



PROJETOS ÁGEIS E ANÁLISE DE SISTEMAS



Luís Otavio Toledo Perin

PROJETOS ÁGEIS E ANÁLISE DE SISTEMAS

1ª edição

Londrina
Editora e Distribuidora Educacional S.A.
2020

© 2020 por Editora e Distribuidora Educacional S.A.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Editora e Distribuidora Educacional S.A.

Presidente

Rodrigo Galindo

Vice-Presidente de Pós-Graduação e Educação Continuada

Paulo de Tarso Pires de Moraes

Conselho Acadêmico

Carlos Roberto Pagani Junior
Camila Braga de Oliveira Higa
Carolina Yaly
Giani Vendramel de Oliveira
Henrique Salustiano Silva
Mariana Gerardi Mello
Nirse Ruscheinsky Breternitz
Priscila Pereira Silva
Tayra Carolina Nascimento Aleixo

Coordenador

Tayra Carolina Nascimento Aleixo

Revisor

Lucas dos Santos Araujo Claudino

Editorial

Alessandra Cristina Fahl
Beatriz Meloni Montefusco
Gilvânia Honório dos Santos
Mariana de Campos Barroso
Paola Andressa Machado Leal

Dados Internacionais de Catalogação na Publicação (CIP)

Perin, Luís Otavio Toledo
P445p Projetos ágeis e análise de sistemas/ Luís Otavio Toledo
Perin, – Londrina: Editora e Distribuidora Educacional S.A.,
2020.

45 p.

ISBN 978-65-5903-084-2

1. Guia do conjunto de conhecimentos em gerenciamento de projetos (Guia PMBOK).
2. Scrum (Desenvolvimento de software). 3. Projeto de sistemas I. Título.

CDD 005

Raquel Torres - CRB: 6/2786

2020
Editora e Distribuidora Educacional S.A.
Avenida Paris, 675 – Parque Residencial João Piza
CEP: 86041-100 — Londrina — PR
e-mail: editora.educacional@kroton.com.br
Homepage: <http://www.kroton.com.br/>

SUMÁRIO

Projeto de software, seu escopo, o PMI e o PMBOK _____	05
Metodologia Ágil e características: DSDM, FDD, XP e <i>Scrum</i> _____	20
Paradigma Orientação a Objetos: histórico e conceitos _____	37
UML: conceitos, diagramas e ferramenta CASE _____	55

Projeto de software, seu escopo, o PMI e o PMBOK

Autoria: Luís Otávio Toledo Perin

Leitura crítica: Lucas dos Santos Araujo Claudino



Objetivos

- Definir o que é projeto de software.
- Compreender o escopo de um projeto de software.
- Assimilar conceitos de PMI e PMBOK.



1. Projeto de software

1.1 Introdução

Neste tema, abordaremos sobre o projeto de software e seu escopo, além de introduzir os conceitos de PMI (*Project Management Institute*) e PMBOK (*Project Management Body of Knowledge*). Então, atente-se aos conteúdos apresentados e não perca nenhum dos próximos parágrafos!

Como sabemos, a grande utilização de dispositivos e o crescente uso de aplicações favorecem a indústria de software, independentemente da área, já que todos os processos que compõem o produto devem ser trabalhados de maneira constante em seu ciclo de desenvolvimento.

Ao se desenvolver um software para um cliente, algumas perguntas estão intrínsecas, como a capacidade em realizar tal atividade, o tempo de desenvolvimento e o custo pelo produto. Para que essas perguntas possam ser respondidas, necessita-se de uma análise inicial de requisitos, além da definição do escopo, realização de estimativas de prazo, levantamento riscos, definição recursos, desenvolvimento do cronograma e documentação detalhada, a fim de a partir disso um planejamento possa ser realizado, o qual deverá ser constantemente revisado durante a execução do projeto, pois, isso é importante para o acompanhamento dos trabalhos, além de possibilitar a correção de eventuais desvios do planejamento inicial.

A todos esses conjuntos de tarefas podemos compreender como parte da gestão de projetos, o qual tem por objetivo o seu gerenciamento e controle, além de contar com o auxílio de técnicas para sua melhoria, como o PMBOX e o PMI. Desse modo, os próximos capítulos tratarão do escopo do software, além de dar continuidade sobre o gerenciamento de projetos.

1.2 Processos e projetos

Para que você possa compreender como a engenharia de software atua na análise, construção e manutenção do software, alguns termos precisam ficar claros logo de início. Nesse sentido, os processos e projetos são amplamente utilizados dentro de toda e qualquer engenharia, não sendo diferente para a área de sistemas computacionais. Por isso, compreender que cada um possui um objetivo e uma função para a elaboração do produto é de extrema importância para seu correto entendimento. Segundo Sbrocco e Macedo (2012, p. 29):

Projetos estão relacionados com algo novo, que tem início e fim bem definidos. Já os processos ocorrem de maneira contínua, com a finalidade de produzir produtos ou serviços idênticos. Processos normalmente são estabelecidos a partir da conclusão de um projeto. Os projetos produzem um produto único, que é diferente em pelo menos uma característica de qualquer outro produto existente. Já produtos decorrentes de processos serão sempre idênticos.

Dessa forma, cada termo possui uma definição conceitualizada e específica, entretanto, ambos possuem algo em comum: a delimitação de escopo e recursos empregados, algo que dentro de toda a atividade deve ser previamente bem definido, evitando dúvidas e desvios durante sua execução.

Dentro da engenharia de software chamamos de ciclo de vida do projeto toda a fase de concepção até a entrega do produto, onde há a existência de métricas bem definidas, além de processos em cada etapa, garantindo que o produto a ser entregue passe por um rigoroso controle de qualidade. No que tange o gerenciamento de projeto, existem dois conjuntos de processos: de gerenciamento de projetos, utilizado para descrever e organizar o trabalho do projeto; e o orientado ao produto desejado, que tem por finalidade especificar e criar o produto do projeto.

Para que haja uma organização do projeto, alguns processos precisam ser definidos e estabelecidos, assim como é mostrado no quadro a seguir.

Quadro 1 – Grupos de processos para organização do projeto


Processo	Descrição
Início	Processos relacionados ao início de determinada fase ou projeto. Deve caracterizar seu início.
Planejamento	Processos relacionados à divisão do projeto em etapas ou fases. Também pode haver a fragmentação em partes menores, as quais facilitam a execução, mantendo os objetivos do projeto.
Execução	Processos destinados à coordenação de pessoas e recursos para a elaboração das atividades previamente definidas.
Controle	Processos com foco na garantia que os objetivos do projeto serão atingidos. Esse controle pode ser feito com técnicas de monitoramento, medição de progresso e tomada de ações corretivas.
Finalização	Processos relacionados ao aceite do projeto ou de certa fase.

Fonte: elaborado pelo autor.

Quanto aos processos de gerenciamento de projetos, eles se dividem em subprocessos, os quais possuem relação com a área de conhecimento que interagem ao longo de todo o projeto. Por exemplo, podemos citar alguns desses tipos de processos: gerenciamento de tempo, de custo, de escopo, de risco, de qualidade, de integração, de comunicação, de recursos humanos e de contratos ou de aquisições. Todos eles possuem um objetivo específico: o gerenciamento do projeto.

1.3 Gerenciamento de projeto de software

Assim como em qualquer outro processo de engenharia, a necessidade por métricas e processos bem definidos é essencial para o bom gerenciamento do projeto. Dessa forma, a engenharia de software tem papel fundamental dentro do gerenciamento de projeto, haja vista que




é um procedimento que envolve prazos e custo, além do gerenciamento da equipe para a correta execução dos trabalhos propostos.

Nesse contexto, é de responsabilidade do gerente de projetos garantir que o projeto de software atenda e até supere as restrições mencionadas anteriormente, além de oferecer um produto de alta qualidade. Entretanto, segundo Sommerville (2011), o bom gerenciamento não é sinônimo de sucesso do produto, já que inúmeros outros fatores estão em jogo, mas um mau gerenciamento pode colocar em risco todo o planejamento efetuado anteriormente, ocasionando em atrasos e, até mesmo, aumento de custo no projeto.

O trabalho do gerente de projetos vai muito além de garantir o andamento do projeto, sendo ele responsável por mais algumas funções, como:

- a. Planejamento de projeto: responsável pelo planejamento, elaboração de estimativa, cronograma de desenvolvimento de projeto e atribuição de certas tarefas para a equipe. A supervisão dos trabalhos também se torna necessária, já que é necessário garantir que os padrões elencados no início do projeto sejam cumpridos, como prazo e custo.
- b. Geração de relatórios: com o objetivo de informar os clientes e o gerente da empresa sobre o andamento do projeto, o gerente de projeto fica responsável por essa função. Esses relatórios devem atender diferentes níveis, ou seja, conter informações resumidas para um rápido entendimento, por exemplo, até dados mais complexos e técnicos, os quais podem ser discutidos durante a revisão do projeto.
- c. Gerenciamento de riscos: responsáveis por avaliar os riscos que podem afetar o projeto, além de controlá-los e intervir quando necessário.
- d. Gerenciamento de pessoas: a gestão da equipe também fica a cargo do gerente de projeto. Ele deve decidir os integrantes desse



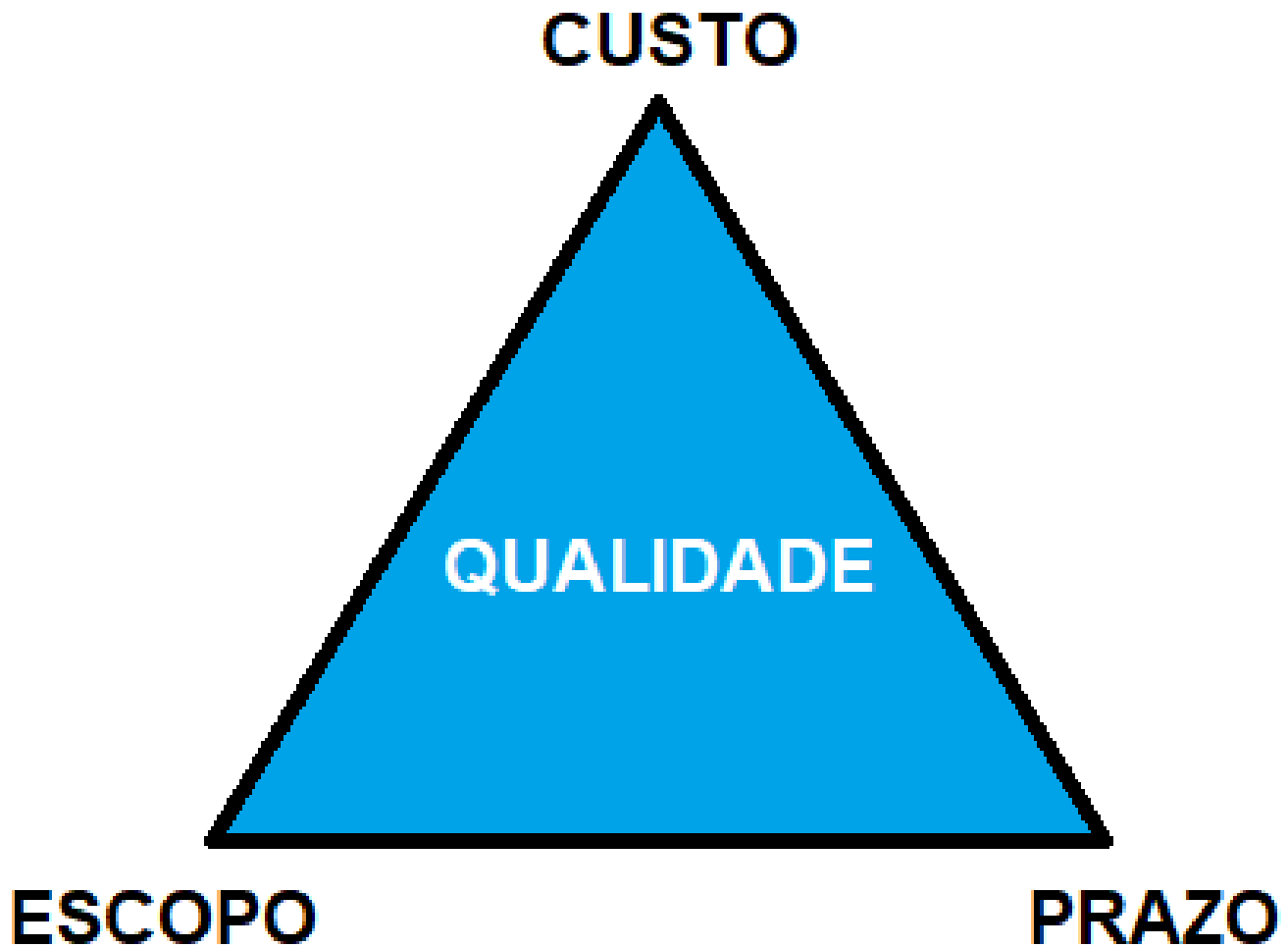
time, além de estabelecer a forma de trabalho, buscando garantir um melhor desempenho de todos os integrantes.

