabilidade, disponibilidade, manutenibilidade, segurança e outros atributos de qualidade? Como as características do sistema devem ser demonstradas para os outros? O sistema deve detectar e isolar defeitos? Qual o tempo médio entre falhas, que foi determinado? Existe um tempo máximo permitido para reiniciar o sistema depois de uma falha? Como o sistema pode incorporar modificações no projeto? A manutenção corrigirá os erros ou também incluirá o aprimoramento do sistema? Que medidas de eficiência serão aplicadas à utilização dos recursos e ao tempo de resposta? Com que facilidade o sistema se deslocará de um local para outro (acesso remoto) ou de um tipo de computador para outro?

Alguns passos podem ser seguidos para descobrir exatamente o que o cliente quer, como é o caso de várias entrevistas e reuniões. Basicamente, durante o processo de especificação de requisitos, devemos:

- Analisar a situação atual da empresa (se existe algum sistema, funcionalidades etc.)
- Fazer com que o usuário aprenda a entender o contexto, os problemas e os relacionamentos do sistema solicitado.
- Entrevistar usuários atuais e potenciais.
- Demonstrar como o novo sistema deve operar.
- Pesquisar documentação existente.
- Realizar um brainstorming com os usuários potenciais e usuais.
- Observar estruturas e padrões.
- Documentar todo o processo e todas as reuniões.

Revisão dos Requisitos

Todo requisito permite ao desenvolvedor explicar seu entendimento de como os clientes querem que o sistema funcione. Eles devem informar aos projetistas quais funcionalidades e características o sistema resultante deve ter; e informar a equipe de teste sobre o que demonstrar para convencer o cliente de que o sistema será entregue de acordo com o que foi solicitado.

Para garantir que os itens especificados sejam alcançados, é importante verificarmos sempre se: os requisitos estão corretos, são consistentes, estão completos, são realistas, cada um descreve algo que é importante para o cliente, podem ser verificados, podem ser rastreados etc.

Com relação ao sistema, após definirmos os requisitos funcionais e não funcionais, é importante revisarmos para verificar se as metas e objetivos do software permanecem consistentes com as metas e os objetivos do projeto, se as interfaces importantes para todos os elementos do sistema foram descritas, se o fluxo e a estrutura da informação são adequadamente definidos para o domínio da informação, se a modelagem e diagrama estão claros, se há risco tecnológico do desenvolvimento, se os requisitos de software alternativos foram considerados, se existem inconsistências, redundâncias ou omissões, se o que o cliente desejava foi contemplado por completo.

Documento de Definição de Requisito

A documentação da definição de requisitos é fundamental e deve sempre ser descrita em linguagem que os clientes entendam. Utilizaremos linguagem técnica apenas na modelagem do sistema.

Este documento é construído ao longo do processo de desenvolvimento. Alguns capítulos serão finalizados antes, outros durante e outros ao final do desenvolvimento do sistema.

Para tanto, Pfleeger (2004) e Sommerville (2011) definem os seguintes tópicos que deverão ser estruturados em capítulos, conforme a Tabela 2:

Tabela 2 – Estrutura do documento de requisitos.

Capítulo	Descrição
Prefácio	Define possíveis leitores do documento assim como o histórico das versões (criação, alteração, exclusão etc.).
Introdução	Descreve a necessidade para o sistema. Explica as funções dos sistemas e como deverão ser integradas (funcionar com) a outros sistemas, além de como atender aos objetivos globais dos negócios do solicitante.
Glossário	Descreve termos técnicos utilizados.
Definição dos requisitos de usuários	Descreve o serviço a ser fornecido aos usuários. Os requisitos não funcionais também são descritos neste item. Esta descrição pode ser em linguagem natural, diagrama ou outra forma, desde que facilmente compreensível ao cliente; afinal, ele terá que aprovar este documento.
Arquitetura do sistema	Apresenta a visão geral e de alto nível da arquitetura do sistema, inclusive mostrando a distribuição de funções entre os módulos. Aqui devem ser descritos os objetos que serão reutilizados no desenvolvimento do projeto.

Especificação dos requisitos do sistema	Descreve em detalhes os requisitos funcionais e não funcionais.
Modelos do sistema	Inclui modelos gráficos que mostram os relacionamentos dos componentes do sistema e do ambiente em que operará. Aqui entra as descrições e documentação de modelagem (fluxo de dados, relacionamentos, objetos etc.).
Evolução do sistema	Descreve os pressupostos fundamentais em que o sistema se baseia e todas as suas mudanças previstas em decorrência da evolução do hardware. Esta função é útil ao projetista do sistema porque poderá dimensionar futuras versões e tecnologias a serem consideradas.
Apêndice	Neste item, informações detalhadas são incorporadas – hardware, banco de dados, modelagem, configurações mínimas exigidas, organização lógica dos dados, modelagens etc.
Índice	Vários índices podem existir: alfabético (normal), de diagramas, de figuras, funções etc.

De posse de todas as especificações de requisitos, podemos dar início ao processo de modelagem. Nesta disciplina, não abordaremos os conceitos de modelagem. Caso não tenha tido acesso a esse assunto, sugerimos uma pesquisa complementar sobre as técnicas existentes, pois a modelagem de um sistema junto com a definição e especificação de requisitos são documentos importantes para os desenvolvedores conseguirem realizar seu trabalho de forma eficiente e eficaz, cumprir os prazos e as estimativas.

Gerenciamento de configurações

Alterar um software é inevitável quando o estamos desenvolvendo. Inúmeras ideias surgem para melhoria, solicitações por parte do cliente ou dos próprios gerentes de desenvolvimento, e a solução é mexer no projeto inicial para adequá-lo.

Geralmente, quando falamos em projeto de desenvolvimento de software, estamos nos referindo a sistemas complexos, feitos por módulos por diversos desenvolvedores que podem estar ou não fisicamente distantes e conhecer ou não todos os membros da equipe de desenvolvimento.

É nesse cenário que problemas em modificações em rotinas, sistema ou até mesmo no escopo do projeto podem ser catastróficos se não forem coordenados e gerenciados.

Imagine você desenvolvendo um módulo, vai para casa descansar no final do expediente e, no dia seguinte, percebe que algumas rotinas foram alteradas? E o pior: percebe que foram alteradas por alguém que não conhecia direito o quanto aquela rotina influenciava outras do próprio sistema e nem que era usada em outro sistema de outro usuário.

Com certeza, nessa cena, algum desconforto entre as equipes surgiria.

Mudanças devem ser analisadas antes da realização, registradas antes da implementação, relatadas aos que precisam tomar conhecimento delas ou controladas de modo que melhorem a qualidade geral do projeto e reduzam erros. (PRESSMAN, 1996).

É nesse cenário de controlar as alterações que surge o Gerenciamento de Configurações de Software (GCS) ou Software Configuration Management (SCM).

GCS é uma atividade abrangente que é aplicada em todo o processo de engenharia de software. Uma vez que uma mudança pode ocorrer a qualquer tempo, as atividades de GCS são desenvolvidas para:

1. Identificar mudanças.
 2. Controlar mudanças.
 3. Garantir que a mudança esteja sendo adequadamente implementada.
 4. Relatar mudanças a outras pessoas que possam ter interesse nelas.
- (PRESSMAN, 1996, p. 916)

GCS pode ser considerada como a arte de coordenar o desenvolvimento de software para minimizar os problemas de mudanças e alterações antecipadamente.

É importante salientar que GCS não é manutenção.

Segundo Pressman (2006, 1996), Sommerville (2011) e Pfleeger (2004), a manutenção é um conjunto de atividades de Engenharia de Software (ES) que acontece depois que o sistema foi entregue para o , enquanto GCS é um conjunto de atividades de ES que acontece no momento que o projeto começa a ser desenvolvido e vai até o momento de ser retirado do mercado. É um conjunto de atividade contínua (Figura 2).

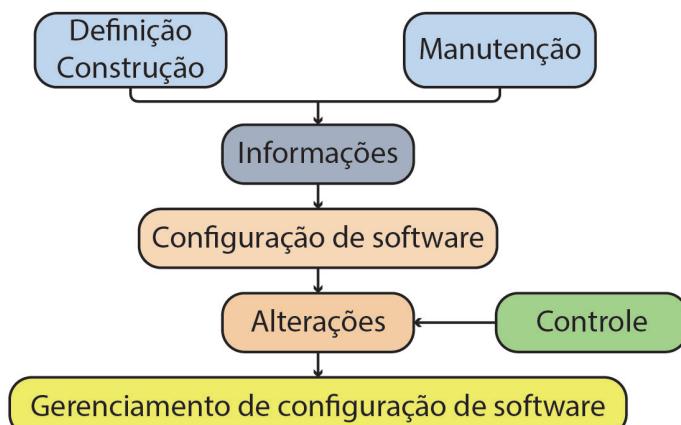


Figura 2 – Descrição ilustrativa sobre GCS.



Diálogo com o Autor

Uma meta primordial da metodologia de Engenharia de Software é melhorar a facilidade com que as mudanças podem ser acomodadas e reduzir a quantidade de esforços despendidos quando as mudanças são feitas. (PRESSMAN, 1996, p. 917)

O resultado de todo processo de Engenharia de Software são informações que podem ser divididas em:

- 1 - Código fonte: executável e fonte, que na verdade é o resultado do projeto contratado.
- 2 - Documentos que apoiam e esclarecem todo o processo e itens relacionados a ele.
- 3 - Estrutura de dados: existente e integrada que apoia e é apoiada (tanto para o item 1, quanto para o 2).

Os itens gerados de todos esses componentes são chamados de **Itens de Configuração de Software (ICS) ou Software Configuration Items (SCI)**, e sua descrição e informações relacionadas a eles deverão fazer parte da documentação do Plano de Projeto.

Seria tranquilo se pudéssemos trabalhar por meio da garantia da não mudança do Plano de Projeto ou da Especificação de Software. Infelizmente, não é isso que acontece. Mudanças ocorrem o tempo todo, aliás, se você está desenvolvendo um projeto de software, não importa em qual momento do ciclo de desenvolvimento você esteja, com certeza, mudanças serão implementadas por você e solicitadas pelos membros da equipe de desenvolvimento e/ou projetista e/ou gerente e/ou qualquer pessoa que conheça e pense no projeto. Isso é tão certo quanto $2+2 = 4$!

Gerenciar as mudanças nos oferece um ambiente estável de desenvolvimento, pois alterações sem controle proporcionam um ambiente caótico. Na nossa unidade 1, vimos que a ES veio para colocar ordem numa atividade que era caótica: a de desenvolvimento de software sem ordem, padrão ou gerenciamento adequado.

Existem técnicas e conceitos que nos ajudam a coordenar essas mudanças e é isso que veremos a partir de agora.

Baseline – linhas básicas

Mudanças: éis uma coisa constante num ambiente tão instável como é o de desenvolvimento de um projeto.

Clientes querem modificar requisitos, administração quer modificar a abordagem do projeto, desenvolvedores querem modificar abordagens técnicas etc.

Com o passar do tempo e a familiaridade com o desenvolvimento de cada projeto, novas ideias, sugestões e mudanças surgem e o engenheiro de software tem que enfrentar uma realidade difícil: muitas das mudanças solicitadas são justificáveis, ou seja, precisam ser feitas.

Uma Linha Básica ou Baseline é um conceito de GCS que ajuda a controlar as mudanças sem impedir seriamente as mudanças justificáveis. Ou seja: se uma mudança é justificável, ela poderá ser feita, mas por meio de uma linha básica.

Pressman (1996) nos descreve o mecanismo da linha básica por meio de uma analogia:

Consideremos as portas da cozinha de um grande restaurante. Para eliminar colisões, uma porta é marcada como entrada e outra como saída. As portas têm quatro paradas que permitem que elas sejam abertas somente na direção apropriada.

Se um garçom pegar um pedido na cozinha, colocá-lo numa bandeja e depois perceber que escolheu o prato errado, ele pode mudar para o prato certo rápida e informalmente antes de sair da cozinha.

Se, entretanto, ele sair da cozinha, entregar o prato ao cliente e depois for informado de seu erro, ele deve seguir uma série de procedimentos:

1. Olhar a conta para determinar se ocorreu o erro;
 2. Desculpar-se profundamente;
 3. Retornar à cozinha pela porta ENTRADA;
 4. Explicar o problema, e assim por diante.
- (PRESSMAN, 1996, p. 918)

Uma baseline é idêntica a portas da cozinha de um restaurante. Antes de um item de configuração de software se tornar uma linha básica, mudanças podem ser feitas de modo rápido e informalmente. Após esse período, mudanças devem ser feitas formalmente e por meio de procedimentos e etapas (Figura 3).

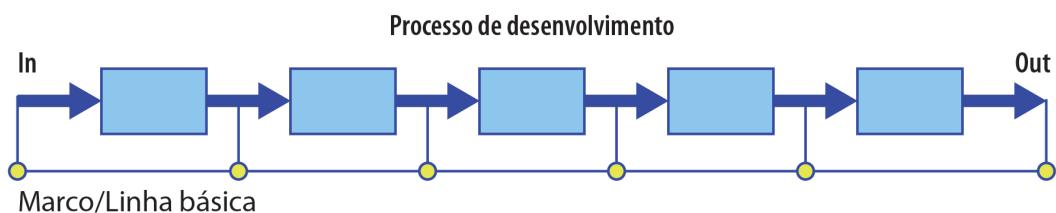


Figura 3 – Definição das linhas básicas.

Uma linha básica é um marco de referência. Antes de um item se tornar uma linha básica, inúmeras alterações podem ocorrer, pessoas diferentes podem alterá-los (além do desenvolvedor), sugestões podem ocorrer; uma vez que o item foi considerado referência/marco, alterações só poderão ser realizadas por meio de solicitação e documentação formal.

Linhas básicas podem ocorrer em qualquer fase do projeto e, uma vez definidas informações sobre elas, são colocadas no que chamamos de repositório de software ou repositório de itens de configuração ou biblioteca de projeto.

Quando alguém da equipe de Engenharia de Software quer fazer uma mudança num SCI definido com linha básica, este item é copiado do da Base de Dados (BD) para a área particular do solicitante e um registro dessa solicitação é guardado num arquivo chamado contábil. O solicitante então poderá alterar o arquivo copiado até que todas as mudanças sejam completadas e só então ele voltará à base de dados inicial já atualizada. Em algumas circunstâncias, o arquivo original é bloqueado e ninguém pode utilizá-lo até que todas as mudanças sejam executadas e o arquivo seja atualizado, revisto, aprovado e liberado novamente para uso de todos da equipe.

As linhas básicas podem ocorrer ao final de cada fase do desenvolvimento ou quando gerentes, projetistas, engenheiros ou desenvolvedores decidirem. Quando um item passa, é definido como linha básica e, pelo baseline, ele é considerado baselined, ou é dito que o item tornou-se uma linha básica.

Características de um item baselined:

- Foi revisto formalmente e teve acordo das partes.
- Serve como base para o trabalho futuro.
- É armazenado no repositório dos itens de configuração (RIC).
- Pode ser alterado somente através de procedimentos formais de controle de mudança.

O processo de gerenciamento de configuração de software



Para controlarmos o processo de GCS, é necessário coordenarmos as alterações de algumas atividades, como veremos a seguir.

O processo consiste em:

- A. Selecionar o item a ser gerenciado.
- B. Identificar objetos.
- C. Controlar versão.
- D. Controlar mudanças.
- E. Fazer auditoria de configuração.
- F. Elaborar relatório de status do sistema.
- G. Controlar interface.
- H. Controlar fornecedores de informação.

A. Selecionar o item a ser gerenciado

Selecionar o item a ser gerenciado é uma das tarefas mais difíceis, pois ele deve ser escolhido por meio de análise minuciosa. Esse é novamente um processo que engloba técnica e criatividade.

