UNIP
UNIVERSIDADE PAULISTA
Interativa

Análise de Sistemas Orientada a Objetos

Autor: Prof. Fábio Rossi Versolatto
Colaboradores: Prof. Luciano Soares de Souza
Profa. Ana Carolina Bueno Borges
Prof. Eduardo de Lima Brito

Professor conteudista: Fábio Rossi Versolatto

Mestre em Engenharia de Software pelo Instituto de Pesquisas Tecnológicas (IPT) da USP, especializado em Arquitetura, Componentização e SOA pela Universidade Estadual de Campinas (Unicamp), pós-graduado em Tecnologia de Construção de *Software* Orientado a Objetos pelo Centro Universitário Senac e bacharel em Ciências da Computação pelo Centro Universitário da FEI.

Professor do curso de Análise e Desenvolvimento de Sistemas da Universidade Paulista (UNIP), no qual leciona e orienta os alunos no programa Projeto Integrado Multidisciplinar.

Atuando na área de pesquisa, possui publicações e participações em eventos na área de Engenharia de Software no Brasil e no exterior, além de projetos de pesquisa aplicados na área social.

Atua na área de desenvolvimento de sistemas de *software* com larga experiência em sistemas estratégicos em empresas de grande porte, com ênfase em análise de requisitos, projeto, arquitetura e desenvolvimento de soluções de *software*.

Dados Internacionais de Catalogação na Publicação (CIP)

V564a Versolatto, Fábio Rossi.

Análise de Sistemas Orientada a Objetos. / Fábio Rossi Versolatto. – São Paulo: Editora Sol, 2015.
172 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ano XIX, n. 2-027/15, ISSN 1517-9230.

1. Análise de sistemas. 2. Projeto de desenvolvimento. 3. Engenharia de requisitos. I. Título.

CDU 681.3.07

A-XIX

Sumário

Análise de Sistemas Orientada a Objetos

Unidade III

Unidade IV

APRESENTAÇÃO

A disciplina *Análise de Sistemas Orientada a Objetos* tem como objetivo apresentar ao aluno as boas práticas para captar e mapear as necessidades das organizações, inseridas em um mercado cada vez mais competitivo, de tal forma que essas necessidades captadas de forma eficiente se tornem subsídios para o futuro projeto de *software*, utilizando para isso o paradigma da orientação a objetos.

Para atingir este objetivo, a disciplina começa contextualizando o aluno com os problemas que a Engenharia de Software se propõe a resolver, traçando um paralelo com a linha de tempo de sua evolução.

Com o aluno contextualizado na temática, a disciplina apresenta e discute a importância das atividades de desenvolvimento a serem executadas de forma organizada dentro de um projeto, apresentando os principais modelos de desenvolvimento e papéis a serem desempenhados pelos integrantes da equipe deste projeto.

Conceitos básicos e motivações postos, debate-se a engenharia de requisitos como fase fundamental dentro de um projeto de desenvolvimento, mostrando as principais técnicas para elicitação, mapeamento, modelagem, documentação e gestão de todos os tipos de requisitos de um processo de negócio.

Por fim, a disciplina apresenta a UML (Unified Modeling Language) como uma ferramenta para a representação e a modelagem de um sistema de *software*, partindo da modelagem dos requisitos e chegando ao projeto dos componentes do *software*, utilizando os diagramas da UML como ferramenta de apoio em todo o processo.

INTRODUÇÃO

Desde que o *software* se estabeleceu como uma ferramenta importante na estratégia competitiva das grandes empresas, a indústria de desenvolvimento vem passando por transformações para atender às necessidades cada vez maiores deste mercado.

O desafio lançado não está mais no simples fato de se desenvolver uma série de linhas de código que, agrupadas, compõem um sistema de *software*, mas sim em desenvolver este *software* como um produto, como uma ferramenta de estratégia competitiva que atenda a uma série de exigências e a determinados padrões de qualidade.

O que se espera de um projeto de *software* é que ele seja iniciado e terminado no menor intervalo de tempo possível e que seja desenvolvido com um custo competitivo e, principalmente, com alta qualidade. Balancear essas três variáveis em uma equação chamada "projeto de *software*" é um dos grandes desafios da Engenharia de Software.

Se o *software* deve ser enxergado como uma ferramenta de estratégia competitiva em uma grande corporação, parece razoável que um projeto não deva tomar um tempo excessivo, afinal de contas, em um mercado cada vez mais competitivo, tem a vantagem aquele que larga na frente.

Seguindo a mesma linha de raciocínio, espera-se que um projeto não tenha custos exorbitantes, o que pode, inclusive, torná-lo inviável. Em casos de projetos malsucedidos, o custo de um processo de negócio realizado de forma não automatizada pode ser mais vantajoso que automatizá-lo. Logo, podemos notar que o *software* deve estar atento à relação custo-benefício.

Sob uma visão ideal, um projeto deve ser executado em um tempo razoável e com um custo viável. Como se isso não bastasse, um sistema de *software* competitivo deve atingir certo nível de qualidade.

As necessidades do negócio devem ser mapeadas de forma eficiente. Automatizar o processo de negócio de uma empresa é uma tarefa árdua, que requer inicialmente e, principalmente, total clareza de todas as regras e necessidades, ou requisitos, que envolvem este processo a ser automatizado. A qualidade do *software* que se pretende entregar passa inicialmente por essa fase.

É possível também, em um primeiro momento, que nos deparemos com a necessidade de atendermos a requisitos não tão claros, mas que em hipótese alguma são menos importantes.

Parece razoável, em um mundo onde a tecnologia de *hardware* evoluiu e segue em constante evolução, que devamos nos preocupar em desenvolver sistemas de *software* que se adaptem às diversas tecnologias e plataformas de computadores pessoais, sistemas operacionais, dispositivos móveis, entre outros. Tudo isso sem perder o foco em fornecer uma experiência rica ao usuário final para que ele tenha segurança e conforto no uso e, também, para que possa realizar suas atividades de maneira produtiva.

Voltando à linha de raciocínio anterior, se o *software* deve ser identificado como uma ferramenta de estratégia competitiva em uma grande corporação, e que deve ser desenvolvido em tempo e custo de tal forma a torná-lo efetivamente competitivo, não é difícil observar a necessidade de pensarmos em um projeto que nos possibilite manutenção rápida e eficiente. Afinal, se o mercado competitivo muda com muita rapidez, é possível imaginar que um sistema de *software* também deva se adaptar e essa realidade.

A partir dessa reflexão inicial, podemos nos questionar: será que realmente estamos dando a devida atenção a todos esses pontos na hora de iniciar um projeto? Por que características de "outras engenharias", como padronização e estabelecimento de um processo produtivo, nos parecem tão distantes da Engenharia de Software?

O paradigma da orientação a objetos, iniciada nesta disciplina, tem como objetivo levantar e responder a essas e outras questões.

Este livro-texto busca oferecer um norte para as melhores práticas de desenvolvimento, de tal forma que possamos atingir, como produto final, o *software* como uma ferramenta de estratégia competitiva de qualidade.

Unidade I

1 INTRODUÇÃO À ANÁLISE DE SISTEMAS

1.1 Introdução

A evolução constante das plataformas de *hardware*, acompanhada do aumento escalar no uso dos computadores pessoais, mudou a forma como se pensava o desenvolvimento de um sistema de *software*.

Com essa evolução, os sistemas de *software* ficaram necessariamente mais completos e complexos. Muito disso se deve à necessidade de se utilizar todos os recursos disponíveis, recursos estes que faltavam em outros momentos.

Segundo Stallings (2003), a evolução dos computadores é evidenciada, ao longo dos anos, pelo aumento da capacidade dos processadores, da memória e da velocidade do mecanismo entrada/saída.

Ainda segundo Stallings (2003), essa evolução deve-se ao fato de que os componentes, também chamados de transistores, que compõem um processador foram diminuindo em tamanho com o passar do tempo.

Com essa diminuição, foi possível reduzir o espaço entre esses componentes. Diminuindo também a distância que uma corrente elétrica percorre entre dois ou mais transistores, obtendo um consequente ganho na velocidade de processamento.

Além do aumento na velocidade do processamento, a diminuição dos componentes gerou um espaço maior na pastilha dos processadores, que, por sua vez, foram preenchidas por mais transistores, havendo, consequentemente, um novo ganho de velocidade.

Podemos notar esse ciclo analisando a Lei de Moore (Moore, 1965), proposta por um dos fundadores da Intel, Gordon Moore, que observou que a quantidade de transistores em um processador duplicava a cada 12 meses.

Observação

Segundo a Lei de Moore, se nós construíssemos um processador hoje (janeiro de 2015) com uma capacidade de processamento de "X", em janeiro de 2016 construiríamos um processador com capacidade 2 "X".