- e. Elaboração de propostas: um dos primeiros passos para a concepção de um projeto de software é sua aceitação por parte do cliente e/ou patrocinador. Desse modo, é de responsabilidade do gerente de projeto desenvolver uma proposta, a qual deverá conter estimativas de custo, prazo e proposta do projeto, tendo como objetivo o aceite por parte do contratante.

Com todas essas responsabilidades, o gerente de projeto se torna peça fundamental em todas as fases de execução, seja antes do desenvolvimento propriamente dito, durante a criação e após a entrega do produto, já que a manutenção é essencial para a continuidade e sobrevivência do software. Logo, diretrizes de escopo, prazo e custo devem servir como embasamento para o gerenciamento do projeto, haja vista que estão diretamente relacionadas com o fator qualidade do produto, conforme a Figura 1.

Neste sentido, qualquer alteração em pelo menos um desses itens afetará ao menos mais um e, conseqüentemente, o fator qualidade. Com isso, saber estabelecer um equilíbrio e dimensionar as atividades é um papel crucial dentro do gerenciamento de projeto, que possui como foco o gerente de projeto. Por isso, utilizar-se de ferramentas para o melhor controle e dimensionamento do projeto é parte fundamental da atividade, fazendo com que gráficos PERT (*Program Evaluation Review Technique*), CPM (*Critical Path Method*) e Gantt ajudem, por exemplo, com a restrição de tempo e cumprimento do cronograma. Quanto ao custo, estabelecer um orçamento e utilizar-se de gráficos como o histograma de recursos contribui para que não haja surpresas indesejáveis ou que um dos lados saia prejudicado (Sbrocco; Macedo, 2012).

Figura 1 – Qualidade e fatores determinantes




Fonte: elaborado pelo autor.

1.4 Ciclo de vida

Apesar de pensarmos apenas no desenvolvimento do software propriamente dito, devemos ter em mente que essa é uma de suas fases de seu ciclo de vida, sendo esse ciclo composto por inúmeras outras, a qual chamamos de ciclo de vida. Dessa forma, o ciclo tem como objetivo descrever as fases pelas quais o software passa desde a sua concepção até a sua descontinuidade.

Como o próprio nome nos sugere, dentro do ciclo de vida de software existem várias fases, as quais remetem a diferentes processos, cada qual com seu foco e objetivo de trabalho, fazendo com que em cada etapa



trabalhada haja métricas e passos a serem seguidos. De acordo com Sommerville (2011), a seguir, listaremos quatro fases mais abrangentes dentro dos diversos ciclos de vida. Em cada fase há um conjunto de atividades ou passos a serem realizados por todos os envolvidos do projeto, sendo elas:

1. Fase de definição.
2. Fase de desenvolvimento.
3. Fase de operação.
4. Fase de retirada.


A seguir, especificaremos cada fase, onde podemos entender como o ciclo de vida funciona.

1.4.1 Fase de definição

A fase de definição do software ocorre juntamente com outras atividades, como a modelagem de processos de negócios e a análise de sistemas. Nessa etapa, há o levantamento, por diversos profissionais, da situação atual, tendo como foco a busca pela identificação de problemas, os quais possam resultar em soluções que envolvam sistemas computacionais. Posteriormente, um estudo de viabilidade, com análise de custo-benefício, deve ser realizado para averiguar a melhor solução a ser escolhida.

De posse dessa informação, outro passo importante é a definição da forma de contratação, ou seja, se o sistema será adquirido ou desenvolvido, incluindo informações como necessidade de hardware, ferramentas de software, mão de obra, procedimentos, informações e documentação.

Se a opção escolhida for pelo desenvolvimento do sistema, no escopo da engenharia de software é obrigatória a elaboração de um documento



que trate sobre tal proposta, podendo ser a base de um futuro contrato de desenvolvimento, por exemplo.


Além disso, nessa fase se inclui um dos principais processos, o levantamento de requisitos do software, além dos modelos de domínio. Os requisitos garantem que o engenheiro de software possa elaborar um plano de desenvolvimento de software de acordo com a necessidade levantada anteriormente, indicando em detalhes os recursos necessários (humanos e materiais), bem como as estimativas de prazos e custos (cronograma e orçamento).

1.4.2 Fase de desenvolvimento

A denominada fase de desenvolvimento, ou de produção do software, tem como objetivo a inclusão de todas as atividades de construção do produto. Ou seja, isso inclui atividades de projeto, implementação, verificação e validação do software.

Dentro da fase de projeto podemos entender toda a concepção e modelagem empregada para descrever como a aplicação será implementada, incluindo:

- a. Projeto conceitual: elaboração de ideias e conceitos básicos para o desenvolvimento do software.
- b. Projeto de interface com o usuário: direcionado a elaborar formas de como o usuário deverá interagir com a aplicação, ou seja, como ele deverá proceder para realizar suas tarefas. Além disso, também é determinado os objetos de *layout*, como botões, menus, *layout* de janelas e telas. Neste momento, o tema usabilidade deverá ser evidenciado, já que um dos pontos de sucesso da aplicação passa por esse item.
- c. Projeto da arquitetura de software: destinado à elaboração de uma visão macroscópica do software em relação aos componentes que interagem entre si.

- 
- d. Projeto dos algoritmos e estrutura de dados: tem como objetivo buscar soluções algorítmicas, assim como estruturas de dados associados. Tal processo deverá ser feito de maneira independente da linguagem de programação a ser utilizada.

A fase de implementação, por sua vez, envolve atividades de codificação, compilação, integração e testes. Assim, podemos entender como codificação, a tradução do projeto em um programa, o qual utiliza linguagens e ferramentas adequadas. Ela deve refletir a estrutura e o comportamento descrito no projeto da arquitetura do software. Neste momento, os testes já podem ser iniciados e a depuração de erros ocorre durante a programação, utilizando técnicas e ferramentas adequadas. Além disso, nesta etapa, um controle e gerenciamento de versões é fundamental para a organização da aplicação, haja vista que o versionamento semântico pode ser executado por um *framework* ou outra plataforma, onde as mudanças são indicadas por meio de incrementos em números específicos. Por exemplo, o Git, onde é possível criar versionamentos, os quais auxiliam no processo de consulta de mudanças de um determinado processo ou, até mesmo, em se utilizar de versões anteriores a atual.

Por fim, a fase de verificação e validação se destina a comprovar que o software está de acordo com as especificações elencadas no início do projeto, bem como a satisfação do usuário frente ao produto desenvolvido. Mais especificamente, a validação visa assegurar que o programa está fazendo o que foi definido em sua especificação. Logo, a verificação visa certificar se o programa está correto, isto é, se não possui erros de execução e está fazendo de forma correta suas funcionalidades.

1.4.3 Fase de operação

Nesta fase podemos elencar algumas atividades, como: distribuição e entrega, instalação e configuração, utilização e manutenção. Quanto a distribuição e entrega, ela pode ser realizada diretamente pelo desenvolvedor, ou por pacote, que pode ser vendido em lojas ou pela internet.

Geralmente, o processo de instalação e configuração pode ocorrer com a ajuda do próprio software adquirido, já que na maioria das vezes são autoexplicativos.


A utilização, por sua vez, está diretamente relacionada ao nome, ou seja, a utilização do sistema. Nesse sentido, fatores como a usabilidade devem ser levados em consideração.

Por fim, a manutenção poderá ocorrer de diferentes maneiras, sendo corretiva, adaptativa, de aperfeiçoamento ou evolutiva. No entanto, elas podem resolver diferentes tipos de problemas, como a qualidade do software, adequação de novos requisitos dos clientes ou adaptar-se às novas tecnologias que surgem com o passar dos anos.

1.4.4 Fase de retirada

Nesta fase, o software obsoleto é descontinuado e sua substituição demanda muito esforço e dedicação, haja vista que inúmeras aplicações ainda funcionam, mas a nível tecnológico estão defasadas. Deste modo, remover um sistema legado não é uma tarefa fácil, já que a aplicação é confiável e os usuários dominam todas as funções do sistema.

Com isso, processos de reengenharia tem o objetivo de facilitar e viabilizar o processo de migração para a nova plataforma, garantindo, por exemplo, a descontinuidade por etapas. Neste momento, o engenheiro de software deverá se utilizar de estratégias, as quais



envolvam camadas de processo, métodos, ferramentas e fases da engenharia de software. Tudo isso pode ser definido como modelo de processo, que leva em consideração alguns fatores, como: a natureza do projeto e da aplicação a ser desenvolvida, os métodos e as ferramentas a serem utilizadas e os controles e produtos que precisam ser entregues.

Dentro do modelo de processo, alguns outros podem estar inclusos, como: modelo em cascata, modelo de prototipação, modelo RAD (*Rapid Application Development*), modelo incremental, modelo espiral, modelo de desenvolvimento baseado em componentes, modelo de desenvolvimento concorrente, modelo de métodos formais e modelos de processos ágeis.

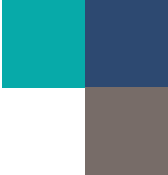
De acordo com Pressman (2011, p. 678):

A manutenção e o suporte de software são atividades contínuas que ocorrem por todo o ciclo de vida de um aplicativo. Durante essas atividades, defeitos são corrigidos, aplicativos são adaptados a um ambiente operacional ou de negócio em mutação, melhorias são implementadas por solicitação dos interessados e é fornecido suporte aos usuários quando integram um aplicativo em seu fluxo de trabalho pessoal ou corporativo.

Portanto, é possível perceber que o ciclo de vida do software é bastante amplo e dinâmico, onde cada fase possui uma particularidade, a qual contribuirá para o melhor direcionamento daquilo que se quer produzir ou prestar suporte, haja vista que o ciclo nunca para, mesmo em casos que o produto já sendo utilizado.

2. PMI e PMBOX

Dentro da área de gerenciamento de projetos, alguns conceitos tornam-se importantes para que o ciclo de vida do software se fortaleça em



todas as suas fases, além de garantir que os profissionais estejam em sintonia com os métodos e métricas mais atuais que o mercado oferta. Com isso, algumas nomenclaturas passam a surgir, como PMI, MPBox e PMP. No entanto, o que cada uma significa e qual sua influência frente a aplicação e seu gerenciamento? Vamos descobrir juntos!


O PMI é uma organização sem fins lucrativos que tem o objetivo de disseminar as melhores práticas de gerenciamento de projetos em todo o mundo. Por meio de publicações, eventos e reuniões o tema é disseminado entre todos da área de gerenciamento, garantindo que o compartilhamento de informações e padrões ocorra de modo universal (Sbrocco; Macedo, 2012).

Contudo, nem sempre foi assim, as ideias surgiram a partir de algumas reuniões e encontros em 1969, por meio da união de gerentes de projetos, onde a possibilidade de trocar informações e compartilhar problemas em comum resultou na criação de uma espécie de associação, a qual denominou-se PMI.

Para Sbrocco e Macedo (2012, p. 30):

Com cerca de 90.000 membros em todo o mundo, o PMI é hoje a organização mais importante da área de gerenciamento de projetos, estabelecendo padrões, provendo seminários, programas educacionais e certificação profissional que cada vez mais as organizações exigem dos seus líderes de projeto.

Ela tem a missão de organizar o gerenciamento de projetos e seus profissionais podem usufruir dessa prática de maneira organizada, já que é separado em capítulos e segue uma categorização segundo regiões geográficas. Desse modo, cada membro do PMI é vinculado a um capítulo, e com os profissionais da mesma região realizam atividades de cunho formativo para o gerenciamento de projetos, garantindo assim que haja uma atualização constante entre os profissionais da área.