Se não selecionar qual item será gerenciado e não souber exatamente o porquê disso, poderá acontecer uma exagerada documentação e um atraso nas funções de desenvolvimento desnecessárias.

O processo para execução dessa atividade pode ser estruturado por:

- Selecione o item.
- Descreva como o item se relaciona a outros no BD.
- Planeje as linhas de referências tendo como base o ciclo de desenvolvimento do projeto.

- Descreva quando pode ser alterado (descrever quais as justificativas que serão aceitas e em que circunstâncias).
- Descreva como pode ser alterado (se haverá necessidade ou não de suspender sua execução e utilização durante o período de mudança).
- Crie um documento e anexe ao plano do projeto.
- Insira essas informações, de forma adequada, no repositório de itens de configuração.

B. Identificação de objetos na configuração de software

Para controlar e administrar ICS, cada um deve ser nomeado separadamente e depois organizado, usando uma abordagem orientada a objeto.

Cada item é considerado um objeto, por isso a necessidade do nome, da descrição, da lista de recursos e identificação, conforme o seguinte:

- Nome: nome dado ao objeto.
- Descrição: é uma lista de itens de dados que identifica o tipo de ICS (documento, programa, dados) que é representado por objeto, identificador do projeto, informações sobre mudanças ou versões.
- Lista de recurso é a lista de entidades fornecidas, processadas, consultadas ou exigidas pelo objeto.
- A identificação do objeto de configuração também deve levar em consideração objetos relacionados ao objeto nomeado.

Exemplo de uma informação existente no ICS:

Item	Id	tipo	nome	versão	nome completo
Especificação do Projeto	ProjAA	EP	Obj1_escopo	V1.0	AAEP Obj1_escopo V10
Especificação de Requisito	ProjAA	ER	Obj2_ReqIntfc	V2.0	AAER Obj2_ReqIntfc V20

Os objetos podem ser:

Básicos: unidade de texto que foi criada por um engenheiro de software durante a análise, projeto, codificação ou teste. Um exemplo pode ser uma seção de especificação de requisitos ou uma listagem fonte para um módulo.

Composto: é uma coleção de objetos básicos e outros objetos compostos.

O controle das alterações desses objetos pode ser documentado por meio de gráficos como podemos ver na Figura 4:

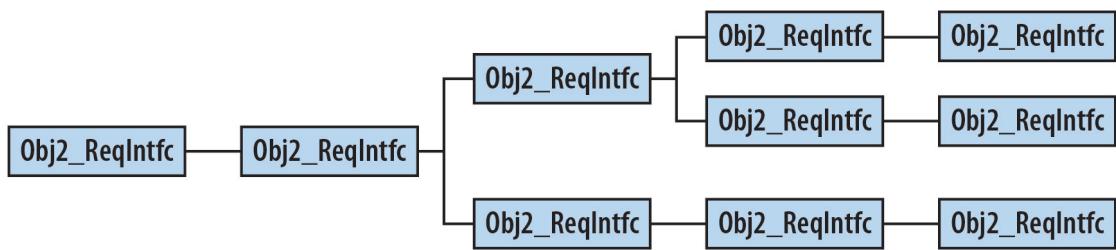


Figura 4 – Representação gráfica do controle de objetos nos ICS também chamado de Gráfico de Evolução.

A representação gráfica do controle de mudanças dos ICS tratados como objetos fará parte da documentação do Plano de Projeto.

C. Controle de versão

Esse controle combina procedimentos e ferramentas para gerenciar diferentes versões de objetos de configurações.

O gerenciamento de configurações permite que o usuário identifique configurações alternativas do sistema de software por meio da escolha de versões apropriadas. Isso é levado ao efeito ao associar atributos a cada versão de software e depois permitir que uma configuração seja especificada [e construída], descrevendo-se o conjunto de atributos desejados. (PRESSMAN, 1996, p. 927)

Como no item anterior, o controle de versões também pode ser representado graficamente, conforme veremos na Figura 5.

Uma das representações existentes para ser usada nesse caso é a chamada “Árvore Delta”. Nela, são inseridas figuras delta, de acordo com sua relação com a mudança, por exemplo:

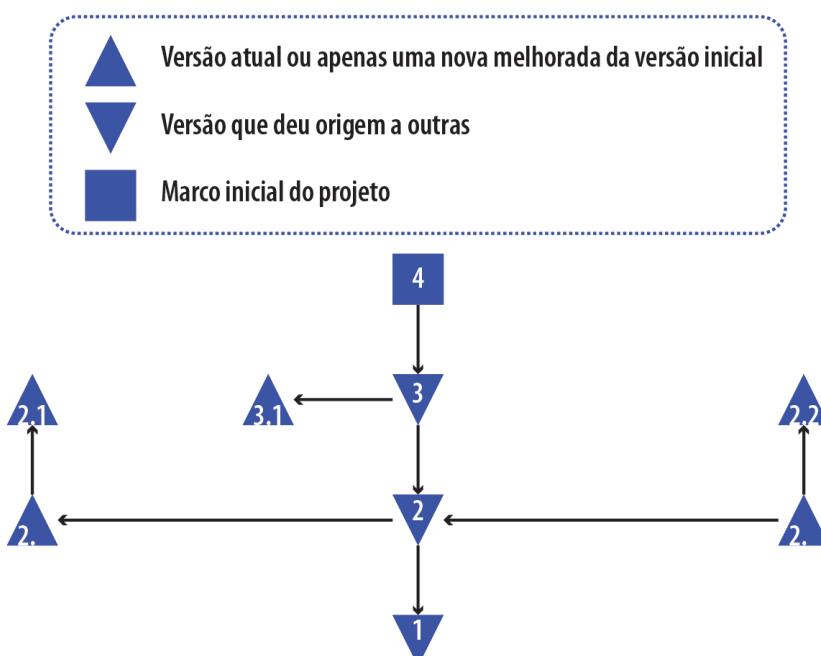


Figura 5 – Representação gráfica de Árvore Delta para o controle de versão.

A representação gráfica do controle de versões dos ICS fará parte da documentação do Plano de Projeto.

D. Controle de mudanças

Não controlar as mudanças significa trabalhar num ambiente caótico e consequentemente realizar trabalhos com qualidade questionável.

O controle de mudanças combina procedimentos humanos e ferramentas automatizadas para proporcionar um mecanismo de controle das mudanças.

Um pedido de mudança sempre será submetido a avaliações técnicas. Devem ser analisados os potenciais efeitos colaterais, o impacto global sobre os outros objetos de configuração e funções do sistema e o custo projetado da mudança.

O resultado dessa avaliação é apresentado num documento chamado relatório de mudanças e uma equipe pertencente à Autoridade Controladora de Mudanças (Change Control Authority – CCA) toma a decisão final sobre o status e a prioridade da mudança.

Uma Ordem de Mudança de Engenharia (Engineering Change Order – ECO) é gerada para cada mudança aprovada. Na ECO, é descrita a mudança a ser feita, as restrições que devem ser respeitadas e os critérios de revisão e auditoria.

Nesse momento, o processo a ser alterado passa por um processo de check-out. Esse processo retira o item do repositório, cataloga-o no sentido de informar que ele está sendo alterado. Posteriormente, após o item ser alterado, ele passa por um período de check-in – inserção do objeto no repositório (figuras 6, 7 e 8).

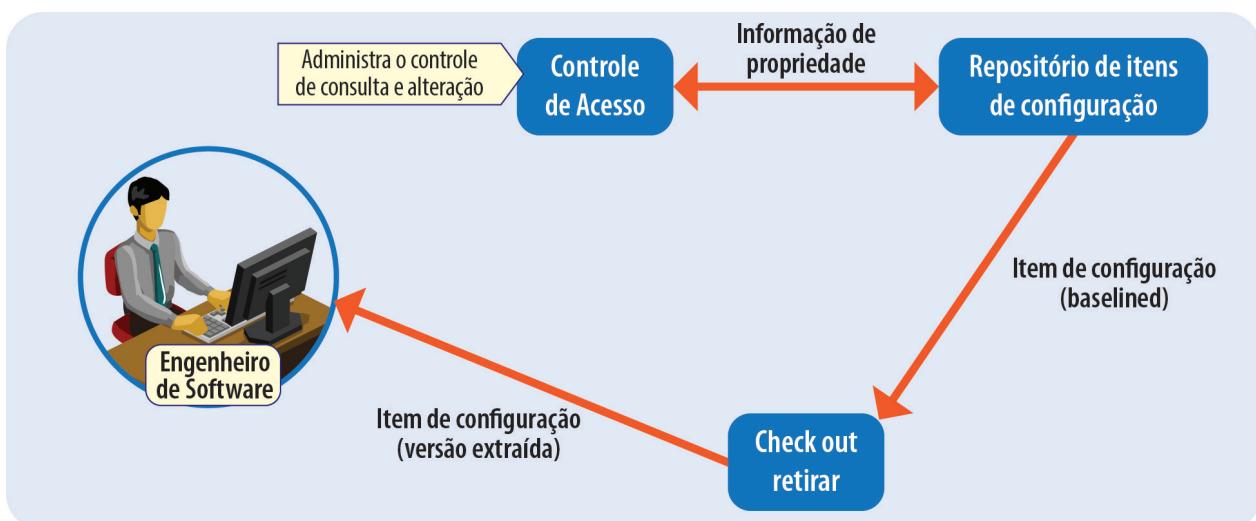


Figura 6 – Descrição ilustrativa sobre *check-out*

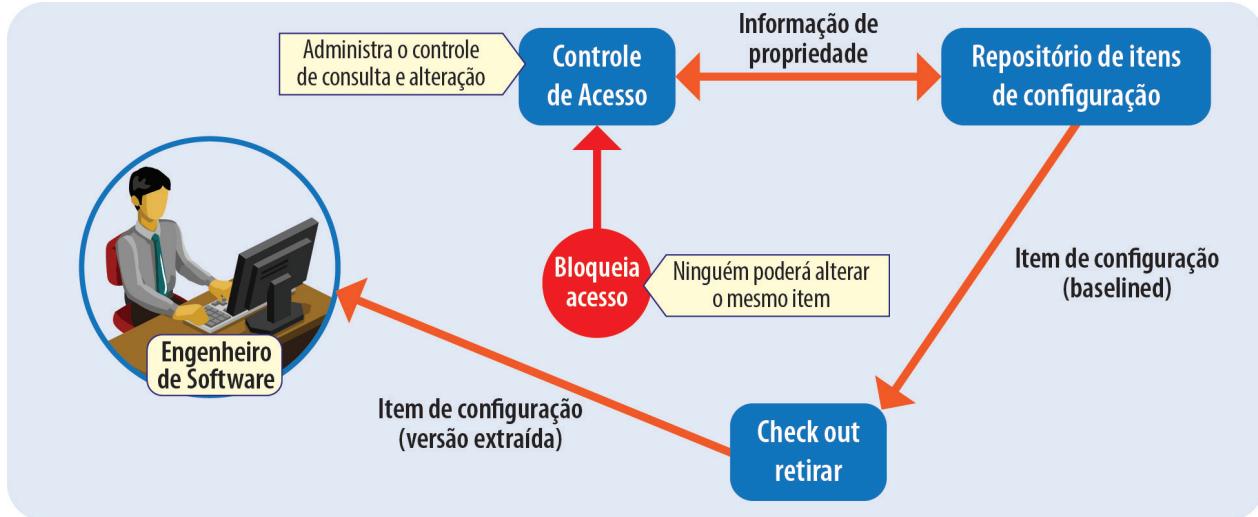


Figura 7 – Descrição figurativa do bloqueio do objeto no repositório.

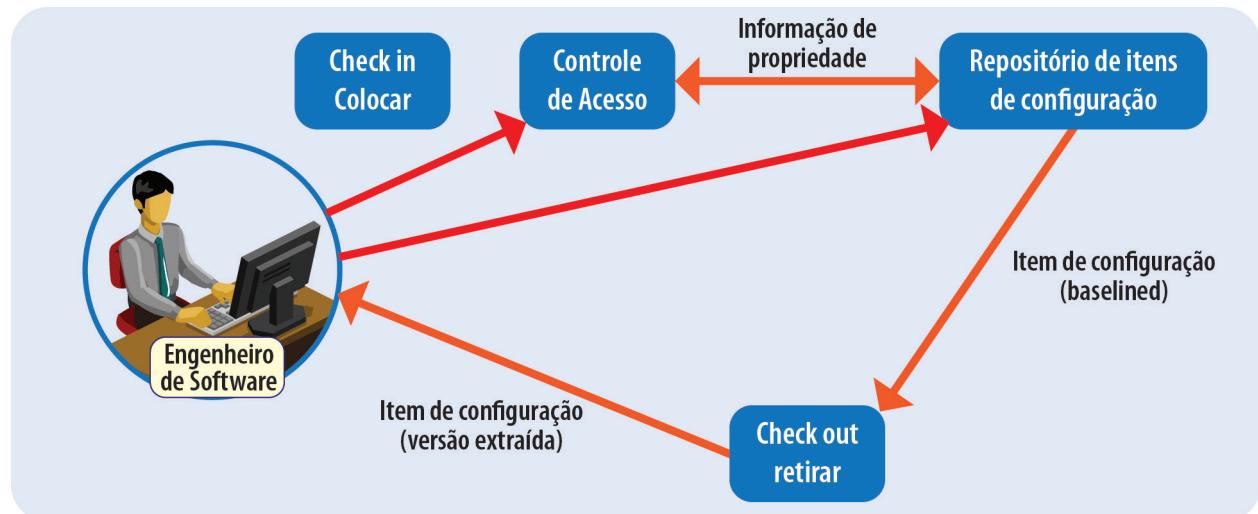


Figura 8 – Item alterado e inserido no repositório.

E. Auditoria de Configurações

A identificação, o controle de versão e o controle de mudanças ajudam o desenvolvedor de software a manter ordem daquilo que, de outro modo seria uma situação caótica e difícil de ser gerenciada.

Entretanto, a mais controlada ação de mudança é capaz de controlar e rastrear as mudanças até o ponto em que uma ECO é gerada.

Para garantirmos que a mudança seja adequadamente realizada é necessário:

4 - Revisões técnicas formais.

5 - Auditoria de configuração de software.

Uma **revisão técnica formal** deve ser realizada para todas as situações. Ela focaliza a exatidão técnica do objeto de configuração que foi alterado e é avaliada sua consistência com relação a outros ICS.

Uma **auditoria de configuração de software** complementa a revisão técnica formal ao avaliar um objeto de configuração quanto às características que geralmente não são consideradas durante a revisão.

[...] a auditoria pergunta e responde às seguintes questões:

A mudança especificada na ECO foi feita? Outras modificações adicionais foram incorporadas?

Uma revisão técnica formal foi realizada para avaliar a exatidão técnica?

Os padrões de ES foram adequadamente seguidos?

A mudança foi “realçada” no ICS? A data da mudança e o autor da mudança foram especificados? Os atributos do objeto de configuração refletem a mudança?

Os procedimentos de GCS para anotar a mudança, registrá-la e relatá-la foram seguidos?

Todos os ICS relacionados foram adequadamente atualizados?

(PRESSMAN, 1996, p. 934)

F. Relato de status de configuração

A construção do Relato de Status de Configuração (RSC) – *Configuration Status Report* (CSR) – é uma tarefa que visa a responder às questões sobre o que aconteceu, quem fez, quando e o que mais é afetado.

Toda vez em que um ICS recebe uma identificação nova, é atualizado; toda vez em que uma auditoria é realizada, uma entrada de RSC é realizada.

Essas informações, geralmente são geradas online e podem ser acessadas pela administração e por profissionais para que eles fiquem a par das mudanças ocorridas.

G. Controle de interface

Este item segue os mesmos padrões dos anteriores: coordena as mudanças de interfaces dos objetos que pertencem ao ICS.