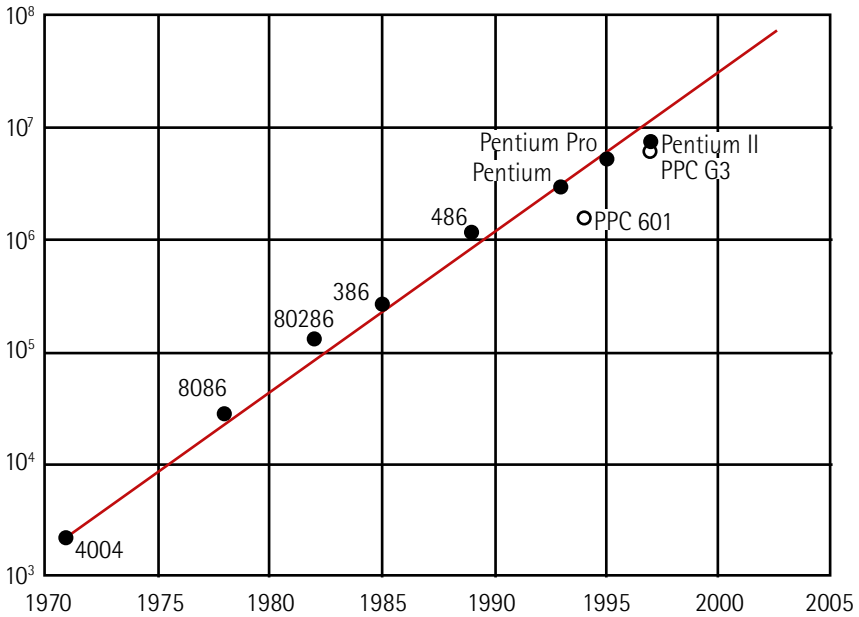

Figura 1 – Crescimento do Número de Transistores x Lei de Moore

A figura anterior mostra o crescimento do número de transistores dos processadores Intel no decorrer do tempo em um comparativo com o que prega a Lei de Moore, representada pela linha central do gráfico.

A evolução do número de transistores segue o que foi previsto por Moore, com a única ressalva que, nos anos 1970, a duplicação se deu a cada 18 meses, em contrapartida aos 12 meses observados na teoria de 1965.

Além do óbvio aumento na capacidade de processamento, alguns dos reflexos da Lei de Moore:

- Diminuição do custo de produção: com o baixo custo de produção dos computadores, foi possível reduzir o preço final e, consequentemente, houve um aumento nas vendas e na produção em escala dos equipamentos.

- Diminuição do consumo de energia: os computadores passaram a consumir menos energia, tornando-se cada vez mais atrativos a um público cada vez mais crescente.

- Computadores menores: com componentes menores, os computadores também reduziram de tamanho. Passamos a não ter mais a necessidade de um local físico específico e especial para o computador.

- Computadores mais confiáveis: com um maior número de transistores em uma pastilha, diminuiu a necessidade de conexões entre pastilhas. Essas conexões são feitas utilizando solda, que é menos confiável se comparadas às conexões feitas entre transistores.

Esses fatores geraram uma atmosfera positiva que culminaria, na década de 1980, com a massificação do uso de computadores por **usuários comuns**, o uso dos **computadores pessoais**.

Com o **problema *hardware*** resolvido, o primeiro desafio posto estava em como desenvolver sistemas que maximizassem o uso do processador. Como utilizar toda a capacidade de processamento paralelo? Como diminuir a ociosidade de um processador?

Após a massificação do uso de computadores, surgiram novos desafios. As máquinas passaram a ser operadas por **usuários comuns**. Não era mais necessário ser um especialista, os computadores passaram a ser mais fáceis de serem usados. Desafio posto: como desenvolver sistemas que acompanhassem a tendência da massificação do uso?

Com esses e muitos outros desafios, passamos a ter uma maior preocupação com a informação. A forma como a informação era trabalhada, trafegada, levada ao usuário e gerenciada passou a ter papel de fundamental importância.

Surgiu então o conceito de **sistemas de informação**.

Saiba mais

Para mais informações sobre dispositivos de entrada/saída, processadores, memória e arquitetura de computadores, leia:

STALLINGS, W. *Arquitetura de computadores*. São Paulo: Prentice Hall, 2003.

1.2 Sistemas de informação x *software*

O que é o sistema imunológico de uma pessoa? De forma resumida e popular, o sistema imunológico biológico é a composição de diferentes tipos de células, cada qual com sua função, que atuam conjuntamente com o objetivo de manter a integridade do organismo que está sendo protegido.

Algumas células têm a função de identificar possíveis ameaças e outras, de tomar determinadas ações em caso de ataque. Essas células são independentes entre si, todavia, quando da execução de alguma tarefa, agem em conjunto (SILVA, 2001).

De forma análoga, um sistema de informação é a composição de informações (ou dados), pessoas, processos e tecnologias que tem por objetivo apoiar ou melhorar o processo de negócio de uma corporação.

Sistemas de informação tem foco na informação gerada em um processo organizacional, de tal forma a originar valor a esse processo (BEZERRA, 2006).

Quando falamos em tecnologia de sistemas de informação, estamos falando de toda plataforma tecnológica que dá suporte ao processo de negócio, como infraestrutura de rede, servidores, computadores, sistemas operacionais e sistemas de *software*.

Observação

Rezende (2005) observa que todo sistema que gera e armazena informação pode ser considerado um sistema de informação, independentemente se utiliza tecnologia ou não.

Imagine um sistema de informação que tem como objetivo controlar o processo de um *call center*. Qual valor estratégico e competitivo esse sistema de informação está produzindo à empresa?

Poderíamos trabalhar com a ideia de que esse sistema proporcionaria atendimento mais eficaz e eficiente ao cliente, e geraria informações importantes que serviriam de apoio gerencial, como a definição do perfil do cliente. Ainda poderíamos pensar como um valor importante estar apto à legislação de apoio e atendimento ao consumidor, que está em constante aperfeiçoamento.

É importante ter em mente quais são os objetivos que o sistema deve atingir, pois eles servem de norte para todo o restante.

No exemplo do processo de um *call center*, é importante o mapeamento do processo a ser executado, como representado de forma simplificada na figura a seguir. Qual é a sequência de atividades a serem executadas a partir do momento que o cliente é atendido, quem são as pessoas que executam cada atividade e qual informação é gerada no decorrer do processo.

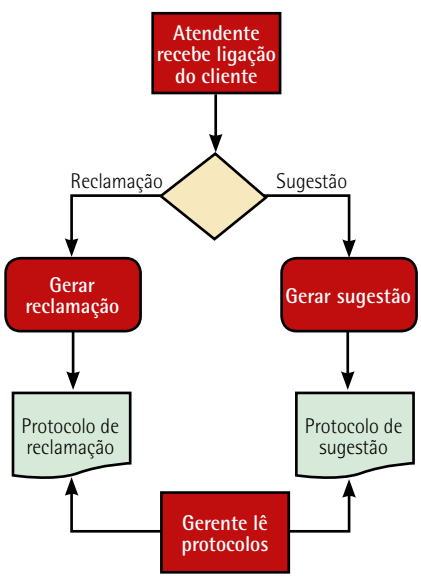

Figura 2 – Mapeamento simplificado de um processo de atendimento

Na plataforma tecnológica, deveríamos nos atentar a detalhes como **como** e **onde** essa informação seria armazenada e qual o melhor projeto de rede para suportar o tráfego dessa informação.

Modelar um sistema de informação não é tarefa das mais simples e, dentro disso tudo, é importante pensarmos qual parte do processo seria automatizada por um sistema de *software*.

Projetar um sistema de *software* compreende termos a noção de que ele é um componente de um sistema de informação, ou seja, que ele é parte de um todo e que precisa estar harmoniosamente encaixado neste todo.

Diversas foram as abordagens e técnicas criadas para modelar sistemas de *software* até chegarmos à abordagem foco deste livro: orientação a objetos. Por ora, entenda que a orientação a objetos é uma forma de se projetar um sistema de *software*. Adiante, detalharemos melhor o assunto.

O quadro a seguir mostra a evolução das formas de se desenvolver sistemas, que anda em paralelo com a evolução dos computadores já debatida anteriormente.

Quadro 1 – Evolução da modelagem de sistemas de *software*

Período	Cenário	Abordagem de modelagem
Décadas de 1950 e 1960	Desenvolvimento sem planejamento inicial, devido à simplicidade dos sistemas de *software*.	Fluxogramas e/ou diagramas de módulos.
Década de 1970	– Surgimento de computadores mais acessíveis e avançados. – Sistemas de *software* mais complexos com necessidades mais específicas, como tempo real e multiusuário. – Surgimento dos primeiros bancos de dados relacionais.	Projeto estruturado e programação estruturada.
Década de 1980	– Massificação do uso de computadores. – Surgimento dos primeiros conceitos de sistemas distribuídos e sistemas inteligentes. – Necessidade de interfaces mais ricas.	Análise estruturada.
Década de 1990	– Computadores mais avançados. – Conceitos de computação paralela passam a se sedimentar. – Surgimento dos conceitos de sistemas especialistas e sistemas para mobilidade. – Necessidade cada vez maior de produtividade. Chave: Conceitos de reúso. – Necessidade de uma linguagem única de representação e modelagem de sistemas (*).	– Paradigma da orientação a objetos. – Criação da linguagem de modelagem unificada – UML.

Adaptado de: Bezerra (2006, p. 12).

Um sistema de *software* também pode ser identificado como o conjunto de vários componentes que, quando executados, produzem um resultado previamente desejado.

Esses componentes podem ser chamados de componentes de *software*, que são desenvolvidos utilizando métodos e processos que estabelecem uma relação entre a necessidade do cliente e um código executado em uma máquina (PRESSMAN, 2006).

Desenvolver um sistema de *software* vai além da simples digitação de linhas de código, mas está ligado ao entendimento das necessidades do cliente, as fronteiras dele dentro de um sistema de informação, de tal forma que *software* possa atender a certos parâmetros de qualidade, sendo, assim, eficaz e eficiente dentro do processo organizacional.

A Engenharia de Software tem por objetivo apoiar o desenvolvimento de *software* fornecendo técnicas para especificar, projetar, manter e gerir um sistema (SOMMERVILLE, 2010). Ela é baseada em três pilares:

- Métodos: provêm um conjunto de atividades com ênfase no **como** fazer para desenvolver um sistema de *software*.