Como consequência dessa participação e contribuição dos membros do PMI, o gerenciamento de projetos só tem a ganhar, já que tudo está reunido em uma compilação de obra única, a qual damos o nome de PMBOK que é um guia das melhores práticas do gerenciamento de projetos já elaborado pelo PMI.

Esse guia traz, em sua última edição, as dez áreas que merecem maior atenção durante o gerenciamento de um projeto, sendo cada uma delas descrita por meio de processos:

1. Gerenciamento da integração do projeto.
2. Gerenciamento do escopo do projeto.
3. Gerenciamento do cronograma do projeto.
4. Gerenciamento dos custos do projeto.
5. Gerenciamento da qualidade do projeto.
6. Gerenciamento dos recursos do projeto.
7. Gerenciamento das comunicações do projeto.
8. Gerenciamento dos riscos do projeto.
9. Gerenciamento das aquisições do projeto.
10. Gerenciamento das partes interessadas do projeto.

Portanto, um projeto é um conjunto de esforços integrados para se obter, ao final do ciclo, o produto desejado. Cada área de conhecimento citada anteriormente se refere a um fator que deve ser considerado dentro da gestão de projetos, sendo sua execução essencial para o sucesso dos trabalhos.

No PMBOX também estão elencados os processos, os quais somam-se 49 e interagem com os grupos de processos e as áreas de conhecimento. Essas áreas e práticas podem ser resumidas nos seguintes processos diferentes:

- a. Iniciação: trata-se do início da implantação do projeto. Neste momento, a elaboração de ações se faz necessária, além da avaliação dos processos já existentes, entre outras ações.
- b. Planejamento: parte destinada à definição do escopo do projeto, bem como o refinamento dos objetivos. Neste momento, as ações tomadas são delimitadas.
- c. Execução: implantação do plano de projeto propriamente dito. Neste momento o objetivo principal é atingir as metas elencadas.
- d. Monitoramento e controle: tem por objetivo supervisionar, rastrear e regular a evolução e a performance do projeto. Áreas que mereçam alteração em seu planejamento estratégico são identificadas.
- e. Encerramento: finalização das atividades do projeto.

Contudo, é interessante mencionar que PMBOX é apenas um guia e, com isso, tem o papel de apoio frente ao gerenciamento do projeto, deste modo, a experiência e o conhecimento do gerente de projetos deve ser levado em consideração para que a execução das atividades possa ocorrer da melhor maneira possível, e que o objetivo seja cumprido, ou seja, um produto com qualidade.

Referências Bibliográficas

PRESSMAN, R. S. **Engenharia de software**: uma abordagem profissional. 7. ed. Porto Alegre: AMGH, 2011.

SBROCCO, J. H. T. de C.; MACEDO, P. C. de. **Metodologias ágeis**: engenharia de software sob medida. São Paulo: Érica, 2012.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

Metodologia Ágil e características: DSDM, FDD, XP e *Scrum*

Autoria: Luís Otávio Toledo Perin

Leitura crítica: Lucas dos Santos Araujo Claudino



Objetivos

- Definir o que é metodologia ágil.
- Compreender as características da metodologia ágil.
- Assimilar métodos ágeis DSDM, FDD, XP e *Scrum*.



1. Metodologia ágil

1.1 Introdução


Nesta leitura abordaremos sobre a metodologia ágil e suas características, além da discussão sobre alguns métodos ágeis, como o DSDM, FDD, XP e *Scrum*. Por isso, fique ligado e não perca nenhum dos próximos parágrafos, pois, eles estão recheados de informações que você precisa saber!

Por meio da engenharia de software, os processos passaram a utilizar métricas e práticas padronizadas para a análise, construção e manutenção de suas aplicações. Neste aspecto, segundo Sommerville (2011), o processo de desenvolvimento, em específico, foi o que mais sofreu modificações ao longo do tempo, partindo do convencional, em cascata ou baseado em especificações, por exemplo, até a metodologia ágil, a qual costuma entregar o software em menos tempo.

Com a evolução da tecnologia e com a dinâmica a qual o mundo está inserido, é essencial para o negócio e a sobrevivência do cliente produzir software em menos tempo, já que o mercado se transforma com rapidez e pede agilidade nas tarefas executadas. Nesse sentido, segundo Sommerville (2011, p. 38):

Nos dias de hoje, as empresas operam em um ambiente global, com mudanças rápidas. Assim, precisam responder a novas oportunidades e novos mercados, a mudanças nas condições econômicas e ao surgimento de produtos e serviços concorrentes. [...] O desenvolvimento e entrega rápidos são, portanto, o requisito mais crítico para o desenvolvimento de sistemas de software.

Um efeito colateral dessa rapidez que o mercado pede quanto ao desenvolvimento de aplicações é a não preocupação com a qualidade ou a ausência de compromisso quanto aos requisitos do software,



algo que futuramente pode ocasionar problemas, mas que para uma implantação rápida pode ser negligenciada em um primeiro momento (SOMMERVILLE, 2011).

Ainda, segundo Sommerville (2011, p. 38-39):

Processos de desenvolvimento de software que planejam especificar completamente os requisitos e, em seguida, projetar, construir e testar o sistema não estão adaptados ao desenvolvimento rápido de software. [...] Quando o software estiver disponível para uso, a razão original para sua aquisição pode ter mudado tão radicalmente que o software será efetivamente inútil.

Desse modo, pensar em soluções que agilizem todas as etapas do ciclo de vida do software, desde o seu planejamento, passando pelo desenvolvimento e entrega, são de extrema importância para ambos, ou seja, para o cliente e a empresa desenvolvedora, já que um tem a aplicação desejada e o outro entrega o que foi acordado, gerando satisfação para ambos.

Nesse sentido, os próximos capítulos contarão como surgiu a necessidade pelo ágil, ou seja, porque o método tradicional foi insuficiente, além de exemplificar sobre os métodos ágeis mais atuais no mercado.

1.2 O início e o manifesto ágil

Já na década de 1980, o assunto “desenvolvimento ágil” era tratado, partindo da IBM a introdução de desenvolvimento incremental, com o apoio da linguagem de quarta geração. No entanto, só no final da década de 1990 que essa ideia ganhou força, graças ao desenvolvimento da noção de abordagens ágeis, como a metodologia de desenvolvimento de sistemas dinâmicos *Dynamic System Development Model* (DSDM)



(STAPLETON, 1997), o *Scrum* (SCHWABER; BEEDLE, 2001) e o *Extreme Programming* (BECK, 1999).


De acordo com Sommerville (2011, p. 40):

A insatisfação com essas abordagens pesadas da engenharia de software levou um grande número de desenvolvedores de software a proporem, na década de 1990, novos ‘métodos ágeis’. Estes permitiram que a equipe de desenvolvimento focasse no software em si, e não em sua concepção e documentação.

Segundo Sommerville (2011), nesta época, havia uma visão universal de que para se produzir o melhor software era necessário um planejamento cuidadoso, com o escopo rigorosamente formalizado, ferramentas CASE (em inglês, *Computer-Aided Software Engineering*), que são todas as ferramentas de computador que auxiliam em atividades de engenharia de software, para sustentar métodos de análise e projeto, além de possuir um rigoroso e controlado processo de desenvolvimento de software.

Entretanto, quando se desenvolve um sistema comercial, ou de pequeno e médio porte, por exemplo, a sobrecarga de atividades e processos é tão alta que domina o processo de desenvolvimento de software, ou seja, a aplicação acaba por ficar engessada. Isso se deve ao fato de se gastar mais tempo com a análise da metodologia de desenvolvimento do software, do que com o real desenvolvimento, testes e documentação. Outro fator importante diz respeito aos requisitos, já que eles se alteram e, desse modo, o sistema a ser entregue ao cliente pode não estar de acordo com as regras mais atuais, visto que se demorou demais para desenvolvê-lo, gerando possíveis retrabalhos desnecessários.

Após a década de 1990, as denominações para software ágil evoluíram, favorecendo a reação contrária aos métodos considerados tradicionais, como o modelo em cascata, que possui uma regulamentação complexa e rígida. Todas essas atividades favoreceram a desburocratização dentro



dos processos para o desenvolvimento de software, as quais eram consideradas lentas e ineficientes, além de irem na contramão do que pensavam os engenheiros de software.

De acordo com Sommerville (2011, p. 40):

Métodos ágeis, universalmente, baseiam-se em uma abordagem incremental para a especificação, o desenvolvimento e a entrega do software. Eles são mais adequados ao desenvolvimento de aplicativos nos quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento.

Inicialmente, os métodos ágeis foram denominados de “métodos leves”. No entanto, em 2001, Kent Beck e outros 16 notáveis profissionais da área de engenharia de software se reuniram e resolveram adotar um nome padrão, a partir daí surgiu o termo “métodos ágeis”. Além disso, nessa mesma reunião foi publicado o manifesto ágil, documento que reúne os princípios e práticas desta metodologia de desenvolvimento. Não suficiente, ainda, eles também formaram a *agile alliance*, ou aliança ágil, uma organização não lucrativa que promove e fomenta o desenvolvimento ágil.

De acordo com o manifesto, os membros declararam os valores da metodologia, sendo:

Estamos descobrindo maneiras melhores de desenvolver software fazendo-o nós mesmos e ajudando outros a fazê-lo. Através desse trabalho, passamos a valorizar:

- **Indivíduos e interações** mais que processos e ferramentas
- **Software em funcionamento** mais que documentação abrangente
- **Colaboração com o cliente** mais que negociação de contratos

- **Responder a mudanças** mais que seguir um plano

Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda. (MANIFESTO, [s.d.], [s.p.])

Ainda neste documento, eles elencaram os 12 princípios do manifesto ágil, sendo eles:

1. Nossa maior prioridade é satisfazer o cliente através da entrega contínua e adiantada de software com valor agregado;
2. Mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento. Processos ágeis tiram vantagem das mudanças visando vantagem competitiva para o cliente.
3. Entregar frequentemente software funcionando, de poucas semanas a poucos meses, com preferência à menor escala de tempo;
4. Pessoas de negócio e desenvolvedores devem trabalhar diariamente em conjunto por todo o projeto;
5. Construa projetos em torno de indivíduos motivados. Dê a eles o ambiente e o suporte necessário e confie neles para fazer o trabalho;
6. O método mais eficiente e eficaz de transmitir informações para e entre uma equipe de desenvolvimento é através de conversa face a face;
7. Software funcionando é a medida primária de progresso;
8. Os processos ágeis promovem desenvolvimento sustentável. Os patrocinadores, desenvolvedores e usuários devem ser capazes de manter um ritmo constante indefinidamente;
9. Contínua atenção à excelência técnica e bom design aumenta a agilidade;
10. Simplicidade—a arte de maximizar a quantidade de trabalho não realizado é essencial;
11. As melhores arquiteturas, requisitos e designs emergem de equipes auto-organizáveis;

12. Em intervalos regulares, a equipe reflete sobre como se tornar mais eficaz e então refina e ajusta seu comportamento de acordo. (MANIFESTO, [s.d.], [s.p.])

Por fim, a partir desses valores e princípios, os softwares atuais se baseiam, formando uma conjuntura sólida para a criação e entrega de aplicações a um determinado espaço de tempo.

1.3 Desenvolvimento ágil

O desenvolvimento ágil de software, ou *agile software development*, tem por objetivo o desenvolvimento de software com foco em minimizar os riscos por meio de desenvolvimento em um curto espaço de tempo ou iteração, de acordo com o exemplo da Figura 1.

A iteração mais comum compreende o desenvolvimento em fases curtas, partindo de uma a quatro semanas, envolvendo todas as tarefas necessárias para implantar uma determinada funcionalidade. Nesse sentido, considerando-se o período curto de cada iteração, a comunicação mantida entre os envolvidos do projeto deve ocorrer em tempo real, sendo, preferencialmente, tratada por meio verbal, embora documentada, ou “face-a-face”, visando minimizar entendimentos parciais ou errôneos. Entretanto, estabelecer um local físico, como uma sala, por exemplo, para que a equipe envolvida possa desenvolver o projeto é extremamente importante, a fim de que não haja desvio de foco do projeto.

