Gestão de Projetos de Software

Brasília-DF.

Elaboração

Jorge Umberto Scatolin

Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

Sumário

Apresentação

Caro aluno

A proposta editorial deste Caderno de Estudos e Pesquisa reúne elementos que se entendem necessários para o desenvolvimento do estudo com segurança e qualidade. Caracteriza-se pela atualidade, dinâmica e pertinência de seu conteúdo, bem como pela interatividade e modernidade de sua estrutura formal, adequadas à metodologia da Educação a Distância – EaD.

Pretende-se, com este material, levá-lo à reflexão e à compreensão da pluralidade dos conhecimentos a serem oferecidos, possibilitando-lhe ampliar conceitos específicos da área e atuar de forma competente e conscienciosa, como convém ao profissional que busca a formação continuada para vencer os desafios que a evolução científico-tecnológica impõe ao mundo contemporâneo.

Elaborou-se a presente publicação com a intenção de torná-la subsídio valioso, de modo a facilitar sua caminhada na trajetória a ser percorrida tanto na vida pessoal quanto na profissional. Utilize-a como instrumento para seu sucesso na carreira.

Conselho Editorial

Organização do Caderno de Estudos e Pesquisa

Para facilitar seu estudo, os conteúdos são organizados em unidades, subdivididas em capítulos, de forma didática, objetiva e coerente. Eles serão abordados por meio de textos básicos, com questões para reflexão, entre outros recursos editoriais que visam tornar sua leitura mais agradável. Ao final, serão indicadas, também, fontes de consulta para aprofundar seus estudos com leituras e pesquisas complementares.

A seguir, apresentamos uma breve descrição dos ícones utilizados na organização dos Cadernos de Estudos e Pesquisa.

Provocação

Textos que buscam instigar o aluno a refletir sobre determinado assunto antes mesmo de iniciar sua leitura ou após algum trecho pertinente para o autor conteudista.

Para refletir

Questões inseridas no decorrer do estudo a fim de que o aluno faça uma pausa e reflita sobre o conteúdo estudado ou temas que o ajudem em seu raciocínio. É importante que ele verifique seus conhecimentos, suas experiências e seus sentimentos. As reflexões são o ponto de partida para a construção de suas conclusões.

Sugestão de estudo complementar

Sugestões de leituras adicionais, filmes e sites para aprofundamento do estudo, discussões em fóruns ou encontros presenciais quando for o caso.

Atenção

Chamadas para alertar detalhes/tópicos importantes que contribuam para a síntese/conclusão do assunto abordado.

Saiba mais

Informações complementares para elucidar a construção das sínteses/conclusões sobre o assunto abordado.

Sintetizando

Trecho que busca resumir informações relevantes do conteúdo, facilitando o entendimento pelo aluno sobre trechos mais complexos.

Para (não) finalizar

Texto integrador, ao final do módulo, que motiva o aluno a continuar a aprendizagem ou estimula ponderações complementares sobre o módulo estudado.

Introdução

A indústria de equipamentos computacionais iniciou fazendo e aperfeiçoando os mais variados hardwares. Por muito tempo um equipamento teve as lógicas de processamento sendo executadas diretamente no hardware. Foi onde o mundo tecnológico pôde conhecer tecnologias de processador como a CISC e RISC.

Com a concorrência aumentando, essa mesma indústria se viu diante da necessidade de inovar pela sobrevivência. Precisou urgentemente começar a diminuir os custos de hardware para poder se tornar competitiva. E como conciliar o aumento da demanda por equipamentos cada vez mais potentes com a necessidade de baixar os custos? Software! Com uma única palavra, os engenheiros criaram uma nova funcionalidade para o software, que foi revolucionário. Processamentos que antes eram feitos no hardware, hoje poderiam ser feitos pelos softwares que, além de possuir uma produção em escala menor, suas atualizações, seja de melhorias ou correções, são mais eficientes e baratas.

Entretanto, com grandes demandas de produção vêm as dificuldades! Desenvolvedores viram seus softwares passar de dezenas de linhas de código para milhões em pouco tempo. Ao mesmo tempo que surgiu a expansão da indústria, as técnicas de gerenciamento, de linha de produção, de gestão financeira, comercial etc., também precisaram de um software para poder ser mais eficientes. Junta toda essa euforia à necessidade de ter acesso às informações remotamente, de onde quer que estivéssemos. Está feito um cenário que terá sua vida dependendo do software.

Técnicas de gerenciamento de desenvolvimento de software foram primordiais para que aplicativos de gestão, de celulares, para web, pudessem se tornar produtos altamente qualificados e capazes de gerar negócios. Muitos modelos de gerenciamento não tiveram vida longa. Outros, porém, estão se reinventando e aproveitando ao máximo muitos conceitos que, ao longo de sua história, marcaram fases de gerações de desenvolvedores. Neste módulo serão explicadas as principais tecnologias de gerenciamento de desenvolvimento de software.

Objetivos

- Compreender melhor as principais técnicas de gerenciamento de projetos de software de mercado.
- Estudar as principais técnicas de gestão de desenvolvimento de um software.
- Compreender os padrões de projetos e técnicas.

O SOFTWARE E MODELOS DE PROCESSOS

UNIDADE I

CAPÍTULO 1

Características do Software

No início do que chamamos a Era do Processamento de dados, o ser humano dispunha de equipamentos munidos de lógicas computacionais baseadas em componentes eletrônicos, e muitas literaturas classificam essas lógicas eletrônicas como software. Porém, se formos classificar todas as sequências lógicas inventadas antes do ENIAC, que foi o primeiro hardware criado para receber um projeto de software, liderará nossa lista aquela que é considerada a primeira programadora da história, a matemática *Ada Lovelace*. Ada desenvolveu algoritmos para efetuar cálculos propostos pelo projeto de um computador mecânico, criado por Charles Babage, um engenheiro mecânico e filósofo, cuja principal função de seu invento, era calcular funções polinomiais. A máquina era composta por uma Unidade Lógica Aritmética, para realizar cálculos e operações booleanas, uma memória para armazenar números, entrada de dados e fórmulas (conhecidos hoje como o software), que seriam injetados na máquina através de cartões perfurados (códigos). A linguagem utilizada já previa operações nativas como estrutura condicional (*if*) e laços de repetição (*loops)*. Apesar de não ter concluído o projeto, Babage foi considerado o pai do computador. Estaria, nesse momento, sendo criado o casamento perfeito, que conhecemos hoje entre hardware e software.

O software moderno é considerado um produto composto por instruções escritas em uma linguagem de programação, executados por microprocessadores e armazenados em memórias. Diante dessa classificação, o primeiro software das gerações eletrônicas foi o do matemático Húngaro *John von Neumann (*lê-se Nóiman), cem anos depois dos projetos de Ada e Babage.

Segundo Pressman (2011, p. 31) o software distribui o produto mais importante de nosso tempo, a informação, além de oferecer poder computacional ao hardware e servir de veículo de distribuição, atuando como base para o controle do computador, como os

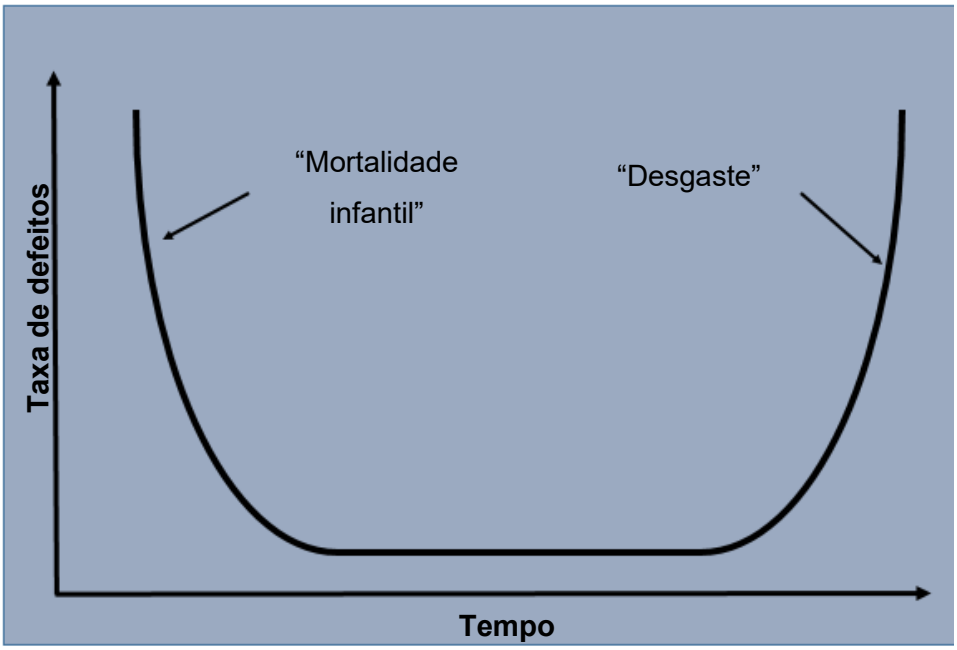
sistemas operacionais, a comunicação através das redes e a criação e controle de outros softwares.

Deterioração do software

A parceria entre hardware e software proporciona ciclos evolutivos com enormes ganhos em processamento, ora pela melhoria na arquitetura de hardware, ora nas funcionalidades de software. Lembrando sempre que, muitas funcionalidades que, antes eram executadas no hardware, foram designadas ao software. Isso se deve aos fatores custo, gerado por tempo para lançamentos, defeitos, falhas de projetos, etc. Esses fatores podem ocorrer tanto no hardware quanto no software, entretanto, o software leva muita vantagem em relação ao custo. Segundo Pressman (2011, p. 32), o alto custo se deve ao fato de o processo de fabricação do hardware gerar problemas de qualidade inexistentes para software. Ambas atividades dependem de pessoas, porém, as pessoas envolvidas e trabalhos realizados são diferentes; ambas constroem produtos, mas as abordagens são diferentes; por não poder ser gerido como processo de fabricação, o custo de desenvolvimento de software está no processo de engenharia.

Veja na figura a seguir o que chamamos de gráfico "curva da banheira", a fase da vida de um hardware. Os defeitos são muito frequentes no início de sua vida, depois, existe uma correção e o mesmo se mantém estável e, ao final, com o desgaste dos componentes.

Figura 1: Ciclo de vida de hardware

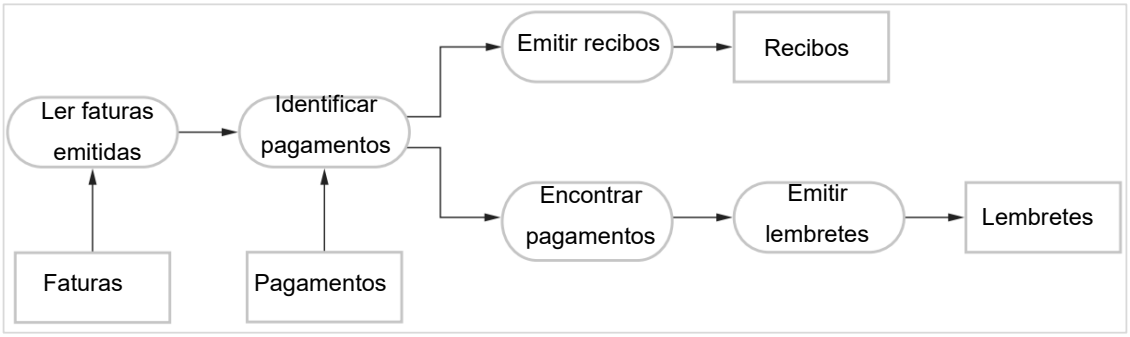


Fonte: Pressman (2011, p. 32)

Segundo Pressman (2011, p. 33), software não se desgasta conforme o hardware, mas sim, deteriora. No início da vida de um software, defeitos não descobertos elevam as taxas. Entretanto, conforme os erros são corrigidos, as taxas se diminuem.

Esses altos e baixos vão se tornando cada vez mais frequentes devido às tarefas de manutenção do software e, segundo Pressman (2011, p. 32), essa complexidade é muito maior que a manutenção de hardware. A figura a seguir demonstra curva idealizada pelo projeto do software e a curva real, com a manutenção e defeitos encontrados.

Figura 2: Curva de efeito para software



Fonte: Pressman (2011)

Campos de aplicação

Segundo Pressman (2011, p. 34), são inúmeras aplicações e campos de aplicações para a utilização do software. Ele divide em campos:

Software de Aplicação

Software sob medida que soluciona uma necessidade específica de um negócio. Normalmente, são softwares *ERPs* que cuidam de transações comerciais, tomadas de decisão, processos de produção etc.

Software de Sistema

Conjunto de ferramentas feitos para atender outros programas, *drivers* de hardware, compiladores, operações concorrentes, estruturas de dados complexas e múltiplas interfaces de hardware externas.

Software de engenharia/científico

São softwares para processamento numérico pesado. Suas aplicações, segundo Pressman (2011, p. 34), vão desde astronomia, à análise de tensões na indústria, à dinâmica orbital de ônibus espacial.

Software embutido

São softwares residentes em dispositivos que possuem funções limitadas e específicas, como sistema de controle de combustível em um carro, forno micro-ondas, TVs etc.

Software para linha de produtos

Projetado para atender utilidades específicas para muitos clientes diferentes. É direcionado para atender um mercado de consumo em massa ou limitar-se em um mercado específico. Exemplo: planilhas eletrônicas, gerenciador de banco de dados etc.

Aplicações para a Web

São sofisticados ambientes computacionais, interconectadas via web, capaz de se conectarem a banco de dados e aplicações comerciais. São conhecidas como as *webapps*.

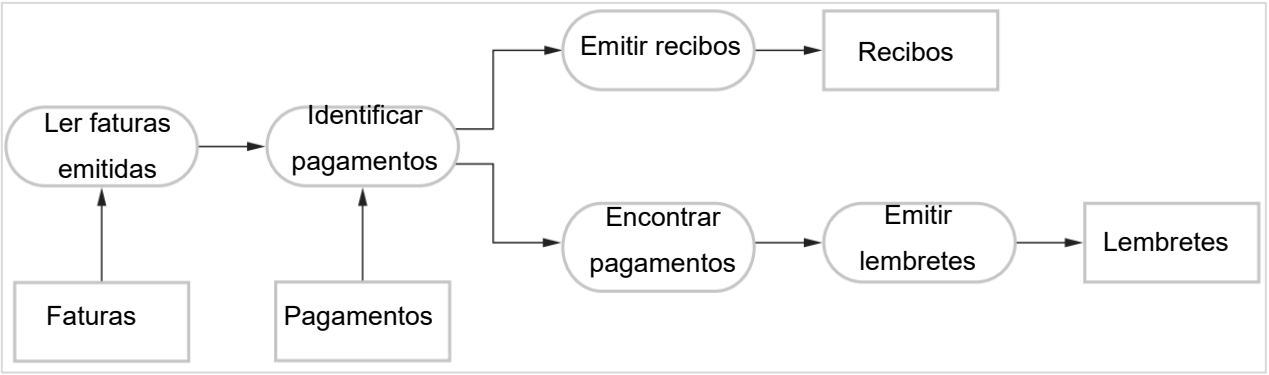
Software de inteligência artificial

São softwares que utilizam algoritmos capazes de aprender padrões, tais como reconhecimento de som e imagens, redes neurais, robótica etc.

Tipos de Sistemas de Aplicações

Os tipos de sistemas são softwares. Apesar de estarem em classificações diferentes, possuem características comuns que apresentam arquiteturas abstratas. Esse caso acontece com os Softwares de Aplicação e está caracterizado ao reúso de arquiteturas do mesmo tipo, como um ERP.

Figura 3: Arquitetura de aplicação genérica



Fonte: Sommerville (2011)

Sistema Legado

Interessante: quando se estudam algumas disciplinas que contenham o assunto Sistema, o software legado sempre aparece. Percebe-se, porém, que seu estudo é muito importante, pois as migrações de dados, de funcionalidades, de processos, serão o divisor de águas para uma troca bem-sucedida, ou não.

Software legado se refere ao software que foi desenvolvido há muito tempo, porém, ainda está em utilização nas organizações as quais, apesar de antigo, dependem muito de suas funcionalidades. Eles possuem pouca ou nenhuma documentação, suas soluções, por mais vitais que sejam, fora da organização, já estão defasadas. Quando o desenvolvimento é feito em departamento próprio da empresa, seu código fonte está em linguagens de programação e arquiteturas antigas. Já foi alterado por inúmeros programadores que não têm mais o propósito de adicionar novas funcionalidades, mas sim, de mantê-lo apenas em funcionamento.

Pressman (2011, p. 36) dá um triste conselho em relação à manutenção e alteração desse tipo sistema: “não faça nada”, pelo menos até que ele tenha que passar por alguma modificação significativa. Se atende às necessidades de seus usuários e roda de forma confiável, não está quebrado e não precisa ser “consertado”.

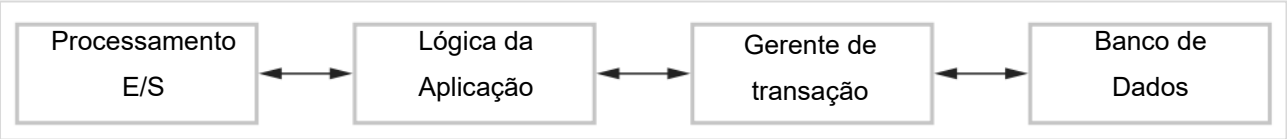
Sistema de Processamento de Transação

Sistemas de processamento de transação controlam a forma com que os dados serão atualizados em um banco de dados, através de solicitações vindas de usuários de diversos sistemas. Segundo Sommerville (2011, p. 115), uma transação é uma sequência de operações tratadas como uma unidade atômica, ou seja, indivisível.

Todas as operações em uma transação devem ser concluídas antes da mudança no banco de dados se tornarem permanentes. Um exemplo, seria um fluxo de faturamento em uma empresa. Imagine que, para dar baixa no estoque, todas as outras operações como, consulta de financeira, transporte etc., deveriam estar em conformidade com a lógica de negócio. Se por acaso alguma operação não estivesse conforme a regra a baixa no estoque não seria gravada no banco de dados. Sommerville (2011, p. 115) nos oferece o exemplo mais tradicional das literaturas:

> Um exemplo de uma transação é um pedido de um cliente para retirar dinheiro de uma conta bancária usando um caixa eletrônico (*ATM — Automated Teller Machine*). Trata-se de obter detalhes da conta do cliente, verificando o saldo e modificando-o com a retirada de uma quantia, e enviar comandos ao ATM para a entrega do dinheiro. Até que todas essas etapas sejam concluídas, a transação é incompleta e o banco de dados de contas do cliente não é alterado.

Figura 4: A estrutura de aplicações de processamento de transações



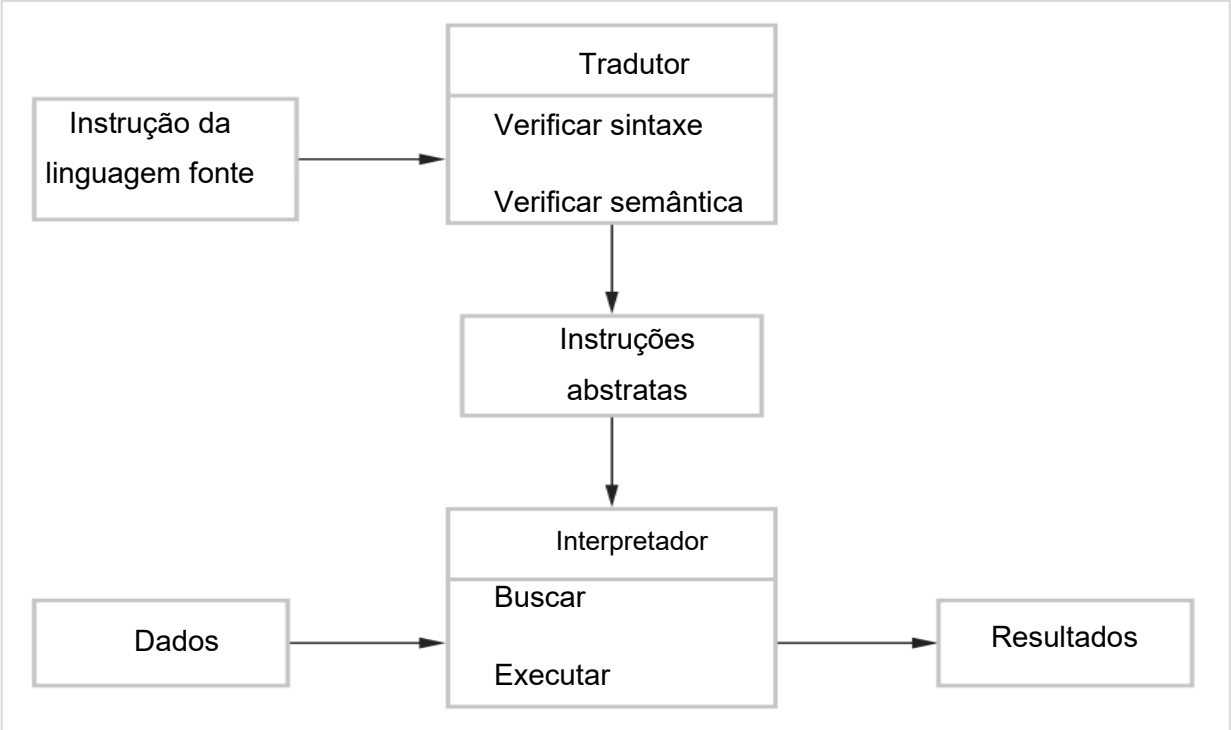
Fonte: Sommerville (2011)

Sistema de Processamento de Linguagens

Os sistemas processadores de linguagens traduzem um determinado tipo de linguagem em outra. Quando se trata de ambiente de programação, eles utilizam linguagens de programação como o Java, C++ ou C#, em linguagens de máquinas.

Outros sistemas de processamento de linguagens podem traduzir uma descrição XML de dados em comandos, para consultar um banco de dados ou uma representação XML alternativa (SOMMERVILLE, 2011, p. 119).

Figura 5. A arquitetura de um sistema de processamento de linguagem



Fonte: Sommerville (2011)

Sistema de Informação

Segundo Sommerville (2011, p. 117), todos os sistemas que envolvem a interação com bancos de dados compartilhados podem ser considerados sistemas de informação baseados em transações. Um sistema de informação permite acesso controlado a uma grande base de informações, como um catálogo de biblioteca, um horário de voo ou os registros de pacientes em um hospital.

Normalmente, o sistema de informação é modelado usando o conceito de camadas, onde as camadas superiores estão mais próximas ao usuário final e as inferiores, ao banco de dados e negócios. A utilização de vários servidores separando as camadas permite um alto desempenho e atende à demanda conforme o sistema cresça e precise de mais recursos. A maioria é composta por um servidor web, que atende às solicitações específicas dos protocolos da web. Um servidor de aplicações, que processa a lógica de negócios – muitos utilizam recursos do próprio servidor web. E um servidor de banco de dados, que armazena e recupera as solicitações da camada de negócios.

CAPÍTULO 2

Projeto de Arquitetura e Componentes do Software

Segundo Pressman (2011, p. 229), o projeto da arquitetura representa a estrutura de dados e os componentes de programas necessários para construir um sistema computacional, considerando o estilo de arquitetura, a estrutura e as propriedades dos componentes e suas inter-relações.