- Ferramentas: mecanismos que dão suporte à execução das atividades descritas no método. Um assunto que está sempre em pauta, quando estamos falando de ferramentas, são as chamadas ferramentas CASE (Computer-Aided Software Engineering), que fornecem e consomem informações geradas nas atividades especificadas nos métodos da Engenharia de Software. Se utilizadas em conjunto, de forma sistêmica e organizada, fornecem suporte ao processo de desenvolvimento de software.

Saiba mais

Saiba mais sobre a seleção e utilização de ferramentas CASE na seguinte norma:

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION/ INTERNATIONAL ELECTROTECHNICAL COMMISSION. *14102*: information technology – guideline for the evaluation and selection of case tools. Geneve: ISO, 2008.

- Processos: proveem o elo entre métodos e ferramentas. Definem o **quando** e **onde** fazer, indicando a sequência de execução das atividades, quais ferramentas utilizar e **quem** deve ser o responsável pela atividade.

Observação

Roger Pressman (PRESSMAN, 2006) e Iam Sommerville (SOMMERVILLE, 2010) são considerados dois dos maiores expoentes da Engenharia de Software.

Em resumo, a Engenharia de Software compreende um conjunto de atividades a serem executadas em uma determinada sequência, com o uso de ferramentas de suporte, dentro de um projeto de desenvolvimento.

2 PROJETO DE DESENVOLVIMENTO

2.1 Processo de desenvolvimento de *software*

O processo de desenvolvimento de *software*, conforme dito anteriormente, resume-se a um conjunto de atividades executadas em uma determinada sequência. Esse conjunto de atividades também pode ser chamado de etapas da Engenharia de Software ou paradigmas da Engenharia de Software (PRESSMAN, 2006).

A escolha de um processo de desenvolvimento passa principalmente pela natureza do problema a ser resolvido. Existem muitos paradigmas aplicados e que detalharemos nas próximas seções. Todavia, qualquer que seja o paradigma, segundo Sommerville (2010), é obrigatória a presença de quatro macroatividades:

Quadro 2 – Atividades fundamentais para a Engenharia de Software

Atividade	Descrição
Especificação de *software*	Fase na qual são definidas e especificadas as funcionalidades do *software*, seus requisitos e suas regras de execução.
Projeto e implementação de *software*	Fase na qual os requisitos são transformados em sistema de *software*. Quando são projetados e construídos os componentes de *software* e suas integrações.
Validação de *software*	Fase na qual o *software* é validado frente aos requisitos especificados.
Evolução de *software*	Fase na qual são traçadas estratégias para que o sistema de *software* possa ser mantido e evoluído na medida.

Adaptado de: Sommerville (2010, p. 28).

Dizemos que essas atividades são **macroatividades** pelo fato de que elas podem, e devem, possuir subatividades e/ou atividades que as suportem, por isso dizemos que são **macro** ou complexas.

Se fizermos uma analogia com a Engenharia Civil, podemos imaginar que a atividade de especificação produz os requisitos de uma casa, como a metragem, a quantidade de quartos, tipo de cozinha, quantidade de banheiros etc. Enquanto a fase de projeto irá produzir a planta da casa, transformar os desejos do proprietário em projeto. Se utilizar boas ferramentas, o projetista da casa poderá exibir um modelo quase que final da casa para o seu cliente, com detalhes que inicialmente poderiam até passar despercebidos, como a cor das paredes.

Na implementação ou construção, a casa começa a se concretizar fisicamente. Os requisitos do proprietário, que viraram um projeto consistente, tomam corpo com tijolos e cimento.

Quem já passou por uma situação parecida saberá muito bem do que estamos falando. Com o andamento das obras, é difícil segurar a ansiedade de "dar uma olhada".

É comum que, com a obra tomando forma, mudemos nossa forma de ver. O que antes era uma planta, um projeto, passa a se tornar algo concreto, palpável. Nessa hora, solicitamos pequenos ajustes no projeto, o que acaba invariavelmente refletindo na própria obra.

Chamamos de validação o ato de analisarmos se o que está sendo construído está em conformidade com o que foi desejado e/ou se está em conformidade com o que foi projetado. A possibilidade de correções e ajustes é grande nessa fase.

Se o projeto e a construção da casa foram feitos utilizando bons padrões e boas ferramentas, temos a garantia, ou uma boa probabilidade, de que não tenhamos grandes problemas.

No entanto, mesmo que aconteça algum tipo de problema ou que desejássemos algum tipo de mudança, temos a garantia de que a manutenção terá o menor custo possível.

Feito esse paralelo, voltemos para o processo de desenvolvimento de *software*. Cada atividade possui:

- Pré e pós-condições: qual é o cenário pré e pós-execução de uma atividade. Por exemplo: para validação de um *software*, é necessário que este esteja implementado, após execução da validação, espera-se que o *software* esteja validado.

- Papéis: quem é o responsável pela execução da tarefa. Por exemplo: o analista de testes é responsável pela validação.

- Produtos: também chamados de artefatos, é o que é produzido como saída de uma determinada atividade. Por exemplo: espera-se como artefato de saída da atividade de especificação de *software* um documento contendo a descrição detalhada de todas as necessidades do negócio.

Saiba mais

A norma IEE 12207 (2006), na seção processo de desenvolvimento, especifica, dentro de um parâmetro de boas práticas, as atividades e a sequência de execução para o desenvolvimento de um produto de *software:*

INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. *12207*: standard for information technology – software life cycle processes. New York: IEEE, 2008.

2.2 Modelos de processo de *software*

Muitos são os modelos de processos aplicados e debatidos atualmente. Nesta seção, abordaremos alguns dos diversos modelos de processos de *software* utilizados atualmente na indústria de desenvolvimento.

2.2.1 Modelo cascata

Também chamado de *waterfall* ou também citado na literatura como ciclo de vida clássico, o modelo cascata é composto de cinco atividades.

Análise de requisitos

Nesta fase são elicitados e documentados os requisitos do sistema. Existem técnicas e processos de elicitação de requisitos, que serão detalhadas mais adiante.

Projeto

Nesta fase são definidos os componentes do *software* e como eles irão interagir para atingir os requisitos elicitados na fase de análise dos requisitos.

É também nesta fase que são definidas as necessidades e as restrições de *software* e *hardware*, como banco de dados, sistemas operacionais e configurações mínimas de espaço em disco e processamento necessárias para o projeto de *software*. É neste momento que a arquitetura do sistema começa a ser definida.

Codificação

Nesta fase os componentes do *software* são desenvolvidos em linguagem de máquina.

É comum que nesta fase sejam feitos testes unitários, ou também chamados de testes de desenvolvimento, que têm como intuito a detecção, se houver, e correção de pequenas falhas unitariamente em cada componente de *software*.

Testes

Nesta fase o *software* construído é submetido à validação com o objetivo de encontrar e corrigir possíveis não conformidades com os requisitos avaliados, com a arquitetura definida ou até mesmo erros de construção.

Saiba mais

Existem inúmeros processos que têm como objetivo a garantia da qualidade do *software*. Saiba mais na obra:

CORTÊS, M. L.; CHIOSSI, T. C. dos S. *Modelos de qualidade de software*. Campinas: Unicamp, 2001.

Implantação e manutenção

A fase de implantação e manutenção é a fase mais longa do ciclo de vida de um *software* e tem início quando o sistema é colocado para uso por parte do usuário final.

Na prática, dizemos que o sistema está em ambiente produtivo, ou seja, o resultado de suas operações ou falhas afetarão diretamente o processo da organização.

Nesta fase, é possível que o sistema passe por alterações, seja para correção de algum erro não encontrado na fase de testes ou mudanças nos requisitos provocadas pelas mais diversas razões.

Lembrete

Uma das importantes características que um sistema de *software* deve possuir é ser manutenível, ou seja, a capacidade e a facilidade de adaptação de um *software*. Quanto melhor a manutenibilidade, melhor a qualidade do *software*.

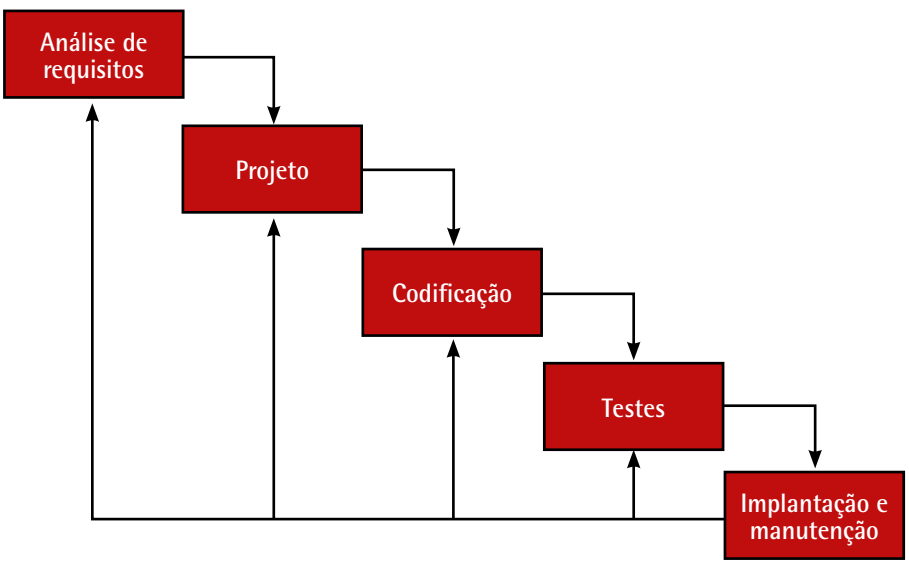

Figura 3 – Sequência de execução das atividades no modelo cascata

No modelo cascata, as atividades são executadas de forma sequencial. Assim, uma atividade não é iniciada até que sua predecessora seja completamente finalizada. Isso nos permite afirmar, por exemplo, que a construção não é iniciada até que seja finalizada a arquitetura do sistema de *software* na fase de projeto, que, por sua vez, não é iniciada até que todos os requisitos sejam levantados, documentados e aprovados pelo usuário.