Figura 1 – Etapas do desenvolvimento ágil




Fonte: adaptado de <https://www.lecom.com.br/blog/valores-da-metodologia-agile/>. Acesso em: 17 nov. 2020.

Como visto, o desenvolvimento ágil possui como foco o tempo em que algo é desenvolvido, não sendo esta metodologia imune a críticas, já que alguns afirmam baixa qualidade ou, até mesmo, inexistência de documentação do projeto.

Além disso, há outro fator diz respeito à interação humana “face-a-face”, que pode trazer informalidade nas definições, o que merece certa atenção, além de sempre contar com o acompanhamento do gestor do projeto para garantir a qualidade dos trabalhos, independentemente do processo utilizado.

De uma forma geral, os processos ágeis atendem aos projetos de software que, normalmente, apresentam alguns pontos em comum. Nesse contexto, de acordo com Sommerville (2011, p. 39), esses pontos são:

1. Os processos de especificação, projeto e implementação são intercalados. Não há especificação detalhada do sistema, e a documentação do projeto é minimizada ou gerada automaticamente pelo ambiente de programação usado para

- 
- implementar o sistema. O documento de requisitos do usuário apenas define as características mais importantes do sistema.
2. O sistema é desenvolvido em uma série de versões. Os usuários finais e outros stakeholders do sistema são envolvidos na especificação e avaliação de cada versão. Eles podem propor alterações ao software e novos requisitos que devem ser implementados em uma versão posterior do sistema.
 3. Interfaces de usuário do sistema são geralmente desenvolvidas com um sistema interativo de desenvolvimento que permite a criação rápida do projeto de interface por meio de desenho e posicionamento de ícones na interface. O sistema pode, então, gerar uma interface baseada na Web para um navegador ou uma interface para uma plataforma específica, como o Microsoft Windows.

2. Métodos ágeis

A existência de diversos métodos ágeis nos faz questionar se existe um melhor que o outro ou, ainda, se cada um possui uma forma, qual devo seguir para obter o melhor resultado? De acordo com Sommerville (2011, p. 40):

Embora esses métodos ágeis sejam todos baseados na noção de desenvolvimento e entrega incremental, eles propõem diferentes processos para alcançar tal objetivo. No entanto, compartilham um conjunto de princípios, com base no manifesto ágil, e por isso têm muito em comum.

Observe o quadro a seguir, ele demonstra os diferentes métodos ágeis, onde são instanciados de maneira diferente, isso faz com que se exista uma similaridade entre os métodos ágeis, possuindo cada qual sua finalidade.


Quadro 1 – Grupos de processos para organização do projeto

Princípios	Descrição
Envolvimento do cliente	Os clientes devem estar intimamente envolvidos no processo de desenvolvimento. Seu papel é fornecer e priorizar novos requisitos do sistema e avaliar suas iterações.
Entrega incremental	O software é desenvolvido em incrementos com o cliente, especificando os requisitos para serem incluídos em cada um.
Pessoas, não processos	As habilidades da equipe de desenvolvimento devem ser reconhecidas e exploradas. Membros da equipe devem desenvolver suas próprias maneiras de trabalhar, sem processos prescritivos.
Aceitar as mudanças	Deve-se ter em mente que os requisitos do sistema vão mudar. Por isso, projete o sistema de maneira a acomodar essas mudanças.
Manter a simplicidade	Focalize a simplicidade, tanto do software a ser desenvolvido quanto do processo de desenvolvimento. Sempre que possível, trabalhe ativamente para eliminar a complexidade do sistema.

Fonte: adaptado de Sommerville (2011, p. 40).

2.1 DSDM

O DSDM ou Metodologia de Desenvolvimento de Sistemas Dinâmicos é uma metodologia ágil, mas, originalmente, baseada em desenvolvimento rápido de aplicação ou *Rapid Application Development* (RAD). Por sua vez, ela é iterativa e incremental, além de enfatizar o envolvimento constante do usuário.



Além disso, essa metodologia tem como objetivo entregar softwares no tempo e com custo estimados, baseando-se no controle e ajuste de requisitos ao longo do desenvolvimento.

Ainda, o DSDM tem nove princípios formados por quatro séries e cinco pontos-chaves. Os nove princípios são:

- **Envolvimento:** ponto principal para a eficiência e eficácia do projeto. Decisões tomadas com mais precisão entre usuários e desenvolvedores.
- **Autonomia:** tomada de decisões importantes, as quais influenciam diretamente no projeto sem aprovação dos superiores.
- **Entregas:** entrega frequente de produtos, que devem ser anteriormente testados e revisados.
- **Eficácia:** a entrega de um sistema que seja capaz de auxiliar nas necessidades atuais da empresa e do negócio é mais importante do que o foco nas funcionalidades propriamente ditas.
- **Feedback:** contar com o retorno do cliente é de extrema relevância, já que o desenvolvimento é iterativo e incremental.
- **Reversibilidade:** alterações feitas no desenvolvimento são reversíveis.
- **Previsibilidade:** definição de escopo e requisitos de alto nível antes do início do projeto.
- **Ausência de testes no escopo:** testes são tratados fora do ciclo de vida do projeto.
- **Comunicação:** comunicação e cooperação de todos os envolvidos no projeto é fator fundamental para maior eficácia e eficiência no projeto.

2.2 FDD

A metodologia ágil *Feature Driven Development*, ou FDD, foi criada em Singapura na década de 1990, sendo utilizada pela primeira vez para o desenvolvimento de um sistema bancário internacional, o qual estava em uma situação bastante crítica, já que o prazo de entrega era apertado, sem contar o tamanho do projeto (SBROCCO; MACEDO, 2012, p. 99).


Como a FDD é uma metodologia bem objetiva e estruturada, ela é composta por cinco processos, baseando-se nos requisitos para iniciar o seu processo de produção. Comumente, o conhecimento tácito é explorado, já que fica restrito a cada pessoa e não está documentado, sendo a maior base de conhecimento dessa metodologia (PRIKLADNICKI; MILANI, 2014, p. 74).

Desse modo, podemos estabelecer duas fases no ciclo de vida da FDD, assim como nos relata Prikladnicki, Willi e Milani (2014, p. 74):

Linear: inicialização. A meta é conceber o produto a ser construído, numa visão inicial de sua estrutura e de suas funcionalidades, e criar um plano inicial de entregas incrementais, que servirá de guia durante a construção. Tipicamente essa fase consome de 2 a 4 semanas, dependendo do porte do projeto. Pode ser útil também para a realização de orçamentos e propostas, antes de se comprometer com o desenvolvimento propriamente dito. Também é conhecida como “Iteração 0” (zero) em outros Métodos Ágeis.

Iterativa: construção. A meta é entregar incrementos do produto de forma frequente, tangível e funcional, com qualidade para serem utilizados pelo cliente, se for preciso. Tipicamente, iterações são de 2 semanas, mas as durações devem ser adaptáveis e não necessariamente fixas durante o projeto. O progresso é reportado de maneira simples, clara e eficiente.


Logo, a FDD está estruturada em cinco processos, sendo eles:

- 
1. Desenvolver modelo abrangente: gerar um modelo de objetos (e/ou de dados) de alto nível, isso a partir de requisitos, análise orientada por objetos, modelagem lógica de dados e demais técnicas para o entendimento do negócio, tudo para guiar a equipe durante o projeto.
 2. Construir lista de funcionalidades: obter uma hierarquia de funcionalidades que representa o produto a ser construído, também conhecido como *product backlog*. Ela deverá se basear em três camadas típicas: áreas de negócio, atividades de negócio e passos automatizados da atividade (funcionalidades).
 3. Planejar por funcionalidade: gerar um plano de desenvolvimento, levando em consideração a complexidade e dependência das funcionalidades, além da prioridade e o valor para o negócio/cliente.
 4. Detalhar por funcionalidade: nesta fase, a equipe detalha os requisitos e outros artefatos para a codificação de cada funcionalidade, incluindo os testes. O objetivo é possuir um modelo de domínio mais detalhado, com esqueletos de códigos prontos para preenchimento.
 5. Construir por funcionalidade: cada esqueleto de código é preenchido, testado e inspecionado. Desse modo, espera-se como resultado um incremento do produto integrado ao repositório principal, com qualidade para ser usado pelo cliente.

2.3 XP

O XP, ou *eXtreme Programming*, é uma metodologia rigorosa e disciplinada, que possui como filosofia de desenvolvimento quatro valores: comunicação, feedback, simplicidade e coragem. Para Prikladnicki, Willi e Milani (2014, p. 38):

A Programação Extrema é a combinação de uma abordagem colaborativa, livre de desconfianças, com um conjunto de boas práticas de engenharia



de software que são eficientes por si só, individual e independentemente do contexto. Cada uma dessas práticas contribui para o aumento da qualidade do software e ajuda a garantir que o produto final agregue valor e atenda às necessidades do negócio. Alguns exemplos dessas práticas são: revisão de código, integração rápida, testes automatizados, feedback do cliente e design simples.

Para que o desenvolvimento do software possa ser efetuado de acordo com as diretrizes da metodologia XP, alguns valores devem ser compreendidos, respeitados e rigorosamente seguidos, sendo eles:

- a. Comunicação: entender o que o cliente deseja não é uma tarefa nada fácil, mas cabe ao responsável pelo projeto compreender as necessidades, desejos e receios, entendendo e organizando a real necessidade do cliente, visando apoiar o negócio. Deve haver um canal de comunicação entre a equipe de desenvolvimento e os usuários, visando o melhor entendimento possível.
- b. Feedback: saber se algo está sendo bem aceito ou não é de suma importância para o andamento dos trabalhos, já que pode cancelar ou não as atividades desenvolvidas. Quanto ao desenvolvimento de software, qualquer feedback é importante e decisivo para que se economize tempo e esforço no médio prazo, recomendado ser o mais cedo possível, sendo o responsável pelo projeto o promotor e divulgador dos resultados obtidos.
- c. Simplicidade: o fato de software ser simples não é sinônimo de algo mal feito ou mal planejado, pelo contrário, o básico e funcional atendem as expectativas do cliente, já que ele deseja que o seu problema seja resolvido, de um modo ou de outro. Com isso, após atender à necessidade inicial, implementações mais sofisticadas podem ser executadas.
- d. Coragem: fazer o certo sempre é a melhor alternativa, mas isso demanda atitudes positivas e corajosas, já que a real situação de algo pode não agradar o cliente, entretanto, mantê-lo informado com dados concretos e verídicos sempre é a melhor alternativa.

2.4 SCRUM

A metodologia *Scrum* tem sua origem na década de 1990, por meio de Jeff Sutherland e sua equipe, que tinham como objetivo inicial atender empresas de desenvolvimento de software, mas que, com o passar do tempo, viram seus conceitos sendo aplicados ao gerenciamento de projetos.

O *Scrum* se baseia em seis características: flexibilidade dos resultados, flexibilidade dos prazos, times pequenos, revisões frequentes, colaboração e orientação a objetos. Além disso, ele possui uma estrutura de funcionamento por ciclos, os quais denominam-se *sprints*, que são iterações de trabalho com duração variável, partindo de duas a quatro semanas (SBROCCO; MACEDO, 2012, p. 161).

Nessa metodologia, há um conjunto de práticas e regras que devem ser seguidas pela equipe, cada qual com suas atribuições e papéis, que são divididos em três: o *Product Owner*, que é o dono do projeto, possuindo como principal objetivo a garantia do retorno sobre o investimento; o *Scrum Master*, que tem a responsabilidade de remover os possíveis impedimentos que a equipe ou time possam ter, garantindo o uso do *framework* e protegendo o time de envolvimento externos prejudiciais; e, por fim, o *Team* ou Equipe *Scrum*, o grupo de pessoas que seleciona e desenvolve as funcionalidades do *product backlog*, que é a relação dos requisitos desejáveis pelo cliente. Observe a Figura 2, que retrata o ciclo mencionado acima.

Figura 2 – Visão geral da dinâmica da metodologia *Scrum*



Fonte: adaptado de Cohn (2008, [s.p.]).