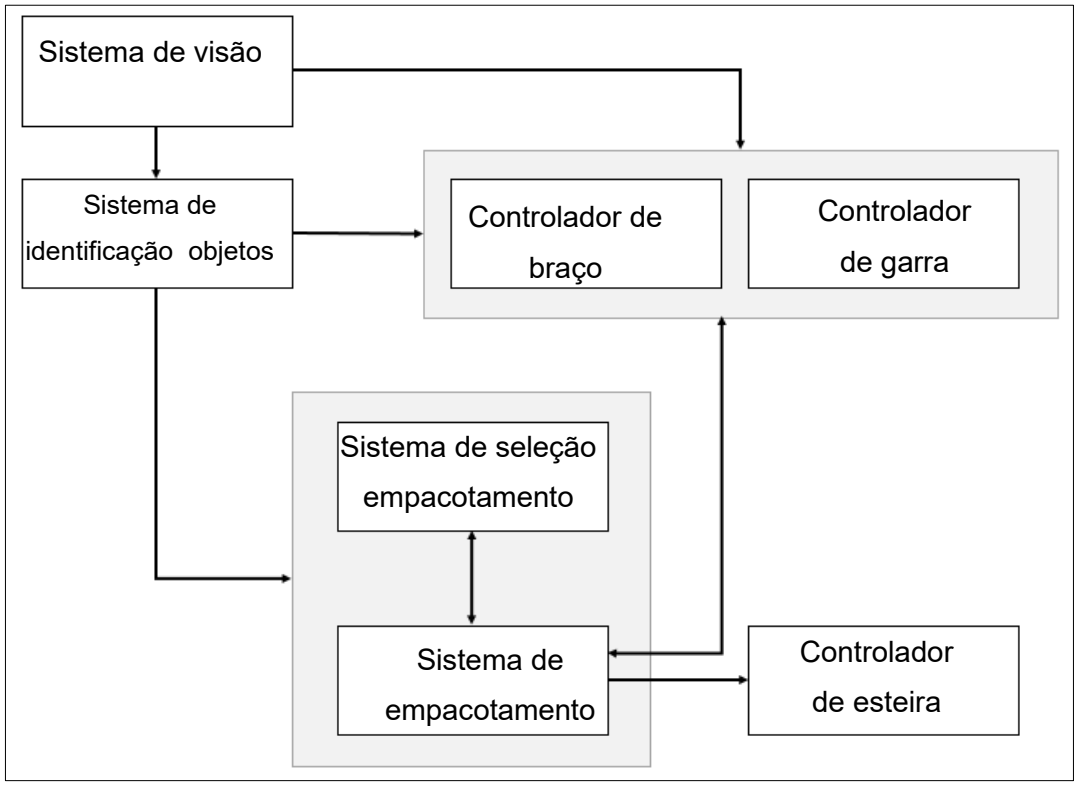
Sommerville (2011, p. 103), diz que o projeto de arquitetura está focado com o entendimento de como um sistema deve ser organizado e com a estrutura geral desse sistema e que, no modelo do processo de desenvolvimento de software, o projeto de arquitetura é o primeiro estágio no processo de projeto de software e, seu resultado é um modelo de arquitetura que descreve como o sistema está organizado em um conjunto de componentes de comunicação. Outra atuação importante do projeto da arquitetura de software está ligada à robustez, ao desempenho, capacidade de distribuição e manutenibilidade.

ISO / IEC / IEEE 42010: 2011 aborda a criação, análise e sustentação de arquiteturas de sistemas através do uso de descrições de arquitetura. Está estabelecida sob a Architectural Description Language, Linguagem de descrição de Arquitetura.

https://www.iso.org/standard/50508.html (acessado em 06/06/2019).

Existe uma sobreposição entre processos de engenharia de requisitos e projeto de arquitetura. Normalmente, especificações do sistema não devem incluir todas as informações do projeto. Nesse caso, poderá ser proposta uma arquitetura abstrata, que poderá servir, num primeiro instante como uma forma de negociação de requisitos do sistema e, em seguida, como forma de estruturar as discussões com os envolvidos: clientes, desenvolvedores etc. Essa arquitetura poderá ser moldada em diagramas de blocos simples.

Figura 6: A arquitetura de um sistema de controle robotizado de empacotamento



Fonte: Sommerville (2011)

No diagrama anterior, a caixa representa graficamente um componente. Quando aparecerem caixas dentro de caixas, ou seja, um componente dentro de outro, significa que ele foi decomposto em subcomponentes. As setas indicam as direções dos dados e quais componentes se comunicam.

Devido ao estreito relacionamento entre requisitos não funcionais e a arquitetura do software e independente do estilo e a arquitetura escolhida, o projeto deverá atender aos seguintes requisitos não funcionais:

» **Desempenho**: se no projeto o desempenho for um fator crítico, projeta-se a arquitetura em função de localizar operações críticas em uma granularidade em pequenos números de componentes.

» **Proteção**: se a proteção for um fator crítico ou mais importante, deverá utilizar a arquitetura de camadas.

» **Segurança**: se a segurança for um fator crítico, a arquitetura deve se concentrar em um único componente, reduzindo custo e validação de segurança.

» **Disponibilidade**: se a disponibilidade for um fator crítico, a arquitetura deve incluir componentes redundantes, podendo ser substituídos ou atualizados sem comprometer a disponibilidade.

Existe uma aparente contradição entra a prática e a teoria, relacionada ao processo de arquitetura. Sommerville (2011, p. 105), cita essas duas formas: para facilitar a discussão sobre o projeto, ou seja, uma visão de alto nível. E a segunda é usada como forma de documentar um projeto de arquitetura, de maneira bem detalhada, de forma que mostre seus diferentes componentes e suas conexões. E conclui que, em muitos projetos é única documentação existente, pelo fato de que muitas pessoas pensam que a documentação detalhada não é útil e não vale o custo.

Padrões de arquitetura

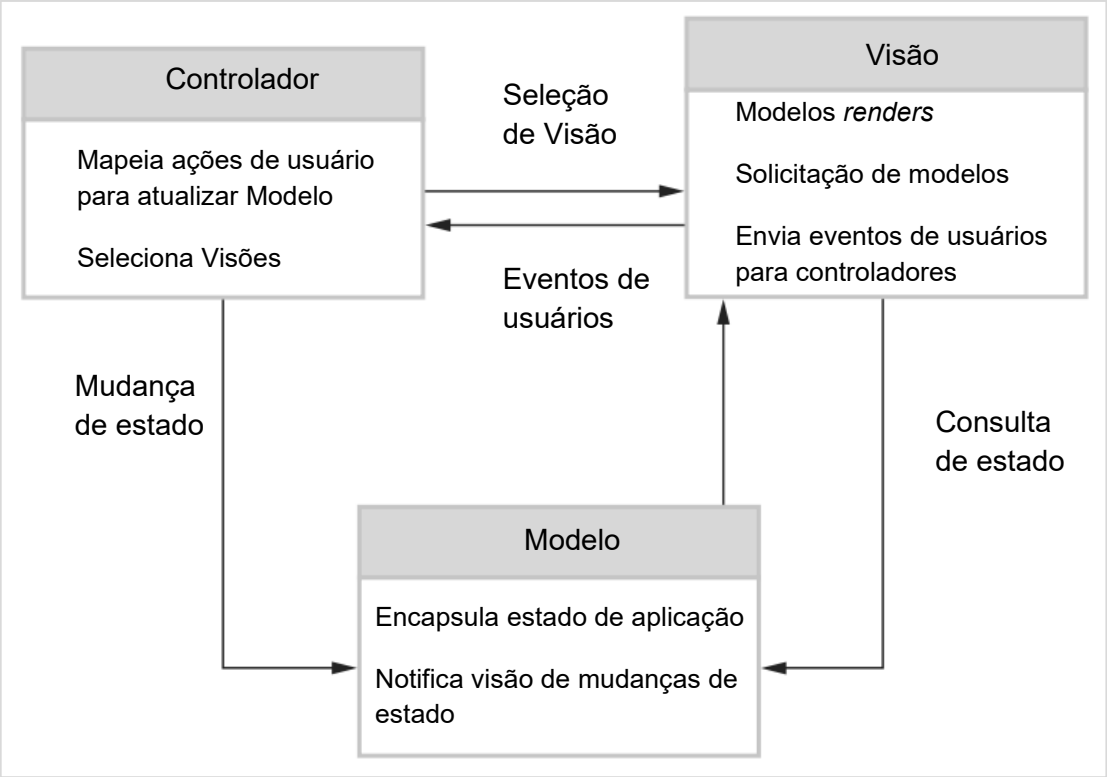
Os padrões de arquitetura tratam um problema específico de aplicações em um contexto específico e sob um conjunto de limitações e restrições. O padrão propõe uma solução de arquitetura capaz de servir como base para o projeto de arquitetura (PRESSMAN, 2011, p. 239). Sommerville (2011, p. 108) complementa: um padrão de arquitetura deve descrever uma organização de sistema bem-sucedida em sistemas anteriores. Deve incluir informações de quando o uso desse padrão é adequado, e seus pontos fortes e fracos.

Arquitetura em camadas

A arquitetura em camadas provê uma interdependência entre elementos de um sistema, permitindo, assim, que suas funcionalidades tenham baixo impacto quando houver alguma modificação.

O padrão MVC é um bom exemplo desse conceito, onde recursos e serviços são oferecidos separadamente.

Figura 7: Organização MVC



Fonte: Sommerville (2011, p. 109)

Veja na tabela a seguir, como descrever o famoso padrão MVC. O padrão MVC é muito utilizado em aplicações web e seu nome é uma sigla que significa *Model*, *View*, *Controller*.

Tabela 1: O padrão modelo-visão-controlador (MVC)

Nome	**MVC – *Model-View-Controller***
Descrição	Separa a apresentação e a interação dos dados do sistema. O sistema é estruturado em três componentes lógicos que interagem entre si. O componente **Modelo** gerencia o sistema de dados e as operações associadas a esses dados. O componente **Visão** define e gerencia como os dados são apresentados ao usuário. O componente **Controlador** gerencia a interação do usuário e passa essas interações para a **Visão** e o **Modelo.**
Exemplo	A Figura 7 mostra a arquitetura de um sistema aplicativo baseado na Internet, organizado pelo uso do padrão MVC.
Quando é usado	É usado quando existem várias maneiras de se visualizar e interagir com dados.
Vantagens	Permite que os dados sejam alterados de forma independente de sua representação, e vice-versa. Apoia a apresentação dos mesmos dados de maneiras diferentes, com as alterações feitas em uma representação aparecendo em todas elas.
Desvantagens	Quando o modelo de dados e as interações são simples, pode envolver código adicional e complexidade de código.

Fonte: Sommerville (2011, p. 109)

A abordagem em camadas auxilia o desenvolvimento incremental. Um exemplo é que, quando se desenvolve ou altera uma camada, seus recursos podem ser disponibilizados aos usuários ou sistemas que estão requisitando. Enquanto sua interface permanecer inalterada, camadas podem ser substituídas por outras

equivalentes. Quando uma camada de interface muda, somente as camadas adjacentes são alteradas. Isso vale para a inserção de novos recursos. As camadas também podem facilitar em implementações multiplataformas, pois somente camadas próximas aos sistemas operacionais e banco de dados poderão ser manipuladas sem afetar as demais.

Tabela 2: Arquitetura em camadas

Nome	Arquitetura em camadas
Descrição	Organiza o sistema em camadas e suas funcionalidades. Uma camada fornece serviços à camada acima dela; assim, os níveis mais baixos de camadas representam os principais serviços suscetíveis de serem usados em todo o sistema.
Exemplo	Um modelo em camadas de um sistema para compartilhar documentos com direitos autorais, em bibliotecas diferentes.
Quando é usado	É usado na construção de novos recursos sob os já existentes; quando o desenvolvimento está distribuído por várias equipes, com a responsabilidade de cada equipe em uma camada de funcionalidade; quando há um requisito de proteção multinível.
Vantagens	Desde que a interface seja mantida, permite a substituição de camadas inteiras. Recursos redundantes (por exemplo, autenticação) podem ser fornecidos em cada camada para aumentar a confiança do sistema.
Desvantagens	Na prática, costuma ser difícil proporcionar uma clara separação entre as camadas, e uma camada de alto nível pode ter de interagir diretamente com camadas de baixo nível, em vez de através da camada imediatamente abaixo dela. O desempenho pode ser um problema por causa dos múltiplos níveis de interpretação de uma solicitação de serviço, uma vez que são processados em cada camada.

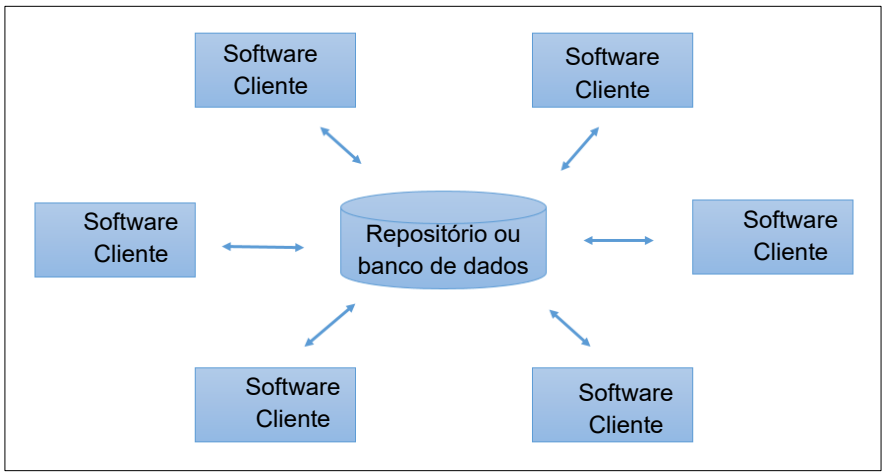
Fonte: Sommerville (2011, p. 110)

Arquitetura de repositório

Segundo Sommerville (2011, p. 111), a maioria dos sistemas que utilizam um banco de dados ou repositório compartilhado se enquadra nesse modelo, que é adequado para aplicações nas quais os dados são gerados por um componente e acessado por outro.

Pressman (2011, p. 235), acrescenta: esse repositório de dados, que também pode ser um arquivo, reside no centro dessa arquitetura, onde os componentes manipulam os dados contidos no repositório e um software cliente acessa um repositório central que, muitas vezes, é um repositório passivo, do qual os clientes acessam os dados independentemente das ações de outros clientes.

Figura 8: Arquitetura centralizada em dados



Fonte: Adaptada pelo autor, de Pressman (2011, p. 236)

Arquitetura cliente-servidor

Uma arquitetura cliente-servidor obedece a uma arquitetura onde exista um conjunto de serviços, servidores e clientes que acessam esse serviço. Segundo Sommerville (2011, p. 113), os principais componentes dessa arquitetura são: conjunto de servidores que oferecem serviços a outros componentes, um conjunto de clientes que podem chamar os serviços oferecidos e uma rede que permite aos clientes acessar o serviço.

Normalmente, arquiteturas cliente-servidor são consideradas sistemas distribuídos. Quando se diz "distribuídos" logo imaginamos equipamentos, serviços ou servidores separados fisicamente. Entretanto, a implementação desse conceito nos permite rodar serviços separados em um mesmo computador. Ainda assim, continua sendo considerado sistema distribuído e conta com os benefícios da separação e interdependência, onde a modificação, a substituição ou mesmo uma possível falha não terão tanto impacto.

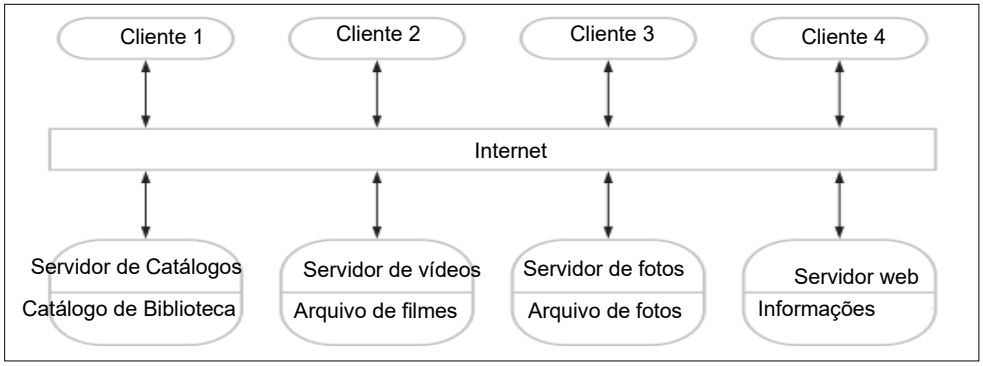
Segundo Sommerville (2011, p. 113), os clientes podem ter de saber os nomes dos servidores disponíveis e os serviços que eles fornecem. No entanto, os servidores não precisam conhecer a identidade dos clientes ou quantos clientes estão acessando seus serviços. Os clientes acessam os serviços fornecidos por um servidor por meio de chamadas de procedimento remoto usando um protocolo de solicitação-resposta, tal como o protocolo HTTP usado na Internet. Essencialmente, um cliente faz uma solicitação a um servidor e espera até receber uma resposta.

Tabela 3: Arquitetura cliente-servidor

Nome	Cliente-servidor
Descrição	Em uma arquitetura cliente-servidor, a funcionalidade do sistema está organizada em serviços — cada serviço é prestado por um servidor.
Exemplo	A Figura 10 é um exemplo de uma biblioteca de filmes e vídeos/DVDs, organizados como um sistema cliente-servidor.
Quando é usado	É usado quando os dados em um banco de dados compartilhado precisam ser acessados a partir de uma série de locais. Como os servidores podem ser replicados, também pode ser usado quando a carga em um sistema é variável.
Vantagens	A principal vantagem desse modelo é que os servidores podem ser distribuídos através de uma rede.
Desvantagens	Cada serviço é um ponto único de falha suscetível a ataques de negação de serviço ou de falha do servidor. Pode haver problemas de gerenciamento se os servidores forem propriedade de diferentes organizações

Fonte: Sommerville (2011, p. 114)

Figura 9: Uma arquitetura cliente-servidor para uma biblioteca de filmes



Fonte: Sommerville (2011, p. 114)

Desenvolvimento Baseado em Componentes

A modelagem orientada a objetos resulta em vários diagramas de classe, uma infinidade de classes, objetos e relacionamentos. Descobrir peças reutilizáveis, nesse universo, é muito difícil entre unidades menores. A ideia por trás do Desenvolvimento Baseado em Componentes, DBC, em português, ou, CBD – *Component Based Development*, é integrar as partes relacionadas e reutilizá-las coletivamente. Essas partes integradas são conhecidas como componentes.

Segundo Sommerville (2011, p. 315):

> Sua criação foi motivada pela frustração de projetistas, pois o desenvolvimento orientado a objetos não levou a um amplo reuso, como se havia sugerido. As classes de objetos foram muito detalhadas e específicas e muitas vezes precisavam ser associadas com uma aplicação em tempo de compilação. Era preciso ter conhecimento detalhado das classes para usá-las e isso geralmente significava que era necessário ter o

> código-fonte do componente, o que significava que vender ou distribuir objetos como componentes reusáveis individuais era praticamente impossível.

Um componente é um objeto de software, destinado a interagir com outros componentes, encapsulando certas funcionalidades ou um conjunto de funcionalidades. É um conjunto modular, portátil e reutilizável de funcionalidades bem definidas que encapsula sua implementação e a exporta como uma interface de nível superior. Ele tem uma interface obviamente definida e está em conformidade com um comportamento recomendado comum a todos os componentes dentro de uma arquitetura. Um componente de software pode ser implantado de forma independente e está sujeito à composição de terceiros. Sommerville (2011, p. 315) completa: Os componentes são abstrações de nível mais alto do que objetos e são definidos por suas interfaces. Geralmente, eles são maiores que objetos individuais e todos os detalhes de implementação são escondidos de outros componentes.

As técnicas de desenvolvimento baseadas em componentes consistem em rotinas de desenvolvimento não convencionais, incluindo avaliação de componentes, recuperação de componentes etc. Um componente encapsula a funcionalidade e os comportamentos de um elemento de software em uma unidade binária reutilizável e autoimplantável.

Além disso, os componentes devem ser padronizados em características sugeridas por Sommerville (2011, p. 317):

- **Padronizado:** a padronização de componentes determina que um componente usado em um processo CBS precisa obedecer a um modelo de componentes padrão, definir as interfaces de componentes, metadados de componente, documentação, composição e implantação.
- **Independente:** com a independência de um componente, deve ser possível compor e implantá-lo sem precisar usar outros componentes específicos.
- **Passível de composição:** Para um componente ser composto, todas as interações externas devem ter acesso por meio de interfaces publicamente definidas.
- **Implantável:** Deve ser capaz de funcionar como uma entidade autônoma em uma biblioteca de componentes que forneça uma implementação do modelo de componentes, o que geralmente significa que o componente não tem como ser compilado antes de ser implantado.

» **Documentado**: A documentação deverá ser clara e bem específica para que os potenciais usuários possam decidir se satisfazem a suas necessidades. A sintaxe e, idealmente, a semântica de todas as interfaces de componentes devem ser especificadas.

Projeto de Componentes

Segundo Pressman (2011, p. 262), alguns princípios básicos são aplicáveis no projeto de componentes e têm sido largamente adotados. Inicia-se a modelagem orientada a objetos, o SOLID:

» **S** - *Single-responsibility principle* - Princípio da responsabilidade única

» **O** - *Open-closed principle* - Princípio aberto-fechado

» **L** - *Liskov substitution principle* - Princípio da substituição de **L** - *Liskov*

» **I** - *Interface segregation principle* - Princípio da segregação de interfaces

» **D** - *Dependency Inversion Principle* – Princípio da inversão de dependência

Princípio *Closed-open*

Baseia-se no princípio de que um componente deve ser aberto para extensão e fechado para modificação. Esse modelo garante que o comportamento da classe ou do componente possa ser estendido. Conforme vão acontecendo as mudanças de requisitos, deve ser fácil e possível que essa classe se comporte para atendê-lo. Entretanto, não é possível que o código-fonte da classe base esteja suscetível a alterações sem permissão.

Devemos especificar o componente para permitir que ele seja estendido sem a necessidade de fazer modificações internas no próprio componente (PRESSMAN, 2011, p. 263).

Princípio da substituição de L - Liskov

Esse princípio define que suas subclasses devem ser substituíveis por suas superclasses. Pressman (2011, p. 263) sugere que um componente que usa uma superclasse deveria continuar funcionando apropriadamente, caso uma subclasse fosse passada para o componente em seu lugar, honrando, inclusive, todos os contratos implícitos entre as classes.

Princípio da inversão de dependência

Esse princípio está baseado não nas dependências de componentes concretos, mas sim, em suas interfaces, ou seja, quanto mais um componente depender de outros componentes, mais difícil será efetuar sua extensão. A inversão de dependência capacita um código a ter sua flexibilização e sua reutilização mais efetiva. As abstrações de níveis superiores não afetarão as de níveis inferiores quando forem alteradas.

Princípio da responsabilidade única

Esse princípio está baseado na responsabilidade de uma classe. A classe só deve ter apenas um motivo para mudar. Quando acontece alguma mudança, seja por requisitos, refatoração ou reconstrução, significa que as classes deverão ser alteradas. Nesse momento, quanto mais responsabilidade a classe tiver, mais difícil será sua modificação. O acoplamento das classes ligará suas responsabilidades e a responsabilidade pode ser definida como um único motivo para mudança. Único significa trabalhar isoladamente, ou seja, seu propósito só faz uma coisa, não faz duas.

Princípio da segregação de interfaces

Esse princípio sugere que é melhor utilizar várias interfaces específicas a uma interface de propósito único. Imagine uma interface chamada Animal, que teria os métodos comer, dormir e andar. Porém, imagine que existam animais que voam. Para que esse exemplo funcione em relação ao princípio da segregação de interfaces, deve-se criar interfaces para cada função dos Animais: Comer, Dormir, Voar. Isso passaria a ser interessante, pois poderíamos combinar ações dos animais. Lembrando que, ao quebrar em interfaces menores, é preferencial a Composição ao invés de Herança, Desacoplamento por Acoplamento, separando os papéis e não acoplando classes derivadas com responsabilidades desnecessárias.

The Golden Age of Software Architecture

Esse trabalho avalia o desenvolvimento da arquitetura de software desde seu início, na década de 1980, até seu uso atual. Contém pouco conteúdo técnico, mas apresenta um interessante panorama histórico. (SHAW, M.; CLEMENTS, P. The Golden Age of Software Architecture. IEEE Software, v. 21, n. 2, mar.-abr. 2006.) Disponível em: http://dx.doi.org/10.1109/MS.2006.58. (Acessado em 06/07/2019).

Handbook of Software Architecture

Esse é um trabalho em andamento realizado por Grady Booch, um dos primeiros evangelizadores da arquitetura de software. Ele vem documentando as arquiteturas de vários sistemas de software para que você possa ver a realidade em vez de abstrações acadêmicas. Disponível em: http://www.handbookofsoftwarearchitecture.com/ (Acessado em 6/7/2019).

Reuse-based Software Engineering. Uma discussão abrangente sobre as diferentes abordagens para o reúso de software. Os autores abrangem questões técnicas de reúso e gerenciamento de processos de reúso. (MILI, H.; MILI, R.; YACOUB, S.; ADDY, E. Reuse-based Software Engineering. John Wiley & Sons, 2002.)

Overlooked Aspects of COTS-Based Development. Um artigo interessante que discute uma pesquisa de desenvolvedores usando uma abordagem baseada em COTS e os problemas que eles encontraram. (TORCHIANO, M.; MORISIO, M. IEEE Software, v. 21, n. 2, mar.-abr. 2004.) Disponível em: http://dx.doi.org/10.1109/MS.2004.1270770.