É justamente nesse ponto que está uma das maiores fragilidades deste modelo.

A fase de projeto não se inicia até que todos os requisitos sejam elicitados, documentados e aprovados pelo usuário. É comum que requisitos novos surjam e sejam alterados no curso do projeto, pelas mais diversas

razões, entre elas a falta de habilidade do analista em extrair e captar as necessidades do processo de negócio. A mudança nos requisitos pode ocorrer durante a fase de projeto, codificação ou até mesmo na fase de testes, assim como problemas na arquitetura podem ser identificados na construção ou na implantação.

Esse cenário nos expõe duas situações:

- Alto custo em cada atividade: cada atividade acaba por levar mais tempo para ser finalizada, justamente para minimizar os impactos de possíveis falhas.

- Alto custo de retrabalho: no caso de encontrarmos alguma falha em algum produto produzido em alguma atividade, o custo de correção é alto.

Por exemplo, se encontrarmos uma desconformidade originada de requisitos mal levantados na fase de testes, teremos que voltar até a fase de análise, arquitetura e a construção para novamente chegarmos aos testes e tudo isso passando pela validação e aprovação de todos os produtos gerados em cada atividade.

Em contrapartida, o modelo espiral é muito eficaz em projetos nos quais temos o escopo e as necessidades bem definidas e com poucas chances de mudança, como em sistemas de missão crítica, cujos modelos matemáticos são feitos e validados à exaustão antes que seja desenvolvido qualquer tipo de modelo ou linha de código.

Observação

Chamamos de retrabalho o processo de realizar novamente uma atividade que já tenha sido dada como concluída em um momento anterior.

2.2.2 Modelo incremental

O modelo incremental, também chamado de iterativo, é, em sua essência, bem parecido com o modelo cascata.

Observação

Não confunda iterativo, com interativo! O termo interação é usado no sentido da ação recíproca entre usuário e equipamento. Iteração significa repetição.

No modelo iterativo, as atividades e a sequencia de execução são exatamente as mesmas do modelo cascata. Mas então qual é a diferença entre os dois modelos?

A diferença é que no modelo incremental não é necessário que todos os requisitos estejam mapeados para se iniciar o ciclo de desenvolvimento.

O ciclo é iniciado a partir de **blocos de requisitos** e não a partir de todos os requisitos, como pregado no modelo clássico. A partir daí, a metodologia é a mesma, ou seja, cada atividade somente é iniciada depois que a predecessora é finalizada e validada.

Cada **bloco de requisitos**, quando implantado, corresponde a uma parte do sistema de *software*, também chamado de incremento. O ciclo é repetido tantas vezes quanto necessário, até que todos os incrementos do *software* sejam implantados, daí o termo **modelo iterativo**, ou seja, modelo que se repete inúmeras vezes.

O modelo iterativo não diminui o custo total do projeto, mas reduz o custo de cada atividade. Com escopo reduzido, o tempo gasto para executar cada tarefa tende a ser menor também. Um exemplo disso pode ser notado na fase de testes. O tempo gasto para validar um número menor de funcionalidades é inversamente proporcional ao tempo gasto na validação de um escopo mais amplo.

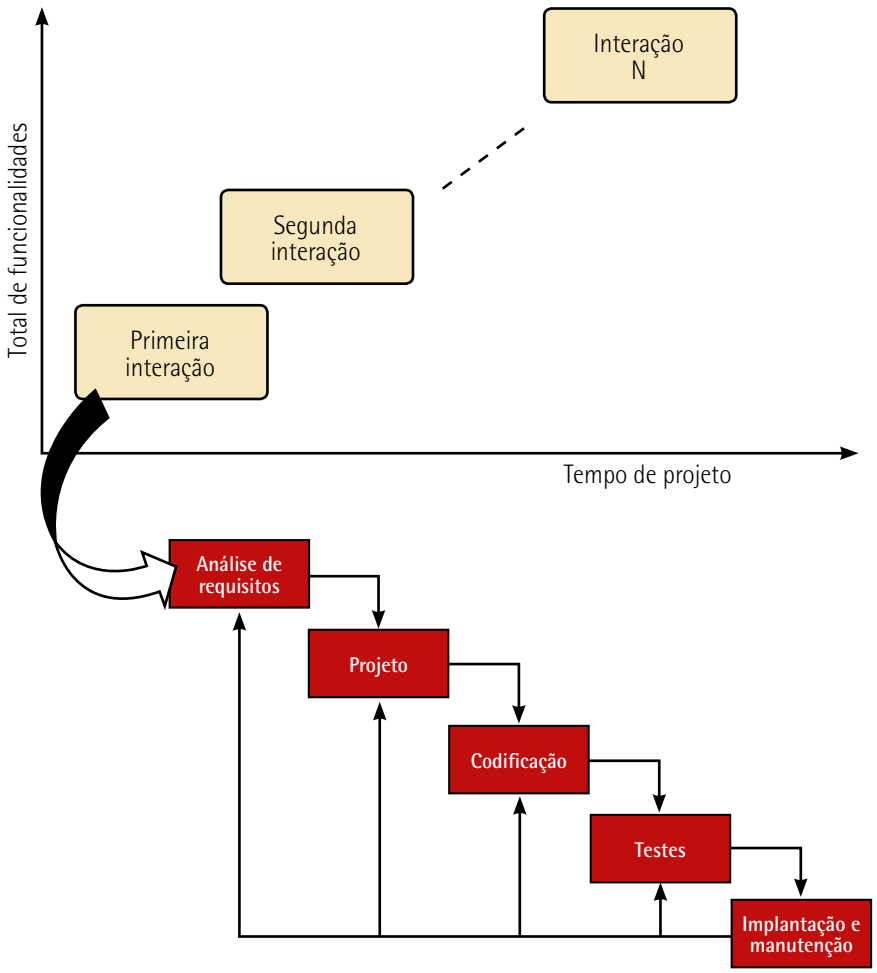

Figura 4 – Modelo iterativo

Um dos pontos positivos na adoção do modelo incremental é a redução do retrabalho. O número menor de requisitos reduz a chance de problemas na análise, projeto e construção.

Ademais, com um bom planejamento de execução, é possível priorizar partes de requisitos críticos para o negócio, requisitos que gerem valor ao processo, abrindo a possibilidade da vantagem competitiva para a organização no mercado.

Nada impede que um ciclo de desenvolvimento seja iniciado antes da implantação de um que já esteja em andamento. Desde que não haja nenhuma dependência entre os requisitos ou componentes de *software*, essa é uma alternativa viável e atrativa do ponto de vista da produtividade.

A chave para aumentarmos a eficiência na produção de um sistema de *software* utilizando o modelo incremental é: bom planejamento.

O modelo incremental serviu de base para muitos outros processos largamente utilizados e debatidos na literatura, como: prototipagem, modelo espiral e processo unificado, que iremos detalhar na sequência.

Saiba mais

O modelo iterativo também serviu como base para o desenvolvimento dos Métodos Ágeis, leia o manifesto ágil para saber mais sobre o que são eles:

<http://agilemanifesto.org/iso/ptbr/>.

2.2.3 Prototipagem

Segundo o dicionário, protótipo é um primeiro tipo, um primeiro exemplar. Em Engenharia de Software podemos pensar que o protótipo é a primeira versão de um sistema de *software* (SOMMERVILLE, 2010).

Esse protótipo de *software* tem como objetivo a prova de conceitos. Verificar e demonstrar se os requisitos mapeados estão de acordo com a necessidade do usuário ou validar uma solução de projeto. É fato que a prototipagem é utilizada com o objetivo de antecipar mudanças que possam vir a ser mais custosas no desenvolvimento de um sistema de *software*. O modelo de prototipagem possui quatro atividades:

- planejamento;
- definição das funcionalidades;
- construção;
- validação.

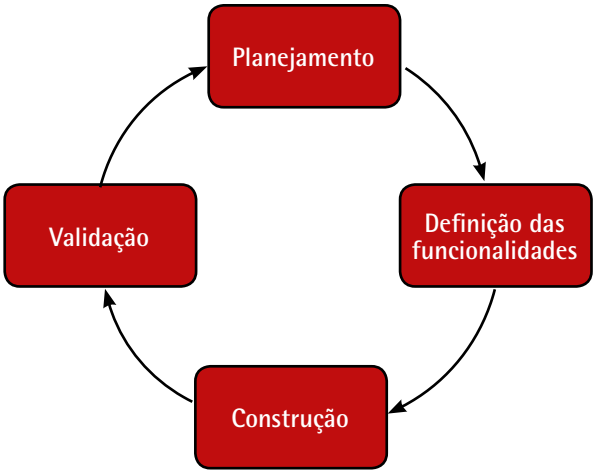

Figura 5 – Ciclo de atividades de prototipação

Como sabemos quando usar a prototipação? Pressman (2006) faz algumas observações bem interessantes:

- Se nosso usuário tem uma boa noção, mas não tem, no momento, uma visão detalhada do que precisa ser feito, é positivo que usamos a prototipação como um "primeiro passo".

- O cliente precisa ter a total noção de que o protótipo não é o sistema de *software*. A comunicação é fundamental.

- Toda a equipe de desenvolvimento também precisa ter a total noção que o protótipo não é o sistema de *software*. Ele foi desenvolvido descartando muitas das boas práticas de projeto, como a manutenibilidade.