Conforme nos relata Sbrocco e Macedo (2012, p.162):

[...] no início de cada projeto, clientes e desenvolvedores se reúnem com o objetivo de definir o Backlog¹ do produto (que representa a lista de requisitos). Esse período também é utilizado para escolher o *SCRUM* Master, eleito entre a equipe alocada para o projeto. Depois que o Product Backlog é definido, a equipe deve se dedicar à definição da Sprint Backlog, que contém uma lista de atividades que serão realizadas na próxima sprint, momento em que também são definidas as responsabilidades de cada membro do time. Após os desenvolvedores discutirem quais padrões serão adotados, as atividades de análise, codificação e testes devem se iniciar. [...] ao final de cada sprint um incremento do produto deve ser apresentado ao cliente para que o time obtenha uma retroalimentação.

Referências Bibliográficas

BECK, K. **Extreme programming explained**: embrace change. Reading, MA: Addison Wesley, 1999.

COHN, M. Mountain goat software. 2008. Disponível em: <http://www.mountaingoatsoftware.com/>. Acesso em: 18 out. 2020.



MANIFESTO para desenvolvimento ágil de software. Disponível em: <https://agilemanifesto.org/iso/ptbr/manifesto.html>. Acesso em: 19 out. 2020.

MARKETING LECOM. Metodologias ágeis–o que é e como aplicar? **Lecom**, 30 jul. 2018. Disponível em: <https://www.lecom.com.br/blog/valores-da-metodologia-agile>. Acesso em: 19 out. 2020.

PRIKLADNICKI, R.; WILLI, R.; MILANI, F. **Métodos ágeis para desenvolvimento de software**. Porto Alegre: Bookman, 2014.

SBROCCO, J. H. T. de C.; MACEDO, P. C. **Metodologias ágeis: engenharia de software sob medida**. São Paulo: Érica, 2012.

SCHWABER, K.; BEEDLE, M. **Agile software development with scrum**. Upper Saddle River, NJ: Prentice Hall, 2002.

SOMMERVILLE, I. **Engenharia de software**. 9. ed. São Paulo: Pearson Prentice Hall, 2011.

STAPLETON, J. **DSDM: dynamic systems development method**. Harlow, England: Addison-Wesley, 1997.

Paradigma Orientação a Objetos: histórico e conceitos

Autoria: Luís Otávio Toledo Perin

Leitura crítica: Lucas dos Santos Araujo Claudino



Objetivos

- Compreender o paradigma da orientação a objetos.
- Definir o que é programação orientada a objetos.
- Assimilar histórico e conceitos da programação orientada a objetos.



1. Paradigma da orientação a objetos


1.1 Introdução

Neste tema, você estudará sobre o paradigma da orientação a objetos, que será relatado a partir de seu histórico e conceitos. Por isso, fique atento e não perca nenhum dos próximos parágrafos, pois, eles estão recheados de informações importantes.

Você já parou para pensar como existem pessoas que fazem coisas maravilhosas e são chamadas de artistas? Este título, geralmente, é atribuído porque eles realizam verdadeiras obras de arte, se destacando naquilo que fazem. O pintor, por exemplo, se utiliza da tela e de tintas, além de suas habilidades e capacidade criativa, para criar belos quadros. O compositor musical, por sua vez, nas letras de uma canção reflete a mais pura realidade e sentimento sobre algo, sem contar o momento em que é cantada, sendo para muitas pessoas a coisa mais harmoniosa que existe.

Porém, você deve estar se perguntando o que tudo isso tem a ver com o paradigma da orientação a objetos, não é mesmo? Os exemplos citados anteriormente refletem a execução de situações bem pensadas, no entanto, a mesma regra pode ser aplicada ao paradigma, já que ele é um meio para que algo seja construído, ou seja, é uma ferramenta que, se bem utilizada, irá gerar resultados de excelente qualidade, sem contar na solução de um problema que ele irá suprir.

Entretanto, o simples ato de produzir software não basta. Para que isso ocorra com todos os padrões e referências nacionais e internacionais, o processo de aprendizagem da programação orientada a objetos demanda esforço e empenho, sem contar a prática. Tudo isso, agregado à correta estruturação dos temas, resultará em uma aplicação que



deverá resolver uma problemática e, conseqüentemente, agradar seu utilizador, ou seja, o usuário.


Nesse sentido, para Dall'Oglio (2007, p. 86):

A orientação a objetos é um paradigma que representa toda uma filosofia para construção de sistemas. Em vez de construir um sistema formado por um conjunto de procedimentos e variáveis nem sempre agrupadas de acordo com o contexto, como se fazia em linguagens estruturadas (Cobol, Clipper, Pascal), na orientação a objetos utilizamos uma ótica mais próxima do mundo real. Lidamos com objetos, estruturas que já conhecemos do nosso dia-a-dia e sobre as quais possuímos maior compreensão.

Desse modo, os próximos capítulos contextualizarão melhor a história e o conceito por trás do paradigma da orientação a objetos, comparando-o com o método tradicional, além de mostrar sua aplicabilidade nos dias de hoje.

1.2 Histórico

Antes de compreendermos o que exatamente o paradigma da orientação a objetos propõe, ou como ocorre seu nascimento até os dias de hoje, é importante entendermos o que significa paradigma. Segundo o dicionário Michaelis (2020), paradigma é “Algo que serve de exemplo ou modelo; padrão”, ou seja, uma referência que pode ou não ser seguida, a depender das afinidades e objetivos desejados. Já no universo da programação de computadores, trata-se de uma maneira, ou seja, um estilo de se programar (HOUAISS; FRANCO; VILLAR, 2001, p. 329). Desse modo, o paradigma nada tem a ver com ferramentas ou o tipo de linguagem utilizada, mas com a forma com que o problema será resolvido, já que o paradigma orientado a objetos faz com que o desenvolvedor modele o problema o mais próximo da realidade.



Nesse contexto, pare e reflita: desde a invenção do primeiro computador até o mais atual, você já notou como a programação tem evoluído? Pois bem, a programação das primeiras máquinas era feita por chaveamento em instruções binárias de máquina, utilizando-se de um painel frontal. Já os programas contavam com centenas de milhares de instruções, o que garantia seu funcionamento.

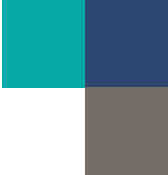
Mais adiante, com a evolução da máquina e da programação propriamente dita, a linguagem *assembly* foi inventada, permitindo que um programador pudesse manipular complexidades, sendo FORTRAN a primeira linguagem de alto nível difundida. Entretanto, ela apenas torna os programas mais fáceis e claros de serem entendidos se comparado com o *assembly*, não apresentando mudanças no seu estilo de programação.

Com o objetivo de resolver problemas da programação estruturada, como a complexidade em dar manutenção em grandes softwares, já que aplicações desse tipo tendem a se tornarem difíceis, a programação orientada a objetos passa a ser uma alternativa, alterando a maneira como se programava, além de garantir a manutenção do programa.

O paradigma orientado a objetos é o que reflete mais fielmente os problemas enfrentados atualmente na construção de um software. Nesse sentido, linguagens do tipo OO (orientação a objetos) são desenvolvidas para atuarem diretamente na solução dos problemas, sendo utilizado a simulação como grande aliado.

Para que você possa entender como a programação orientada a objetos (POO) está constituída hoje, além do que ela representa, você precisa compreender como se deu a sua criação e o que ocorreu durante todos esses anos.

A primeira linguagem de programação (LP) a utilizar o conceito de objetos em sua estrutura foi a Simula 67, passando a introduzir os



conceitos de classes e herança. Segundo Douglas (2015), a Simula 67 ocorreu na década de 1960, sendo criada por Kristen Nygaard e Ole-Johan Dahl, no centro Norueguês de Computação em Oslo. Já nas décadas seguintes, os avanços e aperfeiçoamentos continuaram para a implementação do modelo orientado a objetos, dando origem a linguagem Smalltalk. Por sua vez, o seu criador foi Alan Kay, sendo considerado um dos criadores desse novo paradigma.


Para Roque Neto (2018, p. 13):

Alan Kay, que atuava na Universidade de Utah naquela época, gostou do que viu na SIMULA. Consta que ele teria vislumbrado um computador pessoal que pudesse fornecer aplicações orientadas a gráficos e intuiu que uma linguagem como a SIMULA poderia oferecer bons recursos para leigos criarem tais aplicações.

Kay então resolveu “vender sua visão” à Xerox e no início dos anos 1970; sua equipe criou o primeiro computador pessoal, o Dynabook. A linguagem Smalltalk, que era orientada a objetos e também orientada a gráficos, foi desenvolvida para programar o Dynabook. Ela existe até hoje, embora não seja largamente usada para fins comerciais.

A partir da década de 1970, a ideia da POO se tornou mais sólida e começou a ser impulsionada. Mais tarde, já em 1980, Bjarne Stroustrup integrou tal paradigma a linguagem C, o que resultou no C++, sendo a primeira linguagem OO usada em massa (THE UNIVERSITY OF TENNESSEE, [s.d.]).

Por volta dos anos 1990, James Gosling liderou um grupo na Sun, o qual desenvolveu uma linguagem mais simples do C++, sendo denominada Java. Como a ideia em utilizá-la em aplicações de vídeo sobre demanda não vingou, ele decidiu voltá-la para aplicações de Internet, o qual obteve sucesso, sendo utilizada até os dias de hoje.



Desse modo, o mundo em que vivemos tem se tornado cada vez mais imediatista, não sendo diferente com a tecnologia. Com isso, softwares cada vez mais funcionais e dinâmicos, e que são entregues em um curto espaço de tempo, são requisitados pelo mercado. Nesse contexto, a engenharia de software é uma grande aliada, já que se utilizando de suas técnicas e práticas vem contribuindo significativamente para a tender a demanda solicitada.

1.3 Programação estruturada versus programação OO

Como visto anteriormente, a orientação a objetos parte da ideia de uma organização de software em termos de coleção de objetos discretos incorporando estrutura e comportamento próprios. Esta abordagem difere da organização voltada ao desenvolvimento tradicional de software, onde estruturas de dados e rotinas são desenvolvidas de forma apenas fracamente acopladas. Como entidade essencial, o paradigma OO tem como objeto o recebimento e envio de mensagens, além da execução de processamentos e a possibilidade de mudar ou não de estado. Assim, os problemas são resolvidos por meio de objetos, enviando mensagens a eles.

Já nos anos 1960, a programação estrutura foi criada, com seu método impulsionado por linguagens como C e Pascal, sendo possível desenvolver programas complexos com certa facilidade. No entanto, quando estamos falando de projetos pequenos a situação é outra, pois, o que muda quando ele atinge um certo tamanho, tornando-se extremamente difícil sua manutenção, além do alto custo para modificá-lo.

Para Dall'Oglio (2007, p. 86), a programação estruturada:

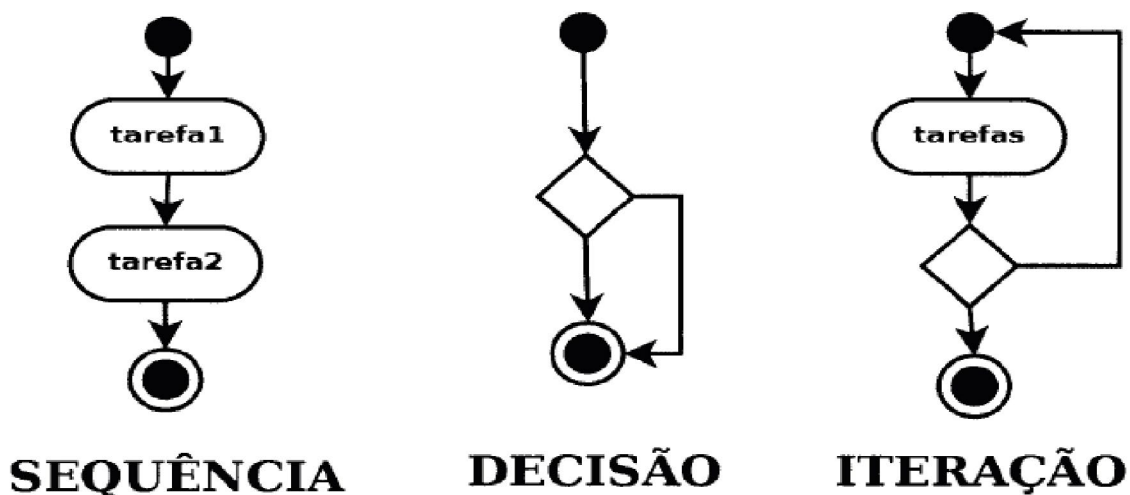
É baseada fortemente na modularização, cuja ideia é dividir o programa em unidades menores conhecidas por procedimentos ou funções. Essas

unidades menores são construídas para desempenhar uma tarefa bem específica e podem ser executadas várias vezes.