Construction by Configuration: A New Challenge for Software Engineering. Esse é um artigo no qual abordo os problemas e as dificuldades de construção de uma nova aplicação configurando sistemas existentes. (SOMMERVILLE, I. Proc. XIX Conferência de engenharia de software australiano, 2008). Disponível em: http://dx.doi.org/10.1109/ASWEC.2008.75.

Architectural Mismatch: Why Reuse Is Still So Hard. Esse artigo faz uma releitura de um anterior, que discutiu os problemas de reúso e integração de vários sistemas COTS. Os autores concluíram que, embora haja alguns progressos, ainda existem problemas em suposições feitas pelos projetistas de sistemas individuais. (GARLAN, M. *et al*. IEEE Software, v. 26, n. 4, jul.- ago. 2009.) Disponível em: http://dx.doi.org/10.1109/MS.2009.86.

CAPÍTULO 3
Modelos de Processos e Ciclo de Vida

Segundo Sommerville (2011, p. 18), um processo de software é um conjunto de atividades, que devem estar relacionadas, que caminham para produzir um software, podendo envolver o desenvolvimento de software a partir do zero em uma linguagem padrão de programação. Porém, atualmente, novos softwares de negócios são desenvolvidos por meio de extensões, modificações, configurações ou componentes. Já Pressman (2011, p.53) estabelece que cada atuação metodológica é composta de uma ação de engenharia de software, que por sua vez, cada ação é determinada por um grupo de tarefas, que representa as tarefas de trabalho a ser completadas, e os elementos de software serão gerados, os fatores de garantia da qualidade serão exigidos e os referência utilizados para estabelecer o progresso. Ainda descreve que a engenharia de software estabelece cinco atividades metodológicas: comunicação, planejamento, modelagem, construção e entrega.

Os processos de software são decompostos em processos menores. Esses processos, por sua vez, são compostos de atividades, onde existe a possibilidade de serem decompostas. Para cada atividade de um processo é importante saber quais as suas subatividades, as pré-atividades, os artefatos de entrada e de saída da atividade, os recursos necessários e os procedimentos a serem utilizados na sua realização.

Muitos são os processos de software que possam ser utilizados para cada caso. Empresas vêm adequando, criando, modificando as diversas boas práticas propostas. Esses processos são muito complexos e dependem de pessoas para tomarem as decisões pertinentes à sua construção. Porém, existem quatro atividades fundamentais, propostas por Sommerville (2011, p. 18):

1. **Especificação de software**. A funcionalidade e restrições devem ser definidas.

2. **Projeto e implementação de software**. O software deve ser criado para atender às especificações.

3. **Validação de software**. O software deve ser validado para garantir que atenda às demandas do cliente.

4. **Evolução de software**. O software deve evoluir para atender às necessidades de mudança dos clientes.

Muitas empresas ainda possuem métodos ultrapassados de processo de desenvolvimento de software que ainda suprem suas necessidades. Porém, à medida que a demanda cresce, cresce também a necessidade de utilizar as boas práticas que garantam um crescimento gradativo e eficaz. Segundo Sommerville (2011, p. 19):

> Os processos de software, às vezes, são categorizados como dirigidos a planos ou processos ágeis. Processos dirigidos a planos são aqueles em que todas as atividades são planejadas com antecedência, e o progresso é avaliado por comparação com o planejamento inicial. Em processos ágeis, o planejamento é gradativo, e é mais fácil alterar o processo de maneira a refletir as necessidades de mudança dos clientes.

Sommerville (2011, p. 18) conclui: é preciso encontrar um equilíbrio entre os processos dirigidos a planos e os processos ágeis, pois tudo dependerá da abordagem de cada software.

Em um processo de desenvolvimento de software, o ponto de início para a arquitetura de um processo é a escolha de um modelo de ciclo de vida. Quando não se escolhe um modelo, o desenvolvedor tende a partir diretamente para a codificação. Esse modelo atrai a maioria dos desenvolvedores como a luz atrai um inseto. Infelizmente, para a maioria o resultado será o mesmo: um desastre total. Esse modelo é de alto risco e nada confiável para projetos de grande porte ou que exigirão muito do sistema.

Ciclo de Vida de Desenvolvimento de Software

O Ciclo de Vida de Desenvolvimento de Software ou *Software Development Life Cycle SDLC* é um processo sistemático para construir software que garanta a qualidade e correção do software construído. Esse processo visa produzir um produto de qualidade, que atenda às expectativas do cliente, nas funcionalidades, no prazo e no custo estimado. Consiste em um plano bem detalhado que visa explicar como planejar, construir e manter um projeto de software. Suas fases têm seu próprio processo de entrega, que alimentam as entradas das fases subsequentes. Suas principais vantagens são:

- Oferece uma boa base de planejamento, programação e estimativa de projetos.
- Padroniza a estrutura de atividades e entrega.
- Elabora mecanismos que efetuam rastreamento e controle de projetos.
- Oferece uma visão do projeto para todos os envolvidos no desenvolvimento.

» Aumenta a velocidade de desenvolvimento, melhora o relacionamento com o cliente, diminuindo, assim, os riscos inerentes a prazos, requisitos e orçamentos.

Suas fases são:

Figura 10: Fases do ciclo de vida de desenvolvimento de software

Fonte: Elaboração própria do autor (2019)

Análise de requisitos

É primeira fase do ciclo de vida do desenvolvimento de software e é conduzido pelos arquitetos e engenheiros seniores. Ela precisa de pessoas mais capacitadas, pois o detalhamento preciso será fundamental para o projeto. Também fornecerá uma visão geral, clara e detalhada do escopo de todo o projeto.

Estudo de viabilidade

Logo após a conclusão da fase de requisitos, o próximo passo é definir e documentar as viabilidades e necessidades do software. Este processo é realizado com a ajuda do documento *Software Requirement Specification,* que inclui tudo o que deve ser documentado no ciclo de vida do projeto. Podemos ainda citar algumas características que definem o estudo:

» **Econômico:** o projeto caberá no orçamento?

» **Legal:** está conforme as legislações?

» **Viabilidade de operação:** temos as funcionalidades esperadas pelos clientes?

» **Técnico:** pode ser utilizado em qualquer arquitetura computacional?

» **Cronograma:** o projeto caberá no cronograma?

Design

Nessa fase os documentos do software são comparados e gerados de acordo com os documentos gerados na fase de requisitos, que definirá a entrada na fase de modelagem e a arquitetura do sistema.

Existem dois tipos de documentos de design desenvolvidos nesta fase:

Design de alto nível (HLD)

- Breve descrição e nome de cada módulo.
- Um resumo sobre a funcionalidade de cada módulo.
- Relação de interface e dependências entre módulos.
- Tabelas de banco de dados identificadas com seus principais elementos.
- Diagramas de arquitetura completos, juntamente com detalhes de tecnologia.

Design de baixo nível (LLD)

- Lógica funcional dos módulos.
- Tabelas de banco de dados, que incluem tipo e tamanho.
- Detalhe completo da interface.
- Aborda todos os tipos de problemas de dependência.
- Listagem de mensagens de erro.
- Entradas e saídas completas para cada módulo.

Codificação

Na fase de codificação, os desenvolvedores, juntamente com os arquitetos, escolhem a linguagem de programação apropriada para o projeto e iniciam a programação, ou seja, a geração do código fonte. Essa pode ser a fase mais longa, pois existem várias subtarefas como testes de aceitação, teste de lógica, design de interface do usuário etc. Algumas ferramentas são primordiais para a codificação, como *frameworks* que facilitam e adiantam bem essa fase.

Testes

Na codificação existem alguns testes efetuados pelo desenvolvedor, enquanto programa. Porém, esse não é um teste definitivo e é muito suscetível a falhas, uma vez que o desenvolvedor já se acostuma com os possíveis eventos de erros e tende a evitá-los. Por isso, assim que o software já tenha algum módulo que seja "*entregável*", ou seja, não está completo, mas já tem funcionalidades, é recomendado criar um ambiente de testes tão próximo ao ambiente que o usuário final utilizará. Isso é para verificar se tudo funciona conforme as necessidades dos clientes. Nessa fase, defeitos são encontrados e reportados aos desenvolvedores, que podem entrar num ciclo de correção e testes até sanar o problema.

Implantação

Terminada a fase de testes, sem erros e, consequentemente, sem alterações, inicia-se a fase de implantação. Em softwares modularizados, a implantação pode ser realizada por etapas, respeitando sempre as diretrizes passadas pelo gestor do projeto.

Manutenção

Mesmo depois de implantado, o software pode apresentar alguns *bugs*, algum ajuste de requisito poderá ser atualizado, melhorado ou criado.

O SDLC tem seu foco a garantir que as necessidades das etapas continuem a serem executadas de acordo com as especificações. Por isso, ao longo do tempo foram testados vários modelos para manter a melhoria contínua no processo de desenvolvimento de software.

Modelos Prescritivos

Cascata

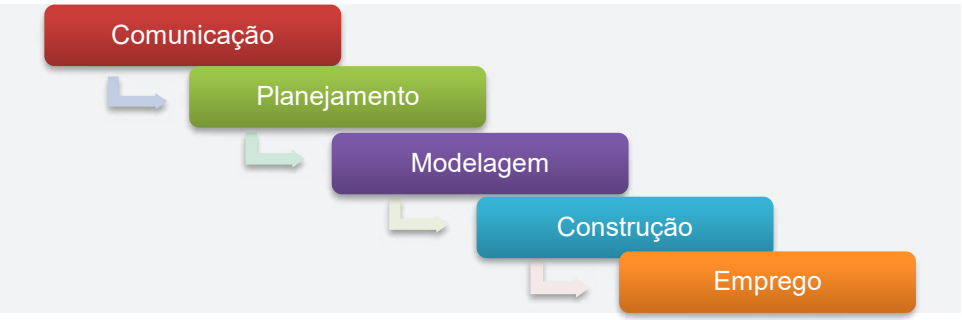
Novamente o modelo Cascata é mencionado. Segundo Filho (2010, p. 24), no modelo cascata os principais subprocessos são executados em sequência, o que permite que seus pontos de controles sejam demarcados. Estes pontos de controle facilitam muito a gestão dos projetos, o que faz com que este processo seja, em princípio, confiável e utilizável em projetos de qualquer escala.

Entretanto, sua burocracia e rigidez fazem com que as atividades de requisitos, análise e desenho não permitam erros, com baixa visibilidade para o cliente. Pelo fato de sua implementação permitir que, em fases posteriores, haja revisão e alteração das especificações, a equipe deverá ser maior, pois seu gerenciamento se torna trabalhoso.

Segundo Sommerville (2011, p 21), seu maior problema é a divisão inflexível do projeto em estágios distintos. Os compromissos devem ser assumidos em um estágio inicial do processo, o que dificulta que atendam às mudanças de requisitos dos clientes.

Processos baseados em transformações formais são geralmente usados apenas no desenvolvimento de sistemas críticos de segurança ou de proteção. Eles exigem conhecimentos especializados. Para a maioria dos sistemas, esse processo não oferece custo-benefício significativo sobre outras abordagens para o desenvolvimento de sistemas (SOMMERVILLE, p. 21).

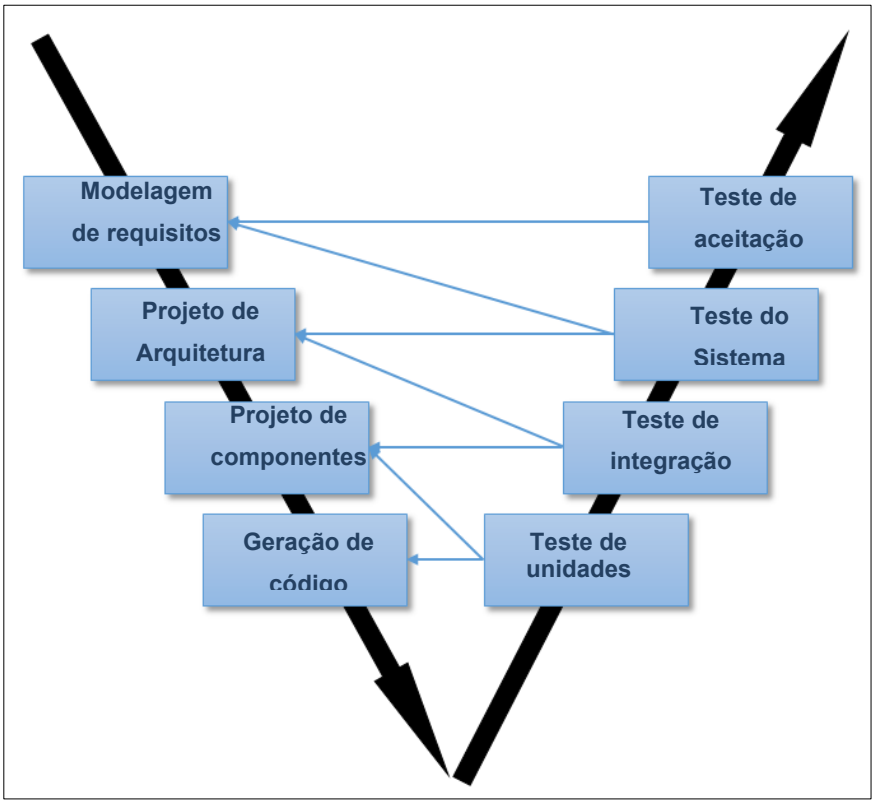
Figura 11: Modelo cascata usado em processos de software



Fonte: Pressman (2011, p. 660)

Existe uma variação do modelo cascata, que é o modelo em V. Segundo Pressman (2011, p. 60), esse modelo descreve a relação entre ações de garantia da qualidade e as ações associadas a comunicação, modelagem e atividade de construção Inicial.

Figura 12: Modelo em V



Fonte: Pressman (2011, p. 60)

Conforme a equipe vai descendo para o lado esquerdo do V, os requisitos básicos do problema são refinados em representações progressivamente cada vez mais detalhadas e técnicas do problema e de sua solução. Assim que o produto esteja com funcionalidades prontas para serem entregues, a equipe se desloca para cima, do lado direito do V. Nesse momento, vários testes são realizados, interagindo com sua etapa correspondente no lado esquerdo do V (PRESSMAN, 2011, p. 60).

Iterativo ou Incremental

O modelo incremental é um processo de desenvolvimento em que os requisitos são divididos em vários módulos independentes do ciclo de desenvolvimento de software. O desenvolvimento incremental é feito em etapas desde o projeto de análise, implementação, teste / validação, manutenção.

Segundo Sommerville (2011, p. 21) o desenvolvimento incremental é baseado na ideia de desenvolver uma implementação inicial, expô-la aos comentários dos usuários e continuar por meio da criação de várias versões até que um sistema adequado seja desenvolvido. Atividades de especificação, desenvolvimento e validação são intercaladas, e não separadas, com rápido feedback em todas as atividades.

O surgimento do desenvolvimento incremental se deu às falhas encontradas no modelo cascata. Ele oferece maior capacidade para incorporar alterações durante o desenvolvimento, é cíclico e não unidirecional.

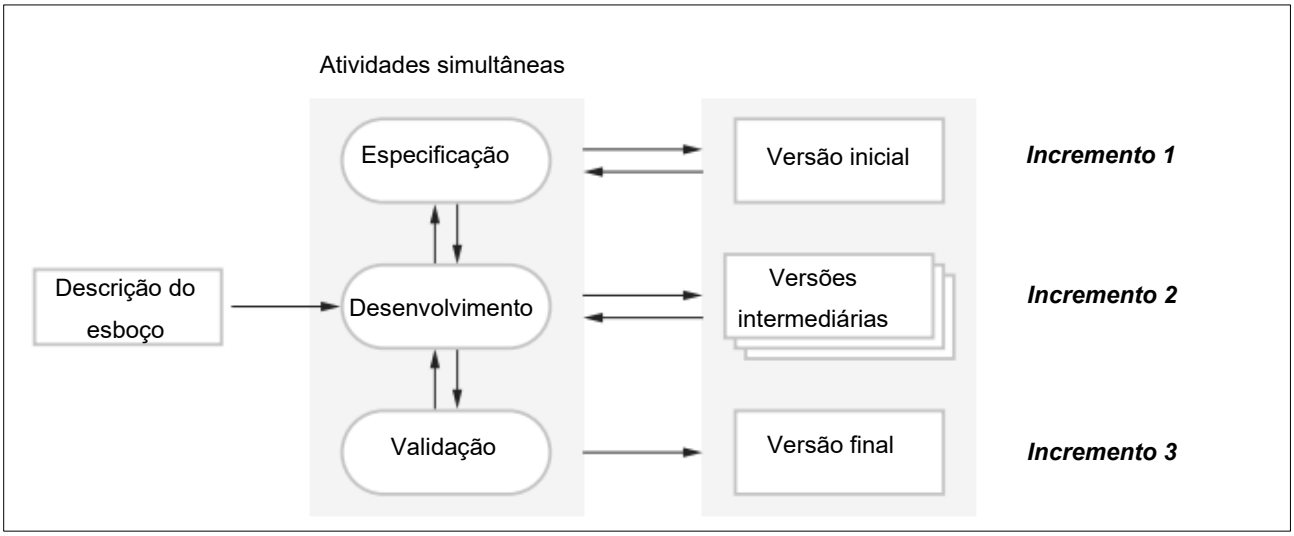
As vantagens do modelo incremental sob o modelo cascata são citadas por Sommerville (2011, p. 22):

» O custo das mudanças nos requisitos do cliente é menor. O retrabalho de refazer análise e documentação é menor do que no modelo em cascata.

» É mais fácil obter retorno dos clientes sobre o desenvolvimento que foi feito.

» É possível obter entrega e implementação rápida de um software útil ao cliente, mesmo se toda a funcionalidade não for incluída.

Figura 13. Desenvolvimento incremental



Fonte: Sommerville (2011, p. 22)

No desenvolvimento incremental, as equipes analisam partes do sistema para revisá-las e melhorá-las. O retorno do usuário é consultado para modificar os alvos de entregas sucessivas.

Sommerville (2011, p. 22) menciona algumas desvantagens do gerenciamento incremental:

1. O processo não é visível. Os gerentes precisam de entregas regulares para mensurar o progresso. Se os sistemas são desenvolvidos com rapidez, não é economicamente viável produzir documentos que reflitam cada uma das versões do sistema.

2. A estrutura do sistema tende a se degradar com a adição dos novos incrementos. A menos que tempo e dinheiro sejam dispendidos em refatoração para melhoria do software, as constantes mudanças tendem a corromper sua estrutura. Incorporar futuras mudanças do software torna-se cada vez mais difícil e oneroso.

Prototipação

A prototipagem de software refere-se à criação de protótipos de software que exibem a funcionalidade do produto em estágio de desenvolvimento, porém, sem a lógica programada em alguma linguagem.

Um protótipo é uma versão inicial de um sistema de software, usado para demonstrar conceitos, experimentar opções de projeto e descobrir mais sobre o problema e suas possíveis soluções. O desenvolvimento rápido e interativo do protótipo é essencial para que os custos sejam controlados e os *stakeholders* do sistema possam experimentá-lo no início do processo de software (SOMMERVILLE, p. 30).

Existem várias vantagens em entregar um protótipo ao cliente:

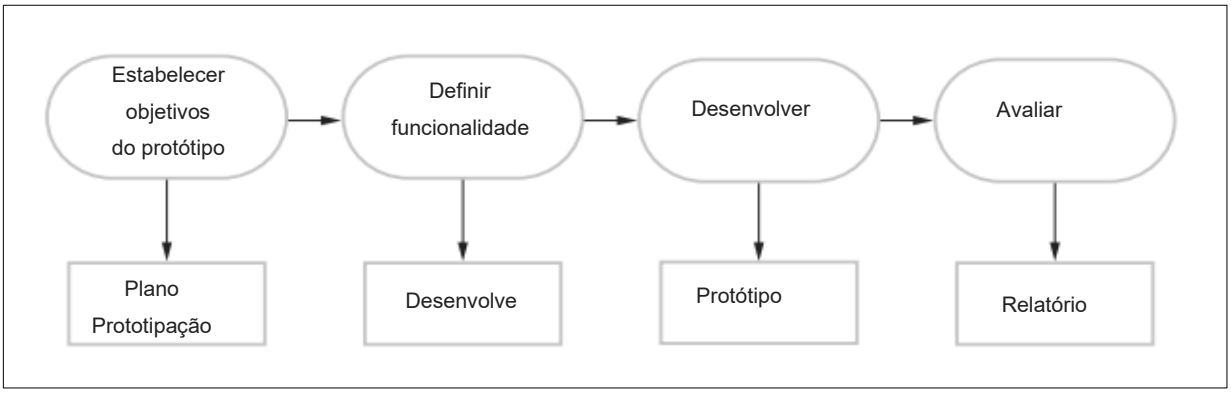
» Desenvolvimento rápido e interativo, pois, como não precisa da lógica de programação, o cliente oferece um feedback muito mais preciso.

» No processo de requisitos, a prototipagem pode elucidar e validar alguns requisitos, inclusive, interfaces de usuário.

» Os usuários podem obter novas ideias e propor uma evolução do sistema.

» Entretanto, devemos definir os objetivos dos protótipos, pois, se o propósito é validar interfaces de usuários, não conseguiremos obter sucesso para validar outas funcionalidades.

Figura 14. O processo de desenvolvimento de protótipo



Fonte: Sommerville (2011, p. 32)

Outra preocupação da prototipagem é no momento dos testes. Apesar de existir um material voltado somente aos testes no curso, é importante destacar que o papel do responsável pelos testes será muito importante, pois vários problemas podem deixar de ser detectados.

Modelo espiral

Segundo Sommerville (2011, p. 33), uma proposta de *framework* dirigido a riscos é denominado espiral. O processo de software é representado como uma espiral, e não como uma sequência de atividades com alguns retornos de uma para outra. Cada volta na espiral representa uma fase do processo de software. Dessa forma, a volta mais interna pode preocupar-se com a viabilidade do sistema; o ciclo seguinte, com definição de requisitos; o seguinte, com o projeto do sistema, e assim por diante.

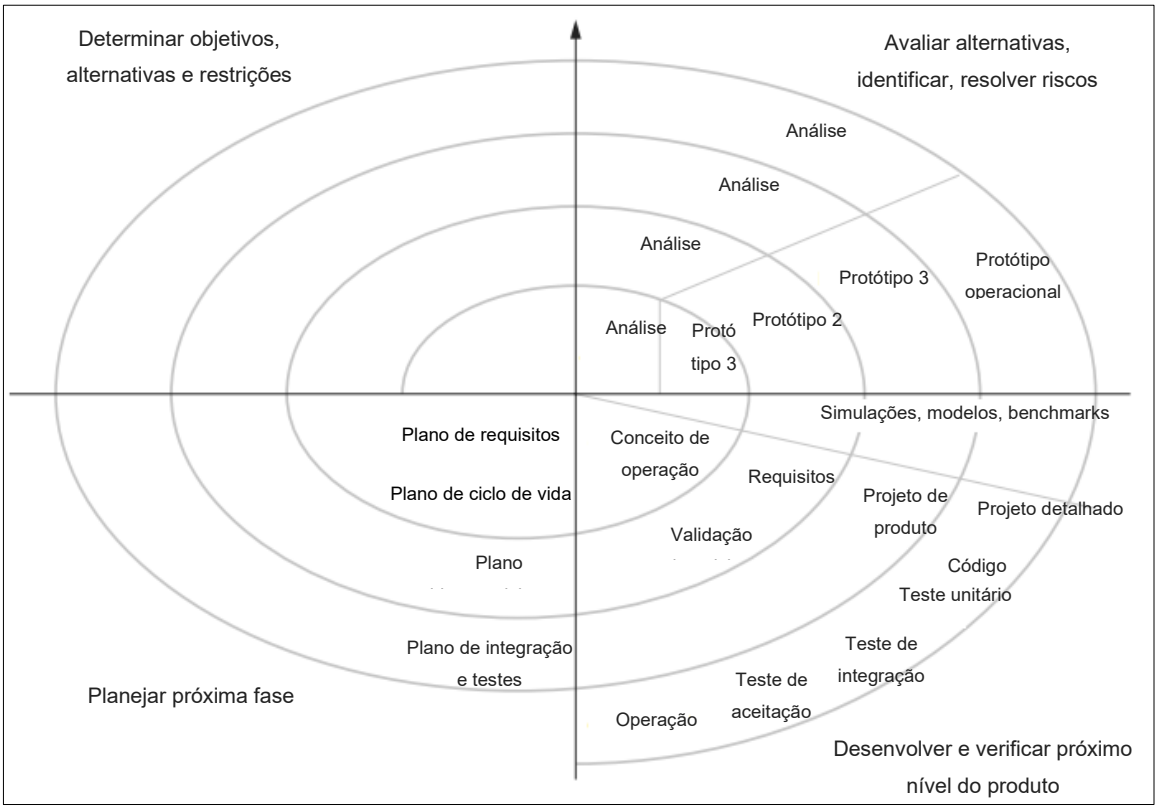
Cada volta da espiral é dividida em quatro setores:

1. Definição de objetivos.
2. Avaliação e redução de riscos.
3. Desenvolvimento e validação.
4. Planejamento.

A diferença entre o modelo espiral e outros modelos de processo de software é seu foco na gestão do risco. Um ciclo da espiral começa com a efetivação de objetivos, como desempenho e funcionalidade. Em seguida, são listadas as formas alternativas de chegar aos objetivos e de lidar com as restrições de cada um deles. Cada alternativa é avaliada em função de cada objetivo, e as fontes de risco do projeto são analisadas.