As mudanças de interfaces solicitadas são coordenadas pelo processo de controle de mudanças e por meio delas é atualizada a base de dados que contém as informações sobre as interfaces/acesso a dados.

H. Controle de fornecedores de informação

Nem sempre toda informação e base de dados acessadas por um projeto faz parte de uma BD pertencente ao cliente. Algumas vezes, informações e BD de terceiros são relacionadas e denominadas Base de Dados de Fornecedor.

Esse item controla mudanças de terceiros acoplados, mudanças essas coordenadas pelos itens anteriores (A-F).

As questões relacionadas à Gerência de Configuração de Software (GCS) têm como objetivo coordenar e auditar todas as mudanças ocorridas no sistema do seu desenvolvimento até o momento que é retirado do mercado.

É importante salientar que não é manutenção e nem procedimentos que ocorrem após a entrega do projeto ao cliente. GCS visa responder, por meio de um processo estruturado, às seguintes questões: o que mudou e quando? Por que mudou? Quem fez a mudança? Podemos reproduzir esta mudança?

Nesse contexto, podemos dizer que o controle de versão é capaz de dizer o que mudou e quando mudou, além de atribuir os motivos a cada uma das mudanças.

A auditoria, por sua vez, responde às duas últimas perguntas: quem fez a mudança? Podemos reproduzir a mudança? Cada uma dessas perguntas está diretamente relacionada ao processo de controle de atualização e mudanças do projeto que é de vital importância para a Engenharia de Software.

Material Complementar



Explore

O objetivo do material complementar é lhe ajudar a entender, sob uma ótica diferente daquela da professora confeudista, assuntos abordados nas unidades teóricas.

É fundamental a leitura deste material para o melhor entendimento sobre o assunto.

Como nesta unidade abordamos os conceitos gerais da Engenharia de Software, nossa sugestão de material complementar é o capítulo 4, intitulado Engenharia de requisitos, no seguinte livro:

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011. p. 57-81.

Referências

Bibliografia Fundamental

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011.

Bibliografia Básica

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação**. 4. ed. Rio de Janeiro: LTC, 1999.

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação Gerenciais**. Administrando a empresa digital. 5. ed. São Paulo: Pearson Education do Brasil, 2006.

PFLEEGER, S. L. **Engenharia de Software**: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 2006.

SOMMERVILLE, I. **Engenharia de Software**. 6. ed. São Paulo: Pearson Addison Wesley, 2005.

Bibliografia Complementar

ALCADE, E.; GARCIA, M.; PENUELAS, S. **Informática Básica**. São Paulo: Makron Books, 1991.

FAIRLEY, R. E. **Software engineering concepts**. New York: McGraw-Hill, 1987.

IEEE. **Software Engineering Standards**. 2013. Disponível em: <http://www.ieee.org/portal/innovate/products/standard/ieee_soft_eng.html>. Acesso em: 10 dez. 2013.

LUKOSEVICIUS, A. P.; CAMPOS FILHO, A. N.; COSTA, H. G. **Maturidade em gerenciamento de projetos e desempenho dos projetos**. Disponível em: <www.producao.uff.br/conteudo/rpep/.../RelPesq_V7_2007_07.doc>. Acesso em: 12 nov. 2013.

MAFFEO, B. **Engenharia de Software e especialização de sistemas**. Rio de Janeiro: Campus, 1992.

MICHAELIS. **Moderno dicionário da língua portuguesa**. São Paulo: Cia. Melhoramentos, 1998.

PARREIRA JÚNIOR, W. M. **Apostila de Engenharia de Software**. Disponível em: <http://www.waltenomartins.com.br/ap_es_v1.pdf>. Acesso em: 13 nov. 2013.

PAULA FILHO, W. P. **Engenharia de Software**: fundamentos, métodos e padrões. 2. ed. Rio de Janeiro: LTC, 2001.

Revista Engenharia de Software. Disponível em: <<http://www.devmedia.com.br/revista-engenharia-de-software-magazine>>. Acesso em: 12 nov. 2013.

VONSTA, A. **Engenharia de Programas.** Rio de Janeiro: LTC, 1983.

WIENNER, R.; SINCOVEC, R. **Software engineering with Modula 2 and ADA.** New York: Wiley, 1984.

WIKIPEDIA (2007a). **ISO 9000.** Disponível em: <http://pt.wikipedia.org/wiki/ISO_9000>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007b). **Melhoria de Processo de Software brasileiro.** Disponível em: <<http://pt.wikipedia.org/wiki/MPS.BR>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007c). **CMMI.** Disponível em: <<http://pt.wikipedia.org/wiki/Cmmi>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007d). **Engenharia de Software.** Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software>. Acesso em: 01 fev. 2007.

Anotações





Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

www.cruzeirodosulvirtual.com.br
Campus Liberdade
Rua Galvão Bueno, 868
CEP 01506-000
São Paulo SP Brasil
Tel: (55 11) 3385-3000



Universidade
Cruzeiro do Sul



UNICID
Universidade
Cidade de S. Paulo



UNIFRAN
Universidade
de Franca



UDF
Centro
Universitário



Módulo
Centro
Universitário

Engenharia de *Software*



Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

Material Teórico



Teste e manutenção de *software*

Responsável pelo Conteúdo:

Prof.^a Dr.^a Ana Paula do Carmo Marchetti Ferraz

Revisão Textual:

Prof.^a Me. Luciene Oliveira da Costa Santos



- Introdução
- Estratégia de testes
- Inspeção de *software*
- Técnicas ou métodos de testes



Nesta unidade, serão apresentadas as estratégias de testes e tipos de manutenção que existem.

Testar um software é fundamental para garantir a qualidade do produto de software comercializado, por isso, em grandes projetos, existem equipes independentes que realizam esta atividade.

Novamente, gostaríamos de lembrar que este Material teórico não esgotará todos os aspectos relacionados a esses conceitos. Portanto, sugerimos sempre pesquisas adicionais.

Contextualização

Discutirmos as questões relativas a teste e manutenção não é tarefa fácil. Todos nós sabemos da importância dessas atividades, mas muito pouco tempo dedicamos a elas.

Hoje, entendemos que as atividades relacionadas a testes e manutenção são fundamentais no processo de desenvolvimento de produto de software.

A seguir, apresentamos duas charges relacionadas a essa atividade. Uma sobre a evolução que essa atividade teve nos últimos tempos e a outra acerca da postura (divertida e, em algumas situações, real) que alguns gerentes têm em relação à equipe de teste.



Testar um produto de software não é tarefa fácil, mas é necessária. Esperamos que ao final desta unidade você também tenha consciência disso.

Introdução



Não temos como falar em desenvolvimento de um produto de *software* sem mencionar fatores relacionados à qualidade.

Qualidade pode ser um conceito subjetivo – o que é qualidade para um, pode não ser a outro. Entretanto, para medirmos a qualidade de *software*, conceitos subjetivos não podem ser considerados. É necessário que tenhamos uma definição precisa do que é um *software* de qualidade, segundo princípios e propriedades, para podermos atestar seu nível.

Entretanto, para garantirmos um bom produto de *software*, é necessário que diferentes testes sejam executados, e é nessa direção que trabalharemos neste capítulo.

Mesmo depois de todo o cuidado durante o desenvolvimento, com revisões periódicas e inspeções para remoção de defeitos, necessitamos realizar a etapa de teste para detectar os defeitos que escaparam das revisões e para avaliar o grau de qualidade de um produto e seus componentes.

O fluxo de teste é extremamente importante para o desenvolvimento de *software*. Segundo Pressman (1996) e Sommerville (2011), o custo de correção de um problema na fase de manutenção pode chegar até a 60% do custo total do projeto. Em outras palavras, realizar testes é fundamental não apenas para atestar qualidade do produto, mas também para diminuição de custos totais do projeto.

Na fase de teste, precisamos detectar a maior quantidade possível de defeitos que não foram apanhados pelas revisões, considerando os limites de custos e prazos (PAULA FILHO, 2005).

Estratégia de testes



Segundo Pressman (2007) e Paula Filho (2005), teste é considerado um conjunto de atividades que deve ser planejado antecipadamente e realizado de forma sistemática.

O planejamento e a realização das atividades de teste planejadas fazem parte do que definimos por **estratégia de teste de software**.

Como sempre falamos, a Engenharia de Software coloca ordem em diversas atividades do desenvolvimento de um produto de software. Esse “colocar ordem” implica planejar antecipadamente atividades e gerenciá-las. No caso de teste, isso não é diferente. Temos que montar uma estratégia de teste de software antes de iniciarmos qualquer atividade relacionada a ela.

Uma estratégia, ou política de teste de software, define técnicas de projeto de casos de teste e métodos de teste específico para cada etapa do processo de Engenharia de Software (PRESSMAN, 1996).

Nenhum produto é testado apenas ao final do seu desenvolvimento, por isso que, para cada etapa do processo de Engenharia de Software, é necessária uma estratégia específica de teste.

Por empirismo, sabemos que:

- A atividade inicia no nível de módulos e prossegue “para fora”, na direção da integração de todo o sistema. Sempre que você fez um programa, durante sua vida acadêmica, que continha módulos ou procedimentos, normalmente você os testava separadamente e depois testava sua conexão. É a isso que este item se refere.
- Diferentes técnicas de teste são apropriadas a diferentes pontos de tempo. Em cada fase do projeto, testes devem ser aplicados e existem técnicas específicas ou que podem trazer melhores resultados em pontos específicos do projeto.

Lembre-se sempre de que “a qualidade das partes garante a qualidade do todo”.

Considerando uma estratégia de teste, ela deve:

- Acomodar testes de **baixo nível** e teste de **alto nível**.
- Oferecer orientação ao profissional.
- Oferecer um conjunto de marcos de referência ao administrador do projeto e de cada fase do sistema.
- Ser mensurável. Existem testes estatísticos que são utilizados para testar desempenho e confiabilidade do programa para checar como ele trabalha sob determinadas condições operacionais.

Vale ressaltar que a atividade de teste num produto de software não é a mesma realizada pela equipe de desenvolvimento.

A equipe de desenvolvimento realiza atividades de revisão e depuração e, somente após atestar que o projeto está adequado, é que a equipe de teste entra em ação.

Toda empresa de desenvolvimento de produto de software de grande porte possui equipes de testes contratadas para os projetos desenvolvidos. Ela é formada por um **grupo independente** de pessoas para atestar a qualidade da avaliação e verificação do que está sendo realizado.

Geralmente, a ação chega a um nível em que desenvolvedores e analistas não encontram mais erros, mas isso não significa que eles não existam. É nesse momento que novos agentes são chamados (equipe de teste que não tenha participado do processo de desenho e implementação do projeto).

Segundo Sommerville (2007), um Grupo Independente de Teste (*Independent Test Group – ITG*) apesar de ser, como o próprio nome já diz, independente, ele faz parte da equipe de projeto de desenvolvimento de software, no sentido que está envolvido durante o processo de especificação e continua envolvido planejando e especificando estratégias e políticas de teste ao longo do projeto.

Devemos mencionar que, embora as atividades de teste e depuração sejam diferentes e independentes, a depuração deve ser sempre acomodada em qualquer estratégia de teste.

Antes de começar a explorar as estratégias de teste, dois conceitos devem ser bem compreendidos: Validação e Verificação.

“Validação e Verificação (V&V) é o nome dado aos processos de verificação e análise que asseguram que o *software* cumpra com as especificações e atenda às necessidades dos clientes”. (SOMMERVILLE, 2005, p. 358)

VALIDAÇÃO: refere-se ao conjunto de atividades que garante que o *software* que foi construído seja rastreável às exigências do cliente.

Basicamente responde à pergunta: estamos fazendo o produto **certo**?

VERIFICAÇÃO: refere-se ao conjunto de atividades que garante que o *software* seja construído da forma correta.

Basicamente responde à pergunta: estamos fazendo o **certo** produto?

Ambas as definições estão diretamente relacionadas à garantia de qualidade de *software*, como podemos observar na Figura 1:



Figura 1 - Relação das atividades relacionadas à garantia de qualidade do processo de *software*.

As principais definições referenciam sobre teste e são conduzidas pelo padrão IEEE (*Software Engineering Standards*). Utilizamo-nos desse padrão, referenciados pelos autores Sommerville (2005, 2007) e Paula Filho (2005), para discutirmos esses assuntos nesta Unidade. Entretanto, não deixe de acessar o site na nossa bibliografia, para conhecer um pouco mais sobre padrões IEEE relacionados aos assuntos de Engenharia de *Software*.



Explore

Faça uma pesquisa sobre IEEE e suas normas.

O processo de V&V ocorre em várias fases do projeto, começa com as revisões de requisitos e continua ao longo de todo o processo de desenvolvimento (codificação) até a fase final de teste.

Muitas pessoas confundem verificação de *software* com validação, mas, como podemos observar, são duas atividades distintas, que devem ser trabalhadas de forma complementar e nunca exclusiva.

O objetivo principal do processo de V&V, segundo Sommerville (2007), é estabelecer um vínculo de confiança entre o sistema de desenvolvimento e o projeto aprovado pelo usuário.

O nível de confiabilidade exigida depende, em grande escala, do propósito do cliente, das expectativas do usuário (que pode ou não ser a mesma pessoa que o cliente) e o atual ambiente/segmento do mercado no qual o *software* será utilizado.

Considerando o processo de V&V, há duas abordagens, conforme Sommerville (2007, p. 342):

1 - Inspeção de software ou revisão por pares: analisa e verifica representações de sistemas como documentos de requisitos, diagramas de projeto e código-fonte de programa. Você pode usar inspeções em todos os estágios do processo. Inspeções podem ser suplementadas por alguma análise automática de texto-fonte de um sistema ou documentos associados. Inspeções de software e análise automatizada são técnicas de V&V estáticas, usadas quando você não precisa executar o *software* em um computador.

2 - Teste de software: envolve executar uma implementação do *software* com dados de teste. Você examina as saídas do *software* e seu computador operacional para verificar se seu desempenho está conforme o necessário. O teste é uma técnica dinâmica de verificação e validação.

Os processos de inspeção e testes desempenham um papel complementar no desenvolvimento do *software* e há técnicas que podem ser utilizadas em momentos específicos da produção. Uma das vantagens de se desenvolver um produto de *software* por meio da técnica incremental (revisar conceitos de Engenharia de Software) é que, ao finalizarmos uma parte do *software*, ele pode ser testado em sua totalidade, naquele incremento (parte), pois testar um sistema só é possível quando há uma versão executável, ou seja, ele não precisa estar completo, mas deve haver partes executáveis e passíveis de teste.

Considerando o abordado no item 1 (inspeção de programas), embora seja amplamente utilizado nos dias atuais, o teste de programa é a principal técnica de verificação e validação.

Testar um *software* significa experimentá-lo, usando dados reais do usuário e examinar a saída esperada para confirmar anomalias.

Segundo Sommerville (2007), há dois tipos de testes que podem ser utilizados:

1 - Teste de validação: que tem a finalidade de mostrar se o *software* é o que o cliente deseja – se atende aos requisitos. Como parte deste processo, podem ser utilizados testes estatísticos para verificar confiabilidade e desempenho.

2 - Teste de defeito: que é destinado a revelar defeitos no sistema. O objetivo é procurar inconsistência entre o programa testado e suas especificações iniciais.

Considerando as definições, podemos perceber que não há uma definição clara entre um e outro. À medida que descobrimos defeitos pelos testes de validação, normalmente, acontecem adequações para sanar inconsistências por meio da análise dos requisitos.

Sendo assim, toda vez que falarmos sobre testes, estaremos falando de verificação e validação de sistemas.

Uma confusão normal acontece em relação ao *debugging* e como este processo interage com as definições de testes.