As funções também são importantes e podem receber parâmetros, o que poderá influenciar no processamento dos resultados internos, já que podem variar de acordo com os argumentos de entrada, que são os parâmetros e, por consequência, ocorrer a execução em diferentes circunstâncias. Segundo Dall'Oglio (2007), a esse tipo de prática, denominamos o conceito de reuso do software, característica muito importante da engenharia de software.


Neste tipo de programação, as unidades de código, que são as funções, se associam por meio de três mecanismos básicos, sendo eles: sequência, decisão e iteração, conforme a figura a seguir.

Figura 1 – Etapas do desenvolvimento ágil



Fonte: Dall'Oglio (2007, p. 87).

De acordo com a Figura 1, cada mecanismo tem sua importância dentro da codificação. Para Dall'Oglio (2007, p. 87):



A sequência representa os passos necessários para executar um programa em função de suas tarefas desempenhadas, por exemplo: 1) Leia os valores digitados; 2) Exiba: a soma na tela.

A decisão permite selecionar um determinado fluxo de processamento baseado em determinadas expressões lógicas, por exemplo: SE o valor digitado for maior que 20, execute a função `alerta_usuario()` SENÃO execute a função `grava_dados()`.

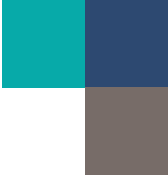
A iteração permite a execução repetitiva de um determinado bloco de comandos do programa. Esta execução geralmente tem um ponto de entrada representado por uma expressão lógica. O bloco de comandos é executado repetitivamente enquanto a expressão for verdadeira.

Agora que você já conhece a programação estruturada, vamos entender a programação orientada a objetos! Nela as funções e os dados são unidos, formando a entidade objeto. Os objetos, por sua vez, diferentemente dos dados passivos das linguagens tradicionais, podem atuar e suas ações são ativadas pelas mensagens enviadas e depois processadas por funções chamadas métodos.

Uma coleção de objetos similares é chamada classe, podendo conter atributos (variáveis) e funções (métodos) para manipular esses atributos (DALL'OGGIO, 2007). Uma instância de uma classe é um objeto. A maior abstração é conseguida pelo uso da hierarquia de classes. Para reutilizar métodos, as hierarquias de classe recorrem ao mecanismo exclusivo da OO que é a herança ou hereditariedade.

Durante a execução de programa OO ocorrem três eventos, sendo eles:

- a. Criação de objetos conforme a necessidade.
- b. As mensagens movimentam-se de um objeto para outro.
- c. Quando os objetos não são mais necessários, eles são apagados e a área na memória recuperada.



Além disso, princípios como encapsulamento, abstração, subtipos, herança e seleção dinâmica de métodos são itens básicos quando trabalhamos com POO (GABBRIELLI; MARTINI, 2010). Esses conceitos são considerados essenciais quando falamos no desenvolvimento de bons programas e sempre devem ser trabalhados a favorecer os aspectos de desempenho, segurança e organização da aplicação.

O compartilhamento de recursos também é uma característica da OO, podendo ser aplicada em diversos níveis distintos. A herança de estruturas de dados e comportamento fazem com que estruturas comuns sejam compartilhadas entre diversas classes similares derivadas, mas sem redundância. O compartilhamento de código por meio de herança é outra vantagem, sendo evidente a clareza conceitual de reconhecer que operações sobre objetos diferentes podem ser a mesma coisa, o que reduz o número de casos distintos que devem ser entendidos e analisados, sem contar a economia de código que é proporcionado.

Com isso, a OO permite mais do que um simples compartilhamento de informação dentro de um projeto, oferecendo, também, a possibilidade de reaproveitar projetos e código em projetos futuros. O grande artifício é utilizar-se de ferramentas como abstração, encapsulamento, polimorfismo e herança, além da definição de bibliotecas de elementos reusáveis, importante para reuso entre projetos.

Contudo, não pense que a OO é uma fórmula mágica para se reaproveitar tudo e construir projetos mais rapidamente, pois, tudo demanda planejamento e disciplina, podendo assim pensar em termos genéricos, os quais não focam apenas uma aplicação específica e, por isso, podem englobar outras.

Veja o Quadro 1, ele sintetiza as diferenças entre a programação estruturada e a orientada a objetos.

Quadro 1 – Diferença programação estruturada X programação OO

Programação estruturada	Programação OO
Procedimentos ou funções	Métodos
Variáveis e dados	Atributos
Chamadas a procedimentos ou funções	Mensagens
Chamadas a procedimentos ou funções	Classes
Hereditariedade	-
Chamadas sob o controle do sistema	Chamadas sob o controle do programador

Fonte: elaborado pelo autor.

2. Programação OO

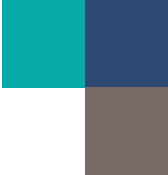
2.1 Abstração

A abstração tem por objetivo focar nos aspectos essenciais inerentes a uma entidade e ignorar propriedades “acidentais”. No desenvolvimento, isso representa concentrar-se no que um objeto é e faz antes de se decidir como ele será implementado. Isso dá liberdade para que o programador tome decisões de desenvolvimento ou de implementação apenas quando há um melhor entendimento do problema a ser resolvido.

De acordo com Roque Neto (2018, p. 14):

A abstração está relacionada à definição precisa de um objeto. Essa definição inclui sua identificação (nome), suas características (ou propriedades) e o conjunto de ações que ele desempenha.

Apesar de muitas linguagens de programação suportarem o conceito de abstração de dados, a junção com polimorfismo e herança na orientação



a objetos é muito mais eficiente e poderosa. Assim, quando falamos em modelagem na linguagem orientada a objetos devemos ter como mais importante a identificação de abstrações que melhor descrevem o domínio do problema. Isso proporciona a identificação de aspectos semelhantes quanto à forma e ao comportamento que permitem a organização das classes.

Para exemplificar, Roque Neto (2018, p. 14) expõe:

Tomemos como exemplo um cachorro: o objeto cachorro deve ser único e não poderá ser repetido. Nesse ponto, o objeto já tem identidade definida. Sua caracterização se dá pela cor do pelo, peso, raça e por aí vai. Por fim, as ações que ele é capaz de desempenhar incluem latir, farejar, pular etc. Pronto! Conseguimos abstrair o objeto cachorro e o temos perfeitamente definido.

2.2 Objetos e classes

Um objeto é definido como uma entidade concreta com limites e significados bem definidos, ou seja, é uma entidade do mundo real que é representado na aplicação a ser desenvolvida. Nesse sentido, um objeto deve possuir:

- a. Identidade: especificidade que o distingue dos demais objetos.
- b. Estado: conjunto de valores de seus atributos em um determinado instante.
- c. Comportamento: reação apresentada às solicitações feitas por outros objetos com os quais se relaciona.

Logo, os objetos apresentam dois propósitos: gerar a compreensão do mundo real e suportar uma base prática para uma implementação computacional. Desse modo, é interessante mencionar que não existe um jeito certo de separar um problema em objetos, já que o projetista e a natureza do problema é quem vão julgar a melhor forma de trabalho.

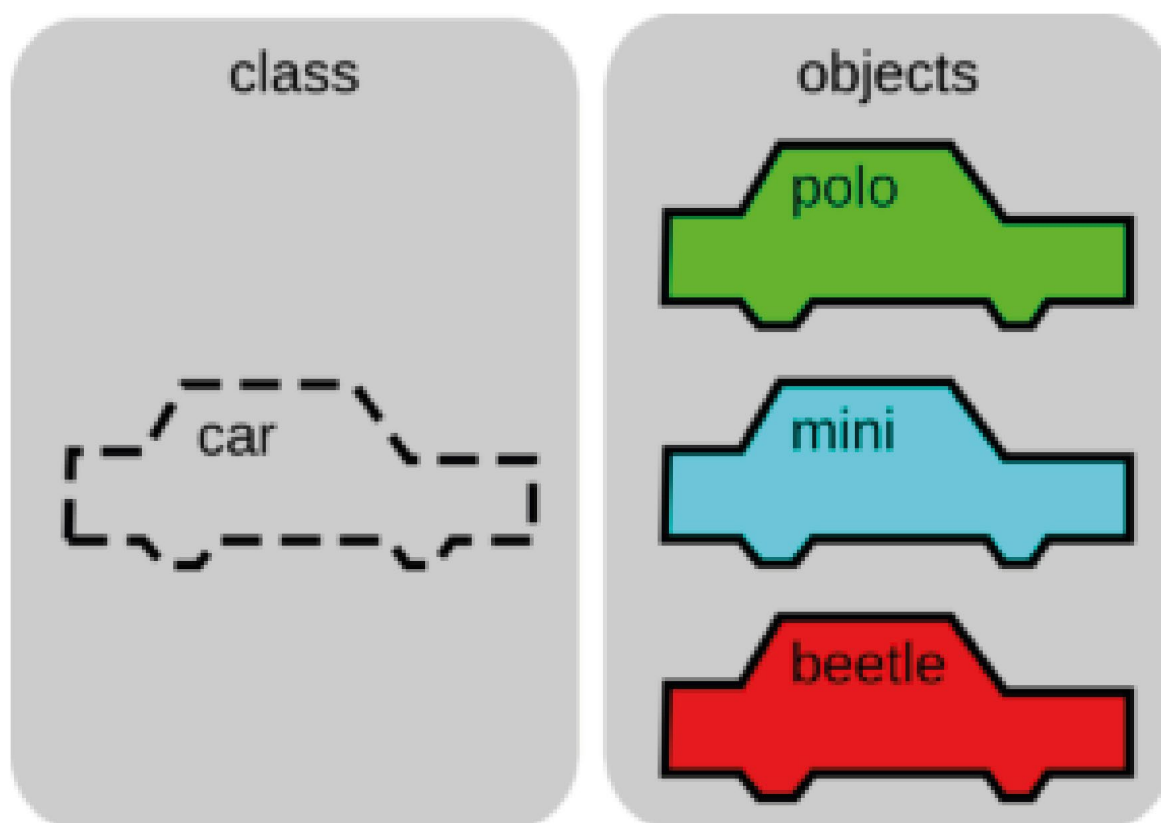
A classe, por sua vez, descreve um grupo de objetos com propriedades (atributos) e comportamento (operações) similares, relacionamentos com outros objetos e uma semântica comum.

Nesse contexto, para Felix (2016, p. 5):

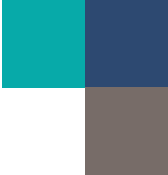
Em resumo: objetos correspondem a elementos da vida real e classes agrupam esses objetos. Assim, quando falamos de carro, e só de carro, estamos falando de uma classe. Isso porque não especificamos as características do veículo, então, várias características podem estar contidas – o que importa é que se trata de um carro.

Observe o exemplo da Figura 2, onde é evidenciado a classe e o objeto. Como classe temos o Carro e como objeto as suas variações, sendo, neste caso, um polo, um mini e um beetle.

Figura 2 – Classe e objeto



Fonte: adaptada de Henrique (2016, [s.p.]).



Deste modo, devemos ter em mente que uma diferença fundamental é que um objeto constitui uma entidade concreta com tempo e espaço de existência, já a classe é somente uma abstração, ou seja, um tipo de dado definido pelo usuário.

2.3 Atributos

Segundo Felix (2016), um atributo é a propriedade (ou dado) da classe. Basicamente, ele é um dado ou informação de estado, possuindo cada objeto em uma classe seu próprio valor. Por exemplo, um objeto da classe PessoaFisica tem um nome e uma idade, desta forma, cada objeto da classe PessoaJuridica também tem um nome e uma idade a partir da sua data de fundação; estes seriam atributos comuns das duas classes.

2.4 Operações e métodos

Uma operação é uma função ou transformação que pode ser aplicada a objetos em uma classe, como abrir, salvar e imprimir são operações que podem ser aplicadas a objetos da classe Arquivo. Todos os objetos em uma classe compartilham as mesmas operações.