Rosemberg *et al.* (2008) utilizaram uma abordagem voltada à prototipação e obtiveram bons resultados em seu projeto. Os autores trabalharam com o que chamamos de **protótipo de baixa fidelização**, que dá ênfase na interação com o usuário e na estrutura geral do sistema de *software*, não se preocupando em fazer protótipos que produzissem uma imagem real do sistema.

Protótipos de baixa fidelização foram eficientes e viáveis para demonstrar para os usuários as funcionalidades a serem implementadas pelo sistema.

Outro fato interessante observado foi a viabilidade econômica do protótipo: barato na construção e na alteração.

Alguns dos bons resultados obtidos nesse projeto: rapidez no processo de captação dos requisitos e antecipação dos problemas.

Saiba mais

Leia mais sobre os resultados obtidos e a metodologia utilizada pelos autores:

ROSEMBERG C. *et al.* Prototipação de software e design participativo: uma experiência do atlântico. *In:* SIMPÓSIO BRASILEIRO DE FATORES HUMANOS EM SISTEMAS COMPUTACIONAIS, 8.,2008, Porto Alegre. *Proceedings of the VIII Brazilian Symposium on Human Factors in Computing Systems*. Porto Alegre: Sociedade Brasileira de Computação, 2008.

Note também como o autor trabalhou o conceito de *design* participativo.

2.2.4 Modelo espiral

O modelo espiral, ou também citado na literatura como modelo de Boehm (1988), tem como raiz o modelo iterativo incremental e como preocupação central a mitigação de riscos.

A diferença fundamental do modelo espiral para os outros modelos que discutimos até agora é que cada ciclo completo, ou cada iteração, no modelo proposto por Boehm em 1988, não produz ou implementa um sistema ou uma parte do sistema de *software*.

Diferentemente do modelo incremental, no qual temos como resultado ao final da fase de implementação, que corresponde ao fim de um ciclo, um conjunto de requisitos especificados, documentados, projetados, construídos e validados, ao final do ciclo de um modelo espiral poderíamos ter como produto apenas a especificação de requisitos.

O modelo espiral é composto de quatro fases:

- Definição dos objetivos.
- Mitigação dos riscos.
- Desenvolvimento do produto.
- Planejamento da próxima fase.

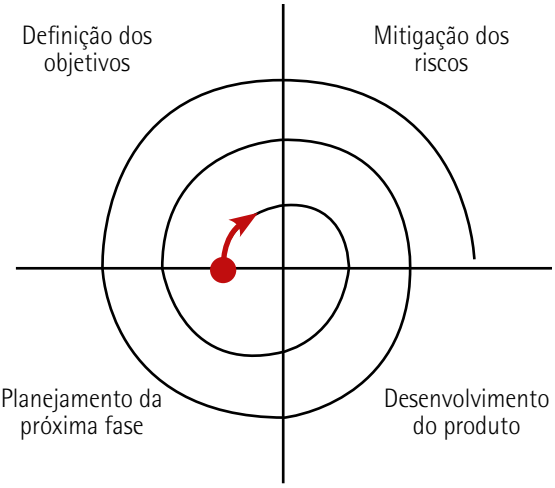

Figura 6 – Modelo espiral

A figura anterior representa o modelo espiral proposto por Boehm (1988). Cada linha da espiral, em cada fase, corresponde a uma sequência de atividades. Por exemplo: na fase de Desenvolvimento do Produto podemos ter Elicitação de Requisitos, Documentação e Validação dos Requisitos.

A ênfase na diminuição do risco se dá pelo fato de que entre a definição do que precisa ser feito e o efetivo desenvolvimento, existe a fase de mitigação dos riscos. Nela, os riscos são identificados, mitigados e, a partir disso, são traçadas estratégias para minimizar os impactos para as fases seguintes.

Saiba mais

Existem algumas variantes do modelo espiral adotadas por outros autores, saiba mais sobre essas variantes e sobre o modelo espiral em:

BOEHM, B.; HANSEN, W. J. *Spiral development*: experience, principles, and refinements. Pittsburg: CMU/SEI,, 2000. Disponível em: <http://csse.usc.edu/csse/TECHRPTS/2000/usccse2000-525/usccse2000-525.pdf>. Acesso em: 18 dez. 2014.

Observação

Software Engineering Institute (SEI) é um centro de pesquisa e desenvolvimento ligado à Universidade Carnegie-Mellow que tem ênfase no estudo da Engenharia de Software. Veja mais em: <http://www.sei.cmu.edu/>.

2.2.5 Processo unificado

Idealizado por Jacobson, Rumbaugh e Booch (1999), o Processo Unificado, também conhecido como UP (Unified Process), tem em seu fundamento a estrutura do modelo iterativo incremental.

Observação

Iar Jacobson, James Rumbaugh e Grady Booch são considerados três dos maiores expoentes da Engenharia de Software.

O UP é composto por quatro fases:

- Iniciação: nesta fase é definido algo muito importante para qualquer projeto de *software*, o escopo do projeto, que é a relação das necessidades que serão atendidas no projeto de *software* e aquelas que não serão.

- Elaboração: o objetivo desta fase é detalhar o escopo, produzindo o modelo de requisitos do sistema. A partir dos requisitos, são elaboradas soluções para atender a essas necessidades, que podemos chamar de arquitetura do sistema. No decorrer de toda essa fase, são identificados os riscos do projeto e quais estratégias serão usadas para redução dos riscos. Nesta fase pode-se produzir o plano de projeto, ou plano de desenvolvimento, que consiste no mapeamento dos requisitos, as soluções de *software* desenvolvidas para solução de cada requisito e o planejamento dos riscos associados para cada solução.

- Construção: esta fase está associada ao desenvolvimento do projeto, que consiste em projetar, codificar e testar o sistema de *software*.

- Transição: nesta fase, o sistema de *software* é entregue para o usuário final, sendo comum que existam correções e adaptações a serem feitas, decorrentes das mais diversas naturezas, como adaptação ao ambiente final do usuário ou novas demandas.

Segundo os próprios autores, o UP é baseado em três pilares:

- Dirigido por casos de uso.

- Centrado na arquitetura.

- Iterativo e Incremental.

Observação

Caso de uso é, de forma genérica, a descrição detalhada dos requisitos de um sistema de *software* que seguem uma determinada metodologia e um determinado padrão. Iremos detalhar essa abordagem adiante.

Uma variação importante do modelo de processo unificado, que é utilizada largamente pelas grandes organizações atualmente, é o RUP (Rational Unified Process) (KRUCHTEN, 2003).

O RUP é um modelo de processos, desenvolvido pela Rational, que associou o Modelo de Processo Unificado às melhores práticas da Engenharia de Software, que são:

- Desenvolva iterativamente.
- Gerencie requisitos.
- Use arquitetura de componentes.
- Modele visualmente.
- Verifique continuamente a qualidade.
- Gerencie mudanças.

O modelo proposto pela Rational especifica, de forma sistemática, a sequência que as atividades, ou também chamadas de disciplinas, devem ser realizadas dentro de cada fase, quem são os responsáveis e quais produtos são resultantes de cada atividade.

Observação

A Rational Software é uma empresa privada, fundada na década de 80 com o objetivo de desenvolver ferramentas, técnicas e práticas para Engenharia de Software. Em 2003, foi adquirida pela IBM.

As fases do RUP são as mesmas definidas no UP: iniciação, elaboração, construção e transição.

A relevância do RUP, no contexto desse trabalho, dá-se pelo fato de que o modelo incorpora como ferramenta de apoio, a modelagem visual, particularmente utilizando a Linguagem de Modelagem Unificada – UML (Unified Modeling Language).

Além do mais, o modelo dá ênfase à utilização da orientação a objetos como boa prática de Engenharia de Software quando incorpora a utilização da arquitetura de componentes.

Abordaremos ambos os assuntos – UML e Orientação a objetos – detalhadamente adiante.

Saiba mais

O RUP é um processo proprietário da Rational, de tal forma que não é permitido o seu uso sem licença. O OpenUP, a exemplo do RUP, é um modelo de processo unificado e seu uso é gratuito. Saiba mais em:

<http://epf.eclipse.org/wikis/openup/>.

2.3 Processo X pessoas

Como abordamos anteriormente, qualquer modelo de processo de Engenharia de Software se preocupa em definir **quem** produz **o que**, **como** e **quando**:

- **O que** será produzido: qual é o produto ou o artefato do processo.
- **Como**: atividade, ou sequencia de atividades, a serem realizadas, em uma determinada sequência para produção do artefato do processo.
- **Quando**: definição de qual momento o artefato será produzido, definição das pré-condições e pós-condições.
- **Quem**: indicação do responsável pela atividade.

Quais as responsabilidades de uma equipe dentro de um projeto? Qual é o papel da pessoa dentro do processo de desenvolvimento?

O RUP, por exemplo, não tem ênfase no indivíduo, mas sim no papel. O papel é o responsável pela execução de uma determinada tarefa, que é responsável por um determinado artefato (KRUCHTEN, 2003). Segundo a empresa Wthreex (s.d.), o papel "define o comportamento e as responsabilidades de um indivíduo ou de um conjunto de indivíduos que trabalham juntos como uma equipe, no contexto de uma organização de engenharia de *software*".

É exatamente o que ocorre nas organizações de desenvolvimento de *software*. Um indivíduo desempenha um ou mais papéis dentro de um projeto de desenvolvimento. Os papéis e as responsabilidades variam de acordo com cada modelo de processo.

No caso do RUP, os papéis são sistematicamente definidos, por exemplo: o analista de sistemas é responsável, entre outras coisas, por desenvolver o plano de gerenciamento de requisitos e o arquiteto de software executa, entre outras atividades, a estruturação do modelo de implantação.