Em seguida é preciso criar uma solução para esses riscos por meio de atividades de coleta de informações, como análise dos detalhes, prototipação e simulação. Após a avaliação dos riscos, algum desenvolvimento é finalizado, seguido por uma atividade de planejamento para a próxima etapa do processo.

Figura 15: Modelo em espiral de processo de software de Boehm



Fonte: Sommerville (2011, p. 33)

Melhorias de processos

As empresas que utilizam um padrão de processo estão na busca por uma entrega mais barata, rápida e melhor. Essas empresas estão voltando-se a melhorias dos processos, para poder melhorar a qualidade, reduzir custos, adiantar as entregas, conseguindo, assim, atender a demanda por um público cada vez mais exigente.

Segundo Sommerville (2011, p. 493), são duas abordagens bastante diferentes para que sejam feitas melhorias e mudanças de processos:

1. **A abordagem de maturidade de processo**: concentra em, além de introduzir boas práticas baseando-se em práticas anteriores, melhorar o gerenciamento de projeto. Seus principais objetivos produtos de melhor qualidade e previsibilidade de processo.

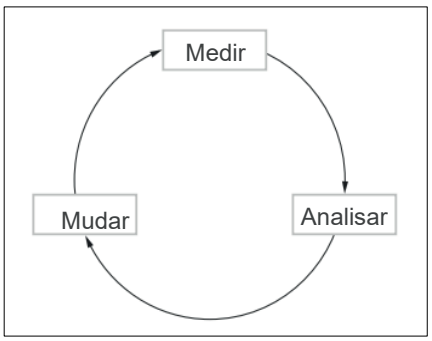
2. **A abordagem ágil**: seus principais objetivos concentram-se na entrega rápida de funcionalidades, capacidade de resposta a mudanças de requisitos, no desenvolvimento interativo e na redução de overheads no processo de software.

Na mesma linha de pensamento, Sommerville (2011, p. 493) dá sua opinião sobre as duas abordagens:

> "Estou convencido de que, para projetos de pequeno e médio portes, a adoção de práticas ágeis provavelmente seja a estratégia mais efetiva de melhoria de processos. No entanto, para grandes sistemas, sistemas críticos e sistemas que envolvem os desenvolvedores em diferentes empresas, muitas vezes as questões de gerenciamento são as razões pelas quais projetos têm problemas. Para as empresas cujo negócio é a engenharia de sistemas grandes e complexos, deve-se considerar uma abordagem centrada na maturidade para melhoria de processos".

O processo de melhoria de processos é um processo cíclico, que envolve três subprocessos:

Figura 16: O ciclo de melhorias de processos



Fonte: Sommerville (2011, p. 497)

Medição

Atributos do projeto são medidos ajudando a melhorar com base em conhecimento. Três métricas podem ser coletadas: tempo necessário para um processo ser concluído, recursos para um determinado processo, número de ocorrência para um determinado evento.

Análise

A análise de processo é o estudo dos processos para ajudar a compreender suas principais características e como esses processos são executados na prática, pelas pessoas envolvidas (SOMMERVILLE, 2011, p. 499).

Mudança

São mudanças propostas para melhorar os pontos fracos detectados na análise. Segundo Sommerville (2011, p. 503), existem cinco estágios principais no processo de mudança de processos:

» Identificação de melhorias.

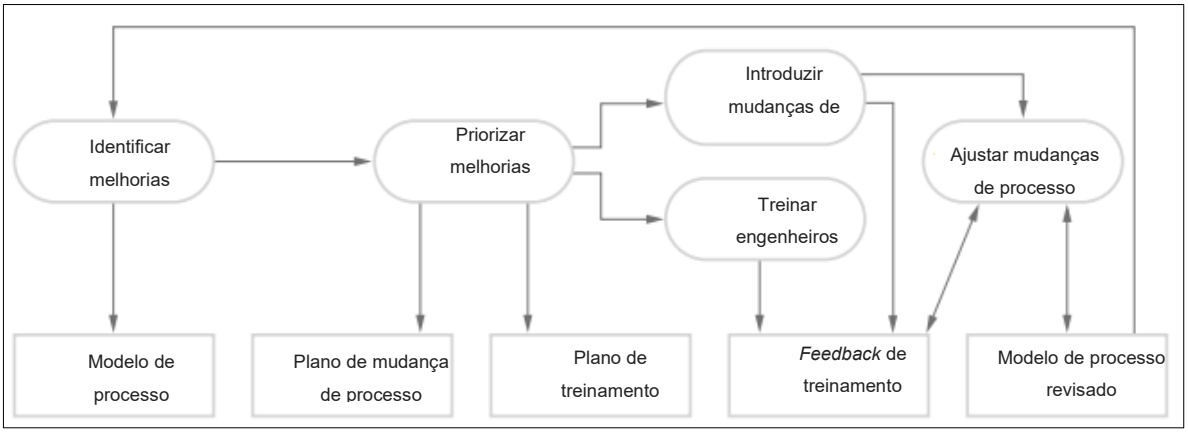
» Priorização de melhorias.

» Introdução de mudanças de processo.

» Treinamento de processos.

» Ajuste de mudanças.

Figura 17: O processo de mudança de processos



Fonte: Sommerville (2011, p. 502)

Após a fase de mudanças, o ciclo recomeça e novas análises recomeçam.

Modelo de Maturidade - *Capability Maturity Model Integration*

O CMMI procura nortear a organização no sentido de implementar a melhoria contínua do processo de software, e o faz através de um modelo que contempla duas representações, divididas em níveis, priorizando de forma lógica as ações a serem realizadas.

O objetivo do CMMI é servir de guia para a melhoria de processos na organização, assim como para auxiliar a habilidade dos profissionais em gerenciar o desenvolvimento de aquisição e manutenção de produtos ou serviços de software, além de proporcionar a visibilidade apropriada do processo de desenvolvimento para todos os envolvidos no projeto. Isso é particularmente importante em grandes projetos que possuem equipes envolvendo dezenas de pessoas, pois, sem o apoio desses modelos de maturidade de processos de software como o CMMI, torna-se ainda mais difícil manter o controle do projeto.

Com a utilização de níveis, o CMMI descreve um caminho evolutivo recomendado para uma organização que deseja melhorar os processos utilizados para construção de seus produtos e serviços. Os níveis também podem resultar de classificações obtidas por meio de avaliações realizadas em organizações compreendendo a empresa toda (normalmente pequenas), ou grupos menores, tais como um grupo de projetos ou uma divisão de uma empresa.

CAPÍTULO 4

Gestão de projeto e aspectos fundamentais

Os capítulos seguintes serão determinantes para que se possa fazer uma eficiente gestão de projeto de software. Entretanto, alguns aspectos fundamentais serão primordiais para que entendamos, de fato, o que é um projeto.

O guia PMBOK

O guia PMBOK é um padrão reconhecido para a profissão de gerenciamento de projetos. Um padrão é um documento formal que descreve normas, métodos, processos e práticas estabelecidas, um vocabulário comum dentro da profissão de gerenciamento de projetos. Ele define o gerenciamento e os conceitos relacionados e descreve o ciclo de vida do gerenciamento de projetos e os processos relacionados.

O Guia PMBOK identifica o subconjunto do conhecimento em gerenciamento amplamente reconhecido como boa prática. “Amplamente reconhecido” significa que o conhecimento e as práticas descritas são aplicáveis à maioria dos projetos na maior parte do tempo e que existe um consenso em relação ao seu valor e sua utilidade. “Boa prática” significa que existe um consenso de que a aplicação correta dessas habilidades, ferramentas e técnicas pode aumentar as chances de sucesso em uma ampla gama de projetos.

O PMBOK define projeto como:

> Um esforço temporário empreendido para criar um produto, serviço ou resultado exclusivo. A sua natureza temporária indica um início e um término definidos. O término é alcançado quando os objetivos tiverem sido atingidos ou quando se concluir que esses objetivos não serão ou não poderão ser atingidos e o projeto for encerrado, ou quando o mesmo não for mais necessário.

Essa deve ser a definição mais utilizada, pois o PMBOK define as boas práticas de gerenciamento de projetos e acabou se tornando uma referência. São dadas como padrão as suas definições.

Os projetos possuem suas particularidades e cada um gera um produto ou serviço e tem impactos sociais, econômicos e ambientais, além de envolver uma única pessoa ou

diversas unidades organizacionais. Embora algumas características em comum possam estar presentes, não muda sua exclusividade. Por exemplo, prédios comerciais, hotéis etc., são construídos pela mesma composição de material, porém, cada um tem sua singularidade. Quando os projetos possuem tarefas novas, surgem as incertezas que demandarão mais desempenho e dedicação no planejamento.

O PMBOK cita alguns exemplos de projetos que incluem, mas não se limitam a:

- Desenvolvimento de um novo produto ou serviço.
- Uma mudança de estrutura, de pessoal ou de estilo de uma organização.
- Desenvolvimento ou aquisição de um sistema de informações novo ou modificado.
- Construção de prédio ou infraestrutura ou
- Implementação de um novo procedimento ou processo de negócios.

Gestão de projeto

Gerenciar um projeto é aplicar todos recursos disponíveis, como ferramentas, técnicas, boas práticas, padrões, a fim de atender aos requisitos propostos no planejamento. Como conceito de boas práticas, o PMBOK dividiu em 5 grupos as etapas do gerenciamento:

- iniciação;
- planejamento;
- execução;
- monitoramento e controle;
- encerramento.

O gerenciamento do projeto inclui várias etapas e nelas estão incluídos:

- escopo;
- qualidade;
- cronograma;
- orçamento;

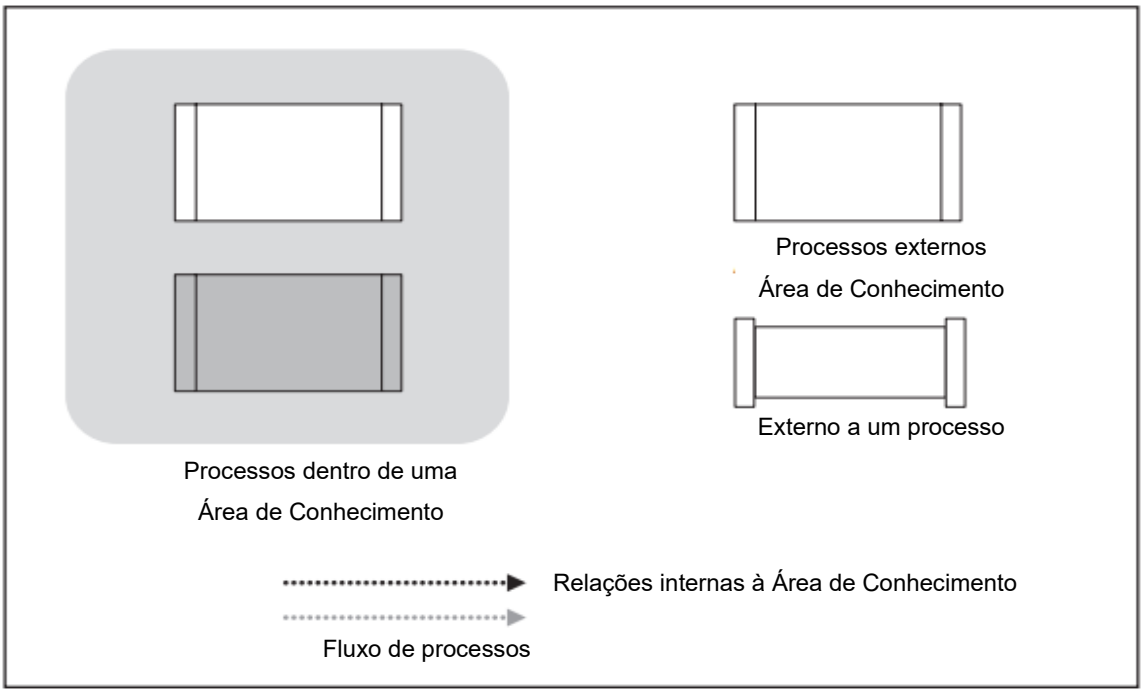
» recursos;

» risco.

Esses tópicos serão tratados individualmente nos capítulos seguintes.

O guia PMBOK subdivide o estudo de cada fase do projeto em Entradas, Ferramentas Técnicas e Saídas, e ilustra o fluxo de informações através de um DFD. Usaremos essa mesma técnica em nossos estudos.

Figura 18: Legenda do diagrama de fluxo de dados



Fonte: PMBOK (2017 p. 69)

Gerenciamento de projetos de software

Gerenciamento de projetos de software refere-se ao ramo de gerenciamento de projetos dedicado ao planejamento, programação, alocação de recursos, execução, rastreamento e entrega de projetos de software.

O gerenciamento de projetos de software segue algumas etapas do gerenciamento de projeto padrão, ou seja, o projeto mais genérico que também é utilizado para produtos e serviços em geral. Porém, os projetos de software têm um processo exclusivo de ciclo de vida que exige várias rodadas de testes, atualizações e feedback dos clientes. A maioria dos projetos relacionados a TI é gerenciada no estilo ágil, a fim de acompanhar o ritmo

crescente dos negócios e fazer interações com base no feedback dos clientes e das partes interessadas.

Como dissemos em ocasiões anteriores, a arquitetura de software faz parte de um conceito maior no universo do desenvolvimento de software, a engenharia de software. Projetar o software em sua plenitude, normalmente é responsabilidade dos engenheiros com a participação de todos os envolvidos, inclusive, os arquitetos. Por isso, é de suma importância o Arquiteto de Softwares entender os conceitos de gestão de projetos, pois, saberá qual é sua função, as etapas, o motivos das decisões dos engenheiros, para poder, enfim, escolher e direcionar para a criação, configuração, customização, a reutilizar componentes, organizar a interação entre componentes e acoplamento dos componentes de software.

Segundo Pressman (2011, p. 207), o objetivo de projetar é gerar um modelo ou representação que apresente solidez e comodidade. Para ele, o projeto de software é a última ação da atividade de modelagem, após a análise e modelagem nos requisitos. Após a análise de requisitos, a especificação completa cria quatro modelos de projetos necessários:

» **Projeto de classes**

Transforma os modelos de classes em realizações de classes de projetos e nas estruturas de dados dos requisitos necessários para implementar o software.

» **Projeto arquitetural**

Define o relacionamento entre os principais elementos estruturais do software, os estilos arquiteturais e padrões de projetos, podendo ser usados nos requisitos e nas restrições.

» **Projeto de interfaces**

Descreve como o software de comunica com sistemas e com os usuários que interagem com ele, implica o fluxo de informações e um típico de comportamento específico.

» **Projetos de componentes**

Transforma elementos estruturais da arquitetura de software em uma descrição procedural dos componentes. Suas informações servem de base para o projeto de componentes.

Antes de ser traçado um plano para o projeto, é necessário reunir com todos os interessados no software para que possam ser definidos o escopo e os objetivos do produto. Também será preciso considerar soluções alternativas e identificar soluções técnicas e de gerenciamento, pois, sem essas informações, será impossível definir precisamente a estimativa de prazo e custo, avaliação de riscos e cronograma. Essa é considerada a primeira etapa da engenharia de requisitos.

Segundo Pressman (2011, p. 567):

> Os objetivos identificam as metas gerais do produto sem considerar como tais metas estão alcançadas. O escopo identifica os principais dados e funções e comportamentos que caracterizam o produto e, mais importante, tenta mostrar as fronteiras e limitações dessas características de maneira quantitativa.

PLANEJAMENTO E GERENCIAMENTO

UNIDADE II

CAPÍTULO 1

Escopo

Escopo de Software

O escopo de software é definido logo no início do gerenciamento do projeto. Ele pode ser elaborado levando em conta as seguintes questões:

» **Contexto**:

Define o grupo de tarefas que determina pontos relevantes do contexto em que será utilizado o produto de software. Dependendo da complexidade e responsabilidade do produto, pode ser resolvida em uma conversa informal com o cliente ou requerer a execução de um processo complexo de definição de produto, engenharia de requisitos de sistema, modelagem de processos de negócio (FILHO, 2001, p. 105).

» **Objetivo da informação**

Define quais os objetivos de entrada e saída de dados que serão utilizados pelos clientes.

» **Função e performance**

Quais as funções e performance do software quando transforma os dados de entrada em dados de saída e o formato que esses dados poderão ser acessados.

Para realizar seu o gerenciamento de escopo, o PMBOK oferece os seguintes processos:

1. coletar os requisitos;
2. definir o escopo;

3. criar a EAP;
4. verificar o escopo;
5. controlar o escopo.

Coletar os requisitos

É o processo responsável por definir e documentar as funcionalidades do sistema e seu sucesso será influenciado pelo esforço em capturar detalhes dos requisitos do sistema. Nos requisitos se encontram as necessidades devidamente documentadas, as expectativas dos interessados, ou seja, o patrocinador do projeto que, normalmente, é o cliente. Depois de coletados, os requisitos geram a EAP, o planejamento de custos e cronogramas. Seu início se dá no termo de abertura do projeto.

Segundo o PMBOK, muitas empresas categorizam os requisitos de projeto e requisitos de produtos, onde os requisitos de projetos podem incluir os de negócios, de gerenciamento de projetos, de entrega etc. Os de produtos incluem informações sobre requisitos técnicos, de segurança, desempenho etc.

- **Entradas:** Termo de abertura do projeto, registro das partes interessadas;
- **Ferramentas e técnicas:** Entrevistas, dinâmicas de grupo, oficinas, técnicas de criatividade em grupo, técnicas de tomadas de decisão em grupo, questionários e pesquisas, observações, protótipos;
- **Saídas:** Documentação dos requisitos, plano de gerenciamento dos requisitos, matriz de rastreabilidade dos requisitos.

Definir o escopo

Segundo o PMBOK (2017, p. 112):

> Definir o escopo é processo de desenvolvimento de uma descrição detalhada do projeto e do produto. A preparação detalhada da declaração do escopo é crítica para o sucesso e baseia-se nas entregas principais, premissas e restrições que são documentadas durante a iniciação do projeto.

Um dos mais importantes passos no gerenciamento de projetos é elaborar uma definição de escopo que identifique e descreva todo o trabalho necessário para produzir o produto final. Ela estabelece o ritmo para o restante dos esforços de planejamento

e, portanto, deve ser suficientemente detalhada. Tenha em mente, no entanto, que ser muito detalhado pode ser tão incômodo quanto não fornecer detalhes suficientes. A definição do escopo destina-se a garantir que todos na equipe entendam o que se espera deles durante o projeto.

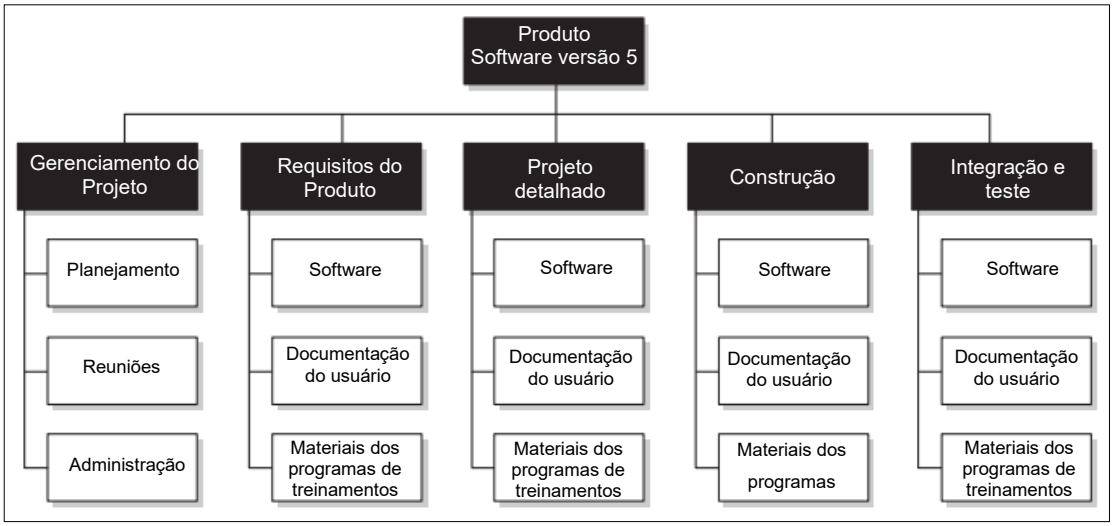
- **Entradas**: Termo de abertura do projeto, documentação dos requisitos, ativos de processos organizacionais.
- **Ferramentas e Técnicas:** Opinião especializada, análise de produto, identificação de alternativas, oficinas.
- **Saídas:** Declaração do escopo do projeto, atualizações dos documentos do projeto.

Criar a EAP

Criar uma EAP é pegar o trabalho e suas entregas e fazer suas subdivisões. Basicamente, é transformar em uma hierarquia, orientada às entregas, e sua direção é sempre descendente. Foi definida uma padronização para construir uma EAP com as melhores práticas: **The Practice Standard for Work Breakdown Structures**, podendo ser acessada em: https://www.pmi.org/pmbok-guide-standards/framework/practice-standard-work-breakdown-structures-2nd-edition.

- **Entradas:** Declaração do escopo do projeto, documentação dos requisitos, ativos de processos organizacionais.
- **Ferramentas e Técnicas:** Decomposição.
- **Saídas:** EAP, Dicionário da EAP, Linha de base do escopo, atualizações dos documentos do projeto.

Figura 19: Exemplo de uma estrutura analítica de projeto organizada por fases



Fonte: PMBOK (2017 p. 119)

Verificar o escopo

A verificação do escopo formaliza a aceitação de entregas concluídas. É feita uma aceitação formal que assegure ao cliente ou patrocinador do projeto que as etapas foram entregues conforme as especificações. Entretanto, essa verificação não é um controle de qualidade, pois não se interessa na precisão da entrega.

- **Entradas:** plano de gerenciamento do projeto, documentação dos requisitos, matriz de rastreabilidade dos requisitos, entregas validadas.
- **Ferramentas e técnicas**: inspeção.
- **Saídas:** Entregas aceitas, solicitação de mudanças, atualização da documentação do projeto.

Controlar o escopo

Controlar o escopo significa monitorar o andamento do projeto e suas mudanças. É assegurado que todas as mudanças que foram solicitadas e ações corretivas são monitoradas por esse processo.

- **Entradas:** plano de gerenciamento do projeto, informações sobre o desempenho do trabalho, documentação dos requisitos, ativos de processos organizacionais.
- **Ferramentas e técnicas**: análise de variação.
- **Saídas:** medição de desempenho, atualização dos ativos dos processos organizacionais, solicitação de mudanças, atualização do plano e dos documentos de gerenciamento do projeto.

CAPÍTULO 2
As Pessoas

Segundo Pressman (2011, p. 567), para que o gerenciamento seja efetivo, é necessário focar nos 4 Ps: Pessoas, Produto, Processo e Projeto.

Desde muito tempo o foco em gerenciamento e motivação de pessoas vem sendo tema na Engenharia de Software. O SEI – Software Engineering Institute – deu tanta importância que até criou um modelo de capacitação de pessoas para o aprimoramento da continuidade de suas habilidades. O People Capability Maturity Model é uma ferramenta que ajuda você a abordar as questões críticas de pessoas em seu time de desenvolvedores. É empregado o processo de estrutura de maturidade do altamente bem-sucedido Capability Maturity Model para Software (SWCMM) como base para um modelo de melhores práticas para gerenciar e desenvolver a força de trabalho de uma organização. O Software CMM tem sido usado por organizações de software em todo o mundo para orientar melhorias drásticas em sua capacidade de melhorar a produtividade e a qualidade, reduzir custos e tempo de lançamento no mercado e aumentar a satisfação do cliente. A filosofia implícita do People CMM pode ser resumida em dez princípios:

6. Em organizações maduras, a capacidade da força de trabalho está diretamente relacionada ao desempenho do negócio.

7. A capacidade da força de trabalho é uma questão competitiva e uma fonte de vantagem estratégica.

8. A capacidade da força de trabalho deve ser definida em relação aos negócios estratégicos da organização.

9. O trabalho intenso em conhecimento desloca o foco dos elementos do trabalho para as competências da força de trabalho.