Primeiro, precisamos entender que o processo de *debugging* é aquele que localiza e corrige os defeitos observados pelo processo de teste (V&V).

Para realizar o processo de *debugging*, podemos utilizar a técnica de verificação de variáveis, estruturas de controle, de repetição etc.

Localizar um defeito em um programa não é tarefa fácil, uma vez que geralmente ele se encontra longe do processo de saída do resultado. Mesmo assim, é necessário realizar esta etapa (*debugging*) com total responsabilidade para que a próxima etapa de V&V possa ser executada de forma segura. Na Figura 2, mostramos o processo de *debugging* de forma simplificada.

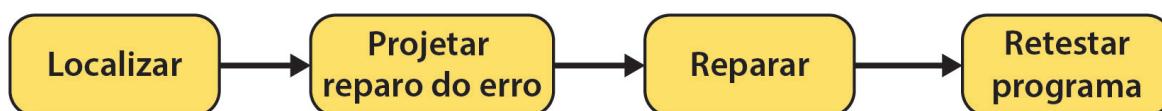


Figura 2 - Processo ilustrativo de debugging (SOMMERVILLE, 2007, p. 52)

Segundo Sommerville (2007), como no desenvolvimento de software, planos de testes devem ser projetados e seguidos para minimizar falhas após entrega do produto ao cliente.

Inspeção de software



Inspeção de software e teste são processos distintos.

Inspeção é processo de V&V estático, no qual um sistema de *software* é revisto para podermos encontrar erros, omissões e anomalias. Geralmente falamos de inspeção relacionada ao código-fonte, mas qualquer representação legível do *software* (requisitos, modelagem, planos e projetos) pode ser inspecionada.

Segundo Sommerville (2007), há três vantagens sobre inspeção de teste:

- Durante os testes, erros podem mascarar (ocultar) outros erros. Uma vez que um erro é descoberto, você nunca pode estar seguro se outras anomalias de saída são devidas a um novo erro ou se são efeitos colaterais do erro já descoberto. Uma única seção de inspeção pode descobrir vários erros.

- b) Versões incompletas de sistemas podem ser inspecionadas sem custos adicionais.
- c) Assim como procurar defeitos de programa, uma inspeção pode também considerar atributos de qualidade mais amplos como conformidade com padrões, portabilidade e facilidade de manutenção. Nesse momento, podem ser localizados, inclusive, algoritmos inapropriados e estilo de programação “pobre”, que dificultaria o processo de manutenção.

A inspeção de software com o objetivo de detecção de defeitos de programação foi formalmente desenvolvida pela IBM na década de 1970.

O processo é realizado por uma equipe independente da de desenvolvedores do produto a ser verificado e com diferentes experiências, que faz revisões linha a linha do código-fonte.

Segundo Sommerville (2007), a diferença fundamental entre inspeção de um programa e outros tipos de revisão de qualidade é que o objetivo especificado na inspeção é encontrar os defeitos de programa. Os defeitos podem ser erros de lógica, anomalias no código que possam indicar situações de não conformidade do software aos padrões do projeto idealizado. Outros tipos de testes, que ainda conheceremos, estão mais relacionados ao cronograma, custos, progresso, marcos de projeto, atendimento de requisitos de clientes etc.

Geralmente a equipe é formada por, pelo menos, quatro pessoas que analisam sistematicamente o código à procura de erros ou defeitos. Esse processo deve ser curto e deve focar detecção de defeitos, conformidade aos padrões. A equipe **não** deve sugerir como resolver inconformidades encontradas, afinal, não é a sua missão.

Todo processo de inspeção deve ser criteriosamente documentado.

Técnicas ou métodos de testes



Para serem eficazes os testes devem ser cuidadosamente planejados e estruturados (estratégia/política de teste).

Durante a realização dos testes, seus resultados devem ser cuidadosamente relatados (documentados) e inspecionados, pois nem sempre os resultados previstos são os obtidos e nem sempre é tão óbvio quando se detecta um erro. Lembre-se de que os erros óbvios foram diagnosticados e adequados durante o processo de depuração feito pelos programadores.

Segundo Paula Filho (2005, p. 184):

[...] testes são indicadores de qualidade do produto, mais do que meios de detecção e correção de erros. Quanto maior o número de defeitos detectados em um software, provavelmente maior também o número de defeitos não detectados. A ocorrência de um número anormal de defeitos em uma bateria de teste indica uma provável necessidade de redesenho dos itens testados.

Devemos lembrar sempre que, em se tratando de testes:

- Se erros graves forem encontrados com regularidade, então a qualidade e a confiabilidade do software são suspeitas.
- Se erros facilmente corrigíveis forem encontrados, então a qualidade e a confiabilidade do software estão aceitáveis, ou os testes são inadequados para revelar erros graves desse software em particular.
- Se não forem encontrados erros, então a configuração de testes não foi suficientemente elaborada e os erros estão escondidos no software.

Baseando-nos nesses parágrafos, podemos então dizer que:

- A atividade de teste é o processo de executar um programa com a intenção de descobrir um erro.
- Se a atividade de teste for conduzida com sucesso, ela descobrirá erros no software.
- Um bom caso de teste é aquele que tem uma elevada probabilidade de revelar um erro ainda não descoberto.
- Um teste bem-sucedido é aquele que revela um erro ainda não descoberto.

Na Figura 3, podemos observar a estrutura das atividades de teste conforme apresentada.

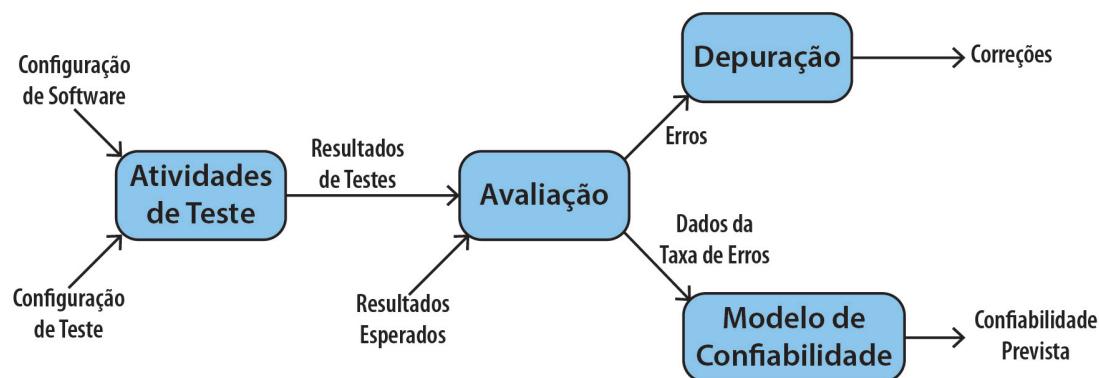


Figura 3 - Estrutura das atividades de teste.

As políticas de configuração de teste agregadas à configuração e documento do software gerarão uma política de atividades de teste que, após executadas, serão verificados os resultados obtidos com os dados esperados e realizada uma avaliação sobre esses dados. A partir daí, a taxa de erros entre o esperado e o obtido será analisada por meio de uma política de confiabilidade (também criada anteriormente no momento da definição da estratégia de teste). Caso a taxa de erro encontrada não seja adequada para o modelo de confiabilidade descrito, os erros serão novamente analisados, depurados e corrigidos.

Estratégias de testes



A partir de agora, veremos modelos de testes de sistemas. Portanto, vale relembrar que **validação** é importante para assegurar que o produto final corresponda aos requisitos do usuário; **verificação** assegura consistência, completitude e corretitude do produto em cada fase e entre fases consecutivas do ciclo de vida do *software*; teste examina o comportamento do produto por meio de sua execução.

Em todo o processo, temos que ter em mente isto:

- **Defeito**, num *software*, é uma deficiência mecânica ou algorítmica que, se ativada, pode levar a uma falha.
- **Falha**, num *software*, é um evento notável em que o sistema viola suas especificações. Geralmente aparece durante a utilização do produto pelos usuários.
- **Erro**, num *software*, é um item de informação ou estado de execução inconsistente. Deve ser corrigido durante a fase de manutenção.

Toda estratégia de teste é válida, desde que considere a verificação de testes de baixo nível (lógica) e testes de alto nível (usabilidade). Deve oferecer a orientação ao profissional, um conjunto de marcos de referência ao administrador, além de ter seu resultado mensurável.

Tipos de estratégias de testes

A realização, com sucesso, da etapa de teste de um *software* deve ter, como ponto de partida, uma atividade de construção de um projeto (dos casos) de teste deste *software*.

Para isso é necessário que:

- Seja conhecida a função a ser desempenhada pelo produto. Os testes devem ser executados para demonstrar que cada função é completamente operacional e sem erro. Esse primeiro princípio, segundo Pressman (2006), deu origem a uma importante abordagem de teste, conhecida como o **teste de caixa preta (black box)**;
- Com base no conhecimento do funcionamento interno do produto, realizam-se testes para assegurar que todas as peças dele estejam completamente ajustadas e realizando a contento sua função. Este segundo princípio, segundo Pressman (2006), é denominado **teste de caixa branca (white box)**, devido ao fato de que maior ênfase é dada ao desempenho interno do sistema (ou do produto).

A. Teste da Caixa Preta

Também chamado de Teste Funcional, é uma abordagem na qual o teste é derivado da especificação de programas ou de componentes. Ele analisa o sistema como uma caixa preta em que seu comportamento só pode ser avaliado por meio da análise das entradas e saídas relacionadas, ou seja, refere-se aos testes que são realizados nas interfaces do software.

O teste é usado para demonstrar que as funções dos softwares são operacionais, que a entrada é adequadamente aceita, a saída é corretamente produzida e que a integridade das informações externas é mantida.

Essa técnica examina aspectos de um sistema **sem** se preocupar muito com a estrutura lógica do software, pois concentra-se nos requisitos funcionais.

O teste procura descobrir erros nas seguintes categorias:

- Funções incorretas ou ausentes.
- Erros de interface.
- Erros nas estruturas de dados ou no acesso a Bancos de Dados (BD) externos.
- Erros de desempenho.
- Erros de inicialização e término.

B. Teste da Caixa Branca ou de Estrutura

O teste tem por objetivo determinar defeitos na estrutura interna do produto com técnicas que exercitem possíveis caminhos e erros de execução.

Nessa técnica, são testados caminhos lógicos através do software, fornecendo casos de testes que colocam à prova conjuntos específicos de condições e/ou laços definidos no sistema.

Ele é também chamado de Teste de Caixa Clara, Teste de Caixa de Vidro e Teste Estrutural. Ele fornece o uso de mais informações sobre o objeto testado do que Teste de Caixa Preta. Esse teste é dispendioso e tem limitada cobertura semântica.

Apesar da importância do Teste de Caixa Branca, não se deve guardar a falsa ideia de que sua realização num produto de software vai oferecer a garantia de 100% de correção deste ao seu final. Isso porque, mesmo no caso de programas de pequeno e médio porte, a diversidade de caminhos lógicos pode atingir um número bastante elevado, representando um grande obstáculo para o sucesso completo desta atividade (PRESSMAN, 2006).

Esse teste baseia-se num minucioso exame dos detalhes procedimentais no qual o “status do programa” pode ser examinado em vários pontos para determinar se o resultado esperado ou estabelecido corresponde ao real.

Devemos ter em mente que testes exaustivos apresentam certos problemas logísticos, porque, mesmo para pequenos programas, o número de caminhos lógicos possíveis pode ser muito grande. Mesmo assim, essa técnica nunca deverá ser descartada, pois cada caso de teste tem sua funcionalidade em momentos específicos estabelecidos pela estratégia de teste definida.

Estratégia de teste ou também chamado de baterias de testes

As políticas de teste podem ser criadas seguindo estas estratégias:

- A) Teste de unidade.
- B) Teste de integridade.
- C) Teste de validação.
- D) Teste de sistema.

Observação: no padrão nomenclatura IEEE, a bateria de teste de validação é chamada de teste de sistema, reservando o termo “teste de validação” para aquele realizado pelo cliente, como parte de um procedimento de aceitação do produto. Por efeitos didáticos, resolvemos definir, separadamente, o que é o teste de validação e de sistemas. Entretanto, você deverá perceber que os conteúdos desses dois grupos são semelhantes.

Nas figuras 4 e 5 podemos observar a estrutura e a relação das estratégias de testes.

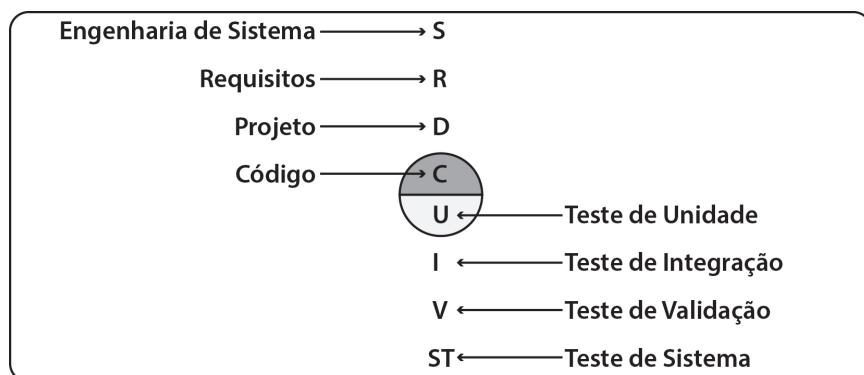


Figura 4 - Estrutura das estratégias de teste.

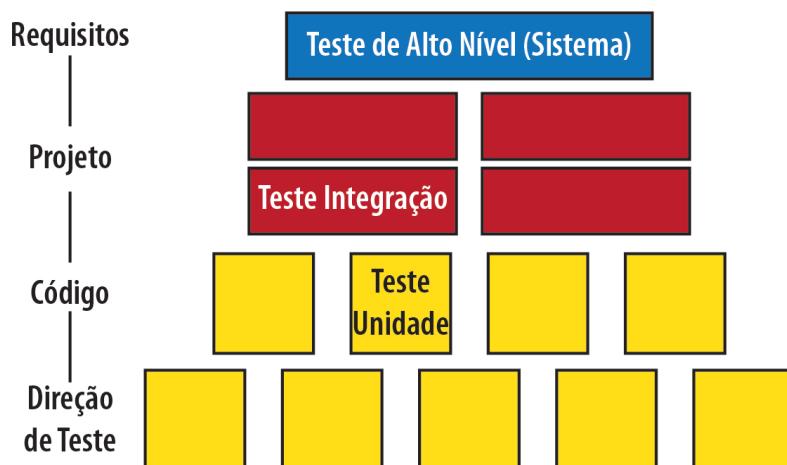


Figura 5 - Estratégia de Teste. No teste de unidade de teste, é verificado o código desenvolvido; no teste de integração, o projeto; e no teste de alto nível, se todos os requisitos funcionais e não funcionais foram satisfeitos.

A. Teste de Unidade

Esse teste tem por objetivo **verificar** os elementos que podem ser testados logicamente. Geralmente, concentra-se em cada unidade do *software*, de acordo com o que é implementado no código-fonte; cada módulo é testado individualmente, garantindo que ele funcione adequadamente.

Geralmente, são utilizadas técnicas de Caixa Branca.

B. Teste de Integração

Tem por objetivo **verificar** as interfaces entre as partes de uma arquitetura do produto. É uma técnica sistemática para a construção da estrutura de programa, realizando, ao mesmo tempo, testes para descobrir erros associados a interfaces e para verificar erros de integração.

O objetivo é, a partir dos módulos testados em nível de unidade, construir uma estrutura de programa que foi determinada pelo projeto.

Utiliza, principalmente, as técnicas de Teste de Caixa Preta.