Bezerra (2006) tem ideia semelhante e destaca que o componente humano é fator importante no desenvolvimento de um sistema de *software*, uma vez que ele é composto de tarefas "altamente cooperativas". Ele ainda destaca, de forma semelhante ao RUP, que uma pessoa pode desempenhar diversas funções que tipicamente um projeto de desenvolvimento de *software* possui, na grande maioria dos casos, independentemente do modelo de processo adotado, que são os seguintes papéis:

Gerente de projetos

Profissional responsável por gerir o projeto, que compreende, entre muitas coisas, coordenar e acompanhar as atividades do projeto e alocar as pessoas para as funções determinadas.

Saiba mais

Saiba mais sobre as atribuições e responsabilidades do Gerente de Projetos no *site* do PMI (Project Management Institute) em: <http://www.pmi.org/>.

Lá você poderá ter os primeiros contatos com o PMBOK, que contém as melhores práticas para a profissão.

Analista

Analista de sistemas, muitas vezes chamado de analista de negócios, é responsável por captar as necessidades do negócio e transformá-las em requisitos do sistema de *software*.

Alguns conhecimentos e habilidades são necessários, como: conhecimento ou familiaridade com o negócio, boa capacidade de comunicação oral e escrita, conhecimento de computação, entre outros.

Bezerra (2006) cita algo de importante destaque: o analista de sistemas deve ser ético, uma vez que, em muitas situações, se depara com informações e contratos que envolvem confidencialidade.

Projetista

Responsável por produzir uma solução computacional para os problemas identificados. Por exemplo: um projetista de interface gráfica do usuário, que é especialista em fornecer soluções para interfaces padrão para sistemas operacionais da Apple ou da Microsoft.

Arquiteto de *software*

Responsável pela elaboração da arquitetura do sistema. Como e quais serão os componentes internos do sistema de *software* e como esses componentes irão se comunicar para atender a uma determinada necessidade.

Geralmente, o arquiteto trabalha em contato com o gerente de projetos, ajudando a organizar o desenvolvimento e estimando tamanho e tempo de desenvolvimento, e com o corpo técnico, orientando decisões técnicas de desenvolvimento.

Desenvolvedores

Também chamados de programadores, os desenvolvedores têm a responsabilidade de implementar o projeto. Transformar a arquitetura e a solução do projeto em solução computacional.

Clientes

O usuário é parte fundamental do processo de desenvolvimento, ele precisa estar envolvido e com foco desde o momento do levantamento de requisitos até a validação.

Avaliadores de qualidade

São responsáveis por garantir que o sistema de *software* atenda a todas as necessidades do negócio.

Saiba mais

A discussão sobre o processo de qualidade é ampla. Leia:

COSTA, I. *et al. Qualidade em tecnologia da informação*. São Paulo. Atlas, 2013.

A participação do profissional no processo de desenvolvimento de um sistema de *software* é fundamental, independentemente do modelo a ser seguido. Ela é fundamental, inclusive, na escolha de um modelo de processo a ser adotado.

Entre os vários processos aqui discutidos, e entre os outros tantos que existem, cabe ao profissional levar em consideração as características do projeto e as características do modelo de processo para saber qual a melhor decisão.

Na Engenharia de Software, o indivíduo tem papel fundamental.

2.4 O paradigma da orientação a objetos

Antes de entendermos o que é o paradigma da orientação a objetos, vamos entender o que é um paradigma: "Um paradigma é um conjunto de regras que estabelecem fronteiras entre o que é certo e errado, entre o que é verdadeiro e o que é falso, entre o que se deve fazer e o que não se deve fazer[...] Ele funciona como um conjunto de categorias que define e orienta o comportamento das pessoas" (CHIAVENATO, 2008, p. 8).

O paradigma da orientação a objetos é uma forma de se desenvolver um sistema de *software* que o enxerga como um conjunto de componentes que interagem entre si para resolver um determinado problema. A cada componente, dá-se o nome de objeto.

A motivação da abordagem orientada a objetos se dá pela tentativa de aproximar o desenvolvimento de *software* àquilo que acontece no mundo real.

Imagine a seguinte situação do nosso cotidiano: um cliente de uma agência bancária vai até um terminal de autoatendimento para sacar uma determinada quantia em dinheiro. O cliente possui algumas características importantes nesse processo, como número de agência, número de conta e senha. Além disso, ele executa algumas ações para efetuar o saque, como inserir cartão, informar senha e retirar dinheiro.

Outro componente importante dessa cena é o próprio terminal de autoatendimento, que, por sua vez, também possui características e comportamentos importantes, como dispensar dinheiro.

Se pensarmos no paradigma da orientação a objetos, podemos interpretar que temos dois objetos no processo: cliente e terminal. Tanto um quanto o outro possuem características, comportamentos, e executam determinadas ações.

Em orientação a objetos, cada objeto também possui características, denominadas atributos, e a ação a ser executada por esse objeto, que tem o nome de método.

Dentro desse cenário, os dois, cliente e terminal, devem interagir para que o processo seja efetuado com sucesso. De maneira análoga, na orientação a objetos, dois objetos interagem entre si para executar uma determinada tarefa computacional por meio da troca de mensagens.

Uma mensagem pode ser interpretada como a requisição que um objeto faz a outro objeto para que ele execute um determinado serviço.

Bezerra (2006) interpreta a orientação a objetos como uma técnica para modelar sistemas que diminui a diferença semântica entre a realidade sendo modelada e os modelos construídos.

Antes do paradigma da orientação a objetos atingir sua plenitude, no início da década de 1990, o paradigma estruturado era amplamente utilizado para modelagem de sistemas.

O paradigma estruturado é baseado em dois elementos: informações (dados) e processos. Nesse pensamento, os processos agem diretamente sobre os dados para a execução de uma determinada tarefa.

Observação

Na década de 1990, o termo processamento de dados era utilizado para descrever a área de desenvolvimento de sistemas, muito em função do paradigma estruturado.

O grande problema enfrentado pelo paradigma estruturado foi a atomicidade dos processos. As unidades de processamento eram organizadas em poucos e grandes blocos – ou muitas vezes em um único bloco – responsáveis pela realização de toda a tarefa.

Essa organização levou a sistemas muito complexos e de difícil manutenção, o que não combinou com a crescente necessidade do mercado, que já debatemos anteriormente.

Em contrapartida, o paradigma orientado a objetos produz modelos com componentes autônomos, chamados objetos, que possuem suas próprias características e informações e seus próprios comportamentos responsáveis pela manutenção dessa informação.

A divisão de responsabilidades proposta pela orientação a objetos proporciona algumas vantagens:

- A modelagem próxima ao mundo real gera modelos mais fáceis de entender e manter.
- Aumento do reúso de códigos.
- Aumento da manutenibilidade: sistemas mais manuteníveis, mudanças nos requisitos não implicam necessariamente na alteração do sistema todo.

O paradigma da orientação a objetos é baseado nos seguintes pilares:

- classes;
- objeto;
- abstração;
- encapsulamento;
- herança;
- polimorfismo;
- ligação, mensagem.

2.4.1 Classe

Classe de objetos pode ser definida como um grupo de objetos com as mesmas propriedades (atributos), comportamentos (operações), relacionamentos e semântica. A classe dos profissionais de engenharia química, por exemplo. Todos devem possuir curso superior em engenharia química, registro no CREA e podem executar apenas determinadas tarefas, como desenvolver laudos.

Ou ainda, voltando ao nosso exemplo do terminal de autoatendimento, podemos entender que o nosso cliente faz parte da classe "Cliente do banco", pois todos os clientes do banco possuem agência, número de conta, senha. Além disso, todos eles podem efetuar saques.

As classes representam o bloco de construção mais importante de qualquer sistema orientado a objetos, pois são utilizadas para capturar o domínio do problema no qual o sistema está sendo desenvolvido.

Classes devem possuir responsabilidades bem definidas, cada responsabilidade representa um contrato ou obrigações dela, sendo assim, podemos entender que uma classe é uma "especificação" de um objeto.

2.4.2 Objeto

Já vimos que um objeto é um elemento do mundo real, que possui relevância para solução de um determinado problema e que esse objeto possui características e executa determinadas ações (métodos).

Esse conjunto de informações, também chamado de atributos, identifica um objeto em uma determinada classe de objetos.

Voltando novamente ao nosso exemplo do terminal de autoatendimento, os clientes que fazem parte da classe "Cliente do banco" possuem agência, número de conta, senha. Podemos entender, portanto, que um objeto do tipo "Cliente" é identificado pelo conjunto dessas informações.

Por exemplo: João é um cliente do banco XYZ, número de agência 1234 e número de conta corrente 12345-0. Não existe, no banco XYZ, nenhum outro cliente com essas informações. Sendo assim, podemos afirmar que todo objeto possui uma identidade, representada por um conjunto de informações atribuídas às suas características. Podemos afirmar também que todo objeto possui um estado, definido também pelo conjunto dessas informações.

Suponhamos que o cliente possua também o atributo "Saldo de conta-corrente". A informação desse atributo define seu estado em função do tempo de vida desse objeto. Por exemplo: João tem um saldo de R$ 900,00 na conta-corrente em um determinado dia. No dia seguinte, João efetua um saque de R$ 100,00, passando a ter um saldo de R$ 800,00.

Diante dessa situação, podemos afirmar também que o estado de um objeto pode ser alterado pelos seus métodos.

2.4.3 Abstração

> No tratamento de problemas mais complexos, onde seja exigido um método de resolução mais longo, torna-se interessante estabelecer etapas menores e mais simples no método de resolução para depois, compondo essas etapas menores, alcançar a resolução do problema (MARTINS; RODRIGUES, 2006, p. 85).