10. Capacidade pode ser medida e melhorada em múltiplos níveis, incluindo indivíduos, grupos de trabalho, competências da força de trabalho e organização.

11. Uma organização deve investir na melhoria da capacidade das competências da força de trabalho que são críticos para sua competência central como um negócio.

12. O gerenciamento operacional é responsável pela capacidade da força de trabalho.

13. A melhoria da capacidade da força de trabalho pode ser buscada como um processo composto de práticas e procedimentos comprovados.

14. A organização é responsável por fornecer oportunidades de melhoria, enquanto indivíduos são responsáveis por tirar proveito delas.

15. Como as tecnologias e as formas organizacionais evoluem rapidamente, as organizações devem continuamente evoluir suas práticas de força de trabalho e desenvolver novas competências de força de trabalho.

Os interessados

Todo projeto, independente de software ou de produtos manufaturados, possui os interessados, normalmente chamados de *stakeholders*. Pressman (2011, p. 569), os classifica em cinco categorias:

- **Gerentes seniores:** definem os itens de negócio.
- **Gerentes técnicos:** gerenciam programadores.
- **Programadores**: devem possuir habilidades para desenvolver o produto.
- **Clientes:** alguns especificam os requisitos e outros têm mais interesse no produto.
- **Usuários:** interagem com o software quando liberado para uso operacional.

Gerente do projeto

O gerente do projeto é a pessoa responsável por fazer o projeto atingir seus objetivos. Segundo Filho (2011, p. 85), o gerente de projetos tem responsabilidade completa por um projeto, inclusive a direção, controle e administração deste. O gerente de um projeto deve ser o único responsável por este, aos olhos do cliente. Cabe ao gerente liderar a equipe do respectivo projeto.

Líderes de equipes

Gerenciar um projeto requer habilidades com pessoas e, geralmente, membros técnicos da equipe, quando promovidos para efetuar tal tarefa, não possuem sensibilidades adequadas com pessoas. Em sua obra On Becomming a Technical Leader, Weinberg (1986, p. 15) sugere o modelo MOI de liderança: Motivação, Organização, Ideias ou Inovação.

Para que a mudança ocorra, o ambiente deve conter três ingredientes:

» **M: motivação** - os troféus ou problemas, o empurrão ou puxão que move as pessoas envolvidas.

» **O: organização** - a estrutura existente que permite que as ideias sejam trabalhadas na prática.

» **I: ideias ou inovação** - as sementes, a imagem do que se tornará.

Motivação de pessoas

Em organizações que desenvolvem produtos manufaturados, existem muitos ativos de alto valor. Ativos como uma máquina de última geração que faz todo o produto sozinha. Em empresa de desenvolvimento de software esse ativo tem seu valor todo voltado para as pessoas. Segundo Sommerville (2011, p. 421), custa muito para essas empresas reterem boas pessoas. Cabe ao gerente de projeto garantir que a empresa aproveite ao máximo o retorno investido e isso só é possível quando as pessoas são respeitadas e lhes são atribuídas responsabilidades que refletem suas habilidades e experiências.

Além de possuir boas habilidades técnicas, os gerentes de projetos de software precisam ter habilidades para poder motivar pessoas. O que em muitas vezes isso não acontece! Na opinião de Sommerville (2011, p. 421), existem quatro fatores críticos no gerenciamento de pessoas:

1. **Consistência**. Membros de uma equipe devem ter o mesmo tratamento. Ninguém espera o tratamento igual para todos, mas as pessoas não podem se sentir subvalorizadas.

2. **Respeito**. Pessoas possuem habilidades diferentes e todos devem ter a oportunidade de contribuir, exceto em casos em que o membro não se encaixa na equipe.

1. **Inclusão**. A atenção é muito importante, pois quando a pessoa sente que está sendo ouvida, sua contribuição é mais efetiva.

1. **Honestidade**. O gerente deve ser honesto consigo e com os membros da equipe em relação a conhecimento técnico e se dispor a submeter a um membro com mais habilidades.

Um gerente de projeto deve estar pronto para motivar a equipe, o seja, para fazer com que as pessoas contribuam com o projeto, o gerente precisa organizar e manter harmônico o ambiente de trabalho. Segundo o PMBOK (2011 p. 239), os conflitos também serão inevitáveis, por diversos fatores, e um gerenciamento de conflitos bem gerenciado resulta em melhor produtividade e em trabalhos positivos. São propostas seis técnicas para resolver conflitos:

» **Retirada/Evitar**: Recuar de uma situação de conflito efetivo ou potencial.

» **Panos quentes/Acomodação**: Enfatizar as áreas de acordo e não as diferenças.

» **Negociação**: Encontrar soluções que trazem algum grau de satisfação para todas as partes.

» **Imposição**: Forçar um ponto de vista às custas de outro; oferece apenas soluções ganha-perde.

» **Colaboração**: Incorporar diversos pontos de vista e opiniões de diferentes perspectivas; resulta em consenso e compromisso.

» **Confronto/Solução de problemas**: Tratar o conflito como um problema que deve ser solucionado com o exame de alternativas; requer uma atitude de troca e diálogo aberto.

Como gerente de projetos, você precisa motivar as pessoas que trabalham com você para que elas contribuam com o melhor de suas habilidades. Motivação significa organizar o trabalho e o ambiente de trabalho para incentivar as pessoas a trabalharem do modo mais eficaz possível.

CAPÍTULO 3
Análise de riscos

Todo projeto tem como principal característica chegar ao fim. Se estamos desenvolvendo algo e nunca termina, existem duas opções, ou existe algo errado ou não é projeto, são simplesmente tarefas. Para a definição de final podemos contar que o cliente ficou satisfeito com o produto, pois, cumpriu o cronograma, as funcionalidades foram satisfatórias conforme os requisitos e o orçamento foi o esperado. Entretanto, em meio a esse resumo, existiram muitas possibilidades de acontecer problemas. Muitos riscos poderiam comprometer a entrega do produto; muitos riscos poderiam fazer com que o projeto não cumprisse alguns de seus objetivos: prazo, custo e funcionalidades.

Mesmo que um projeto não se depare com um problema, o risco de acontecer é muito alto. Segundo Pressman (2011, p. 658), o risco é um problema potencial e, independente do resultado, é aconselhável identificá-lo, avaliar sua probabilidade de ocorrência, estimar seu impacto e estabelecer um plano de contingência caso ele ocorra.

Gerenciamento de riscos

Gestão de riscos significa contenção e mitigação de riscos. Inicialmente é preciso identificá-lo para poder planejar as ações de contenção ou eliminação. Estar preparado para agir quando o risco acontecer e contar com a experiência e conhecimento de toda a equipe para minimizar o impacto no projeto.

Independente de qual produto esteja sendo desenvolvido, seja ele manufaturado ou software, o gerenciamento de riscos inclui as seguintes tarefas:

- Identifique os riscos.
- Classifique e priorize todos os riscos.
- Elabore um plano que vincule cada risco a uma mitigação.
- Monitore os gatilhos de risco durante o projeto.
- Implemente a ação de mitigação se algum risco se concretizar.
- Comunique o status de risco em todo o projeto.

Sommerville (2011, p. 416) sugere três categorias de riscos:

1. Riscos de projeto. Riscos que afetam o cronograma ou os recursos de projeto. Um exemplo de um risco de projeto é a perda de um projetista experiente. Encontrar um projetista substituto com competência e experiência adequados pode demorar muito tempo e, por conseguinte, o projeto de software vai demorar mais tempo para ser concluído.

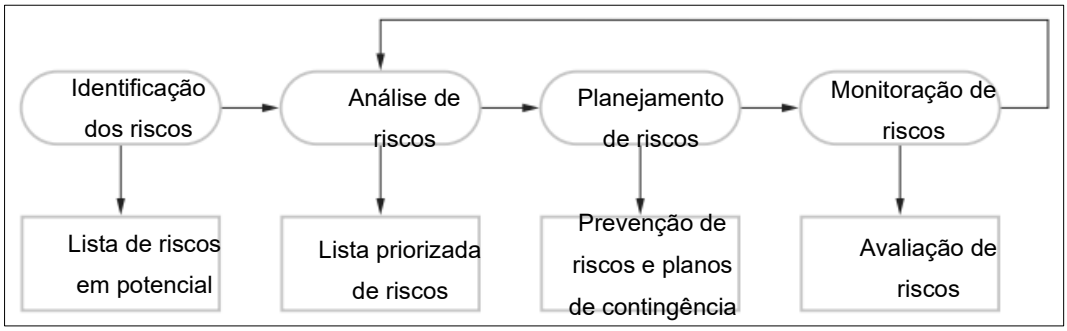
2. Riscos de produto. Riscos que afetam a qualidade ou o desempenho do software que está sendo desenvolvido. Um exemplo de um risco de produto é a falha de um componente comprado para o desempenho esperado, podendo afetar o desempenho geral do sistema de forma mais lenta do que o esperado.

3. Riscos de negócio. Os riscos que afetam a organização que desenvolve ou adquire o software. Por exemplo, um concorrente que introduz um novo produto é um risco empresarial. A introdução de um produto competitivo pode significar que as suposições feitas sobre vendas de produtos de software existentes podem ser excessivamente otimistas.

Identificação do risco

A identificação dos riscos começa com a definição de uma lista de riscos em potencial, que serão priorizados e analisados. Para que os riscos sejam identificados, um plano de gerenciamento deve ser criado a fim de identificar as possíveis ações de contenção, a redução de probabilidade de ocorrência do risco, reduzindo assim o impacto, caso o risco se efetive, se tornando um problema. A identificação também inclui a tarefa de rastrear a probabilidade de o risco acontecer. Essa etapa envolve monitorar o status do risco conhecido e o resultado das ações de redução. O status poderá ser um indicador de que esse risco se transformará em um problema. Por fim, adicionam-se os riscos identificados em uma base de conhecimento para servir de contingência para problemas conhecidos para a posteridade.

Figura 20. Processo de Gerenciamento de Riscos



Fonte: Sommerville (2011, p. 417)

Segundo o PMBOK identificar os riscos é o processo de determinação dos riscos que podem afetar o projeto e de documentação de suas características. O processo deve envolver a equipe do projeto de modo que possa desenvolver e manter um sentido de propriedade e responsabilidade pelos riscos e pelas ações associadas de resposta a riscos. As partes interessadas externas à equipe do projeto podem fornecer informações objetivas adicionais.

» **Entradas:** Plano de gerenciamento de riscos, cronograma, documentos.

» **Ferramentas e técnicas:** Revisão de documentação, técnicas de coleta de informações, análise de premissas, análise SWOT.

» **Saídas:** Registro dos riscos.

Classificação ou análise de risco

A classificação ou análise de risco converte os dados brutos dos riscos em informações para tomadas de decisão e fornece ao gerente de projeto, que é o responsável pelas tomadas de decisão, uma base rica em detalhes.

Análise qualitativa

Nesse processo, cada risco deve ser julgado baseando-se em considerações de experiências anteriores. O PMBOK sugere uma classificação qualitativa de riscos onde faz uma avaliação do impacto e investiga o efeito potencial sobre um objeto. Essa avaliação de probabilidade é feita em cada risco identificado. Realizar a análise qualitativa dos riscos é o processo de priorização de riscos para análise ou ação adicional através da avaliação e combinação de sua probabilidade de ocorrência e impacto.

» **Entradas:** Registro dos riscos, plano de gerenciamento dos riscos, declaração do escopo do projeto, ativos de processos organizacionais.

» **Ferramentas e técnicas:** Registro dos riscos, plano de gerenciamento dos riscos, declaração do escopo do projeto, ativos de processos organizacionais.

» **Saídas:** Atualizações do registro dos riscos.

Segundo o PMBOK (2011, p. 292), existe a possibilidade da criação de uma matriz de probabilidades, cuja funcionalidade é priorizar os riscos, através de uma regra de classificação por prioridades. Ela combina probabilidade e impacto e mostra o resultado com prioridades alta, moderada ou baixa.

» A área cinza escura (com os números maiores) representa alto risco.

» A área cinza média (com os números menores) representa baixo risco.

» A área cinza claro (com os números intermediários) representa risco moderado.

Figura 21. Matriz de probabilidade e impacto.

Matriz de probabilidade

Probabilidade	Ameaças					Ameaças				
0.90	0.05	0.09	0.18	0.36	0.72	0.72	0.36	0.18	0.09	0.05
0.70	0.04	0.07	0.14	0.28	0.56	0.56	0.28	0.14	0.07	0.04
0.50	0.03	0.05	0.10	0.20	0.40	0.40	0.20	0.10	0.05	0.03
0.30	0.02	0.03	0.06	0.12	0.24	0.24	0.12	0.06	0.03	0.02
0.10	0.01	0.01	0.02	0.04	0.08	0,08	0.04	0.02	0.01	0.01
	0.05	0.10	0.20	0.40	0.80	0,80	0.40	0.20	0.10	0.05

Impacto (escala numérica) em um objetivo (por exemplo, custo, tempo, escopo ou qualidade

Cada risco é avaliado de acordo com a sua probabilidade de ocorrência e o impacto em um objetivo se este realmente ocorrer. Os limites de tolerância da organização para riscos baixos, moderados ou altos são mostrados na matriz e determinam se o risco é alto, moderado ou baixo para aquele objetivo.

Fonte: PMBOK (2017 p. 292)

Análise quantitativa

É o processo de analisar numericamente o efeito dos riscos identificados nos objetivos gerais do projeto. Ela é aplicada nos riscos priorizados na análise qualitativa e tem impacto nas demandas concorrentes do projeto. Os eventos são analisados e podem ser usados para atribuir uma classificação numérica aos riscos, individualmente ou todos os riscos que afetam o projeto.

Esse processo deve ser repetido depois de planejar as respostas aos riscos e também como parte do processo de monitorar e controlar os riscos, para determinar se o risco geral do projeto diminuiu satisfatoriamente. As tendências podem indicar a necessidade de mais ou menos ações de gerenciamento dos riscos (PMBOK, 2017, p. 295).

» **Entradas**: Registro dos riscos, plano de gerenciamento dos riscos, plano de gerenciamento dos custos, plano de gerenciamento do cronograma, ativos de processos organizacionais.

» **Ferramentas e técnicas**: Técnicas de coleta e apresentação de dados, técnicas de modelagem e análise quantitativa de riscos, opinião especializada.

» **Saídas**: Atualizações do registro dos riscos.

Planejamento

Planejamento transforma informações de risco em decisões e ações. O planejamento envolve o desenvolvimento de ações para tratar de riscos individuais, priorizar ações de risco e criar um plano de gerenciamento de riscos integrado.

» **Entradas**: Declaração do escopo do projeto, plano de gerenciamento dos custos, plano de gerenciamento do cronograma, plano de gerenciamento das comunicações, fatores ambientais da empresa, ativos de processos organizacionais.

» **Ferramentas e técnicas**: Reuniões de planejamento e análise.

» **Saídas**: Plano de gerenciamento dos riscos.

Segundo o PMBOK (2011, p. 301), esse processo aborda os riscos pela prioridade, inserindo recursos e atividades no orçamento, no cronograma e no plano de gerenciamento do projeto, conforme necessário. As respostas planejadas devem ser adequadas à relevância do risco, ter eficácia de custos para atender ao desafio, ser realistas dentro do contexto do projeto, acordadas por todas as partes envolvidas e ter um responsável designado.

O plano para um risco específico pode assumir muitas formas. Por exemplo:

» **Mitigar** o impacto do risco, desenvolvendo um plano de contingência, caso o risco ocorra.

» **Eliminar**: a eliminação do risco altera o plano de gerenciamento para atualizar a informação sobre a eliminação total do risco.

» **Aceitar:** o aceite da equipe diz que não será tomada mais nenhuma ação, mesmo que o risco volte, pois, ou não conseguiram identificar o risco ou o aceitaram.

» **Transferir**: transferir o risco simplesmente passa a responsabilidade pelo gerenciamento para outra parte, mas não o elimina.

Esse processo leva em consideração que os riscos foram identificados e ele está pronto para que estratégias sejam desenvolvidas. Sommerville (2011, p. 420), considera três categorias:

1. **Estratégias de prevenção**. Seguir essas estratégias indica que a probabilidade de um risco ocorrer será reduzida.
2. **Estratégias de minimização**. Seguir essas estratégias indica que o impacto do risco será reduzido.
3. **Planos de contingência**. Seguir essas estratégias indica que você está preparado e tem uma estratégia para lidar com o pior.

Monitoramento do risco

Monitorar os riscos significa implementar planos de respostas, acompanhar, monitorar, identificar novos e avaliar o processo de monitoramento. Esse processo deve ser feito ao longo de todo o projeto. Segundo Sommerville (2011, p. 416), é importante avaliar regularmente os riscos e seus planos para mitigação de riscos e atualizá-los quando souber mais sobre os riscos.

Segundo PMBOK (2017, p. 308), outras finalidades do processo de monitorar e controlar os riscos determinam se:

» As premissas do projeto ainda são válidas.

» A análise mostra um risco avaliado que foi modificado ou que pode ser desativado.

» As políticas e os procedimentos de gerenciamento dos riscos estão sendo seguidos.

» As reservas para contingências de custo ou cronograma devem ser modificadas de acordo com a avaliação atual dos riscos.

› **Entradas:** Registro dos riscos, plano de gerenciamento dos riscos, informações sobre o desempenho do trabalho, relatórios de desempenho.

› **Ferramentas e técnicas:** Reavaliação de risco, auditorias de riscos, análises de variação e tendências, medição de desempenho técnico, análise de reservas, reuniões de andamento.

› **Saídas:** Atualizações do registro dos riscos, atualizações dos ativos de processos organizacionais, solicitações de mudança, atualizações do plano de gerenciamento do projeto, atualizações dos documentos do projeto.

CAPÍTULO 4
Cronograma

Princípios Básicos

Gerenciar o cronograma é basicamente gerenciar para que o projeto seja entregue no prazo combinado.

Segundo Pressman (2011, p. 631), o cronograma do projeto é uma atividade que distribui esforços estimados por toda a duração planejada do projeto alocando esse esforço para cada tarefa específica.

O cronograma é uma ferramenta gerenciada pelo gerente de projetos, que o utiliza para comprometer as pessoas envolvidas, pois, se uma tarefa está no cronograma, a equipe estará comprometida a fazê-la, ou seja, ele coloca a equipe e o projeto sob controle. Também irá mostrar ao cliente como o trabalho será desenvolvido, comunicar os prazos finais e, possivelmente, comunicar as necessidades de recursos, seja financeiro, de pessoal ou financeiro.

O cronograma do projeto é um calendário que vincula as tarefas a serem realizadas com os recursos que as farão. Antes que um cronograma de projeto possa ser criado, o gerente de projeto deve ter uma estrutura analítica de trabalho (EAP), uma estimativa de esforço para cada tarefa e uma lista de recursos com disponibilidade para cada recurso.

Segundo PMBOK, os processos de gerenciamento do tempo são:

- **Definir as atividades:** identifica as ações para produzir as entregas.
- **Sequenciar as atividades:** identifica e documenta o relacionamento entre as atividades.
- **Estimar os recursos das atividades:** estima os tipos e quantidades de material, pessoas, equipamentos ou suprimentos que serão necessários para realizar cada atividade.
- **Estimar as durações das atividades:** estima o mais próximo possível o número de períodos de trabalho que serão necessários para terminar atividades específicas com os recursos estimados.

» **Desenvolver o cronograma:** o processo de análise das sequências visa criar o cronograma do projeto.

» **Controlar o cronograma:** o processo de monitoramento do andamento do projeto.

Definir as atividades

Para PMBOK (2017, p. 133), atividades é o processo de identificação das ações específicas a serem realizadas para produzir as entregas do projeto. O processo 'Criar a EAP' identifica as entregas no nível mais baixo da estrutura analítica do projeto (EAP), as atividades. As atividades proporcionam uma base para a estimativa, desenvolvimento do cronograma, execução e monitoramento e controle do trabalho do projeto.

» **Entradas:** Linha de base do escopo, fatores ambientais da empresa, ativos de processos organizacionais.

» **Ferramentas e técnicas:** Decomposição, planejamento em ondas sucessivas, modelos, opinião especializada.

» **Saídas:** Lista das atividades, atributos das atividades, lista dos marcos.

Sequenciar as atividades

Para PMBOK (2017, p. 136), é processo de identificação e documentação dos relacionamentos entre as atividades do projeto, onde cada atividade é conectada a pelo menos um predecessor e um sucessor, excetuando o primeiro e último. O sequenciamento pode ser feito por um software de gerenciamento de projetos.

» **Entradas:** Lista das atividades, atributos das atividades, lista dos marcos, declaração do escopo do projeto, ativos de processos organizacionais.

» **Ferramentas e técnicas:** Método do diagrama de precedência (MDP), determinação de dependência, aplicação de antecipações e esperas, modelos de diagrama de rede de cronograma.

» **Saídas:** Diagramas de rede do cronograma do projeto, atualizações dos documentos do projeto.

Estimar os recursos das atividades

Segundo PMBOK (2017, p. 141), é o processo de estimativa dos tipos e quantidades de material, pessoas, equipamentos ou suprimentos que serão necessários para realizar cada atividade. O processo 'Estimar os Recursos das Atividades' é estreitamente coordenado junto com o processo 'Estimar os Custos'.

» **Entradas:** Lista das atividades, atributos das atividades, calendários dos recursos, fatores ambientais da empresa, ativos de processos organizacionais.

» **Ferramentas e técnicas:** Opinião especializada, análise de alternativas, dados publicados para auxílio a estimativas, estimativa "*bottom-up*", software de gerenciamento de projetos.

» **Saídas:** Requisitos de recursos das atividades, estrutura analítica dos recursos, atualizações dos documentos do projeto.

Estimar as durações das atividades

Segundo o PMBOK (2017, p. 146), é o processo de estimativa do número de períodos de trabalho que serão necessários para terminar as atividades específicas com os recursos estimados. Utiliza informações sobre as atividades do escopo do projeto, tipos de recursos necessários, quantidades estimadas de recursos e calendário dos recursos. A maior parte dos pacotes de software de gerenciamento de projetos para elaboração de cronogramas manipulará essa situação através do uso de um calendário do projeto e calendários alternativos de recursos de trabalho-período que são normalmente identificados pelos recursos que requerem períodos de trabalho específicos.

» **Entradas:** Requisitos de recursos das atividades, estrutura analítica dos recursos, atualizações dos documentos do projeto.

» **Ferramentas e técnicas:** Opinião especializada, estimativa análoga, estimativa paramétrica, estimativas de três pontos, análise de reservas.

» **Saídas:** Estimativas de duração das atividades, atualizações dos documentos do projeto.

Desenvolver o cronograma

Segundo PMBOK (2017, p. 152), é o processo de análise de sequências das atividades, suas durações, recursos necessários e restrições do cronograma visando criar o cronograma do projeto. A entrada das atividades, durações e recursos na ferramenta de elaboração de cronograma gera um cronograma com datas planejadas para completar as atividades do projeto. Determina as datas planejadas de início e de término para as atividades e marcos do projeto. A revisão e a manutenção de um cronograma realista continuam sendo executadas durante todo o projeto à medida que o trabalho progride, o plano de gerenciamento do projeto muda e a natureza dos eventos de riscos evolui.

» **Entradas:** Lista das atividades, atributos das atividades, diagramas de rede do cronograma do projeto, requisitos de recursos das atividades, calendários dos recursos, estimativas de duração das atividades, declaração do escopo do projeto, fatores ambientais da empresa, ativos de processos organizacionais.

» **Ferramentas e técnicas:** Análise de rede do cronograma, método do caminho crítico, método da corrente crítica, nivelamento de recursos, análise de cenário *E-se*, aplicação de antecipações e esperas, compressão de cronograma, ferramenta de elaboração de cronograma.

» **Saídas:** Cronograma do projeto, linha de base do cronograma, dados do cronograma, atualizações dos documentos do projeto.