Os Testes de Integração podem ser feitos por:

Integração incremental: o programa é construído e testado em pequenos segmentos, onde os erros são mais fáceis de serem isolados e corrigidos; as interfaces têm maior probabilidade de serem testadas completamente e uma abordagem sistemática ao teste pode ser aplicada.

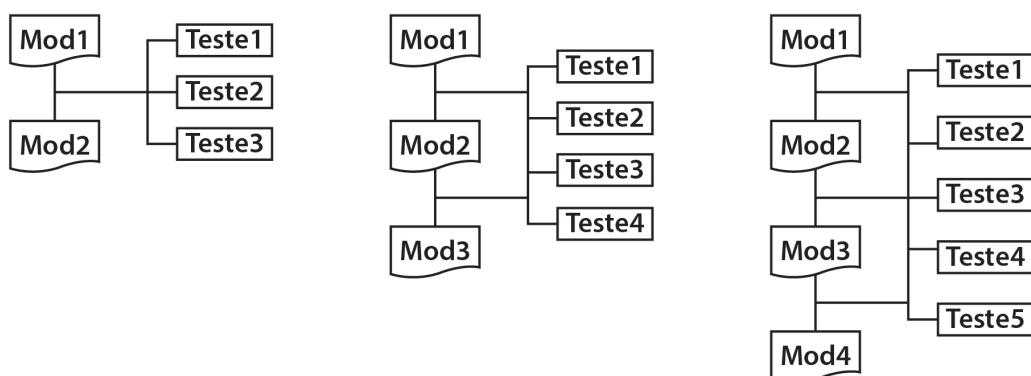


Figura 6 - Teste de Integração incremental.

Fonte: (SOMMERVILLE, 2007, p. 358).

Integração não incremental: abordagem *big-bang*. O programa é testado como um todo, mas prepare-se, o resultado é o caos. Quando erros são encontrados, a correção é difícil porque o isolamento das causas é complicado devido à amplitude por estar sendo testado o programa completo.

Integração top-down: os módulos são integrados movimentando-se de cima para baixo, através da hierarquia de controle. Inicia-se por meio de um módulo de controle principal e a partir dele os outros são incorporados à estrutura de uma maneira *depth-first* (primeiro pela profundidade) ou *breadth-first* (primeiro pela largura).

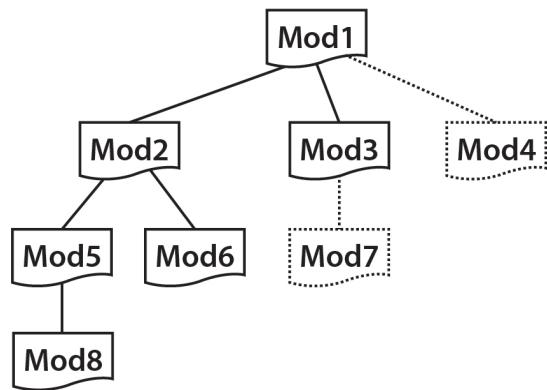


Figura 7 - Integração Top-Down. Depth-first: MOD2, MOD5, MOD8.
Breadth-first: MOD2, MOD3, MOD4.

C. Teste de Validação

Ao término da atividade de teste de integração, o *software* está completamente montado como um pacote, erros de interface foram descobertos e corrigidos e uma série final de testes de *software* – os testes de validação – podem ser inicializados.

Esse teste tem por objetivo **validar** o produto, ou seja, verificar os requisitos estabelecidos como parte da análise de requisitos de software.

É considerado um teste de alto nível, pois verifica se todos os elementos combinam-se adequadamente e se a função/desempenho global do sistema é obtida.

- A validação é bem-sucedida quando o software funciona de uma maneira razoavelmente esperada pelo cliente.
- A especificação de requisitos do software contém os critérios de validação que formam a base para uma abordagem ao Teste de Validação.
- A validação é realizada por meio de uma série de Testes de Caixa Preta que demonstram a conformidade com os requisitos.
- O plano de Teste de Validação visa garantir:
 - Que todos os requisitos **funcionais** sejam satisfeitos.
 - Que todos os requisitos de desempenho sejam conseguidos.
 - Que a documentação (projeto, programador e usuário) esteja correta.
- Requisitos não funcionais (portabilidade, compatibilidade remoção de erros, manutenibilidade etc.) sejam cumpridos.
- Revisão da configuração.
 - Verifica se os elementos de configuração de software foram adequadamente catalogados e desenvolvidos a ponto de apoiar a fase de manutenção.

Quando **um projeto de software é personalizado para vários clientes**, podemos realizar os testes de aceitação por meio das técnicas denominadas Alfa e Beta.

- **Teste Alfa:** o cliente testa o produto **nas instalações do desenvolvedor**. O software é usado num ambiente controlado com o desenvolvedor “olhando sobre os ombros” do usuário e registrando erros e problemas.
- **Teste Beta:** é realizado o teste nas **instalações do cliente** pelo usuário final do software. O desenvolvedor não está presente. O cliente registra os problemas que são encontrados e relata-os ao desenvolvedor em intervalos regulares.

Observação: quando um produto de software é desenvolvido de forma personalizada (para um cliente), não se utiliza o Teste Alfa e nem o Beta. Os testes de aceitação são realizados pelo usuário final para capacitá-lo e para validar todos os requisitos e variam entre um teste informal e uma série de testes planejados que podem ser executados nas dependências do usuário ou do desenvolvedor de acordo com o estipulado entre as partes.

D. Teste de Sistema

Nesta categoria, o *software* e outros elementos do sistema são testados como um todo. Envolve uma série de diferentes testes, cujo propósito primordial é provar completamente o sistema baseado em computador.

Teste de recuperação: um sistema deve ser capaz de tratar determinados erros e voltar à função global sem grandes problemas. Esse teste força o sistema a provocar esses erros para verificar como se comportaria.

Teste de regressão: realiza novos testes previamente executados para assegurar que alterações realizadas em partes do produto não afetem as partes já testadas. Alterações realizadas, especialmente na fase de manutenção, podem introduzir erros nas partes previamente testadas.

Teste de segurança: tenta verificar se todos os mecanismos de proteção embutidos num sistema o protegerão, de fato, de acessos indevidos.

Teste de estresse: também chamado de “teste de tortura”, é utilizado para testar a estabilidade de um sistema. Esse teste direciona a capacidade operacional do sistema a ponto de verificar quando e se acontece o que é denominado ponto de ruptura, ou seja, em qual situação o sistema para de funcionar ou deixa de funcionar de modo seguro.

Teste de desempenho: após o sistema ter sido integrado completamente, é possível testá-lo em relação à sua confiabilidade e desempenho. Testes de desempenho são testes de aplicações de software para garantir que o sistema funcionará como esperado, independentemente da carga de trabalho.

Teste de release: também chamado de teste funcional, garantirá a entrega do sistema ao cliente com funcionalidade, confiabilidade e desempenho especificados e esperados e, principalmente, a inexistência de falhas.

As duas grandes dúvidas quando começamos a conhecer e a perceber a importância dos testes são:

- 1) Quando realizamos testes?**
- 2) Como saber se já testamos o suficiente?**

Uma resposta definitiva é difícil de oferecemos, mas, segundo Sommerville (2007), existe uma resposta pragmática que pode ser dada: jamais será completada uma atividade de teste; na verdade, o que acontece é que são transferidas essas responsabilidades do projetista para a equipe de teste e desta para o cliente. Outra resposta aceitável pode ser que você deverá interromper as suas atividades de teste quando seu tempo provisionado para isso se esgotar ou acabar o dinheiro previsto para essa etapa.

Até agora, apenas apresentamos algumas atividades de teste e reconhecemos sua importância no projeto de software. Entretanto, devido ao nosso tempo, algumas técnicas não puderam ser abordadas. Sugerimos que você pesquise um pouco mais sobre:

- Particionamento de Equivalência ou Partição de Equivalência.
- Teste de caminho (básico).
- Análise do valor limite.
- Testes top-down e bottom-up.
- Testes orientados a objetos.
- Teste de classe de objetos:
 - Teste de integração de objetos.
 - Procure fazer a relação dessas técnicas com as estratégias abordadas.

Manutenção de software

A manutenção de software é o processo geral de modificações de um sistema depois que ele começa a ser utilizado.

A manutenção pode ser simples, corrigir erros de código, ou complexa, com a finalidade de acomodar novos requisitos.

Geralmente, a manutenção não está relacionada a grandes alterações do *software*, estando mais associada à acomodação de novos componentes de sistemas.

A manutenção pode ser considerada a fase mais problemática do ciclo de vida de um *software*, pois os sistemas devem continuar rodando e as alterações são inevitáveis.

Embora na prática não haja muita diferença entre tipos de manutenção, podemos dividi-las nas seguintes categorias:

- **Corretivas:** manutenção para corrigir defeitos no software.
- **Adaptativas:** adaptar o software a um ambiente operacional diferente.
- **Perfectivas:** mudanças para fazer acréscimos a funcionalidades do sistema ou modificá-lo. Geralmente, pode incluir atender aos pedidos do usuário para:
 - Modificar funções existentes.
 - Incluir novas funções.
 - Efetuar melhorias em geral.
- **Melhorar** a manutenibilidade ou confiabilidade futura e fornecer uma base melhor para posterior aprimoramento.

O custo de manutenção deve sempre ser provisionado no valor total do projeto.

Segundo Sommerville (2005, p. 519):

Os custos de manutenção, com uma proporção dos custos de desenvolvimento variam de um domínio de aplicação para outro. Para sistemas de aplicação de negócio [...] os custos de manutenção eram amplamente comparáveis aos custos de desenvolvimento de sistemas. Para os sistemas embutidos de tempo real, os custos de manutenção podem ser até quatro vezes maiores do que os custos de desenvolvimento. Os elevados requisitos de confiabilidade e desempenho desses sistemas podem exigir que os módulos sejam estreitamente vinculados e, como consequência, difíceis de serem modificados.

Sempre a melhor opção para diminuir os custos de manutenção é investir em esforço ao projetar e implementar um sistema, pois é muito mais caro inserir novas mudanças no sistema (adicionar funcionalidades) depois da entrega, do que durante o desenvolvimento. “Boas técnicas de Engenharia de Software,, como a especificação precisa, o uso do desenvolvimento orientado a objetos e do gerenciamento de configurações, contribuem para a redução do custo de manutenção”. (SOMMERVILLE, 2005, p. 519)

Existem alguns problemas clássicos que podem interferir numa boa manutenção após a entrega do sistema para o cliente:

- É impossível ou difícil traçar a evolução do software devido às várias versões existentes.
- As alterações feitas não são adequadamente documentadas.
- É difícil rastrear o processo pelo qual o sistema foi criado.
- É muito difícil entender programas de outras pessoas, principalmente quando elas não estão por perto para explicar.
- Documentação inexistente, incompreensível ou desatualizada.
- A maioria dos softwares não foi projetada para apoiar alterações.
- A manutenção não é vista como trabalho valorizado e sim como um retrabalho.

A manutenção varia de acordo com o tipo do software, dos processos de desenvolvimento e do pessoal envolvido no processo. Algumas vezes, ela pode surgir de conversas informais, o que pode levar a um processo de **manutenção informal**. Entretanto, o mais adequado e utilizado em grandes projetos é a **manutenção formal**, com documentos específicos descrevendo etapas e características de cada fase ou etapa.

- a) Basicamente, podemos dividir o processo de manutenção em etapas:
- b) O usuário, a gerência ou clientes solicitam a mudança.
- c) Se as mudanças forem aceitas, um novo *release* do sistema será planejado. Durante esse planejamento, todas as solicitações e implicações são analisadas e o procedimento de alteração idealizado. Nessa etapa, os novos requisitos do sistema que foram propostos são analisados e validados.
- d) Mudanças são implementadas e validadas em uma nova versão do sistema.
- e) Nova versão do sistema é liberada.

Tais etapas podem melhor ser compreendidas na Figura 9.

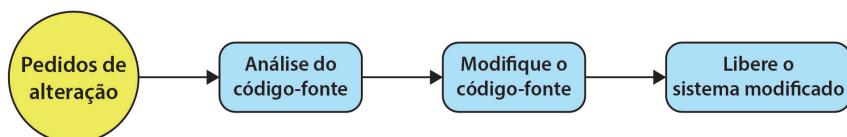


Figura 8 - Processo de reparos de emergência.

Fonte: Sommerville (2005, p. 522)

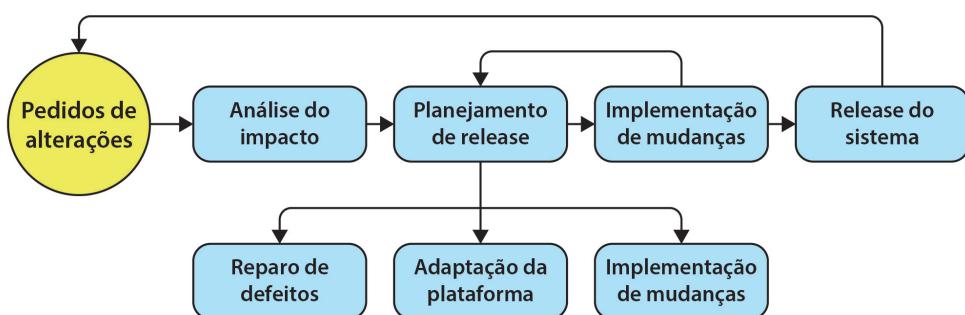


Figura 9 - Visão geral do processo de mudança.

Fonte: Sommerville (2005, p. 522).

Claro que há mudanças. Elas implicam o bom funcionamento imediato do sistema. Embora seja feito com um pouco menos de burocracia, o processo de análise e validação das alterações é sempre fundamental para que uma versão adaptada do sistema seja liberada para o cliente e não cause maiores danos. Entretanto, para assegurar que os requisitos, o projeto de software e o código se tornem inconsistentes, essas mudanças, assim como as categorizadas informais, devem sempre ser documentadas e a documentação geral do projeto de software ser atualizado (Figura 10).

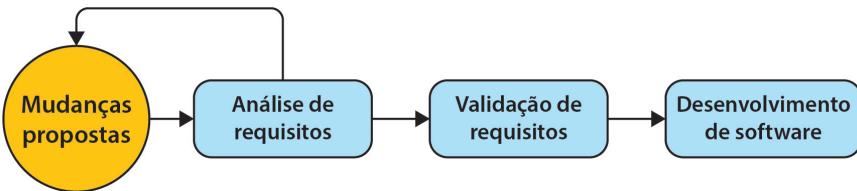


Figura 10 - Implementação de mudanças.

Fonte: Sommerville (2005, p. 522).

Gerentes odeiam surpresas. Dessa forma, uma boa equipe de desenvolvimento sempre procurará prever quais mudanças no sistema possivelmente serão solicitadas, quais partes podem causar maior dificuldade de alteração.

Segundo Sommerville (2005), considerando as previsões de mudanças, podemos dividi-las em 3 e, a cada uma delas, associá-las a determinadas perguntas:

- A) Previsão de facilidade de manutenção: que partes do sistema serão mais dispendiosas para manter?
- B) Previsão de mudanças no sistema: que partes do sistema provavelmente serão mais afetadas pelos pedidos de mudanças? Quantos pedidos de mudanças podem ser esperados?
- C) Previsão de custo de manutenção: quais serão os custos de manutenção durante o tempo de vida útil do sistema? Quais serão os custos de manutenção do sistema no próximo ano?

Prever as respostas a essas perguntas garantirá um maior controle das consequências que podem ser geradas, no sistema e na documentação, a partir da solicitação de mudanças.

Conclusão

Como podemos observar, planejar testes e manutenção faz parte dos conceitos fundamentais da idealização de um sistema.

Realizar testes está muito além de apenas compilar: está relacionado ao perfeito funcionamento, num prazo longo, de um sistema.