Além disso, é possível que uma mesma operação seja aplicada a diversas classes diferentes. Neste caso, chamamos de operação polimórfica, ou seja, ela pode assumir distintas formas em classes diferentes.

Já o método é a implementação de uma operação para uma classe. A operação imprimir é exemplo disso, já que pode ser implementada de forma distinta, dependendo da maneira como o arquivo é impresso, podendo conter apenas texto não formatado, sendo um processador de texto ou um arquivo binário. Com isso, todos esses métodos executam a mesma operação, que é imprimir o arquivo, mas cada método é implementado por um código diferente.

2.5 Encapsulamento

De acordo com Felix (2016), o encapsulamento torna possível que as variáveis da classe e seus métodos sejam agrupados em conjuntos, segundo o seu grau de relação. Desse modo, o seu uso permite que a implementação de um objeto possa ser modificada sem afetar as aplicações que utilizam este objeto. Por sua vez, melhoria de desempenho, correção de erros, mudança de plataforma de execução, são alguns dos exemplos para essa modificação.

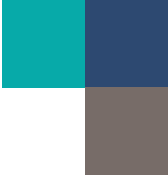
O conceito de “ocultação de informação” é um grande aliado, pois, ele pode ocultar detalhes de uma estrutura complexa, a qual poderia interferir durante o processo de análise. Outro ponto forte é a disponibilização do objeto com toda a sua funcionalidade, mas sem saber como é o seu funcionamento interno ou a armazenagem dos dados recuperados, por exemplo.

Para que você compreenda melhor, podemos utilizar o exemplo do telefone, em que ao discar o número e, posteriormente, a pessoa do outro lado atender, uma série de processos foram necessários para que isso pudesse ocorrer, como a linha telefônica ativa, o fio que transmite os dados e a central telefônica. Porém, você não precisa saber de tudo isso, basta apenas que a ligação ocorra, sem que saiba de todo o processo interno.

2.6 Herança

A herança é o mecanismo que uma classe pode estender a uma classe-mãe (superclasse) e se aproveitar de seus métodos e atributos. Para Felix (2016, p. 93):

Mecanismo que permite a criação de uma nova classe que estende uma outra já definida pelo programador, o que torna possível a reutilização de dados e comportamentos da classe ancestral.



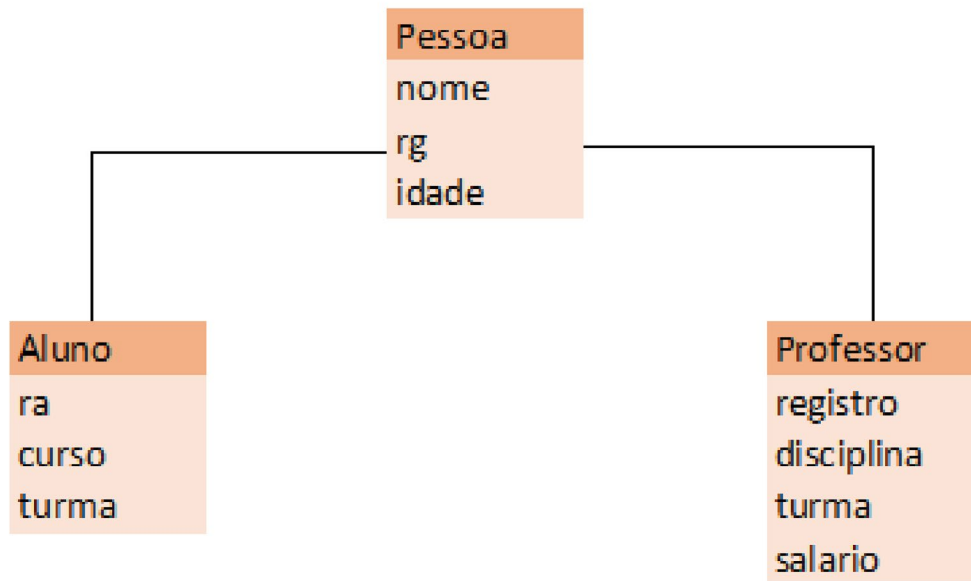
O compartilhamento de similaridades é algo inerente na herança e generalização, sendo abstrações poderosas entre classes, preservando ao mesmo tempo suas diferenças. A generalização é o relacionamento entre uma classe e uma ou mais versões refinadas (especializadas) desta classe. Com outras palavras, a classe sendo refinada é chamada de superclasse ou classe base, enquanto a versão refinada da classe é chamada de subclasse ou classe derivada. Além disso, atributos e operações comuns a um grupo de classes derivadas são colocadas como atributos e operações da classe base, sendo compartilhados por cada classe derivada.

Nesse sentido, de acordo com Felix (2016, p. 98):

Herança é uma característica do paradigma de orientação a objetos por meio da qual um objeto filho herda características e comportamentos do objeto mãe. É pela implementação da herança que conseguimos estabelecer relação entre uma classe mãe e uma classe filha, de modo que a segunda se torne uma extensão da primeira, herdando diretamente os métodos e os atributos públicos da classe mãe.

Observe a Figura 3, onde podemos exemplificar a herança na classe Pessoa, Nome, RG e idade são atributos da classe base Pessoa, que serão herdados pelas classes derivadas Aluno e Professor.

Figura 3 – Herança



Fonte: elaborada pelo autor.

2.7 Polimorfismo

O polimorfismo é a capacidade de uma variável se referir durante sua execução a objetos de diversas classes, ou seja, o usuário pode enviar uma mensagem genérica e deixar os detalhes de implementação para o objeto receptor. Uma mensagem de impressão, por exemplo, pode acessar métodos diferentes, isso se for enviada a uma figura ou para um texto.

Para que ele possa ser aplicado, é necessária uma relação de herança entre classes, ou seja, utilizar o mesmo método em situações diferentes, de acordo com a classe em que foi definido. Assim, chamamos de sobrecarga de método (*overloading*) a execução do polimorfismo em uma mesma classe.

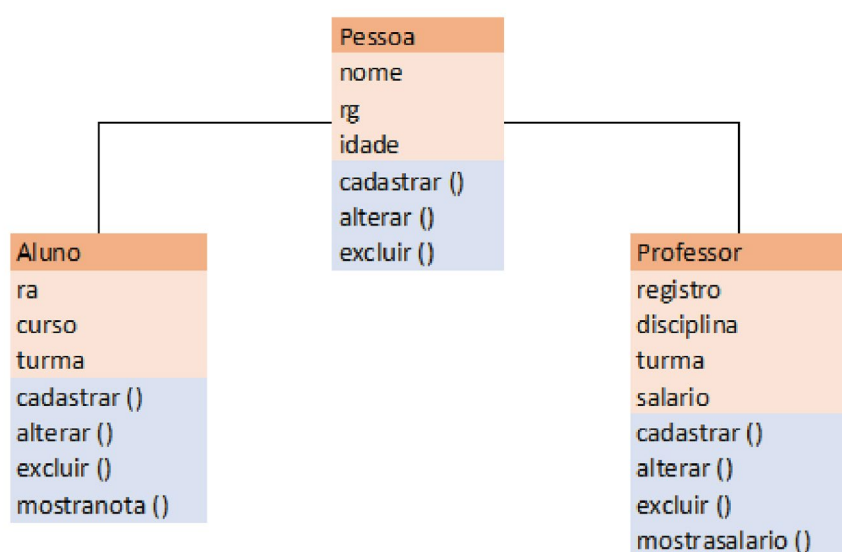
Para Santos (2003, p. 140):

O polimorfismo permite a manipulação de instâncias de classes que herdam de uma mesma classe ancestral de forma unificada: podemos

receber métodos que recebam instâncias de uma classe C, e os mesmos métodos serão capazes de processar instâncias de qualquer classe que herde da classe C, já que qualquer classe que herde de C é-um-tipo-de C.

Observe a Figura 4, onde os métodos cadastrar, alterar e excluir são os mesmos herdados da classe Pessoa, mas as operações entre Aluno e Professor são bastante distintas, em função de seus dados associados.

Figura 4 – Polimorfismo



Fonte: elaborada pelo autor.


Referências Bibliográficas

DALL’OGLIO, P. **PHP: programando com orientação a objetos**. São Paulo. Novatec, 2007.

DOUGLAS, M. Você sabe, com certeza, o que é orientação a objetos? **Object Pascal Programming**, 2015. Disponível em: <http://objectpascalprogramming.com/posts/o-que-e-orientacao-a-objetos/>. Acesso em: 31 ago. 2017.

FELIX, R. **Programação orientada a objetos**. São Paulo. Pearson Education do Brasil, 2016.

GABBRIELLI, M.; MARTINI, S. **Programming languages: principles and paradigms**. [S. l.]: Springer, 2010.



HENRIQUE, J. POO: o que é programação orientada a objetos? **Alura**, 23 out. 2019. Disponível em: <https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos>. Acesso em: 29 out. 2020.

HOUAISS, A.; FRANCO, F. M. M.; VILLAR, M. S. **Dicionário houaiss da língua portuguesa**. São Paulo: Objetiva, 2001.

MICHAELIS. Dicionário brasileiro da língua portuguesa. Disponível em: <https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/paradigma>. Acesso em: 29 out. 2020.

ROQUE NETO, M. **Programação orientada a objetos**. Londrina: Editora e Distribuidora Educacional S.A., 2018.

SANTOS, R. **Introdução à programação orientada a objetos usando Java**. Rio de Janeiro: Campus, 2003.

THE UNIVERSITY OF TENNESSEE. A brief history of object-oriented programming. Disponível em: <http://web.eecs.utk.edu/~huangj/CS302S04/notes/oo-intro.html>. Acesso em: 31 ago. 2017.

UML: conceitos, diagramas e ferramenta CASE


Autoria: Luís Otávio Toledo Perin

Leitura crítica: Lucas dos Santos Araujo Claudino



Objetivos

- Definir o que é UML.
- Compreender conceitos e principais diagramas da UML.
- Apresentar ferramenta CASE para UML.



1. UML


1.1 Introdução

Neste tema, abordaremos sobre a *Unified Modeling Language* (UML), onde você poderá entender o seu conceito, além dos principais diagramas que a norteiam. Além disso, também apresentaremos uma ferramenta CASE para auxiliar no momento da criação. Então, fique ligado nos conteúdos apresentados e não perca nenhum dos próximos parágrafos!

Já percebeu que quando falamos no processo de desenvolvimento de sistemas, ou tudo que está ligado a ele, a engenharia de software está presente? No caso da modelagem, isso não é diferente! Ela tem a função de construir modelos que expressem ou demonstrem as características ou comportamento de um determinado software ou função a ser desenvolvida.

Essa transferência do mundo real para o conceitual é necessária e bastante requisitada, pois, o profissional que está à frente do projeto precisa ter pleno domínio sobre aquilo que está sendo construído, sem contar os outros diversos segmentos que precisam compreender com clareza todo o processo, gerando a menor quantidade de dúvidas ou incertezas sobre o que deve ser analisado.

Nesse sentido, a UML (*Unified Modeling Language*), ou Linguagem de Modelagem Unificada, foi criada. Desse modo, ela é uma linguagem do tipo visual, sendo baseados no paradigma de orientação a objetos, com o intuito de modelar softwares. Segundo Guedes (2011), como seu domínio é amplo, ele pode ser aplicado em todas as partes do ciclo de desenvolvimento do produto, garantindo assim um melhor detalhamento em cada parte específica da análise e/ou desenvolvimento.



Portanto, os próximos capítulos serão surpreendentemente interessantes e irão tratar da UML e de seus principais diagramas. Não bastando, apresentaremos uma super ferramenta CASE, com o objetivo de auxiliar na construção e manipulação de tais artefatos. Por isso, não perca!


1.2 Criação da UML

Como já visto anteriormente, a UML, ou Linguagem de Modelagem Unificada, é baseada em projetos orientados a objetos. De acordo com o próprio nome, trata-se de uma linguagem de modelagem e não um método, metodologia, processo ou linguagem de programação.

Por se tratar de uma linguagem padrão de notação, ou seja, ela se utiliza de instrumentos para especificar, visualizar e documentar os elementos de um sistema orientado a objetos – OO, sua participação é muito importante em projetos, já que:

1. Como linguagem, ela tem o objetivo de expressar decisões de projeto não tão claras ou evidentes durante a codificação.
2. Por meio de sua semântica, ela permite obter decisões estratégicas para o projeto.
3. É passível de ser compreendida pelas pessoas e manipulada pela máquina.
4. Não tem vínculo com linguagens de programação ou métodos de desenvolvimento, tornando-a independente e acessível a qualquer projeto.