O conceito de abstração está ligado à nossa capacidade, enquanto analistas, de resolver problemas, de selecionar determinados aspectos do problema e isolar o que é importante do que não é para um determinado propósito. Abstração é dar ênfase àquilo que é essencial.

Se fizermos uma analogia com as fases de um projeto de construção de uma casa, seria algo como pensarmos na planta em vez dos tijolos. Pensar em como organizar o sistema elétrico e hidráulico em vez de pensar em luminárias, lâmpadas e torneiras.

Além disso, abstração está ligada à nossa capacidade de estabelecer um modelo de objetos bem decomposto, de tal forma que cada objeto seja uma unidade autônoma, estabelecer a dependência apenas entre objetos que tenham alguma colaboração em um determinado momento do processo.

2.4.4 Encapsulamento

Encapsulamento significa deixar visível apenas o que é necessário para a comunicação entre dois objetos, como detalhes da implementação ou a lógica algorítmica de um método.

Voltando ao nosso exemplo do terminal de autoatendimento, o objeto do tipo Cliente precisa saber o que o terminal faz para dispensar o dinheiro? Se ele efetua contagem das notas, se existem outros objetos que fazem parte do processo? A resposta é não. Contanto que a comunicação entre eles, cliente e terminal, esteja funcionando corretamente, não existem problemas pelo fato do cliente não conhecer detalhes da implementação do método de dispensar dinheiro.

2.4.5 Herança

No mundo real existem objetos que possuem características semelhantes, mas que estão em classes diferentes? Clientes Pessoa Física (PF) e Clientes Pessoa Jurídica (PJ) de um banco, por exemplo, possuem características e operações semelhantes?

Sim, no mundo real existem objetos que possuem características semelhantes, mas que estão em classes diferentes. Temos, nesses casos, o conceito de superclasse e subclasse.

Por exemplo, cães, gatos e roedores são exemplos de mamíferos, possuem características e comportamentos semelhantes aos de todos os mamíferos, mas possuem também suas diferenças. Nessa situação podemos dizer que cães, gatos e roedores são subclasses de mamífero, que é uma superclasse.

Se pensarmos em uma estrutura hierárquica, podemos dizer que cães, gatos e roedores são especializações de mamíferos, ou seja, possuem tudo o que os mamíferos possuem e algo a mais que os diferenciam, enquanto isso, mamífero é uma generalização de cães, gatos e roedores, seguindo exatamente a mesma lógica.

Em orientação a objetos, temos exatamente o mesmo conceito. Todos os métodos e atributos da classe mãe, ou superclasse, estarão presentes nas classes filhas, ou subclasses, sem que haja a necessidade de reescrevê-los.

A figura a seguir mostra um exemplo de herança de objetos. A classe Cliente possui os seguintes atributos: *ContaCorrente*, *NumeroAgencia* e *NumeroBanco* além de possuir o método *EfetuarSaque*.

As subclasses ClientePF e ClientePJ possuem os mesmos atributos e métodos da classe Cliente e outros que os diferenciam.

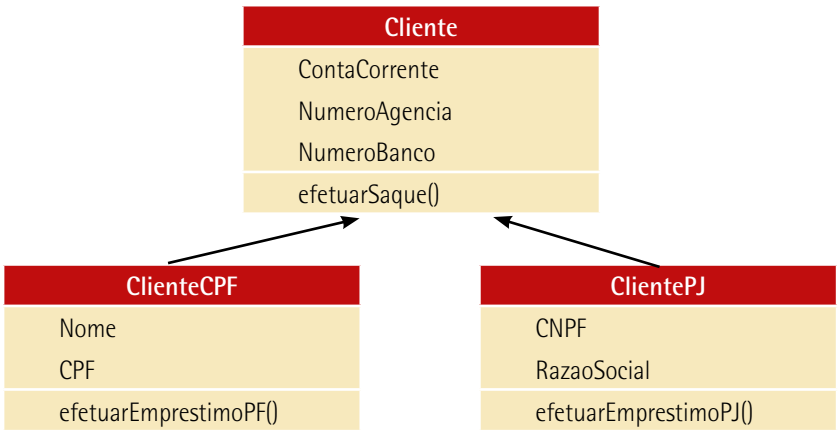

Figura 7 – Exemplo de herança

Além de possuir os atributos da classe mãe, ClientePF e ClientePJ possuem suas próprias características: *CPF* e *Nome*; *CNPJ* e *RazaoSocial*, respectivamente. E seus próprios métodos: *efetuarEmprestimoPF* e *efetuarEmprestimoPJ*, respectivamente.

Existem dois tipos de herança:

- Simples: quando uma classe filha herda características de apenas uma classe mãe.
- Composta: quando uma classe filha herda características de mais de uma classe mãe.

O mecanismo de herança, também chamado de generalização-especialização, é um dos fundamentos da orientação a objetos mais importantes, principalmente por proporcionar reúso.

2.4.6 Polimorfismo

Polimorfismo é quando um objeto tem um comportamento diferente para a mesma ação.

Fazendo novamente uma analogia aos animais, podemos ter a classe caranguejo e a classe cão. Ambas são subclasses de animais, que realizam a ação **andar**. Se fizermos com que um caranguejo ande cinco metros, certamente o comportamento será diferente se enviarmos a mesma mensagem para o cão, muito embora eles respondam à mesma mensagem e o resultado seja exatamente o mesmo: ambos andarão cinco metros.

Em orientação a objetos, polimorfismo significa que um método definido na classe mãe é redefinido, tem uma nova implementação, na classe filha, sendo obrigatória a mesma assinatura para o método.

Observação

A assinatura de um método, assim como a assinatura de uma função em algoritmos, é composta de: tipo de retorno + nome do método + parâmetros de entrada (tipo de dado, nome do parâmetro).

2.4.7 Ligação e mensagem

Mensagem é um meio de comunicação entre objetos com o objetivo de ativar um processamento. Para que um objeto execute um método, ou seja, tenha um determinado comportamento, é necessário enviar a este objeto uma mensagem solicitando a execução deste método.

Se um objeto envia uma mensagem para outro, eles possuem uma ligação, chamada de associação simples, ou seja, uma ligação que acontece apenas em um determinado momento.

Pode acontecer que um objeto tenha em sua composição outro objeto. Neste caso existe uma relação todo-parte. Por exemplo: o objeto *terminal de autoatendimento* é composto dos objetos *monitor*, *leitora de cartão* e *dispensador de dinheiro*. Assim, seguindo na mesma nomenclatura, a *leitora de cartão* é parte do objeto *terminal de autoatendimento*.

A composição todo-parte entre dois objetos determina o ciclo de vida de um objeto. Quando o objeto que compõe outro não existe fora desse contexto, dizemos que existe uma associação do tipo composição. Por exemplo: o objeto *item de nota fiscal* faz parte do objeto *nota fiscal*, todavia, ele, por si só, não existe sem a existência de uma nota fiscal.

Quando o objeto que compõe o outro existe fora do contexto, dizemos que há uma associação do tipo agregação. Podemos citar o objeto *livro*, que faz parte do objeto *biblioteca*, todavia, ele existe mesmo se não existir o objeto *biblioteca* para um determinado contexto, por exemplo, a automação de uma universidade, onde o livro existe mesmo que não esteja na biblioteca, mas ela é composta por uma coleção de livros.

2.5 A linguagem de modelagem unificada

O que é uma linguagem? O dicionário Michaelis (1998) define linguagem como um conjunto de sinais falados (glótica), escritos (gráfica) ou gesticulados (mímica), de que se serve o homem para exprimir suas ideias e sentimentos.

Vamos nos atentar à linguagem escrita. Eu, por exemplo, escolhi um conjunto de caracteres para escrever este material. Em uma sequência lógica, esses caracteres expressam uma linguagem escrita. Você, que está lendo este material, olha esses caracteres e entende a linguagem utilizada, ou seja, o que eu estou querendo dizer, porque sabe interpretar a sequência deles.

Quando estamos desenvolvendo um sistema de *software*, escolhemos uma linguagem de programação. Essa linguagem também é uma sequência lógica de símbolos. Algumas composições de símbolos constituem uma palavra reservada, que, via de regra, cada linguagem possui a sua.

O compilador interpreta essa linguagem, esse conjunto de caracteres, inclusive as palavras reservadas, e produz o resultado que você, desenvolvedor, esperava.

Caso você não tenha escrito algo que faça coerência, seu código não é compilado. Isso acontece porque o modelo de código produzido não é interpretado pelo compilador.

A Linguagem de Modelagem Unificada, ou Unified Modeling Language, conhecida como UML, é uma linguagem para modelar sistemas orientados a objetos.

Por se tratar de uma linguagem, ainda que visual, também é composta por elementos (gráficos) que também seguem algumas regras.

- Regra de sintaxe: cada elemento deve ser desenhado, representado, seguindo um padrão.
- Regra semântica: define o significado de cada elemento dentro de um contexto.

Por exemplo: se combinarmos que a figura a seguir representa uma impressora, todas as vezes que você, leitor, ver o símbolo, vai entender que estou querendo representar uma impressora e que qualquer símbolo diferente dele não é a representação de uma impressora.

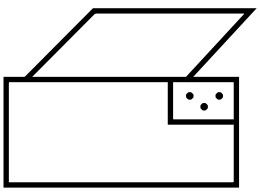

Figura 8 – Representação gráfica de uma impressora

Seguindo a mesma lógica, se combinarmos que um retângulo é a representação de uma folha de papel e que uma linha tracejada representa uma agregação, você irá olhar a figura a seguir e irá interpretar que esse modelo representa algo como "uma impressora possui folhas de papel".