Entretanto, embora projetemos sistemas para ter uma vida, relativamente, longa, manutenções sempre ocorrerão. Muitas serão solicitadas pelos usuários e temos que avaliar a pertinência desses pedidos, outras serão oferecidas pelo próprio fabricante do produto com o objetivo de atualizar e melhorar a eficiência do produto de software.



Importante

Não importa como essas mudanças, ou casos de testes, são solicitados, o importante é que tanto um quanto o outro façam parte de atividades bem planejadas e principalmente, quando executadas, sejam adequadamente documentadas.

Material Complementar

Como nesta unidade abordamos os conceitos gerais da Engenharia de Software, nossa sugestão de material complementar é:

Teste:

- DIAS NETO, A. C. **Introdução aos testes de software**. Disponível em: <http://www.comp.ita.br/~mluisa/TesteSw.pdf>

Manutenção:

- ESPINDOLA, R. S.; MAJDENBAUM, A.; AUDY, J. L. N. **Uma análise crítica dos desafios para Engenharia de Requisitos em Manutenção de Software**. Disponível em: http://wer.inf.puc-rio.br/WERpapers/artigos/artigos_WER04/Rodrigo_Espindola.pdf

Referências

Bibliografia fundamental

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011.

Bibliografia básica

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação**. 4. ed. Rio de Janeiro: LTC, 1999.

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação Gerenciais**. Administrando a empresa digital. 5. ed. São Paulo: Pearson Education do Brasil, 2006.

PFLEEGER, S. L. **Engenharia de Software**: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 2006.

SOMMERVILLE, I. **Engenharia de Software**. 6. ed. São Paulo: Pearson Addison Wesley, 2005.

Bibliografia complementar

ALCADE, E.; GARCIA, M.; PENUELAS, S. **Informática Básica**. São Paulo: Makron Books, 1991.

FAIRLEY, R. E. **Software engineering concepts**. New York: McGraw-Hill, 1987.

IEEE. **Software Engineering Standards**. 2013. Disponível em: <http://www.ieee.org/portal/innovate/products/standard/ieee_soft_eng.html>. Acesso em: 10 dez. 2013.

LUKOSEVICIUS, A. P.; CAMPOS FILHO, A. N.; COSTA, H. G. **Maturidade em gerenciamento de projetos e desempenho dos projetos**. Disponível em: <http://www.producao.uff.br/conteudo/rpep/.../RelPesq_V7_2007_07.doc>. Acesso em: 12 nov. 2013.

MAFFEO, B. **Engenharia de Software e especialização de sistemas**. Rio de Janeiro: Campus, 1992.

MICHAELIS. **Moderno dicionário da língua portuguesa**. São Paulo: Cia. Melhoramentos, 1998.

PARREIRA JÚNIOR, W.M. **Apostila de Engenharia de Software**. Disponível em: <http://www.waltenomartins.com.br/ap_es_v1.pdf>. Acesso em: 13 nov. 2013.

PAULA FILHO, W. P. **Engenharia de Software**: fundamentos, métodos e padrões. 2. ed. Rio de Janeiro: LTC, 2001.

Revista Engenharia de Software. Disponível em: <<http://www.devmedia.com.br/revista-engenharia-de-software-magazine>>. Acesso em: 12 nov. 2013.

VONSTA, A. **Engenharia de Programas.** Rio de Janeiro: LTC, 1983.

WIENNER, R.; SINCOVEC, R. **Software engineering with Modula 2 and ADA.** New York: Wiley, 1984.

WIKIPEDIA (2007a). **ISO 9000.** Disponível em: <http://pt.wikipedia.org/wiki/ISO_9000>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007b). **Melhoria de processo de Software brasileiro.** Disponível em: <<http://pt.wikipedia.org/wiki/MPS.BR>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007c). **CMMI.** Disponível em: <<http://pt.wikipedia.org/wiki/Cmmi>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007d). **Engenharia de Software.** Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software>. Acesso em: 1 fev. 2007.

Anotações





Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

www.cruzeirodosulvirtual.com.br

Campus Liberdade
Rua Galvão Bueno, 868
CEP 01506-000
São Paulo SP Brasil
Tel: (55 11) 3385-3000



Universidade
Cruzeiro do Sul



UNICID
Universidade
Cidade de S. Paulo



UNIFRAN
Universidade
de Franca

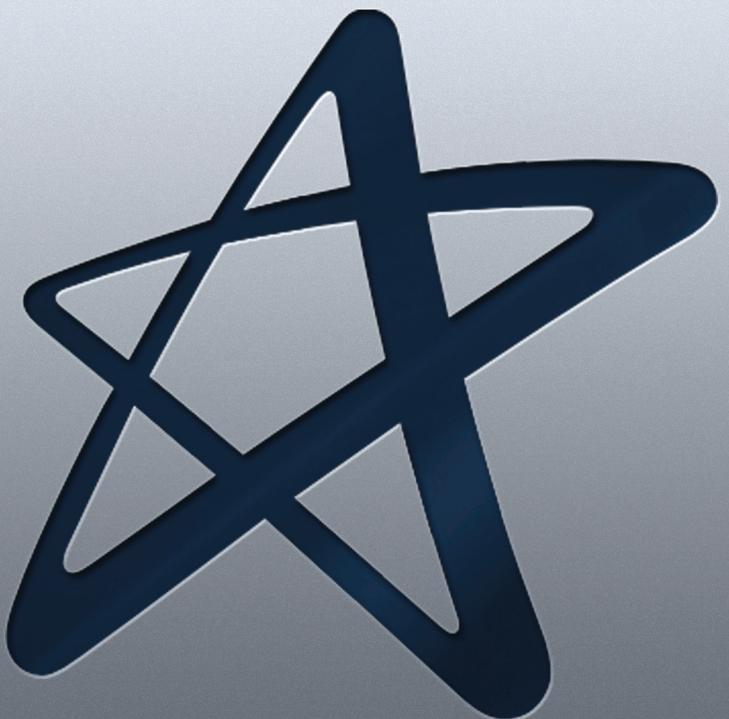


UDF
Centro
Universitário



Módulo
Centro
Universitário

Engenharia de Software



Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

Material Teórico



Ferramentas CASE e qualidade de *software*

Responsável pelo Conteúdo:

Prof.^a Dr.^a Ana Paula do Carmo Marchetti Ferraz

Revisão Textual:

Prof.^a Me. Luciene Oliveira da Costa Santos

UNIDADE

Ferramentas CASE e qualidade de software



- Introdução
- Ferramentas CASE
- Qualidade de software



Objetivo de APRENDIZADO

- Compreender as características relacionadas às ferramentas CASE.
- Refletir a importância da qualidade de software.
- Perceber a existência e as peculiaridades das técnicas e estratégias que focam a qualidade do software.
- Compreender que a qualidade do produto é dependente da qualidade do processo.
- Conhecer modelos para a melhoria da qualidade do software.

Este guia de estudo sobre a forma de organizar suas atividades para cumprir os propósitos da disciplina *online* objetiva salientar algumas informações importantes sobre esta unidade, bem como sobre o estudo *online* em si.

Organize-se também de forma a não deixar para o último dia a realização das atividades (AS e AP), pois podem ocorrer imprevistos. Encerrada a unidade, encerra-se a possibilidade de obter a nota relativa a cada atividade.

Para ampliar seu conhecimento, bem como aprofundar os assuntos discutidos, pesquise, leia e consulte os livros indicados nas Referências.



A Bibliografia Fundamental para esta Disciplina é: SOMMERVILLE, Ian. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011. A Bibliografia Complementar está indicada em item específico em cada unidade.

Contextualização

Discutiremos as questões relacionadas à produção de *software*, assim como métricas para garantir as qualidades de *software*, não são tarefas fáceis, entretanto, são fundamentais para satisfação do cliente ou dos usuários.

Nesta unidade, abordaremos as ferramentas que automatizam o trabalho do desenvolvedor de *software* durante as diversas fases do processo produtivo e até que ponto estas ferramentas interferem na produtividade da indústria do *software*, além das características de qualidade do produto e do processo de *software*, mostrando aspectos e fatores que afetam a qualidade do *software*. Além disso, você conhecerá as principais normas que visam à melhoria da qualidade de *software*.

Entender esses conceitos ajudará a compreender melhor os aspectos relacionados à Engenharia de *Software* apresentados nesta disciplina.

Introdução



Todos os assuntos relacionados à Engenharia de *Software* estão integrados a métodos, ferramentas e procedimentos para o desenvolvimento de produtos de *software* com qualidade.

As ferramentas proporcionam um apoio automatizado ou semiautomatizado aos métodos. Quando as ferramentas são integradas de forma que a informação criada por uma ferramenta possa ser usada por outra, é estabelecido um sistema de suporte ao desenvolvimento de *software* chamado de Engenharia de *Software* Auxiliada por Computador, ou CASE, do inglês *Computer-Aided Software Engineering* (PRESSMAN, 1995).

As ferramentas proporcionam um apoio automatizado aos métodos. Hoje em dia, já existem ferramentas capazes de sustentar todas as etapas e métodos relacionados às várias fases do processo de desenvolvimento de *software* de forma automatizada.

Uma delas é a Ferramenta CASE. Nesta unidade, *não apenas apresentaremos as questões históricas e funcionais da ferramenta*, como também daremos início a conceitos relacionados à qualidade de produto de *software*.

Qualidade pode ser um conceito subjetivo – o que é qualidade para um, pode não ser para outro. Entretanto, para medirmos a qualidade de um *software*, conceitos subjetivos não podem ser considerados.

Para que possamos excluir ou definir a falta parcial ou total da qualidade num produto de *software*, é necessário que haja uma definição precisa do que é qualidade ou, pelo menos, quais são as propriedades relacionadas a partir dos princípios da Engenharia de *Software*.

Nessa linha de raciocínio é que discutiremos os pontos de qualidade, inclusive apresentando alguns conceitos e padrões definidos pelo Instituto de Engenharia de *Software* – *Software Engineering Institute* (SEI) – que, dentre tantos modelos, definiu o de certificação de qualidade de *software* – *Capability and Maturity Model Integrator* (CMMI) – que veremos, em linhas gerais, mais adiante nesta unidade.

Ferramentas CASE



Antes de apresentarmos as características das ferramentas CASE, vamos ler algumas definições, de diferentes autores, para essas ferramentas:

- CASE refere-se a uma ampla gama de diferentes tipos de programas utilizados para apoiar as atividades do processo de *software*, como análise de requisitos, modelagem de sistema, depuração e testes (SOMMERVILLE, 2011).
- CASE é uma ferramenta ou conjunto de técnicas facilitadoras de desenvolvimento de *software* moderno, que utiliza técnicas para ajudar no trabalho dos engenheiros de *software* (REZENDE, 2005).
- Uma ferramenta CASE é um instrumento ou sistema automatizado utilizado para realizar uma tarefa da melhor maneira possível. Essa melhor maneira pode significar que a ferramenta nos torna mais precisos, eficientes e produtivos, ou que exista melhora na qualidade do produto resultante (PFLEEEGER, 2004).

De acordo com Sommerville (2011), os engenheiros fazem os produtos funcionarem, aplicando teorias, métodos e ferramentas nas situações apropriadas de modo seletivo. O *software* não é apenas um programa de computador, é, também, toda a documentação associada e os dados necessários para fazer com que esses programas funcionem corretamente.

A Engenharia de *Software* não cuida apenas do desenvolvimento de um *software*, mas também do desenvolvimento de novas ferramentas ou da melhoria das ferramentas que já existem para suporte e apoio ao *software*.

Então, chegamos à seguinte pergunta: ferramentas CASE são softwares que auxiliam a desenvolver novos softwares?



Sugerimos que você pare alguns minutos e tente formular sua própria resposta sobre isso.

Você pode perceber que o objetivo de qualquer indústria é satisfazer às necessidades de seus clientes entregando produtos com qualidade e aumentando, assim, a produtividade de seus processos de produção. Na indústria de *software*, esses objetivos não são diferentes.

A Engenharia de *Software* tem como objetivo o aperfeiçoamento da qualidade dos softwares desenvolvidos e o aumento da produtividade dos engenheiros que os desenvolvem, visando sistematizar sua manutenção, de modo que aconteça dentro de prazos estimados, com progresso controlado e usando métodos, tecnologias e processos em contínuo aprimoramento (REZENDE, 2005).

Passamos, no início da década de 1970 pela Crise do *Software*. Crise do *Software* é o termo que resume todos os problemas que permearam o desenvolvimento de *software* e que, de alguma forma, a Engenharia de *Software*, com seus processos e procedimentos estruturados tentou minimizar e, como muitos autores afirmam, colocou ordem numa atividade que aparentemente era caótica (atividade de desenvolver um *software*).

Um dos sintomas da crise – alguns dos quais permeiam até os dias atuais – é a dificuldade de suprir a demanda por novos softwares. As ferramentas CASE são promessas da Engenharia de Software para automatizar tarefas, diminuir o tempo de desenvolvimento e, assim, atender melhor à crescente demanda por novos softwares, sem contudo esquecer da qualidade necessária ao produto.

Em outras palavras, a preocupação em desenvolver ferramentas que automatizam o trabalho de engenheiros de software é uma das tentativas de aumentar a produtividade e a qualidade da indústria de software.

Contexto histórico

Pressman (1995) lembra-nos do velho ditado sobre os filhos do sapateiro, ele passa a maioria do tempo fazendo sapatos para os outros e seus próprios filhos não têm sapatos feitos por ele.

Esse ditado é análogo ao ditado popular “em casa de ferreiro, o espeto é de pau”.

Segundo Pressman (1995), nos últimos 20 anos, a grande maioria dos engenheiros de softwares tem sido como os filhos do sapateiro: constroem softwares, sistemas complexos que automatizam o trabalho para os outros, mas, para si mesmos, não têm usado quase nada para automatizar o ambiente de trabalho. Recentemente, a Engenharia de Software era fundamentalmente uma atividade manual em que ferramentas eram usadas somente em últimos estágios do processo.

Por volta da década de 1950, engenheiros utilizavam régulas de cálculo e calculadoras mecânicas, livros e tabelas que continham as fórmulas e algoritmos que precisavam para efetuar a análise de um problema de engenharia. Tudo era feito manualmente, com qualidade, mas manualmente.

Passou uma década e esses engenheiros começaram a experimentar os benefícios da informática, principalmente do computador, na realização destas tarefas.

Já na década de 1970, todas as fórmulas que os engenheiros necessitavam estavam embutidas em um conjunto de programas de computador, usados para analisar uma ampla variedade de problemas, o que tornou inevitável o uso de tal ferramenta no dia a dia da profissão. Foi nesse contexto que a tecnologia começou a estreitar seus laços com o processo de manufatura, tendo os computadores como ferramenta oportunizadora do processo. Assim surgiu o primeiro elo entre o projeto auxiliado por computador – *Computer-Aided Design (CAD)* – e a manufatura auxiliada por computador – *Computer-Aided Manufacturing (CAM)* (PRESSMAN, 1995).

Com o passar dos anos e a institucionalização da produção de software como indústria, essas necessidades se intensificaram e, finalmente, os engenheiros de software ganharam sua primeira ferramenta auxiliada por computador, a Engenharia de Software Auxiliada por Computador – *Computer-Aided Software Engineering (CASE)*.

Atualmente, as ferramentas CASE fazem parte do dia a dia de trabalho do engenheiro de software. Por meio dela foi possível automatizar as atividades e melhorar a produtividade no desenvolvimento do produto de software (PRESSMAN, 2011).