Controlar o cronograma

Segundo PMBOK (2017, p. 106), controlar o cronograma é o processo de monitoramento do andamento do projeto para atualização do seu progresso e gerenciamento das mudanças feitas na linha de base do cronograma.

O controle do cronograma está relacionado a:

› Determinação da situação atual do cronograma do projeto.

› Influência nos fatores que criam mudanças no cronograma.

› Determinação de que o cronograma do projeto mudou.

› Gerenciamento das mudanças reais conforme ocorrem.

» **Entradas:** Plano de gerenciamento do projeto, cronograma do projeto, informações sobre o desempenho do trabalho, ativos de processos organizacionais.

» **Ferramentas e técnicas:** Análise de desempenho, análise de variação, software de gerenciamento de projetos, nivelamento de recursos, análise de cenário *E-se*, ajuste de antecipações e esperas, compressão de cronograma, ferramenta de elaboração de cronograma.

» **Saídas:** Medições de desempenho do trabalho, atualizações dos ativos de processos organizacionais, solicitações de mudança, atualizações do plano de gerenciamento do projeto, atualizações dos documentos do projeto.

MEDIDAS E MÉTRICAS

UNIDADE III

CAPÍTULO 1

Medidas e métricas

Pressman (2011, p. 538) cita uma definição de medida proposta por Fenton (1991):

> Medição é o processo pelo qual números ou símbolos são anexados aos atributos de entidades no mundo real para defini-los de acordo com regras claramente estabelecidas.

Já o IEEE no seu glossário de terminologias de Engenharia de Software, define "medida quantitativa do grau com o qual um sistema, componente ou processo possui determinado atributo".

Para um arquiteto de softwares e demais membros da equipe, é importante saber qual regra que está sendo aplicada para a definição dessas medidas, pois, o status do projeto, seja de tempo ou custo, estará diretamente ligado ao desempenho de cada membro, sendo avaliado por medidas, métrica e indicadores, através de várias fases do projeto.

Antes de se iniciar qualquer projeto de produto, é necessário formular um conjunto de regras para estabelecer a criação de medidas e métricas apropriadas de acordo com o software projetado, além de mecanismos usados para guardar os dados necessários, ferramentas matemáticas e comunicação com a equipe de software.

As métricas criadas só serão úteis se tiverem sua interpretação clara e for validada pela equipe sobre seus atributos e resultados. Alguns atributos foram propostos ao longo do tempo, porém alguns não muito eficazes. Segundo Pressman (2011, p. 542), os atributos devem ser simples e computáveis, empiricamente e intuitivamente persuasivos, consistentes e objetivos no seu uso das unidades e dimensões, independente da linguagem de programação, e terem um mecanismo efetivo de comunicação com a equipe.

O modelo GQM

A abordagem GQM – *Goal, Question, Metric* baseia-se no pressuposto que, para que haja uma medição, deve haver primeiro a especificação dos objetivos do projeto, depois rastrear as metas para os dados que pretendem se definir e, finalmente, fornecer uma estrutura para interpretar esses dados, ou seja, na prática estabelecer um sistema de medição direcionado às metas, levantar questões para atingir os objetivos, e identificar as métricas que gerem as respostas para as perguntas.

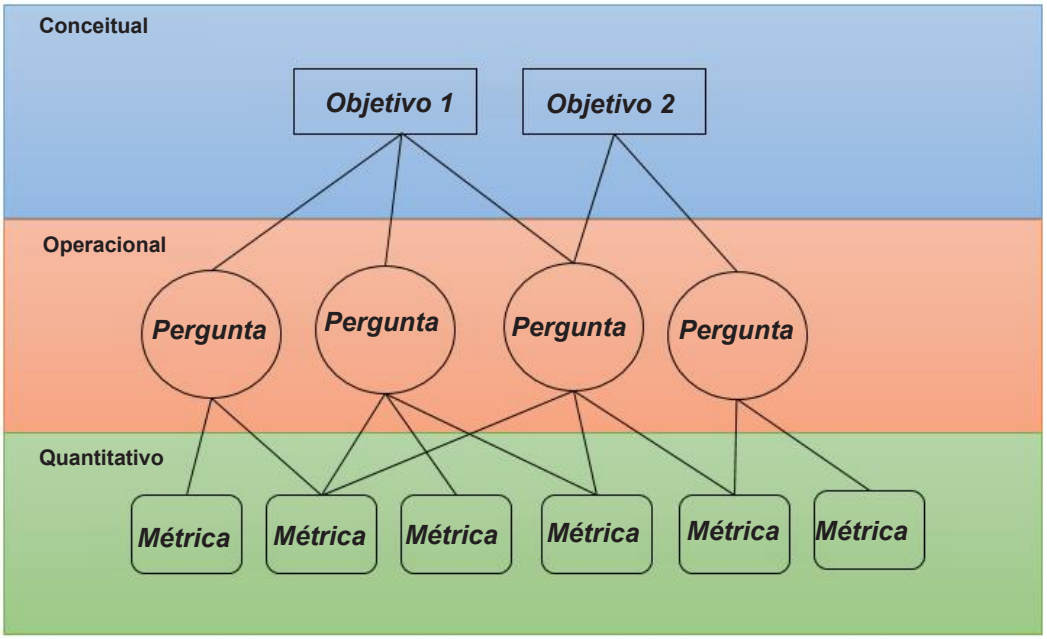
Como resultado, o GQM fornece uma especificação de um sistema de medição hierárquico, visando a um conjunto de perguntas e um conjunto de regras para interpretar os dados da medição. Esse modelo está divido em três níveis:

» **Conceitual (*Goal*):** objetivo das metas que envolvem produtos, processos e recursos.

» **Operacional (*Question*):** conjunto de perguntas usadas para chegar aos objetivos. Elas tentam caracterizar o objetivo da medição em relação a um problema.

» **Quantitativo (*Metric*):** as métricas identificam as medidas necessárias para responder às questões.

Figura 22: Modelo hierárquico GQM



Fonte: Basili (1998, p. 529)

O GQM está dividido em seis etapas:

- Desenvolver um conjunto de metas organizacionais e medições associadas para produtividade e de qualidade.
- Gerar perguntas que definam os objetivos de forma quantificável.
- Especificar medidas necessárias para se obter respostas às questões.
- Desenvolver mecanismos de coletas de dados.
- Colear, validar e analisar os dados em tempo real.
- Analisar o dado para avaliar a conformidade com as metas.

Pressman (2011, p. 541) propõe um *template* para o método GQM:

Analisar {*o nome da atividade ou atributo a ser medido*} **com a finalidade de** {*o objetivo geral da análise*} **com relação a** {*o aspecto da atividade ou atributo considerado*} **do ponto de vista de** {*a pessoa que tem interesse na medição*} **no contexto de** {*o ambiente no qual a medição ocorre*}.

Métricas para o modelo de requisitos

Segundo Pressman (2011, p. 543), como o projeto de software se inicia na criação do modelo de requisitos, obter métricas nessa fase é importante para prever o tamanho do sistema resultante, que pode ser um indicador da complexidade do projeto e quase sempre é um indicador do trabalho cada vez maior de codificação, integração e testes.

Métricas para projeto orientado a objetos

Métricas orientadas a objeto são orientadas pelo tamanho que, segundo Pressman (2011, p. 550), se refere à contagem das entidades orientadas a objetos, como classe e operações, a extensão de uma árvore de herança e funcionalidades da classe.

Orientada à classe

Segundo Pressman (2011, p. 551), a classe é o bloco de código fundamental em um sistema orientado a objetos. Nesse caso, medidas e métricas são muito valiosas quando for necessário avaliar as operações e atributos das superclasses, das classes bases ou filhas, desde o nó até a raiz.

Orientada ao Tamanho – *Class Size* (CS)

O tamanho de uma classe é usado para avaliar a facilidade de compreensão do código pelos programadores. O tamanho pode ser medido pela contagem de todas as linhas de código, o número de declarações, o número de linhas em branco e o número de linhas de comentário.

Orientada a Métodos - *Weighted Methods per Class*

O WMC é uma contagem dos métodos implementados em uma classe ou a soma das complexidades dos métodos. A segunda medição é difícil de implementar, já que nem todos os métodos são passíveis de avaliação hierarquia de classes devido à herança. O número de métodos e a complexidade dos métodos envolvido é uma estimativa de quanto tempo e esforço são necessários para desenvolver e manter a classe. Quanto maior o número de métodos em uma classe, maior o impacto potencial sobre as subclasses, pois os filhos herdam todos os métodos definidos na classe pai.

Orientada a Número de Filhos – NOC (Number of children)

O número de filhos é o número de subclasses imediatas subordinadas a uma classe em sua hierarquia. É um indicador da influência potencial que uma classe pode ter no projeto e no sistema. Quanto maior o número de filhos, maior a probabilidade de abstração do pai e pode ser um caso de uso indevido de subclassificação. Mas quanto maior o número de filhos, maior a reutilização, uma vez que a herança é uma forma de reutilização. Se uma classe tem muitos filhos, pode exigir mais testes dos métodos dessa classe, aumentando assim o tempo de teste.

Orientada à Coesão - *Lack of Cohesion* (LCOM)

A falta de coesão (LCOM) mede a dissimilaridade dos métodos em uma classe por instância variável ou atributos. Um módulo altamente coeso deve ficar sozinho; alta coesão indica boa subdivisão de classe. Alta coesão implica simplicidade e alta reutilização, indica boa subdivisão de classe. Falta de coesão ou baixa coesão aumenta a complexidade, aumentando assim a probabilidade de erros durante o processo de desenvolvimento. As classes com baixa coesão poderiam ser subdivididas em duas ou mais subclasses com maior coesão.

CAPÍTULO 2
Ponto de função

Uma métrica efetiva que pode ser utilizada nessa etapa é ponto de função. Ela ajuda a medir uma funcionalidade fornecida por um sistema. Os pontos de funções são baseados em medidas calculáveis do domínio da informação do software e avaliações qualitativas da complexidade das funcionalidades. A Análise do Ponto de função fornece um método padronizado para dimensionar funcionalmente o produto de trabalho de software. Este produto de trabalho é a saída de novos projetos de desenvolvimento e melhoria de software para versões subsequentes. É o software que é realocado para o aplicativo de produção na implementação do projeto. Ele mede a funcionalidade do ponto de vista do usuário, ou seja, com base no que o usuário solicita e recebe em retorno. Seus conceitos básicos, históricos e atualidades podem ser acessadas em https://www.ifpug.org.

A Análise de Pontos de Função foi inicialmente desenvolvida por Allan J. Albercht em 1979 na IBM e foi modificada pelo *International Users Point Users Group* (IFPUG). O FPA fornece um número adimensional definido em pontos de função que consideramos uma medida relativa efetiva do valor da função entregue ao nosso cliente.

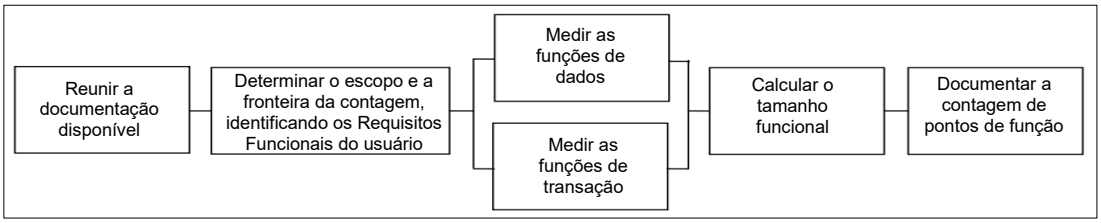
Segundo Pressman (2011, p. 543) os valores do domínio da informação sã definidos da seguinte maneira:

- **Número de entradas externas (EE):** processa dados ou informações de controle que vêm de fora do limite do aplicativo.
- **Número de saídas externas (SE):** é um processo que gera dados ou informações de controle enviadas fora do limite do aplicativo.
- **Número de consultas externas (CE):** é um processo composto de uma combinação de entrada-saída que resulta na recuperação de dados.
- **Número de arquivos lógicos internos (ALI):** Um grupo identificável de dados logicamente relacionados ou informações de controle mantidas dentro do limite do aplicativo.
- **Número de arquivos de interfaces externos (AIE):** Um grupo de dados logicamente relacionados, referentes ao software, mas mantidos dentro dos limites de outro software.

No processo de medição de ponto de função, segundo o IFPUG (2010, p. 9), para se conduzir uma contagem de pontos de função, devem ser executadas as seguintes atividades, a fim de identificar e classificar os componentes funcionais básicos (ALI, AIE, EE, SE, CE):

1. Reunir a documentação disponível.
2. Determinar o escopo e a fronteira da contagem, identificando os Requisitos Funcionais do usuário.
3. Medir as funções de dados.
4. Medir as funções de transação.
5. Calcular o tamanho funcional.
6. Documentar a contagem de pontos de função.
7. Reportar o resultado da contagem de pontos de função.

Figura 23: Diagrama de procedimento para contagem de pontos de função



Fonte: IFPUG (2010, p. 66)

Reunir a documentação disponível

Segundo IFPUG (2010, p. 9), a documentação de suporte a uma contagem de pontos de função deve descrever a funcionalidade entregue pelo software ou a funcionalidade impactada pelo projeto de software medido. Ela pode incluir requisitos, modelos de dados/objetos, diagramas de classes, diagramas de fluxos de dados, casos de uso, descrições de procedimentos, formatos de relatórios, manuais de usuário e outros artefatos do desenvolvimento de software.

Determinar o escopo e fronteira da contagem, identificando os Requisitos Funcionais do Usuário

Segundo IFPUG (2010, p. 10), para executar essa etapa conforme especificações, será necessário executar as seguintes atividades:

- Identificar o propósito da contagem.
- Identificar o tipo de contagem, com base no propósito, como um dos seguintes:
  - uma contagem de pontos de função de projeto de desenvolvimento;
  - uma contagem de pontos de função de aplicação;
  - uma contagem de pontos de função de um projeto de melhoria.
- Determinar o escopo da contagem, com base no propósito e tipo de contagem.
- Determinar a fronteira de cada aplicação contida no escopo da contagem com base na visão do usuário e não em considerações técnicas.
- Os requisitos do usuário podem incluir uma mistura de requisitos funcionais e não funcionais; identificar quais requisitos são funcionais e excluir os não funcionais.

Medir funções de dados

Essa etapa está relacionada aos requisitos do software referentes a armazenar e/ou referenciar dados. Segundo IFPUG (2010, p. 11), devem ser executadas as seguintes funções:

Identificar e agrupar todos os dados lógicos em funções de dados

Para identificar as funções de dados, devem ser executadas as seguintes atividades:

- Identificar todos os dados ou informações de controle logicamente relacionados e reconhecidos pelo usuário, dentro do escopo da contagem.
- Excluir entidades que não sejam mantidas por nenhuma aplicação.
- Agrupar entidades que sejam entidades dependentes, NOTA 2. Entidades independentes são consideradas como grupos lógicos de dados distintos.
- Excluir as entidades abaixo, denominadas dados de código:
  - entidade de dados de substituição, que contém um código e um nome ou descrição explicativos;

- entidade de ocorrência única, que contém um ou mais atributos que raramente ou nunca mudam;
- entidade que contém dados basicamente estáticos, ou que muito raramente mudam;
- entidade de valores *default*, que contém valores para atributos populares;
- entidade de valores válidos, que contém valores disponíveis para seleção ou validação;
- entidade que contém uma faixa de dados para validação.

» Excluir entidades que não contenham atributos requeridos pelo usuário.

» Remover entidades associativas que contenham atributos adicionais não requeridos pelo usuário e entidades associativas que contenham apenas chaves estrangeiras; agrupar os atributos referentes a chaves estrangeiras com as entidades primárias.

Classificar cada função de dados como um ALI ou AIE

Uma função de dados deve ser classificada como:

» Arquivo Lógico Interno (ALI), se for mantida pela aplicação medida.

» Arquivo de Interface Externa (AIE) se for referenciada, mas não mantida, pela aplicação medida, e identificada como um ALI em uma ou mais aplicações.

Contar os DERs para cada função de dados

A fim de contar os DERs - Dados Elementares Referenciados (tipos de dado elementares) correspondentes a uma função de dados, devem ser executadas as seguintes atividades:

a. Contar um DER para cada atributo único, reconhecido pelo usuário e não repetido, mantido na função de dados ou recuperado da mesma por meio da execução de todos os processos elementares pertinentes ao escopo da contagem.

b. Quando duas ou mais aplicações mantiverem e/ou referenciarem a mesma função de dados, contar apenas os DERs utilizados pela aplicação medida.

c. Contar um DER para cada atributo requerido pelo usuário para estabelecer um relacionamento com outra função de dados.

d. Revisar os atributos relacionados a fim de determinar se eles devem ser agrupados e contados como um único DER, ou como vários DERs. O agrupamento irá depender de como os processos elementares utilizam os atributos dentro da aplicação.

Contar os RLRs para cada função de dados

A fim de contar RLRs - Registros Lógicos Referenciados (tipos de arquivo referenciados) para uma função de dados, devem ser executadas as seguintes atividades:

» Contar um RLR para cada função de dados (i.e., por *default*, cada função de dados possui um subgrupo de DERs que é contado como um RLR).

» Contar um RLR adicional para cada um dos seguintes subgrupos lógicos de DERs (dentro da função de dados) que contenham mais de um DER:

› entidade associativa com atributos não chave;

› subtipo (sem ser o primeiro subtipo);

› entidade atributiva, em um relacionamento que não seja 1-1 obrigatório.

Se um modelo de dados não estiver disponível, procurar grupos de dados repetidos para identificar RLRs.

Determinar a complexidade funcional de cada função de dados

A complexidade funcional de cada função de dados deve ser determinada utilizando-se o número de DERs e RLRs, em conformidade com a tabela:

Tabela 4: Complexidade das funções de dados

1 - 19		DERs		
		20 - 50	> 50	
RLRs	1	Baixa	Baixa	Média
	2 - 5	Baixa	Média	Alta
	> 5	Média	Alta	Alta

Fonte: IFPUG (2010, p. 14)

Determinar o tamanho funcional de cada função de dados

O tamanho funcional de cada função de dados deve ser determinado utilizando-se o tipo e a complexidade funcional, de acordo com a tabela:

Tabela 5: Tamanho das funções de dados

ALI		Tipo	
		AIE	
Complexidade funcional	Baixa	7	5
	Média	10	7
	Alta	15	10

Fonte: IFPUG (2010, p. 14)

Medir funções de transação

A funcionalidade de transação satisfaz os Requisitos Funcionais do Usuário que processam dados. Toda a funcionalidade de transação dentro do escopo da contagem deve ser avaliada, a fim de identificar cada processo elementar único. Para medir funções de transação, as seguintes atividades devem ser executadas:

» Identificar processos elementares.

» Contar ALRs para cada função de transação.

» Contar DERs para cada função de transação.

» Medir a funcionalidade correspondente a melhorias.

Calcular o tamanho funcional

O objetivo e escopo da contagem deverão ser considerados na seleção e utilização da fórmula apropriada para calcular o tamanho funcional.

Fórmula para o tamanho funcional de um projeto de desenvolvimento:

$$DFP = ADD + CFP$$

» **DFP** é a contagem de pontos de função do projeto de desenvolvimento;

» **ADD** é o tamanho das funções a serem entregues ao usuário pelo projeto de desenvolvimento;

» **CFP** é o tamanho da funcionalidade de conversão.

Fórmula para ser medido após o projeto de desenvolvimento:

$$AFP = ADD$$

» **AFP** é a contagem de pontos de função da aplicação;

» **ADD** é o tamanho das funções a serem entregues ao usuário pelo projeto de desenvolvimento, ou a funcionalidade existente no momento da contagem da aplicação.

Fórmula para medir o tamanho funcional de um projeto de melhoria:

$$EFP = ADD + CHGA + CFP + DEL$$

» **EFP** é a contagem de pontos de função do projeto de melhoria;

» **ADD** é o tamanho das funções incluídas pelo projeto de melhoria;

» **CHGA** é o tamanho das funções alteradas pelo projeto de melhoria;

» **CFP** é o tamanho da funcionalidade de conversão;

» **DEL** é o tamanho das funções excluídas pelo projeto de melhoria.

Fórmula para medir o tamanho funcional após um projeto de melhoria:

$$AFPA = (AFPB + ADD + CHGA) - (CHGB + DEL)$$

» **AFPA** é a contagem de pontos de função da aplicação após o projeto de melhoria;

» **AFPB** é a contagem de pontos de função da aplicação antes do projeto de melhoria;

» **ADD** é o tamanho das funções incluídas pelo projeto de melhoria;

» **CHGA** é o tamanho das funções alteradas pelo projeto de melhoria;

» **CHGB** é o tamanho das funções alteradas pelo projeto de melhoria;

» **DEL** é o tamanho das funções excluídas pelo projeto de melhoria.

Documentar a contagem de pontos de função

» A contagem de pontos de função deve ser documentada como segue:

» o propósito e o tipo da contagem;

» o escopo da contagem e a fronteira da aplicação;

» a data da contagem;

» uma lista de todas as funções de dados e de transação, incluindo o respectivo tipo e complexidade, bem como o número de pontos de função atribuído a cada uma;

» o resultado da contagem;

» quaisquer suposições feitas e questões resolvidas.

A documentação da contagem de pontos de função também pode incluir o seguinte:

» a identificação da documentação de origem na qual a contagem foi baseada;

» a identificação dos participantes, seus papéis e qualificações;

» para cada função de dados, o número de DERs e RLRs;

» para cada função de transação, o número de DERs e de ALRs;

» uma referência cruzada de todas as funções de dados para as funções de transação;

» uma referência cruzada de todas as funções de dados para as abstrações relacionadas na documentação de origem;

» uma referência cruzada de todas as funções de transação para as abstrações relacionadas na documentação de origem.

Reportar o resultado da contagem de pontos de função

A prática de reportar consistentemente os resultados das contagens de pontos de função permitirá que os leitores identifiquem o padrão com o qual elas mantêm conformidade.

Os resultados que mantenham conformidade com este Padrão Internacional deverão ser reportados como segue: S FP (IFPUG–IS)

S é o resultado da contagem de pontos de função;

FP é a unidade de tamanho do método FSM do IFPUG;

IS é este Padrão Internacional (ISO/IEC 20926:200x).

EXEMPLO: 250 FP (IFPUG-ISO/IEC 20926:200x).

CAPÍTULO 3
Estimativas

Segundo Pressman (2011, p. 604), o gerenciamento de projeto de software começa com uma série de atividades chamadas de planejamento de projeto. Nessa etapa, a equipe de software deverá fazer uma estimativa do trabalho, os recursos e os tempos necessários para a conclusão do projeto.

Certas características podem trazer benefícios às estimativas do projeto de software, como a experiências dos envolvidos, as métricas etc. e também existem alguns problemas inerentes ao projeto que podem dificultar os cumprimentos dos prazos e custos, como as incertezas relacionadas à complexidade e tamanho do projeto por profissionais que não tiveram contatos com experiências passadas, a falta de informações históricas, etc. Segundo Pressman (2011, p. 604), os riscos das estimativas são medidos pelo grau de incerteza nas estimativas quantitativas estabelecidas para recursos, custo e cronograma. Se o escopo do projeto for mal-entendido, a incerteza e o risco das estimativas tornam-se altos.

Existem algumas tarefas relacionadas ao planejamento do projeto:

- Estabeleça o escopo - Unidade II, Capítulo 5.
- Determine a viabilidade.
- Analise os riscos – Unidade II, Capítulo 7.
- Defina os recursos necessários – Unidades I e II:
  - pessoas – Capítulo 6;
  - reúso de software – Capítulo 2.
- Estime o custo e a mão de obra.
- Desenvolva o cronograma – Unidade II, Capítulo 4.

Decomposição

Na tentativa de estimar um projeto de software, na maioria das vezes, o problema a ser resolvido como um todo é muito complexo e imprevisível. Por essa razão, utiliza-se a

decomposição de um "grande" problema em problemas menores e mais fáceis de serem resolvidos.

Entretanto, para sabermos a dimensão do projeto, ou seja, seu tamanho, devemos ter uma precisão baseada em quatro níveis, como sugere Pressman (2011, p. 610):

» o grau de como foi estimado adequadamente o tamanho do projeto;

» a habilidade para traduzir a estimativa em recursos humanos, tempo e financeiro;

» o grau com que o plano do projeto reflete as habilidades da equipe;

» a estabilidade dos requisitos do produto e o ambiente que suporta o projeto;

Estimativa LOC e FP

Segundo Pressman (2011, p. 10), no contexto do planejamento do projeto, o tamanho pode ter uma abordagem direta, ou seja, por tamanho de linhas de código (LOC), ou indireta, por pontos de função (FP).