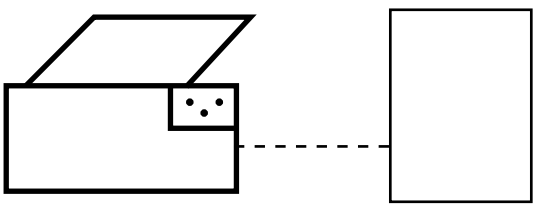

Figura 9 – Representação gráfica de um modelo impressora/papel

Concorda que esse é um modelo que vale apenas para mim e para você, leitor, nos comunicarmos? Isso porque nós "combinamos o jogo". Em contrapartida, qualquer outro que olhar esse modelo vai enxergar apenas "um tipo de máquina ligada a um retângulo por uma linha tracejada". Portanto, a minha linguagem de representação não é eficiente para que eu me comunique em larga escala.

A UML surgiu da necessidade de se desenvolver uma linguagem que fosse padrão para modelagem de sistemas orientada a objetos e que fosse interpretável por qualquer um: usuários, desenvolvedores, arquitetos e gerentes.

Os elementos gráficos são utilizados para representar uma determinada visão do sistema, que vai desde a representação dos requisitos até a visão de implementação. O que nos permite afirmar que, se pensarmos no modelo de processo da Engenharia de Software, enquanto o paradigma da orientação a objetos tem sua área de contribuição restrita à fase de projeto, a UML é uma ferramenta que dá suporte a todo o processo, desde a análise de requisitos, passando pelo projeto, construção e até a validação.

Segundo os criadores da UML, Booch, Jacobson e Rumbaugh (2006), um sistema de *software* pode ser dividido em cinco visões, sendo que, dependendo da complexidade, nem todas precisam ser desenvolvidas:

- Caso de Uso: representa o sistema de um ponto de vista externo, como ele interage com agentes externos, como usuários ou outros sistemas.
- Projeto: representa a arquitetura do sistema de *software*.
- Implementação: representa os componentes e subsistemas.
- Implantação: representa a distribuição física do sistema e seus componentes e como estes irão se comunicar.
- Processo: dá ênfase na representação do paralelismo e sincronização dos componentes do sistema.

Cada visão é composta por um ou mais diagramas UML. Um diagrama UML é uma composição de elementos gráficos com o objetivo de representar alguma perspectiva do sistema.

Nas páginas seguintes, detalharemos cada uma das visões e diagramas, sempre traçando um paralelo com as atividades do processo de desenvolvimento da Engenharia de Software.

Lembrete

Com a evolução da tecnologia e dos computadores, a forma como a informação era trabalhada, trafegada, levada ao usuário e gerenciada passou a ter papel de fundamental importância.

Saiba mais

A especificação e a definição completas da UML estão disponíveis no *site* da OMG. Saiba mais em: <http://www.uml.org/>.

Resumo

Esta unidade apresentou os conceitos introdutórios e fundamentais sobre análise e desenvolvimento de sistemas.

O aumento no uso dos computadores pessoais, ou PCs, associados à notória evolução das estruturas de *hardware*, mudou as estruturas de como os sistemas de *software* eram construídos. Um dos motivadores do aumento do uso de computadores pode ser atribuído à teoria de Moore, também conhecida como Lei de Moore, que resultou em computadores de maior capacidade, menores e mais baratos.

A informação passou a ter papel fundamental nas grandes organizações. Passa ao papel principal a maneira como essa informação chega até o usuário, como ela é trabalhada, trafegada e gerenciada até que chegue ao seu destino final.

Surgiu então o conceito de **sistemas de informação**, um conceito formado por alguns pilares: informações (ou dados), pessoas, processos e tecnologias com o objetivo apoiar ou melhorar o processo de negócio de uma empresa.

Sistemas de informação dão ênfase na informação gerada em um processo de uma organização, com o objetivo de que o tratamento dessa informação gere valor ao processo e a organização (BEZERRA, 2006).

Ao tratarmos de tecnologia de sistemas de informação, abordamos irrestritamente toda plataforma tecnológica que apoia o processo de negócio: infraestrutura de rede, servidores, computadores, sistemas operacionais e sistemas de *software*.

O desenvolvimento de sistemas de *software* também passou, então, por uma fase de desafios. Como desenvolver sistemas de *software* que atendessem às crescentes necessidades?

O objetivo da Engenharia de Software é apoiar o desenvolvimento de *software* a partir de técnicas para especificar, projetar, manter e gerir um sistema de *software* (SOMMERVILLE, 2010). Ela é baseada em três pilares: métodos, ferramentas e processos.

O processo de desenvolvimento de *software* se resume a um conjunto de atividades executadas em uma determinada sequência e que tem por objetivo garantir a qualidade do produto que será entregue no final do processo: o sistema de *software*.

Existem muitos modelos de processo que podemos aplicar a um projeto de desenvolvimento, como o modelo cascata e o modelo iterativo, cada um com as suas características.

A escolha de um processo de desenvolvimento passa principalmente pela natureza do problema a ser resolvido. Todavia, todos possuem as seguintes atividades fundamentais: especificação de *software*, projeto e implementação de *software*, validação de *software* e evolução de *software*.

Qualquer que seja o modelo adotado, o indivíduo tem papel fundamental. Os profissionais que fazem parte de uma equipe de projeto desempenham determinados papéis.

Papéis esses que são responsáveis pela execução de determinadas atividades e produção de determinados resultados. Alguns dos papéis comuns em um projeto são: gerente de projetos, analista de sistemas, projetista, arquiteto, desenvolvedor, analista de qualidade e, principalmente, cliente.

Mudando nossa ênfase para métodos na Engenharia de Software, vimos que o paradigma da orientação a objetos surgiu como uma alternativa para uma modelagem próxima ao mundo real, para gerarmos modelos mais fáceis de entender e manter. Além de proporcionar ganhos que outros paradigmas não proporcionam, como aumento do reúso de código e da manutenibilidade, que é a capacidade de um sistema de *software* ser facilmente manutenível.

O paradigma da orientação a objetos é apoiado nos seguintes pilares: classes, objeto, abstração, encapsulamento, herança, polimorfismo, ligação e mensagem.

Com enfoque em ferramentas, vimos a necessidade de termos uma ferramenta que desse suporte a todo o processo de desenvolvimento de um *software* orientado a objetos. Uma linguagem que pudesse ser padrão para a comunicação entre todos os envolvidos em um projeto.

Surgiu a UML, uma linguagem de modelagem visual de sistemas orientada a objetos, que possuem elementos gráficos de modelagem que representam visões de um sistema de *software* e que possuem regras de sintaxe e semântica.

Por tudo isso, notamos uma evolução e uma maturidade no desenvolvimento de *software*.

Exercícios

Questão 1. (Enade 2008) O gerenciamento de configuração de *software* (GCS) é uma atividade que deve ser realizada para identificar, controlar, auditar e relatar as modificações que ocorrem durante todo o desenvolvimento ou mesmo durante a fase de manutenção, depois que o *software* for entregue ao cliente. O GCS é embasado nos chamados itens de configuração, que são produzidos como resultado das atividades de engenharia de *software* e que ficam armazenados em um repositório. Com relação ao GCS, analise as duas afirmativas apresentadas a seguir:

I – No GCS, o processo de controle das modificações obedece ao seguinte fluxo: começa com um pedido de modificação de um item de configuração, que leva à aceitação ou não desse pedido e termina com a atualização controlada desse item no repositório.

PORQUE

II – O controle das modificações dos itens de configuração baseia-se nos processos de *check-in* e *check-out* que fazem, respectivamente, a inserção de um item de configuração no repositório e a retirada de itens de configuração do repositório para efeito de realização das modificações.

Acerca dessas afirmativas, assinale a alternativa correta:

A) As duas afirmativas são verdadeiras e a segunda é uma justificativa correta da primeira.

B) As duas afirmativas são verdadeiras, mas a segunda não é uma justificativa correta da primeira.

C) A primeira afirmativa é uma proposição verdadeira e a segunda é uma proposição falsa.

D) A primeira afirmativa é uma proposição falsa, enquanto a segunda é uma proposição verdadeira.

E) As duas afirmativas são proposições falsas.

Resposta correta: alternativa B.

Análise da questão

As duas afirmativas são verdadeiras. A primeira trata da responsabilidade da gestão de configuração de acompanhar uma modificação ou alteração desde o seu início, ou seja, de sua solicitação, passando por sua aprovação e também implementação, até seu fim, caracterizado pelo registro no repositório. Esse repositório permite a visualização de todas as modificações ocorridas e, por conseguinte, a configuração atual do referido *software*.

A segunda afirmativa explica como são realizados os registros (inserção e retirada) dos itens de configuração no repositório.

Sendo assim, apesar de verdadeiras, pode-se afirmar que uma não é justificativa de outra, portanto, a alternativa B é a correta.

Questão 2. Considere que você trabalha em uma empresa de desenvolvimento de *software* e que a empresa tenha decidido desenvolver um novo editor de texto para colocar no mercado. Esse editor deve fornecer recursos adicionais de apoio à autoria, embasado no estilo de escrita do usuário, o que o torna um *software* de funcionalidade mais complexa. Considere que a empresa deseje disponibilizar o produto no mercado em versões que agreguem esse suporte de forma gradativa, fazendo análises de risco para avaliar a viabilidade de desenvolvimento de uma nova versão. Tendo de escolher um modelo de processo para desenvolver esse editor e conhecendo as características dos modelos existentes, entre os modelos a seguir, qual é o modelo apropriado para esse caso?

A) Cascata.

B) Espiral.

C) RAD (Rapid Application Development).

D) Prototipação.

E) Cleanroom.

Resolução desta questão na plataforma.