A definição do que é produtividade na indústria de software ainda é controversa, pois a produtividade em software não é uma medida direta. Assim como nas outras engenharias, a

Engenharia de Software propõe algumas métricas para deter dados tangíveis sobre o processo e o produto de software, como, por exemplo, linhas de código – *Line-of-Code* (LOC) – e pontos por função – *Function-Point* (FP) –, que veremos adiante (PRESSMAN, 2011). Entretanto, há, no mínimo, cinco fatores que interferem na produtividade do software, são eles:

- » **Fatores humanos:** o tamanho e a experiência da organização de desenvolvimento.
- » **Fatores do problema:** a complexidade do problema a ser resolvido e o número de mudanças nos requisitos ou restrições de projeto.
- » **Fatores do processo:** técnicas de análise e projeto que são usadas, como: linguagens e ferramentas CASE disponíveis e técnicas de revisão.
- » **Fatores do produto:** confiabilidade e desempenho do sistema baseado em computador.
- » **Fatores relacionados ao recurso:** disponibilidade de ferramentas CASE, recursos de hardware e software.

Portanto, a disponibilidade de ferramentas CASE é considerada relevante, influenciando em até 40%, segundo Pressman (2011), na busca pelo aumento da produtividade do processo de software.

Para Sommerville (2005), as ferramentas de tecnologia CASE possuem facilidades gráficas para o planejamento, projeto e construção de sistemas. Elas podem ser utilizadas para gerar um esboço do programa, a partir de um projeto. Isso inclui código, implementação interfaces e, em vários casos, o desenvolvedor precisa apenas acrescentar pequenos detalhes da operação de cada componente do programa. Elas também podem incluir geradores de códigos, que, automaticamente, originarão código-fonte com base no modelo de sistema, e, também, algumas orientações de processo, que fornecem conselhos ao engenheiro de software sobre o que fazer em seguida.

Classificação das CASE

De acordo com Sommerville (2005), em uma perspectiva de processo, isto é, quanto às fases do processo que a ferramenta automatiza, as CASE podem se dividir em três categorias:

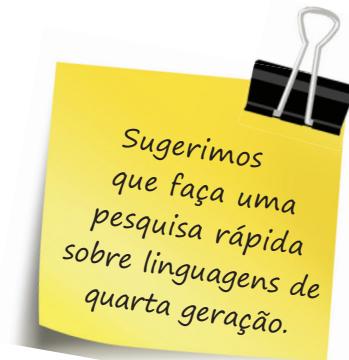
- **Front End ou Upper-CASE:** são aquelas ferramentas que dão apoio à análise e ao projeto, isto é, às fases iniciais do desenvolvimento do software.
- **Back End ou Lower-CASE:** são aquelas ferramentas destinadas a dar apoio à implementação e aos testes, como depuradores, sistemas de análise de programa, geradores de casos de testes e editores de programas.

- **I-CASE ou Integrated CASE:** são as ferramentas que têm como objetivo unir a Upper-CASE à Lower-CASE, isto é, cobrem todo o ciclo de vida do software.

Já Pressman (1995), em uma perspectiva de função, ou seja, de acordo com sua função específica que automatiza, as CASE podem ser classificadas em oito ferramentas. Observe a seguir a descrição de cada uma.

Ferramenta de planejamento de sistemas comerciais:

essa ferramenta tem como objetivo melhorar a compreensão de como a informação circula entre as várias unidades organizacionais. É considerado um “meta-modelo”, com base na qual sistemas de informação específicos são derivados. A informação comercial é considerada e modelada a partir da forma como circula pelas entidades comerciais não sendo considerada como requisito.



Ferramenta de gerenciamento de projetos: a maioria concentra-se em um elemento específico do gerenciamento ao invés de apoiar diversas atividades simultâneas. Ao ser usado um conjunto selecionado de ferramenta CASE, o gerente de projetos pode gerar estimativas de esforço, custo e duração, além de definir a estrutura de divisão de trabalho, planejar o cronograma e acompanhar os projetos continuamente. Além disso, essa ferramenta pode ser usada para compilar métricas e rastrear os requisitos.

Ferramenta de apoio: essa categoria, como o nome já diz, é de apoio. Suas funcionalidades são, dentre outras, apoiarativamente atividades de suporte, tais como a documentação de rede e de sistema, de garantia de qualidade, de gerenciamento de banco de dados e de configuração.

Ferramentas de análise e projeto: possibilitam que o engenheiro de software crie um modelo do sistema que será construído. Essas ferramentas também auxiliam na criação do modelo e na avaliação da qualidade do modelo.

Ferramenta de programação: são aquelas relacionadas aos compiladores, editores e depuradores que são necessárias nas atividades de desenvolvimento. Estão nessa categoria as linguagens denominadas de quarta geração, os geradores de aplicações e as linguagens de consulta a banco de dados.

Ferramenta de integração e teste: auxilia na aquisição de dados de testes, na análise do código-fonte, no planejamento, no gerenciamento e controle de testes.

Ferramenta de prototipação: dá suporte à criação de modelos para prototipação.

Ferramenta de manutenção: auxilia na execução de engenharia reversa, análise e reestruturação de código e na reengenharia.

Veja que há várias classificações para as ferramentas CASE, cada uma relacionada à funcionalidade específica (manutenção, prototipação, apoio etc.). Esta classificação pode ser feita tanto pela sua função quanto pelos papéis que desempenha como suporte aos desenvolvedores, gerentes e outros integrantes da equipe.

A Engenharia de Software Auxiliada por Computador pode ser tão simples quanto uma única ferramenta que suporte uma atividade de Engenharia de Software específica ou tão complexa quanto um ambiente completo que abrange ferramentas, banco de dados, pessoas, hardware, rede, sistemas operacionais, padrões e uma infinidade de outros componentes (PRESSMAN, 2011). Enfim, todas as ferramentas “softwares” que, de alguma forma, auxiliam nos trabalhos de um engenheiro de software, podem ser consideradas como CASE.

Há várias classificações de ferramentas CASE que apoiam as diversas fases do processo de software. Para cada uma dessas classificações, há, no mercado, inúmeras ferramentas disponíveis. Entre elas, podemos citar: *Poseidon, Rational, ErWin, Oracle Designer, Genexus, Clarify, Dr.CASE, MultiCASE, Paradigm, PowerDesigner, Together, Cognos, CoolGen, Smart, Theseus, BPWin, Arena, Visio, Brio, Microstrategy*.



Informação

O SEI (*Software Engineering Institute – Instituto de Engenharia de Software*), *Carnegie Mellon University, Pittsburgh, Pennsylvania*, Estados Unidos, desenvolveu um processo de adoção de ferramentas CASE. O modelo tem como postulados seis estágios para um processo de adoção de ferramentas CASE.



Explore

Mais informações sobre o modelo SEI podem ser encontradas no site do Instituto Nacional de Pesquisas Espaciais. Disponível em: www2.dem.inpe.br/ijar/GuiaCASE.doc. Acesso em: 5 jul. 2010.

Idealmente, as CASE têm como promessas:

- encorajar um ambiente interativo;
- reduzir custos de manutenção;
- melhorar a qualidade do produto de software;
- agilizar o processo de desenvolvimento;
- aumentar a produtividade.

Com os seguintes benefícios decorrentes da utilização de ferramentas CASE:

- CASE de gerenciamento de configuração e documentação são, geralmente, mais aceitas como mecanismo de melhoria do software;
- benefícios controversos de CASE de análise e projeto, engenharia reversa e ferramentas de geração de códigos disponíveis comercialmente;

- ganhos variando de 10% a 30% resultante do uso de CASE na análise e projeto;
- ganhos verdadeiros ocorrem somente depois de um ou dois anos de experiência;
- ganhos variáveis de produtividade;
- modestos ganhos de qualidade;
- documentação melhorada (aumento da manutenibilidade);
- melhoria na comunicação;
- imposição de metodologia e padrões.

Como os benefícios das CASE são ainda relativos, as empresas que se dispuserem a adquirir uma ferramenta para automatizar o processo de *software*, devem considerar alguns itens da CASE em questão, como, por exemplo:

- custo (investimentos) para adotar a tecnologia CASE;
- consistência entre os processos e métodos suportados pelas ferramentas CASE e os processos e métodos utilizados na organização;
- mecanismos de suporte necessários para ferramentas CASE (por parte do fornecedor);
- limites da ferramenta quanto ao tamanho do projeto;
- complexidade da adoção e usabilidade mínima. Complexidade dos processos de adoção das ferramentas CASE;
- capacidade de acomodar mudanças, uma vez que os requisitos se modificam;
- permissão da engenharia reversa dos softwares desenvolvidos sem usar uma ferramenta CASE.

O sucesso ou falha do esforço de adoção da CASE depende muito da habilidade de uma organização para gerenciar custos de curto e de longo prazo.

Portanto, devem ser consideradas algumas implicações de curto prazo:

- um potencial decaimento na produtividade;
- insatisfação de parte dos funcionários ao adotar a nova tecnologia;
- mudanças nos processos e métodos;
- treinamento potencialmente extensivo;
- custos significativos.

Apesar de serem consideradas, muitas vezes, ferramentas de custo elevado e de utilização complexa, uma vez inserida e institucionalizada no processo de desenvolvimento de *software*, certamente, trarão benefícios ao projeto por meio do aumento da produtividade.

Assim, a Engenharia de Software oferece métodos e técnicas para desenvolver ferramentas automatizadas para auxiliar no trabalho de profissionais das mais diversas áreas de atuação. Tais ferramentas – softwares – são bem aceitas, prova disso é a crescente busca por novos e cada vez mais complexos sistemas.

Qualidade de software



Vimos no início deste Material Teórico que as ferramentas CASE auxiliam no processo de desenvolvimento de produtos de *software* com qualidade, entretanto, durante muito tempo, as questões de qualidade eram subjetivas e, aos poucos, esse ambiente foi sendo alterado.

Quando falamos em certificação de qualidade em empresas, geralmente pensamos em certificações ISO, especificamente as relacionadas à ISO 9000 e suas derivações.

Certificações ISO possuem restrições, listas de atributos e níveis que as empresas devem obter relacionadas à qualidade para que possam ser certificadas e tudo parte do princípio da definição do que é qualidade dentro do contexto a ser avaliado.

Segundo a **norma ISO 9000**, versão 2000, a qualidade é o grau em que um conjunto de características inerentes a um produto, processo ou sistema cumpre os requisitos inicialmente estipulados para estes (WIKIPEDIA, 2007a).

Com relação aos produtos de *software* isso não será diferente. Precisamos verificar o que é qualidade nesse contexto para entendermos porque empresas “correm tanto” atrás de um *software* com qualidade.

Para Pressman (2011), Qualidade de *Software*, que é uma área que pertence à Engenharia de *Software*, objetiva atingir e garantir a qualidade final do produto, por meio das definições e normatizações dos processos de desenvolvimento.

Apesar dos diversos modelos aplicados na questão “Qualidade de *Software*” atuarem durante todo o processo de desenvolvimento, o foco principal está na satisfação do cliente ao receber o produto pronto. É importante que se procure garantir que o sistema cumpra com todas as especificações acordadas anteriormente entre a empresa desenvolvedora e o cliente. É nesse momento que percebemos a importância de uma definição de requisitos bem elaborada.

Entretanto, é uma visão simplista dizer que avaliação de qualidade de um *software* só pode ser feita depois que o *software* foi entregue ao cliente. Deve-se garantir a qualidade desde o início da construção do *software*, pois controlamos a sua fabricação passo a passo e medimos a sua qualidade antes que ele saia da fábrica.

Você provavelmente já ouviu falar que a qualidade das partes garante a qualidade do todo; é mais ou menos isso que acontece: ao garantirmos a qualidade das partes do *software*, estamos garantindo a qualidade total do produto.

Existem fatores internos e externos que estão relacionados à qualidade.

- Fatores de qualidade externa são aqueles que podem ser percebidos por pessoas fora da equipe de desenvolvimento – cliente ou eventuais usuários. A partir da observação de fatores específicos, o cliente pode perceber a qualidade ou não do produto de *software*. Enquadram-se nesta classe de fatores de qualidade externa: desempenho, facilidade de uso, correção, confiabilidade, extensibilidade, dentre outros.
- Fatores de qualidade interna são aqueles que estão mais relacionados à visão de um programador, particularmente aquele que vai assumir as tarefas de manutenção do *software*. Enquadram-se nesta classe de fatores de qualidade interna: modularidade, legibilidade, portabilidade, manutenibilidade, dentre outros.

Mesmo observando que são diferentes entre si, é a garantia da qualidade de fatores externos e internos que nos assegura um bom produto de *software*.

Fatores de qualidade de *software*

Os fatores que afetam a qualidade de *software* podem ser categorizados em:

- Fatores que podem ser medidos diretamente (erros de execução).
- Fatores que podem ser medidos apenas indiretamente (usabilidade do *software*).

Vejamos alguns fatores que devem ser considerados (PRESSMAN, 2006, 2011). Para cada um dos fatores apresentados, algumas perguntas podem ser realizadas para perceber a existência ou não deles:

Corretitude (Ele faz aquilo que eu quero?)

É a capacidade dos produtos de *software* de executarem suas funções precisamente, conforme definido nos requisitos e na especificação.

Confiabilidade (Ele se comporta com precisão o tempo todo?)

É a capacidade de o sistema funcionar mesmo em condições anormais. É um fator diferente da corretitude, pois um sistema pode ser correto sem ser confiável, ou seja, ele funciona, mas não o tempo todo e em todas as condições.

Flexibilidade (Posso mudá-lo?)

É a facilidade com a qual podem ser introduzidas modificações nos produtos de *software*. Todo *software* é considerado, em princípio, “flexível” e, portanto, passível de modificações. No entanto, esse fator nem sempre é muito bem entendido, principalmente em se tratando de pequenos programas.

Por outro lado, para softwares de grande porte, esse fator atinge uma importância considerável e pode ser alcançado a partir de dois critérios importantes:

- A **simplicidade de projeto**, ou seja, quanto mais simples e clara a arquitetura do *software*, mais facilmente as modificações poderão ser realizadas.
- A **descentralização**, que implica na maior autonomia dos diferentes componentes de *software*, de modo que a modificação ou a retirada de um componente não implique uma reação em cadeia que altere todo o comportamento do sistema, podendo inclusive introduzir erros antes inexistentes.

Reusabilidade (Serei capaz de reutilizar parte do software?)

É a capacidade de os produtos de *software* serem reutilizados, totalmente ou em parte, para novas aplicações. Esse é um conceito fundamental nos dias de hoje.

Essa característica de reusabilidade é uma necessidade e vem da observação de que alguns dos componentes de *software* obedecem a um padrão comum de execução, restrições e requisitos, o que permite que essas similaridades possam ser exploradas e incorporadas para solucionar outras classes de problemas.

Esse fator permite, principalmente, diminuir o tempo de desenvolvimento de um *software*, gerando economia e qualidade maiores, ou seja, ao utilizarmos objetos (programas ou parte deles) já desenvolvidos, menos algoritmos precisam ser escritos, o que significa menos esforço e menor risco de ocorrência de erros.

Compatibilidade (Serei capaz de compor uma interface com outro sistema?)

A compatibilidade corresponde à facilidade com a qual produtos de *software* podem ser combinados com outros. Esse é um fator relativamente importante, dado que um produto de *software* é construído (e adquirido) para trabalhar convivendo com outros *softwares*.

Eficiência (Ele rodará em meu hardware tão bem quanto possível?)

A eficiência está relacionada com a utilização racional dos recursos de hardware e de sistema operacional da plataforma onde o *software* será instalado.

Portabilidade (Serei capaz de utilizá-lo em outra máquina?)

A portabilidade consiste na capacidade de um *software* em ser instalado para diversos ambientes operacionais e de hardware. Por termos várias plataformas de processadores e sistemas operacionais, essa não é uma característica fácil de ser atingida.

Usabilidade (Ele foi projetado para o usuário?)

Esse fator é certamente um dos mais fortemente detectados pelos usuários do *software*, ou seja, é medido por meio da facilidade de se utilizar o produto.