Tanto a estima LOC quanto a FP iniciam com uma definição delimitada do escopo do software e daí tentam decompor e definir em funções de problemas que podem ser estimados individualmente. Métricas de produtividade são aplicadas à variável de estimativa apropriada (LOC ou FP) e, assim, se obtém o custo ou esforço para a função.

A diferença entre elas, no entanto, é a diferença em nível de detalhe requerido para a decomposição e no alvo do particionamento. Quando é usada a LOC como variável de estimativa, a decomposição é essencial e muitas vezes adotada com níveis consideráveis de detalhes. Quanto maior o particionamento, maior a probabilidade de serem desenvolvidas estimativas LOC precisas (PRESSMAN, 2011, p.611).

Tabela 6: Tabela de estimativas LOC

Função	LOC estimado
Interface de usuário e recurso de controle	2.300
Cadastro de pedidos de vendas	5.300
Consulta de informações financeiras do cliente	6.800
Consulta de informações de produtos no estoque	3.350
Gerenciamento de base de dados	4.950
Emissão de Nota Fiscal	2.100
Recebimento de Nota Fiscal	8.400
Linhas de código estimadas	33.200

Fonte: Elaborada pelo autor, baseada em Pressman (2011, p. 613)

Na estimativa FP, em vez de focalizar a função, é utilizado o domínio de informação, como entradas, saídas, consultas, interfaces externas etc. As estimativas resultantes podem ser usadas para derivar um valor de FP que pode ser relacionado a dados anteriores e usados para gerar uma estimativa (PRESSMAN, 2011, p.611).

Tabela 7. Estimando valores do domínio de informações FP.

Valor do domínio	Saídas	Estimativa	Consulta	Estimativa computada	Peso	FB computado
Número de entradas externas	20	24	30	24	4	97
Número de saídas externas	12	15	22	16	5	78
Número de consultas externas	16	22	28	22	5	88
Número de arquivos lógicos	4	4	5	4	10	42
Número de arquivos de interfaces	2	2	3	2	7	15
Contagem total						320

Fonte: Pressman (2011, p. 614)

COCOMO - *Constructive Cost Model*

É um modelo de estimativa de custo procedural para projetos de software e frequentemente usado como um processo de prever com segurança os vários parâmetros associados à realização de um projeto, como tamanho, esforço, custo, tempo e qualidade. Foi proposto por Barry Boehm em 1970 e baseia-se no estudo de 63 projetos, o que o torna um dos modelos mais bem documentados.

Os principais parâmetros que definem a qualidade de qualquer produto de software, que também são resultado do Cocomo, são principalmente o Esforço e o Planejamento:

- **Esforço**: quantidade de mão de obra que será necessária para concluir uma tarefa. É medido em unidades de pessoa-meses.
- **Programação**: significa simplesmente a quantidade de tempo necessária para a conclusão do trabalho, que é, naturalmente, proporcional ao esforço colocado. É medido em unidades de tempo, como semanas, meses.

O modelo COCOMO II foi desenvolvido a partir de um modelo COCOMO anterior de estimativas de custo, o qual foi em grande parte baseado no desenvolvimento do código original (BOEHM, 1981; BOEHM e ROYCE, 1989).

Segundo Sommerville (2011, p. 444), o modelo COCOMO II leva em consideração abordagens mais modernas para o desenvolvimento de software, tais como desenvolvimento rápido usando linguagens dinâmicas, desenvolvimento por composição de componentes e uso de programação de banco de dados, suporta o modelo

de desenvolvimento em espiral e incorpora submodelos que produzem estimativas altamente detalhadas:

» o modelo de composição de aplicação;

» o modelo de projeto preliminar;

» o modelo de reúso;

» o modelo de pós-arquitetura.

Estimativa para projetos orientados a objetos

Pressman (2011, p. 622) sugere para projetos orientados a objetos, a seguinte abordagem:

» Desenvolva estimativas usando decomposição de esforço, FP e qualquer outro método convencional.

» Usando o modelo de requisitos, desenvolva e determine uma contagem. Atualize sempre que o número de casos de uso mudar.

» Determine o número de classes-chaves de acordo com os requisitos.

» Classifique o tipo de interface para a aplicação, desenvolva um multiplicador para classes de apoio e multiplique o número de classe-chave pelo multiplicador para obter uma estimativa do número de classes de apoio.

» Multiplique o número total de classes-chave e apoio, pelo número médio de unidade de trabalho por classe.

CAPÍTULO 4
Definição de Custos

Segundo (IFPUG, p. 168), estimar os custos é o processo de desenvolvimento de uma estimativa dos recursos monetários necessários para executar as atividades do projeto.

Um bom planejamento da gestão de custos seria elaborar, primeiramente, uma regra de medição de desempenho, utilizando a EAP e os pontos onde os controles serão feitos, estabelecer técnicas de medição de valor agregado, criar as equações de cálculos do valor agregado para determinar as previsões, retorno do investimento, fluxo de caixa descontado e análise de recuperação de investimento.

Estimativas de Custos

Estimar custos é um processo que visa fazer uma previsão dos recursos necessários ao projeto e convertê-los em valores monetários.

Todos os custos são estimados para todos os recursos que serão cobrados do projeto. Isso não se limita a mão de obra, materiais, equipamentos, serviços e instalações, assim como categorias especiais como provisão para inflação ou custos de contingências. Uma estimativa de custo é uma avaliação quantitativa dos custos prováveis dos recursos necessários para completar a atividade. Porém, conforme o projeto vai ganhando maturidade, as estimativas precisam ser ajustadas, pois, assim serão refletidos com mais exatidão os detalhes. Segundo (IFPUG, p. 168), a estimativa de custos é um processo interativo de fase para fase. Por exemplo, um projeto na fase inicial poderia ter uma ordem de grandeza estimada na faixa de ±50%. Mais tarde, conforme mais informações são conhecidas, as estimativas podem estreitar para uma faixa de ±10%. Estimativas de custos são geralmente expressas em unidades de alguma moeda.

- **Entradas**: Linha de base do escopo, cronograma do projeto, plano de recursos humanos, registro dos riscos, fatores ambientais da empresa, ativos de processos organizacionais.
- **Ferramentas e Técnicas**: Opinião especializada, estimativa análoga, estimativa paramétrica, estimativa "*bottom-up*", estimativas de três pontos, análise de reservas, custo da qualidade, software de estimativa de gerenciamento de projetos, análise de proposta de fornecedor.

» **Saídas**: Estimativas de custos das atividades, bases das estimativas, atualizações dos documentos do projeto.

Com base no escopo, documento que fornecerá todas as informações necessárias, será possível definir uma premissa se as estimativas se limitarão somente aos custos diretos ou incluirão os indiretos também. Os indiretos são aqueles que não podem ser diretamente associados a um projeto específico e, portanto, seus custos são rateados entre vários projetos. Algumas informações adicionais que podem gerar impacto na estimativa de custo do projeto são: saúde, segurança, proteção, desempenho, ambiente, seguro, direitos de propriedades intelectuais, licenças e autorizações.

Outro fator importante que impactará no custo do projeto é o cronograma. As estimativas de quantidade de tempo, quantidade de recursos e suas durações, onde os recursos são aplicados por unidade de tempo para duração da atividade. O risco do projeto deverá ser revisto a cada etapa, pois, quando um projeto se depara com algum problema, o custo de curto prazo normalmente aumenta e acarretará atrasos.

Alguns ativos organizacionais poderão influenciar o processo de estimar os custos. São eles: políticas de estimativas de custos, modelo de estimativas, informações históricas e lições aprendidas.

Segundo (IFPUG, p. 172), a precisão de ajuste das estimativas pode ser aperfeiçoada usando a PERT (Técnica de Revisão e Avaliação de Programa):

» **Mais provável**: custo baseado em esforço de avaliação realista.

» **Otimista**: custo baseado na análise de melhor cenário.

» **Pessimista**: custo baseado na análise de pior cenário.

Determinar o orçamento

Segundo (IFPUG, p. 176), determinar o orçamento é o processo de agregação dos custos estimados de atividades individuais ou pacotes de trabalho para estabelecer uma linha de base dos custos autorizada. Essa linha de base inclui todos os orçamentos autorizados, mas exclui as reservas de gerenciamento.

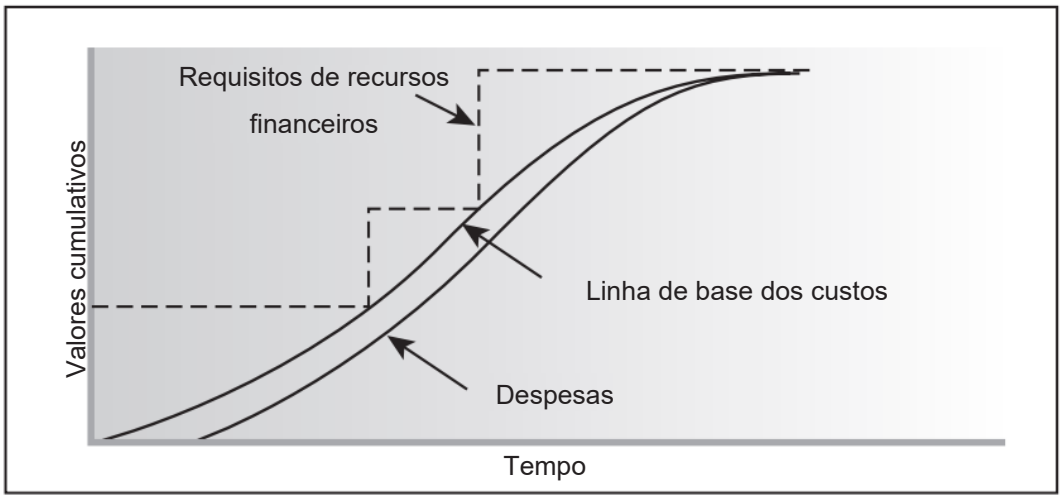
Entradas: Estimativas de custos das atividades, bases das estimativas, linha de base do escopo, cronograma do projeto, calendários dos recursos, contratos, ativos de processos organizacionais.

Ferramentas e Técnicas: Agregação de custos, análise de reservas, opinião especializada, relações históricas, reconciliação dos limites de recursos financeiros.

Saídas: Linha de base do desempenho de custos, requisitos de recursos financeiros do projeto, atualizações dos documentos do projeto.

A linha de base de desempenho é um orçamento no final, sincronizado no tempo, para medir, controlar o desempenho de custo geral do projeto.

Figura 24. Linha de base de custos, gastos e requisitos de recursos financeiros



Fonte: IFPUG (p. 178)

Ao final, o controle atualizará os seguintes documentos do projeto: registro dos riscos, estimativas de custos e cronograma.

Controlar os custos

Controlar os custos é o processo de monitoramento do progresso do projeto para atualização do seu orçamento e gerenciamento das mudanças feitas na linha de base dos custos.

Entradas: Estimativas de custos das atividades, bases das estimativas, linha de base do escopo, cronograma do projeto, calendários dos recursos, contratos, ativos de processos organizacionais.

Ferramentas e Técnicas: Agregação de custos, análise de reservas, opinião especializada, relações históricas, reconciliação dos limites de recursos financeiros.

Saídas: Linha de base do desempenho de custos, requisitos de recursos financeiros do projeto, atualizações dos documentos do projeto.

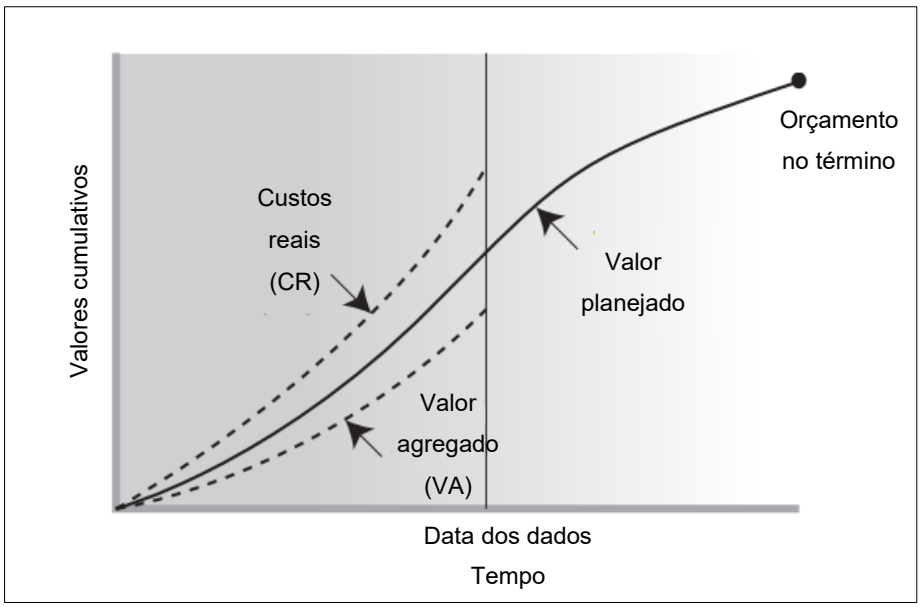
As ferramentas e técnicas para o controle de custos são conceitos muito importantes e muito precisos. Dentre elas, podemos citar:

- GVA (Gerenciamento do Valor Agregado)
- VP (Valor Planejado)
- VA (Valor Agregado)
- CR (Custo Real)

Variações a partir da linha de base aprovada também serão monitoradas:

- Variação de Prazos
  - Equação: VPR = VA – VP
- Variação de Custos
  - Equação: VC = VA – CR

Figura 25: Valor planejado, valor agregado e custos reais



Fonte: IFPUG (p. 183)

CODIFICAÇÃO E IMPLEMENTAÇÃO

UNIDADE IV

CAPÍTULO 1

Características das Linguagens de Programação

Após todos os processos de projetos e engenharia, finalmente se chega a seu objetivo: transformar toda representação do software para algo em que os sistemas computacionais possam entender. A esse processo damos o nome de Codificação.

Segundo Pressman (1998, p.682), o processo de codificação inicia-se depois que um projeto de detalhes foi definido, revisto e modificado, se necessário. Teoricamente, a geração do código-fonte a partir da especificação deve ser direta e, sua facilidade, de conversão do projeto em código oferece uma indicação de quão estreita a linguagem representará o projeto.

Existem algumas etapas às quais podemos nos referir como um processo de tradução. Isso dá pelo fato de extrairmos lógicas de linguagens simbólicas, como UML, passarmos para uma linguagem de programação e em seguida, transformar em algo que tenha suas funcionalidades expressas através de processamentos. Essa última etapa, algumas tecnologias tratam como processo de compilação. Entretanto, veremos seu verdadeiro significado.

As primeiras linguagens de programação surgiram em meados da década de 1940, foram linguagens de máquinas, ou seja, linguagens que eram desenvolvidas para atuar diretamente próximo à eletrônica e eram denominadas linguagens *assembly* (montagem), também conhecida como linguagens de baixo nível. Durante esse crescimento, muitas linguagens foram desenvolvidas, muitos dialetos e semânticas foram criadas e os propósitos já foram sendo distribuídos, assim como o fracasso de algumas e a influência de outras.

Em seguida foi a vez das linguagens de alto nível, que são linguagens cuja sintaxe e semântica estão mais próximas das linguagens naturais, ou seja, mais próximas dos seres humanos.

Tucker (2010, p. 1) faz uma comparação entre linguagem de programação e linguagens naturais:

> Da mesma forma que as nossas linguagens naturais, as linguagens de programação facilitam a expressão e a comunicação de ideias entre pessoas. Entretanto, linguagens de programação diferem das linguagens naturais de duas maneiras importantes. Em primeiro lugar, linguagens de programação também permitem a comunicação de ideias entre pessoas e computadores. Em segundo lugar, as linguagens de programação possuem um domínio de expressão mais reduzido do que o das linguagens naturais. Isso quer dizer que elas facilitam apenas a comunicação de ideias *computacionais*.

Veja a comparação entre um código feito em Assembly e um código feito em Python. Ambos imprimirão a mensagem "*Hello World*" na tela:

Tabela 8: Diferença entre linguagem de baixo e alto nível

Código em Assembly	Código em Python

section .data
msg db 'Hello World', 0AH
len equ $-msg
section .text
global _start
_start: mov edx, len
mov ecx, msg
mov ebx, 1
mov eax, 4
int 80h
mov ebx, 0
mov eax, 1
int 80h
	print("Hello World")

Fonte: Elaboração própria do autor (2019)

Pressman (1985, p 677) acrescenta que as linguagens funcionam como veículos de comunicação entre os seres humanos e computadores. O processo de codificação é uma

atividade humana e que, como tal, as características psicológicas exercem influência sobre a qualidade da comunicação.

Por agir sobre influência psicológica, podemos citar alguns fatores como facilidade de uso, simplicidade de aprendizagem, confiabilidade, aceitação de mercado, portabilidade etc. O fator psicológico vai fazer com que os projetistas encaixem as abordagens dos requisitos do sistema nas restrições impostas por uma determinada linguagem de programação. Segundo Pressman (1985, p. 678), uma vez que os fatores humanos são criticamente importantes no projeto de uma linguagem de programação, as características psicológicas de uma linguagem têm forte relação com o sucesso da conversão do projeto e a implementação em códigos.

Critérios de avaliação

Sebesta (2011, p. 20) cita vários motivos para o estudo de conceito linguagem de programação, como capacidade aumentada para expressar ideias e aprender novas linguagens, embasamento na escolha da linguagem certa, entendimento da importância da implementação, melhor uso das linguagens já conhecidas e avanço geral da computação.

Além de entender os vários conceitos das linguagens de programação, é necessário decidir o critério de avaliação dos recursos antes de escolher qual linguagem utilizar. Esses recursos estão ligados ao processo de desenvolvimento, manutenção e tempo de vida de um software.

Sebesta (2011, p. 26) sugeriu uma tabela com algumas características mais importantes, que podem influenciar na escolha de uma determinada linguagem:

Tabela 9: Critérios de avaliação de linguagens e as características que os afetam

	Critérios		
Característica	**Legibilidade**	**Facilidade de Escrita**	**Confiabilidade**
Simplicidade	*	*	*
Ortogonalidade	*	*	*
Tipos de dados	*	*	*
Projeto de sintaxe	*	*	*
Suporte para abstração		*	*
Expressividade		*	*
Verificação de Tipos			*
Tratamento de exceções			*
Apelidos restritos			*

Fonte: Sebesta (2011, p. 26)

Legibilidade

Facilidade como o código fonte do programa pode ser lido e entendido. Segundo Sebesta (2011, p. 27), devido aos custos de desenvolvimento e manutenção em códigos feitos em linguagens de baixo nível, ocorreu uma transição na qual não se programava mais orientado para as máquinas e sim para pessoas.

» **Simplicidade**

Uma linguagem com multiplicidade de recursos e muitas construções é mais difícil de aprender. Exemplo de incremento na linguagem Java:

```
count = count + 1
count += 1
count++
++count
```

» **Ortogonalidade**

Segundo Sebesta (2011, p. 28):

> Ortogonalidade significa que um conjunto relativamente pequeno de construções primitivas pode ser combinado a um número relativamente pequeno de formas para construir as estruturas de controle e de dados da linguagem. A ortogonalidade é fortemente relacionada à simplicidade: quanto mais ortogonal o projeto de uma linguagem, menor é o número necessário de exceções às regras da linguagem. Menos exceções significam um maior grau de regularidade no projeto, o que torna a linguagem mais fácil de aprender, ler e entender. Essa liberdade de combinações permite construções extremamente complexas.

Truck (2011, p. 17) exemplifica ortogonalidade:

> Considere a passagem de argumentos em uma chamada de função. Uma linguagem integralmente ortogonal permite que qualquer tipo de objeto, incluindo uma definição de função, seja passado como argumento. Outras linguagens restringem os tipos de objetos que podem ser passados em uma chamada.

» **Tipos de dados**

É muito importante a presença de mecanismos adequados para definição de tipos de dados. Sebesta (2011, p. 31) cita um exemplo da falta do tipo de dado booleano e a tentativa, nada clara, de se tentar adaptar essa implementação.

```
timeOut = 1     // Sem a presença do tipo booleano
timeOut = true  // Com a presença do tipo booleano
```

» **Projeto de sintaxe**

Sintaxe de como são formados os identificadores ou variáveis em uma determinada linguagem. Normalmente se restringe ao tamanho, palavras reservadas, formas e significados. Quanto mais fácil de entender um identificador, mais legível será o código. Exemplo:

```
id_aluno        // código ou idade do aluno?
idade_do_aluno
codigoDoAluno
```

Facilidade de escrita

Podemos medir a facilidade de escrita quando analisamos o modo de se escrever um programa para um determinado domínio. Muitas características da facilidade da legibilidade e da escrita são as mesmas, pois partem do princípio que o programado releia o código várias vezes quanto necessário.

A facilidade de escrita deve levar em conta o contexto da aplicação, ou seja, não podemos comparar o Visual Basic e o C para criar interfaces gráficas ao usuário ou comparar uma escrita de um programa para o Sistema Operacional entre essas duas linguagens, na qual o C levará muita vantagem.

» **Suporte à abstração**

Segundo Sebesta (2011, p. 33):

> Na linguagem de programação o programador poderá definir e usar estruturas, dados ou operações complicadas de forma a permitir que muitos dos detalhes sejam ignorados. Um exemplo simples da abstração de processos é o uso de um subprograma para implementar um algoritmo de ordenação necessário diversas vezes em um programa.

Confiabilidade

» **Verificação de tipos**

A verificação de tipos é a execução de testes para detectar erros de tipos em estrutura de dados e identificadores, tanto por parte do compilador quanto durante a execução de um programa. Essa verificação é um fator muito importante, pois garante que os dados se mantenham íntegros. A verificação enquanto o programa está sendo executado é mais cara quando está sendo feita a sua compilação (SEBESTA, 2011, p.35).

» **Tratamento de exceções**

É a possibilidade de um programa capturar erros, em tempo de execução, e tomar alguns caminhos criados pelo programador. Esses caminhos podem ser medidos de desvios ou de interrupções. Essas interrupções podem ser tratadas pelo fato, por exemplo, de o programa não conseguir abrir um arquivo de texto, ou não encontrar o caminho do banco de dados na rede ou tentar acessar um objeto que não está no endereço indicado.

» **Utilização de apelidos**

Em uma definição bastante informal, apelidos são permitidos quando é possível ter um ou mais nomes para acessar a mesma célula de memória. Atualmente, é amplamente aceito que o uso de apelidos é um recurso perigoso em uma linguagem de programação. O programador deve sempre lembrar que trocar o valor apontado por um dos dois ponteiros modifica o valor referenciado pelo outro. Em algumas linguagens, apelidos são usados para resolver deficiências nos recursos de abstração de dados. Outras restringem o uso de apelidos para aumentar sua confiabilidade (SEBESTA 2011, p. 36).

» **Legibilidade e facilidade de escrita**

Tanto a legibilidade quanto a facilidade de escrita influenciam a confiabilidade. Um programa escrito em uma linguagem que não oferece maneiras naturais para expressar os algoritmos requeridos irá necessariamente usar abordagens não naturais, menos prováveis de serem corretas em todas as situações possíveis. Quanto mais fácil é escrever um programa, mais provavelmente ele estará correto. A legibilidade afeta a confiabilidade tanto nas fases de escrita quanto nas de manutenção do ciclo de vida. Programas difíceis de ler são também difíceis de escrever e modificar (SEBESTA 2011, p. 36).

CAPÍTULO 2
Aspectos Fundamentais

Tipos de dados

Os tipos de dados são valores sobre quais é possível efetuar operações. Em toda linguagem de programação existem tipos básicos em comum, por exemplo, inteiro, reais, booleanos e caracteres.

Os tipos de dados estão relacionados ao tamanho alocado de memória e a maioria tem um limite estabelecido. Entretanto, operações podem gerar resultados que excedem os limites alocados em memória ocorrendo erros de tipos.

Erros de tipos

Segundo Trucker (2011, p. 103), um *erro de tipo* é qualquer erro que surja porque uma operação é tentada sobre um tipo de dado para o qual ela não está definida. O *sistema de tipo* pode fornecer uma base para a detecção precoce (em tempo de compilação) do uso incorreto de dados por um programa. Um *sistema de tipo* é uma definição precisa das associações entre o tipo de uma variável, seus valores e as operações possíveis sobre esses valores.

O propósito de estabelecer tipos de dados em linguagens de programação é oferecer ao programador formas de construir programas com a melhor solução.