Manutenibilidade (Posso consertá-lo?)

Esse fator é relacionado ao esforço exigido para localizar e reparar erros num programa, além de adequá-lo a novas versões e atualizá-lo de forma eficaz e eficiente.

Testabilidade

Relacionado ao esforço despendido para testar um *software* a fim de garantir que execute todas as funções para qual foi projetado.

Integridade (Ele é seguro?)

Se o sistema pode ser facilmente acessado por pessoas não autorizadas.

Esses itens citados podem, de acordo com referências existentes, ser medidos e, por meio do resultado dessa medição, ser definido seu fator de qualidade.

Garantia de qualidade de software

A garantia da qualidade é uma atividade fundamental para qualquer negócio que gere produtos ou serviços.

A garantia de qualidade de *software* engloba algumas atividades como: teste, padronizações e procedimentos formais que são aplicados ao processo de engenharia de *software* para o desenvolvimento de *software* com qualidade, controle de mudança, medição e manutenção.

Algumas dessas atividades veremos a seguir, outras, sugerimos que faça uma pesquisa, porque é de fundamental importância conhecê-las, mas devido à nossa carga horária da disciplina, restrições de conteúdo tiveram que ser realizadas.

Métrica de qualidade de software

A possibilidade de estabelecer uma medida da qualidade é um aspecto importante para a garantia de um produto de *software* com algumas das características definidas anteriormente.

Mas como medir, por exemplo, o quanto um *software* será fácil ou não de dar manutenção, ou será seguro?

É nesse contexto de “como medir” que um novo conceito na área de ES é inserido: conceito de métrica.

Uma vez que as medidas quantitativas (mensuráveis) têm-se provado eficientes em vários ramos da ciência, cientistas de computação têm trabalhado arduamente para aplicar métodos similares no desenvolvimento de *software*.

Segundo Wikipédia (2007d, 2007e), a **métrica de software** é a medida de alguma propriedade do *software* ou da sua especificação. A métrica é utilizada para calcular orçamentos, desempenho dos programadores etc.

Para Cavano e MacCall (*apud* PRESSMAN, 1996, p. 753):

A determinação da qualidade é fundamental nos eventos cotidianos – concursos de degustação de vinhos, eventos esportivos, concursos de talento, etc. Nessas situações, a qualidade é julgada na maneira mais fundamental e direta: uma comparação lado a lado dos objetos sob condições idênticas e com conceitos predeterminados. O vinho pode ser julgado de acordo com a clareza, cor, buquê, sabor, etc. Porém, esse tipo de julgamento é muito subjetivo; para ter qualquer valor absoluto, ele deve ser feito por um especialista.

A subjetividade e a especialização também se aplicam na determinação da qualidade de *software*. Para ajudar a resolver esse problema, uma definição mais precisa de qualidade de *software* é necessária, bem como uma forma de derivar medições quantitativas de qualidade de *software* para análise objetiva [...]. Uma vez que não existe essa coisa de conhecimento absoluto, ninguém deve esperar medir qualidade de *software* exatamente, porque cada medição é parcialmente imperfeita. Jacob Bronowsky descreveu esse paradoxo do conhecimento desta maneira: “Ano a ano deparamo-nos com instrumentos cada vez mais precisos com os quais podemos observar a natureza com mais precisão. E quando olhamos para as observações ficamos desconsertados ao ver que elas ainda são vagas, e achamos que elas continuam tão incertas como sempre”.

Por essa citação, percebemos que medir qualidade não é coisa fácil, mas nem por isso especialistas deixaram de tentar criar modelos para que chegássemos o mais próximo possível de uma medição eficaz.



Atenção

É importante salientar que não medimos diretamente a qualidade de *software*, mas a manifestação dessa qualidade durante sua execução.

Nesse momento, não teremos como abordar com profundidade as técnicas de medição, mas citaremos algumas para que você dedique algum tempo para pesquisar sobre estas teorias/técnicas de medição.

Métricas de dimensão e complexidade

Têm sido propostas inúmeras métricas para medir a dimensão e complexidade de um programa. Elas são apresentadas em conjunto pois, na maior parte das vezes, a mesma métrica é apresentada quantificando ora a dimensão, ora a complexidade.

A- Linhas de Código Fonte – Lines of Code (LOC)

É uma métrica de dimensão que, apesar de criticada, é ainda a mais utilizada.

Embora aparentemente simples, obriga a uma definição inequívoca das regras de contagem de linhas, nomeadamente no tocante ao tratamento a dar às linhas em branco e de comentário, instruções não executáveis, diretivas de compilação, múltiplas instruções por linha ou múltiplas linhas por instrução, bem como no caso de reutilização de código.

B- Métricas de Halstead

É um conjunto de métricas proposto por Maurice Halstead que se baseia na teoria da informação e que o autor designou por “Software Science”.

Ele usa medidas primitivas para desenvolver expressões para o comprimento global do programa, o volume mínimo potencial para um algoritmo, o volume real (medido em bits), o nível do programa e outras características, como o esforço do desenvolvimento, o tempo de desenvolvimento e até mesmo o número projetado de falhas no *software*.

São utilizadas bases de cálculos matemáticos para se chegar a um valor métrico da qualidade e complexidade do *software*.

Pressman (2011) defende que a métrica de comprimento de Halstead é objetiva e melhor que a LOC.

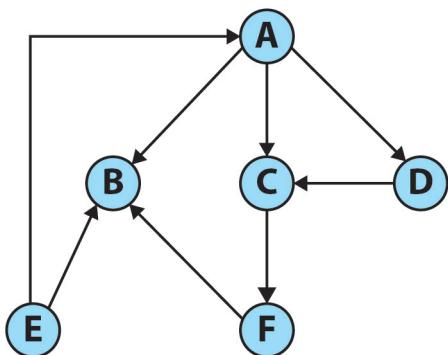
C- Métricas de McCabe

A primeira e mais conhecida métrica proposta por Thomas McCabe é a *métrica de complexidade ciclomática*.

Ela pressupõe que a complexidade depende do número de decisões (complexidade ciclomática), é adimensional e corresponde ao número máximo de percursos linearmente independentes através de um programa.

Os caminhos podem ser representados através de um gráfico orientado em que os nós sinalizam uma ou mais instruções sequenciais e os arcos orientados indicam o sentido do fluxo de controle entre várias instruções (Figura 1).

Figura 1: Complexidade do gráfico de fluxo de controle.



Outras métricas que valem a pena pesquisar e conhecer:

- **Métrica dos Nós:** proposta por Woodward.
- **Métrica dos Fluxos de Informação:** proposta por Henry e Kafura.
- **Métrica voltada para Orientação Objeto.**

Estimativa de software

Quando falamos em estimativa de *software*, uma coisa deve ficar clara: é difícil estabelecer se é possível desenvolver o produto desejado pelo cliente antes de conhecer os detalhes do projeto.

Por isso que uma boa definição de requisitos e os encontros periódicos com os clientes são fundamentais para estimar o tempo de desenvolvimento, custo, tamanho do projeto etc.

O desenvolvimento de um *software* é um processo gradual de refinamento e devemos sempre lembrar que:

- A incerteza da natureza do produto contribui para a incerteza da estimativa.
- Requisitos e escopo mudam.
- Defeitos geralmente são encontrados e demandam retrabalho.
- A produtividade varia de pessoa para pessoa.

O **processo de estimativa** envolve 5 etapas básicas:

- Estimar o Tamanho do Produto.
- Estimar o Tempo.
- Estimar o Esforço.
- Estimar o Custo (envolve 4 fatores).
- Estimar o Prazo.

É viável fornecer estimativas dentro de uma faixa permitida e, com o passar do tempo, a partir do momento em que se conhece mais e mais o projeto, refinar essa faixa.

Estimativa de Tamanho

- É a dimensão do *software* a ser produzido. Seu tamanho e quantidade, por exemplo:
 - N° linhas de código, n° pontos de função, n° de requisitos, pontos de casos de uso etc.

Estimativa de Tempo

- Após desenvolver uma estimativa do volume de trabalho a ser feito, não é fácil estimar o período em que o projeto será executado. Para que não estimemos outros fatores como custo de forma inadequada, a relação entre tempo e pessoa deve ser bem dimensionada/estimada.

Estimativa de Esforço

- É a técnica mais comum para apurar os custos de qualquer projeto de desenvolvimento.
- A estimativa de esforço tem início com a definição do escopo do projeto e as funções que deverão conter.
- O planejador estima o esforço (por exemplo, pessoas/mês), que seria exigido para conclusão de cada tarefa de Engenharia de *Software*, para cada função de *software*. Taxas de mão de obra (isto é, custo/esforço unitário) são aplicadas em cada uma das tarefas de Engenharia de *Software*.

Estimativa de Custo

- Aqui o objetivo é calcular, antecipadamente, todos os custos associados ao sistema: construção, instalação, operação e manutenção.

O Custo da Construção

- Uma vez que o custo está associado ao número de pessoas envolvidas no desenvolvimento do sistema – tais como burocratas, diretores, membros da comunidade usuária, consultores e programadores, membros da auditoria, do controle de qualidade ou da equipe de operações –, ele deve ser cuidadosamente mensurado.

O Custo da Instalação do Sistema

- Esse custo está relacionado ao modo como o cliente terá acesso ao produto. Se ele for instalar sozinho o sistema, é um valor, se for um sistema de grande porte e que precisa de uma equipe de instalação e treinamento, é outro. Quanto maior o sistema, maiores os custos, pois teremos que prever: custo de treinamento do usuário, custo de conversão de banco de dados, custo de instalação do fornecedor, custo da aprovação legal etc.

O Custo Operacional

- Entra em ação após a instalação do sistema. Haverá um custo para o usuário manter a operação do produto. Nesse custo, deve ser previsto como e quando o cliente, com o novo produto, poderá economizar dinheiro, a partir da utilização desse novo instalado. Os custos operacionais mais comuns são: custos de hardware e suprimentos, custos de software, custo de pessoal, custo de manutenção etc.

O Custo de Manutenção ou Falhas

- Por não termos sistemas perfeitos, esse custo deve ser bem dimensionado. O preço (direto ou indireto) a ser pago por todos se um sistema crucial ficar horas ou dias sem operação pode ser incomensurável e gerar inúmeros incômodos no momento em que isso acontece.

Estimativas de Prazo

- Geralmente são dirigidas a datas fornecidas pelo cliente e devem, sempre que possível, serem respeitadas.

Fator Humano

- Quando os objetivos para o desenvolvimento de sistemas não são claros, as pessoas passam a deduzir e criar o produto a partir do que acreditam que seja necessário, desenvolvendo, em inúmeros casos, sistemas inadequados e, consequentemente, métricas falhas, gerando uma expectativa negativa entre o cliente e os técnicos responsáveis, isto é, uma estimativa irreal.
- As pessoas são sensíveis aos estímulos externos e por eles são influenciadas. Um analista, ou um grupo de analistas, disposto a estimar o tempo e custo de um projeto não poderia deixar de dar a devida relevância a esse fato.

Engenharia Humana

- Para tentar amenizar a dificuldade e estabelecer critério para a estimativa em relação às pessoas, surge o conceito de Engenharia Humana, que consiste em aplicar conceitos de psicologia para projetar uma interação homem-computador de alta qualidade. Do ponto de vista do especialista em Engenharia Humana ou interface homem-computador, o homem é tratado como elo de coleta e processamento de dados.

Assim, podemos concluir que as estimativas jamais poderão ser precisas e exatas, pois não são compostas apenas por fatores técnicos, “contáveis” e palpáveis que fazem parte de um projeto, mas também por fatores humanos integrados (sentimentos, políticas, crenças, percepção, experiência etc.), assim como o ambiente e outras características mais, que não podemos estimar de forma absoluta. Entretanto, devem ser analisadas por meio dos embasamentos teóricos existentes sobre o tema. Afinal, estimar não é adivinhar e estimativas mal dimensionadas geram problemas.

Material Complementar



Explore

O objetivo do material complementar é lhe ajudar a entender, sob uma ótica diferente daquela da professora conteudista, assuntos abordados nas unidades teóricas.

É fundamental a leitura deste material para o melhor entendimento sobre o assunto.

Como nesta unidade abordamos os conceitos gerais da Engenharia de Software, nossa sugestão de material complementar é o capítulo 27, intitulado Gerenciamento de qualidade, no seguinte livro:

SOMMERVILLE, I. **Engenharia de Software**. 8. ed. São Paulo: Pearson, 2007. p. 423-438.

Referências

Bibliografia fundamental

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson, 2011.

Bibliografia básica

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação**. 4. ed. Rio de Janeiro: LTC, 1999.

LAUDON, K. C.; LAUDON, J. P. **Sistemas de Informação Gerenciais**. Administrando a empresa digital. 5. ed. São Paulo: Pearson Education do Brasil, 2006.

PFLEEGER, S. L. **Engenharia de Software**: teoria e prática. São Paulo: Prentice Hall, 2004.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 1995.

PRESSMAN, R. S. **Engenharia de Software**. São Paulo: Makron Books, 2006.

SOMMERVILLE, I. **Engenharia de Software**. 6. ed. São Paulo: Pearson Addison Wesley, 2005.

Bibliografia complementar

ALCADE, E.; GARCIA, M.; PENUELAS, S. **Informática Básica**. São Paulo: Makron Books, 1991.

FAIRLEY, R. E. **Software engineering concepts**. New York: McGraw-Hill, 1987.

IEEE. **Software Engineering Standards**. 2013. Disponível em: <http://www.ieee.org/portal/inno_vate/products/standard/ieee_soft_eng.html>. Acesso em: 10 dez. 2013.

LUKOSEVICIUS, A. P.; CAMPOS FILHO, A. N.; COSTA, H. G. **Maturidade em gerenciamento de projetos e desempenho dos projetos**. Disponível em: <www.producao.uff.br/conteudo/rpep/.../RelPesq_V7_2007_07.doc>. Acesso em: 12 nov. 2013.

MAFFEO, B. **Engenharia de software e especialização de sistemas**. Rio de Janeiro: Campus, 1992.

MICHAELIS. **Moderno dicionário da língua portuguesa**. São Paulo: Cia. Melhoramentos, 1998.

PARREIRA JÚNIOR, W. M. **Apostila de Engenharia de Software**. Disponível em: <http://www.waltenomartins.com.br/ap_es_v1.pdf>. Acesso em: 13 nov. 2013.

PAULA FILHO, W. P. **Engenharia de Software**: fundamentos, métodos e padrões. 2. ed. Rio de Janeiro: LTC, 2001.

Revista Engenharia de Software. Disponível em: <<http://www.devmedia.com.br/revista-engenharia-de-software-magazine>>. Acesso em: 12 nov. 2013.

VONSTA, A. **Engenharia de Programas**. Rio de Janeiro: LTC, 1983.

WIENNER, R.; SINCOVEC, R. **Software engineering with Modula 2 and ADA**. New York: Wiley, 1984.

WIKIPEDIA (2007a). **ISO 9000**. Disponível em: <http://pt.wikipedia.org/wiki/ISO_9000>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007b). **Melhoria de processo de software brasileiro**. Disponível em: <<http://pt.wikipedia.org/wiki/MPS.BR>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007c). **CMMI**. Disponível em: <<http://pt.wikipedia.org/wiki/Cmmi>>. Acesso em: 22 jun. 2007.

WIKIPEDIA (2007d). **Engenharia de Software**. Disponível em: <http://pt.wikipedia.org/wiki/Engenharia_de_software>. Acesso em: 01 fev. 2007.

Anotações





Educação a Distância
Cruzeiro do Sul Educacional
Campus Virtual

www.cruzeirodosulvirtual.com.br
Campus Liberdade
Rua Galvão Bueno, 868
CEP 01506-000
São Paulo SP Brasil
Tel: (55 11) 3385-3000