Tipos estáticos e dinâmicos

Normalmente os sistemas de tipos impõem restrições na escrita ou compilação de uma determinada linguagem. Por exemplo, os valores de uma operação de soma devem ser numéricos. Existem linguagens de programação em que a verificação de tipos é feita no momento da compilação e outras no momento da execução. Java, por exemplo, a verificação de tipos ocorre no momento da compilação, porém as conversões desses tipos são feitas em tempo de execução. Muitas das linguagens requerem que um único tipo de dados seja associado à uma variável, desde o momento que ela for declarada até o final da execução do código em que ela foi criada. Nesse caso, o tipo de valor será determinado em tempo de compilação.

Nesses casos, existe um conceito chamado linguagem tipada estaticamente e linguagem tipada dinamicamente. Em uma linguagem tipada estaticamente, seus tipos são declarados em tempo de compilação, ou seja, se a variável foi criada do tipo inteiro, permanecerá assim pelo "resto de sua vida". Já uma linguagem tipada dinamicamente, o tipo da variável poderá mudar em tempo de execução. Para isso, basta atribuir um outro valor, de outro tipo. Por exemplo, se uma varável foi criada do tipo inteiro, em seguida poderá ser atribuído um valor de caractere sem oferecer nenhum tipo de erro.

Uma linguagem de programação é fortemente tipada se o seu sistema de tipo permitir que todos os erros de tipos em um programa sejam detectados em tempo de compilação ou em tempo de execução (TRUCKER, 2011, p. 104).

Conversão de tipos

Em determinado momento os tipos deverão ser convertidos para uma questão de lógica utilizada. Por exemplo, se for desenvolver um programa com interface de usuário e ela possuir uma caixa de texto, para digitar determinado valor numérico, esse valor será tratado pelo compilador como um caractere. Em seguida, será preciso utilizá-lo para efetuar uma determinada operação matemática. Se não houver uma conversão de tipos, um erro será emitido informando a necessidade de conversão. Essa conversão pode ser limitadora se o valor resultante tiver a quantidade de bit menor que o original e ampliadora se a quantidade de bit for maior. Um exemplo de conversão limitadora é converter um resultado float (2,65) para inteiro (2).

Tratamento de exceções

Segundo Trucker (2011, p. 179), uma *exceção* é uma condição de erro que ocorre a partir de uma operação que não pode ser resolvida por si só. Manipular uma exceção é a possibilidade de um programa poder lidar com erros inesperados durante a compilação, ao invés de causar o encerramento dele prematuramente.

Basicamente existem dois tipos de exceções: as de hardware e de software. Hardwares costumam detectar erros em tempo de execução, como estouro de pilhas, overflow etc. As primeiras linguagens de programação foram insuficientes ao acesso do programador à captura e tratamento de erros. Isso só foi possível com as linguagens de alto nível.

Quando uma exceção é executada, uma mensagem de erro, com endereços de memória, é emitida ao usuário. Isso gera um problema de usabilidade, pois, o

usuário ficará sem saber que ação tomar diante da mensagem. Quando isso ocorre, o programador trata essa exceção, capturando-a, ou seja, transfere o controle para um manipulador de exceções e define a resposta de acordo com o problema ocorrido. Por exemplo, quando a tentativa de leitura de um arquivo falha pelo fato de não existir no disco, é emitido um erro de I/O, que o programador captura, lança uma mensagem mais amigável e compreensível ao usuário.

A maioria das linguagens de programação usa a mesma sintaxe da linguagem C++ para tratar as exceções:

```
try {
   // código que pode gerar uma exceção
}
catch ( // tipo da exceção)
{
   //código a ser executado quando esse tipo de exceção
   ocorrer
}
```

Concorrência

Concorrência significa que dois ou mais processos concorrem pelo mesmo recurso de hardware ou de software, ao mesmo instante, e sua distribuição é feita intercalando o processador, usando fatias de tempo.

Segundo Trucker (2011, p. 485), o fatiamento de tempo divide o tempo em pequenos blocos e distribui esses blocos entre os processos de uma maneira uniformizada. A programação concorrente foi usada nos primeiros sistemas operacionais para suportar o paralelismo no hardware e para suportar a programação múltipla e o tempo partilhado.

Os primeiros computadores possuíam um único processador principal e os processos passavam um por vez em seu núcleo. Não havia concorrência e o próximo processo só seria executado depois que o anterior liberasse os recursos.

Processos

Segundo Tanenbaum (2016, p. 60), um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis. Sistemas operacionais precisam criar processos para seu funcionamento

ou para execução de outros programas. Quatro eventos principais fazem com que os processos sejam criados:

1. Inicialização do sistema.
2. Execução de uma chamada de sistema de criação de processo por um processo em execução.
3. Solicitação de um usuário para criar um processo.
4. Início de uma tarefa em lote.

E seu término, cedo ou tarde, acontece pelos seguintes eventos:

1. Saída normal (voluntária).
2. Erro fatal (involuntário).
3. Saída por erro (voluntária).
4. Morto por outro processo (involuntário).

Como foi visto, o processo concorre pelo mesmo recurso de hardware, ou seja, o mesmo espaço de endereçamento está sendo concorrido por miniprocessos simultaneamente. A esses miniprocessos damos o nome de Thread.

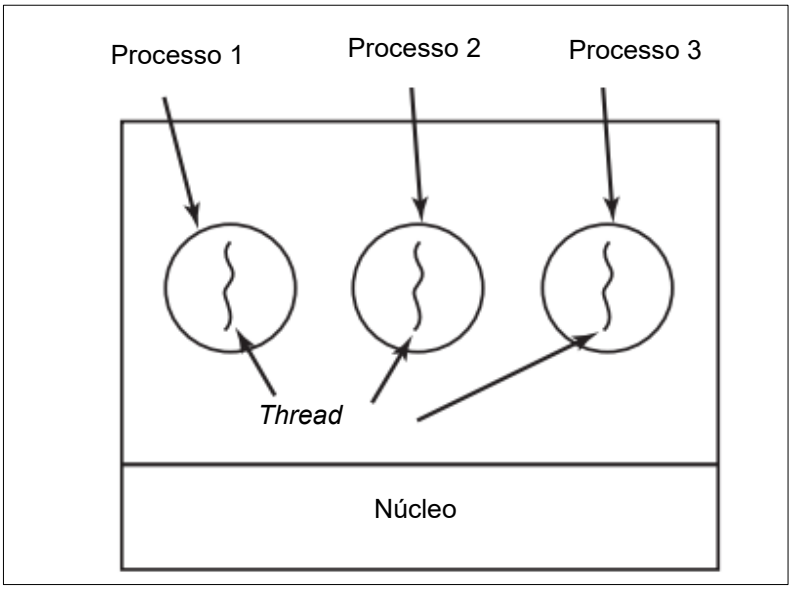
Thread

Segundo Tanenbaum (2011, p.60), o conceito de *threads* se assemelha a subprocessos, porém, diferentemente destes, não possuem identidade própria e, portanto, não são independentes. Cada *thread* possui seus próprios atributos, porém compartilham todos o mesmo espaço de endereçamento, isto é, como se fossem uma única entidade.

Os *threads* não são tão independentes como os processos e os subprocessos, uma vez que compartilham de um mesmo espaço de endereçamento e, por conseguinte, compartilham das mesmas variáveis globais, dos mesmos arquivos, das mesmas tabelas etc.

Uma vez que todo *thread* pode acessar todo o espaço virtual de endereçamento do processo pai e dos demais *threads*, ele pode "ler e escrever" em qualquer local, mesmo na pilha dos outros *threads*.

Figura 26: Processos e *Threads*



Fonte: Tanenbaum (2016, p. 60)

Deadlock

Em determinado momento o processo precisará de algum recurso, porém pode acontecer de esse recurso ou evento nunca ficar disponível. A essa situação damos o nome de *deadlock*. Esse problema torna-se frequente e crítico quando os Sistemas Operacionais evoluem no sentido de implementar o paralelismo de forma intensiva e permitir a alocação dinâmica de um número ainda maior de recursos.

Para que ocorra o *deadlock,* quatro situações são necessárias **simultaneamente.**

1. **Exclusão mútua**: cada recurso deverá estar alocado a um único processo em um determinado instante, somente.
2. **Espera por recurso:** um processo, além de possuir recursos alocados, poderá ficar esperando por outros.
3. **Não preempção:** um recurso não será liberado, quando alocado a determinado processo, só porque outros processos desejam usar o mesmo recurso.
4. **Espera circular:** um processo espera por um recurso alocado a outro processo e vice-versa.

CAPÍTULO 3
Interface do usuário

Segundo Pressman (2011, p. 287), o projeto de interfaces do usuário cria um meio de comunicação efetivo entre o ser humano e o computador, onde, seguindo um conjunto principal de projeto de interfaces, o projeto identifica objetos e eventos de interface e então confecciona um layout que forma a base para um protótipo de interface com o usuário.

Regras de ouro

Existem algumas regras que foram denominadas regras de ouro que definem conceitos básicos para o projeto e interfaces de usuário:

Deixar o usuário no comando

» **Ações indesejadas**

Defina maneiras de interações entre a interface e o usuário de modo a não forçar o usuário a realizar ações desnecessárias ou indesejadas. Por exemplo, se o usuário escolher clicar determinado botão específico, ele deverá ter as opções de parar ou pausar a tarefa a qualquer momento e, durante seu uso, não poderá haver outros tipos de funcionalidades que não sejam pertinentes.

» **Interação**

Proporcione interação facilitada através de vários meios, como comandos de teclado, movimentação de mouse, comando de voz etc.

» **Detalhes técnicos**

Oculte os detalhes técnicos como preocupação com o sistema operacional, ou seja, jamais permita que o usuário interaja com mensagens que eventualmente possam deixá-lo sem poder tomar alguma decisão. Isso ocorre normalmente quando o programador não trata as exceções e possíveis erros.

» **Sensação de controle**

A verdadeira sensação do usuário estar no controle é quando ele tem a certeza de que as interações com os objetos são exatamente como aparecem na tela.

Reduzir a carga de memória do usuário

Uma interface de usuário bem projetada não faz com que se fique tentando lembrar das funcionalidades ou informações pertinentes. Resultados passados ou recordações de ações podem ser facilitadas aos usuários através de dicas visuais.

Termos, ícones, botões e cores do mundo real, relacionados a um determinado domínio, permitem ao usuário que se apoie em indicações visuais compreensíveis, ao invés de memorizar uma sequência de interações misteriosas. Foi o que citou Pressman (2011, p. 287) como metáforas do mundo real.

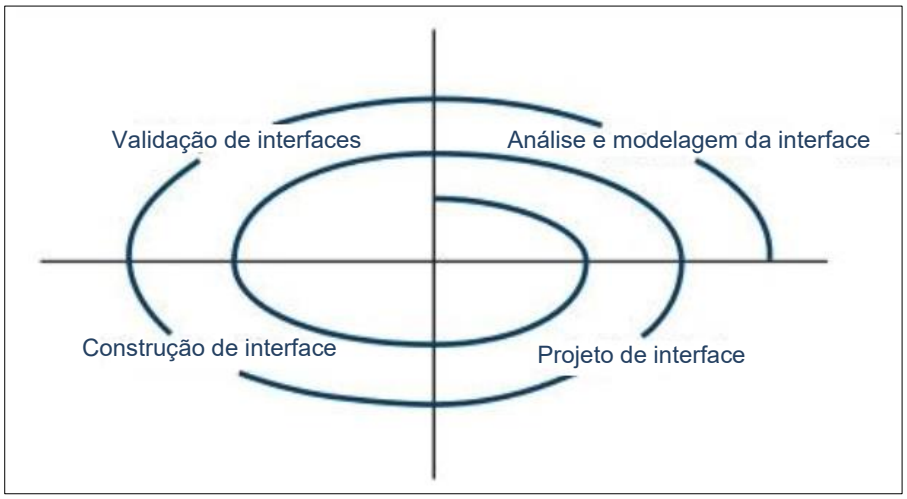
Tornar a interface consistente

Tornar a interface consistente significa que todas as informações visuais serão organizadas conforme determinado assunto e mantidas assim ao longo das exibições das telas. Também, a mudança de operação, tarefa ou funcionalidade deve seguir padrões, ser de forma intuitiva e com avisos prévios, como aviso de exclusão de registro, aviso de perda de dados ao mudar de tela etc.

Projeto de interface

O projeto de interface de usuário, segundo Pressman (2011, p. 292), poderá ser interativo e representado utilizando o modelo espiral. As etapas envolverão a análise e modelagem de interfaces, projeto de interfaces, construção de interfaces e validação de interfaces.

Figura 27: Processo de projeto de interface



Fonte: Pressman (2011, p. 293)

» **Análise da interface**

Na análise de interface é feito um levantamento do perfil dos usuários, como nível de habilidade, conhecimento do assunto, receptividade, que irão interagir com o sistema. Essas análises poderão ser feitas através de entrevistas.

» **Construção da interface**

É feito um protótipo que permite simulações a serem avaliadas.

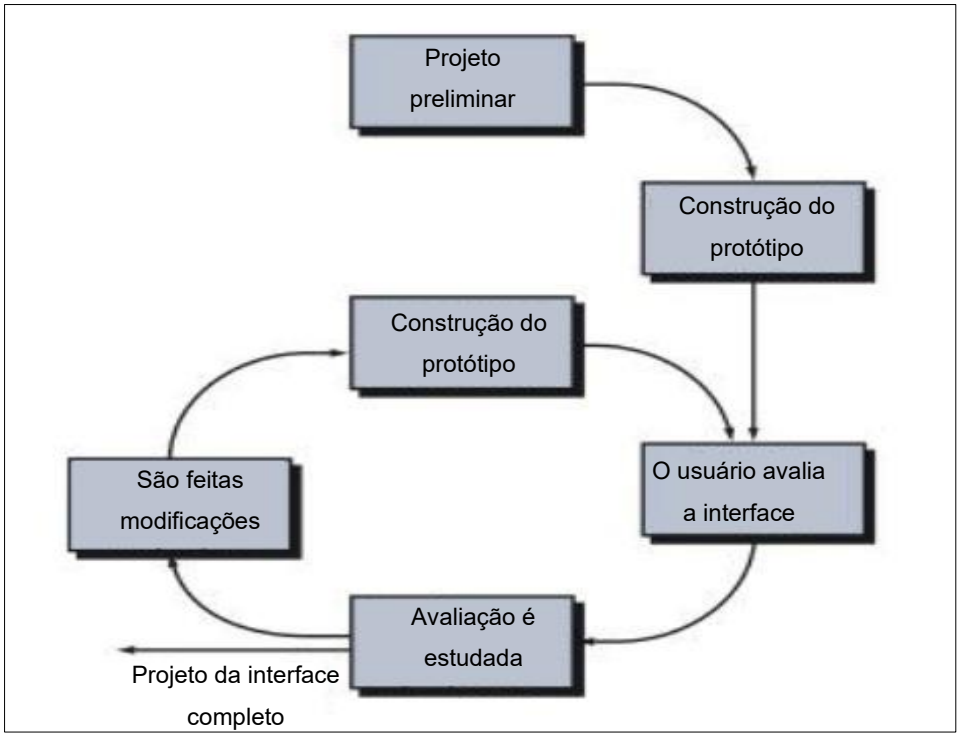
» **Validação de interface**

É a validação do usuário em relação à capacidade dos requisitos da interface sempre atendidos. Também deve ser avaliado o grau de dificuldade da usabilidade da interface e a geração de possíveis falhas.

» Projeto da interface

Definir os objetivos das ações do usuário que permitam que eles realizem todas as tarefas de usabilidade.

Figura 28: Ciclo de avaliação de projeto de interfaces



Fonte: Pressman (2011, p. 312)

CAPÍTULO 4

Suporte para a programação orientada a objetos

O conceito de programação orientada a objetos tem suas raízes no SIMULA 67, mas não foi completamente desenvolvido até a evolução do *Smalltalk* que resultou no *Smalltalk* 80 (SEBESTA, 2011, p. 546).

Para uma linguagem de programação ser considerada com suporte a orientação a objetos, segundo Sebesta (2011, p. 547), ela deve dar suporte a três recursos: herança, tipos de dados abstratos e vinculação dinâmica de chamadas a métodos.

Herança

Segundo Sebesta (2011, p. 547), a pressão por produtividade sob os desenvolvedores, em meados de 1980, fez com que muitos buscassem soluções mais promissoras, e tipos de dados, encapsulamento e controle de acessos eram candidatos para a reutilização de software. Por muito tempo, problemas de tipos abstratos, de relacionamento entre classes pai e filhas, foram constantes.

A herança veio para solucionar tanto o problema do reúso de tipos abstratos quanto para a organização de programas. Surgem, então, nesse momento, as nomenclaturas a que os programadores atuais estão familiarizados. Esses nomes são oriundos do SIMULA 67. Os dados abstratos são chamados de classe, as instâncias das classes de objetos, a classe que foi definida através de outra chama-se subclasse e a classe derivada é a classe pai ou superclasse. Os subprogramas que definem as operações em objetos de uma classe são chamados de **métodos**. As chamadas a métodos são algumas vezes chamadas de **mensagens**. A coleção completa de métodos de um objeto é chamada de **protocolo de mensagens** ou **interface de mensagens**.

Outras definições são mencionadas por Sebesta (2011, p. 548):

1. A classe pai pode definir alguns de seus membros como tendo acesso privado, fazendo com que esses métodos não sejam visíveis na subclasse.

2. A subclasse pode adicionar membros àqueles herdados da classe pai.

3. A subclasse pode modificar o comportamento de um ou mais métodos herdados. É dito que o novo método sobrescreve o método herdado, chamado então de método sobrescrito.

Vinculação dinâmica

É um tipo de polimorfismo fornecido pela vinculação dinâmica de mensagens às definições de métodos.

Veja o exemplo de Sebesta (2011, p. 549):

> Existe uma classe base A, que define um método que realiza uma operação em objetos da classe base. Uma segunda classe, B, é definida como uma subclasse de A. Objetos dessa nova classe precisam de uma operação parecida com a fornecida por A, mas um pouco diferente porque eles são levemente diferentes. Então, a subclasse sobrescreve o método herdado. Se um cliente de A e B tem uma referência para objetos da classe A, essa referência também pode apontar para objetos da classe B, tornando-a uma referência **polimórfico**.

O propósito da vinculação dinâmica é permitir a extensão do software em seu desenvolvimento ou manutenção. Além disso, existem as classes abstratas. Uma classe abstrata não pode ser instanciada, pois alguns de seus métodos podem ser declarados, mas não implementados. Elas só servem para servir de escopo para implementações hierárquicas. Qualquer subclasse de uma classe abstrata a ser instanciada deve fornecer implementações para todos os métodos abstratos herdados.

Suporte para programação orientada a objetos em C++

C++ é uma linguagem híbrida, pois suporta tanto métodos quanto funções não relacionadas a classes específicas, suporta tanto linguagens imperativas quanto orientadas a objetos e suporta tanto programação procedural quanto orientada a objetos.

Herança

Uma classe em C++ pode ser derivada de uma classe pai, alguns ou todos os membros podem ser herdados, além de poder adicionar ou modificar novos membros. Os objetos

em C++ devem ser inicializados antes de serem utilizados e devem ter ao menos um método construtor. Isso vale também caso a classe seja uma subclasse.

Vinculação dinâmica

Um objeto em C++ pode ser manipulado por uma variável de valor, em vez de ponteiro ou referência.

Suporte para programação orientada a objetos em Java

Herança

A diferença da linguagem C++ e Java é que em Java um método pode ser definido como ***final*** e não pode ser sobrescrito em suas classes filhas. Ela também não pode ser pai de nenhuma classe. Também significa que as vinculações de chamadas a métodos da subclasse podem ser estaticamente vinculadas. O Java requer que o construtor na classe pai seja chamado antes do construtor da subclasse.

O Java aceita somente herança simples, sendo que para dar suporte parcial à herança múltipla, utiliza uma classe abstrata chamada interface e que não poderá conter definições.

Vinculação dinâmica

Segundo Sebesta (2011, p. 573), em Java, todas as chamadas a métodos são dinamicamente vinculadas, a menos que o método chamado tenha sido declarado como final, de modo que não pode ser sobrescrito e todas as vinculações são estáticas. A vinculação estática é usada também se o método for estático (***static***) ou privado (***private***), no qual ambos os modificadores não permitem sobrescrita.

Suporte para programação orientada a objetos em C#

Herança

C# usa a sintaxe de C++ para definir classes. Um método herdado da classe pai pode ser substituído na classe derivada marcando sua definição na subclasse com **new**. O

método **new** oculta o método com o mesmo nome na classe pai para acesso normal. Entretanto, a versão da classe pai ainda pode ser chamada prefixando a chamada com **base**. O suporte de C# para interfaces é o mesmo de Java (SEBESTA, 2011, p. 573).

Vinculação dinâmica

Para permitir vinculação dinâmica de chamadas a métodos em C#, tanto o método base quanto seus métodos correspondentes nas classes derivadas devem ser especialmente marcados. O método da classe base deve ser marcado como **virtual**, como em C++.

Para evitar sobrescrita acidental, os métodos correspondentes nas classes derivadas devem ser marcados com *override*. A inclusão de *override* torna claro que o método é uma nova versão de um método herdado (SEBESTA, 2011, p. 577).

Referências

AMBLER, Scott W. **User Interface Design Tips, Techniques, and Principles**. Disponível em: http://www.ambysoft.com/essays/userInterfaceDesign.html. Acesso em: 1 jul. 2019.

BASILI, Victor R. **Goal Question Metric Paradigm**, 1994.

BOEHM, Dr. Barry. **The Spiral Model as a Tool for Evolutionary Acquisition.** Pittsburgh, PA: Carnegie Mellon University, July, 2000.

FILHO, Wilson de Pádua Paula. **Engenharia de Software: fundamentos, métodos e padrões**. Editora LTC: março de 2000.

IFPUG (International Function Point Users Group). **Manual Prático de ponto de função.** Disponível em: <http://www.ifpug.org>. Versão 4.3.1, 2010. Acesso em: 1 jul. 2019.

MARTIN, C. Robert; MICAH, Martin. **Agile Principles, Patterns, and Practices in C#.** Prentice Hall: 2006. p: 768

PRESSMAN, Roger S. **Engenharia de Software:** uma abordagem profissional. 7ª ed. Porto Alegre: AMGH, 2011

PRESSMAN, Roger S. **Engenharia de Software.** São Paulo: Makron Books, 1985

PMBOK: Guia do Conhecimento em gerenciamento de projetos. 6 ed., 2017, Project Management Institute.

SEBESTA, Robert W. **Conceitos de linguagens de programação**. 9. ed. – Dados eletrônicos. – Porto Alegre: Bookman, 2011.

SOMMERVILLE, Ian. **Engenharia de Software** / Ian Sommerville: tradução Ivan Bosnic e Kalinka G. de O. Gonçalves; revisão técnica Kechi Hirama. — 9. ed. — São Paulo: Pearson Prentice Hall, 2011. p. 551.

TANENBAUM, Andrew S. **Sistemas operacionais modernos.** 4. ed. – São Paulo: Pearson Education do Brasil, 2016.

TUCKER, Allen B., NOONAN, Robert E. **Linguagens de programação:** princípios e paradigmas Porto Alegre: AMGH, 2010.

WEINBERG, G. **On Becomming a Technical Leader.** Dorset House, 1986. p 304.

Sites

http://handbookofsoftwarearchitecture.com . Acesso em: 1 jul. 2019.

https://www.iso.org/standard/50508.html. Acesso em: 1 jul. 2019.

http://dx.doi.org/10.1109/MS.2006.58. Acesso em: 1 jul. 2019.

http://dx.doi.org/. Acesso em: 1 jul. 2019.

http://www.tutorialspoint.com/software_architecture_design/component_based_architecture.htm. Acesso em: 1 jul. 2019.

http://secretsofconsulting.blogspot.com/. Acesso em: 1 jul. 2019.

http://www.geraldmweinberg.com/Site/Home.html. Acesso em: 1 jul. 2019.

https://www.oreilly.com/library/view/applied-software-project/0596009488/ch04.html. Acesso em: 1 jul. 2019.

https://www.guru99.com/software-development-life-cycle-tutorial.html. Acesso em: 1 jul. 2019.

https://www.pmi.org/pmbok-guide-standards/framework/practice-standard-work-breakdown-structures-2nd-edition. Acesso em: 1 jul. 2019.

https://www.ifpug.org. Acesso em: 1 jul. 2019.

https://www.brighthubpm.com/project-planning/51681-accurate-estimations-with-the-dephi-method/. Acesso em: 1 jul. 2019.