No entanto, para que todo esse desenvolvimento e aprimoramento da linguagem de modelagem pudesse ocorrer foi necessário uma série de aperfeiçoamentos ao longo dos anos, bem como uma unificação de tantos outros métodos existentes, os quais coexistiam com notações muitas vezes conflitantes entre si. A década de 1990 é prova disso,



conhecida como “guerra dos métodos”, sendo os métodos abaixo os mais conhecidos:

1. OMT (*Object Modeling Technique*), de Rumbaugh.
2. Método de Booch.
3. OOSE (*Object Oriented Software Engineering*), de Jacobson.

Para Guedes (2011, p. 19):

A UML surgiu da união de três métodos de modelagem: o método de Booch, o método OMT (*Object Modeling Technique*) de Jacobson, e o método OOSE (*Object-Oriented Software Engineering*) de Rumbaugh. Estes eram, até meados da década de 1990, os métodos de modelagem orientada a objetos mais populares entre os profissionais da área de desenvolvimento de software. A união desses métodos contou com o amplo apoio da Rational Software, que a incentivou e financiou.

A da união do método de Booch ao OMT de Jacobson culminou no Método Unificado, no final de 1995. Posteriormente, Rumbaugh juntou-se a Booch e Jacobson na Rational Software, sendo seu método OOSE incorporado à nova metodologia. Segundo Guedes (2011), conhecidos como “Os Três Amigos”, Booch, Jacobson e Rumbaugh, lançaram em 1996 a primeira versão da UML.

Com a ajuda de inúmeras empresas da área de modelagem e desenvolvimento de software, a linguagem pode ser aperfeiçoada e melhorada com o passar dos anos. Já em 1997, UML foi adotada pela OMG (*Object Management Group* ou Grupo de Gerenciamento de Objetos), como a linguagem-padrão de modelagem (GUEDES, 2011).

Mais adiante, em julho de 2005, ocorreu o lançamento da versão 2.0 do UML, estando hoje na 2.5. Ao longo desse tempo, alterações diagramáticas foram efetuadas, como no diagrama de atividades e no conjunto de diagramas de interação, bem como no diagrama de estrutura composta e classe estruturada.

1.3 Organização UML

Assim como em qualquer outro processo, seja ele na área da tecnologia ou não, é necessário que uma forma de organização seja utilizada para o bom andamento dos trabalhos. Nesse sentido, a UML possui um conjunto de técnicas de notação gráfica para criar modelos visuais de software, somando-se a técnicas de modelagem de dados, negócios, objetos e componentes. Isso garante que ela seja uma linguagem de modelagem unificada, comum e amplamente utilizável.

O objetivo da UML é mostrar ao responsável e ao time alguns aspectos que merecem reflexão e direcionam para a construção da aplicação, como “o que deve se fazer”, “como fazer”, “quando fazer” e “porque deve ser feito”. Ela não é responsável por mostrar que tipo de implementação deve ser feita, no sentido de direcionar quanto a detalhes do processo de desenvolvimento.

De acordo com Medeiros (2004, p. 10):

Percebemos, logo à primeira vista, que a UML não nos indica como devemos fazer um software. Ela indica apenas as formas que podem ser utilizadas para representar um software em diversos estágios de desenvolvimento.

Utilizando a UML, conseguimos ‘pensar’ um software em um local e codificá-lo em outro. É evidente que alguma outra comunicação adicional se fará necessária, porém deve ser minimalista. [...]

Nesse sentido, a UML se torna peça fundamental para uma boa modelagem daquilo que se quer construir, ou seja, do software. Por mais habilidoso e experiente que o profissional seja, planejar e prever situações é a melhor opção para se pensar e dinamizar todo o produto a ser construído, garantindo assim a harmonia e assertividade das informações.



Segundo Guedes (2011, p. 20):

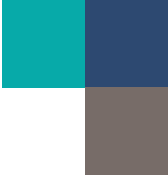
Na realidade, por mais simples que seja, todo e qualquer sistema deve ser modelado antes de se iniciar sua implementação, entre outras coisas, porque os sistemas de informação frequentemente costumam ter a propriedade de “crescer”, isto é, aumentar em tamanho, complexidade e abrangência.

Logo, sistemas devem ser construídos para sempre estarem em constante evolução, ou seja, nunca param de atualizar e reinventar aquilo que já faziam anteriormente, mas passam a fazer de uma maneira diferente. Essa evolução pode ser oriunda de diversos fatores, como:

1. Os clientes desejam constantemente modificações ou melhorias no sistema;
2. O mercado está sempre mudando, o que força a adoção de novas estratégias por parte das empresas e, conseqüentemente, de seus sistemas;
3. O governo seguidamente promulga novas leis e cria novos impostos e alíquotas ou, ainda, modifica as leis, os impostos e alíquotas já existentes, o que acarreta a manutenção no software. (GUEDES, 2011, p. 21)

Com tantas evoluções e constantes modificações para atender as demandas, é importantíssimo que tudo seja documentado de forma a conter um grande detalhamento, bem como seja preciso e receba atualizações daquilo que foi modificado. De acordo com Guedes (2011), isso garantirá a facilidade e rapidez durante o seu gerenciamento, além de garantir que novos erros não sejam criados em decorrência dos antigos.

Assim, modelar um sistema é uma forma bastante prática e eficiente para se adiantar um resultado que se espera com o produto, em parte ou em sua totalidade. Por isso, a UML dispõe de modelos ou diagramas



(representações gráficas do modelo parcial de um sistema) para serem utilizados, de forma combinada, com o objetivo de deslumbrar todas as visões e aspectos do sistema.

Quanto a sua organização, os diagramas da UML estão divididos em estruturais e comportamentais, sendo eles:

1. Diagramas estruturais:

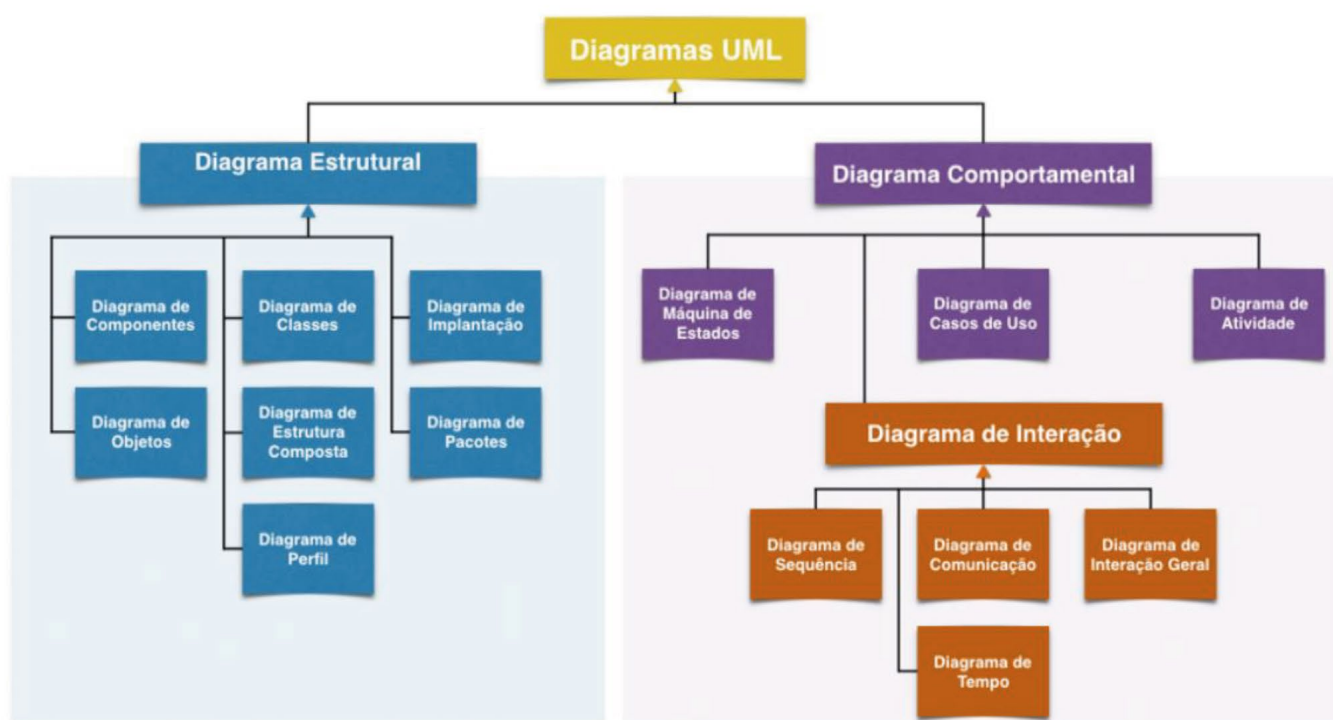
- a. Diagrama de classe.
- b. Diagrama de objeto.
- c. Diagrama de componentes.
- d. Diagrama de implantação.
- e. Diagrama de pacotes.
- f. Diagrama de estrutura composta.
- g. Diagrama de perfil.

2. Diagramas comportamentais:

- a. Diagrama de casos de uso.
- b. Diagrama de máquina de estados.
- c. Diagrama de atividade.
- d. Diagramas de interação, que se dividem em:
 - a. Diagrama de sequência.
 - b. Diagrama de interação geral.
 - c. Diagrama de comunicação.
 - d. Diagrama de tempo.

A partir da UML 2.4 existe um total de 14 diagramas, assim como demonstrado na figura a seguir.

Figura 1 – Diagramas UML



Fonte: Vieira (2015, [s.p.]).

Nesse sentido, é importante mencionar que os diagramas estruturais são responsáveis pelos aspectos estáticos do sistema, ou seja, constituídos de estrutura que permanece inalterada por não levar o tempo em consideração na sua representação. Já os diagramas comportamentais se baseiam em aspectos dinâmicos do sistema e seu relacionamento com o decorrer do tempo. Além disso, sua subcategoria, os de interação, também, possui esse dinamismo entre os objetos do sistema e as trocas de mensagens entre eles.

Desde a sua criação, a UML tem passado por evoluções e aprimoramentos, como pode ser visto na Tabela 1. Em sua versão 1.X eram cerca de 9 diagramas, passando para 14 a partir da versão 2.4. Porém, essa atualização não se resume apenas em novos diagramas, há atualização também em sua estrutura e documentação, a qual está sempre atenta as necessidades do mercado.


Tabela 1 – Comparativo versões UML

UML 1.X	UML 2.4
Componentes	Componentes
Objetos	Objetos
Classes	Classes
----- -----	Estrutura Composta
----- -----	Perfil
Implantação	Implantação
----- -----	Pacotes
Máquina de Estados	Máquina de Estados
Casos de Uso	Casos de Uso
Atividade	Atividade
Sequência	Sequência
Comunicação	Comunicação
----- -----	Tempo
----- -----	Interação Geral

Fonte: elaborada pelo autor.

1.4 Diagramas UML

A UML possui uma vasta quantidade de diagramas em sua estrutura, os quais são uma representação gráfica de um conjunto de elementos, como: classes, interfaces, colaborações, componentes, nós, entre outros. Eles são utilizados para analisar o sistema sob diferentes óticas, facilitando o processo de desenvolvimento da aplicação.



Para Guedes (2011, p. 30):

O objetivo disso é fornecer múltiplas visões do sistema a ser modelado, analisando-o e modelando-o sob diversos aspectos, procurando-se, assim, atingir a completitude da modelagem, permitindo que cada diagrama complemente os outros.

1.4.1 Diagrama de casos de uso

Um dos diagramas mais usuais da UML, o diagrama de casos de uso é utilizado para fazer a modelagem do contexto de um sistema e a modelagem de seus requisitos. Segundo Silva (2007), ele especifica um conjunto de funcionalidades, por meio do elemento sintático “casos de uso”, e os elementos externos que interagem com o sistema, por meio do elemento sintático “ator”. Além de casos de uso e atores, também, há relacionamentos de dependência, generalização e associação.

Para Guedes (2011, p. 30):

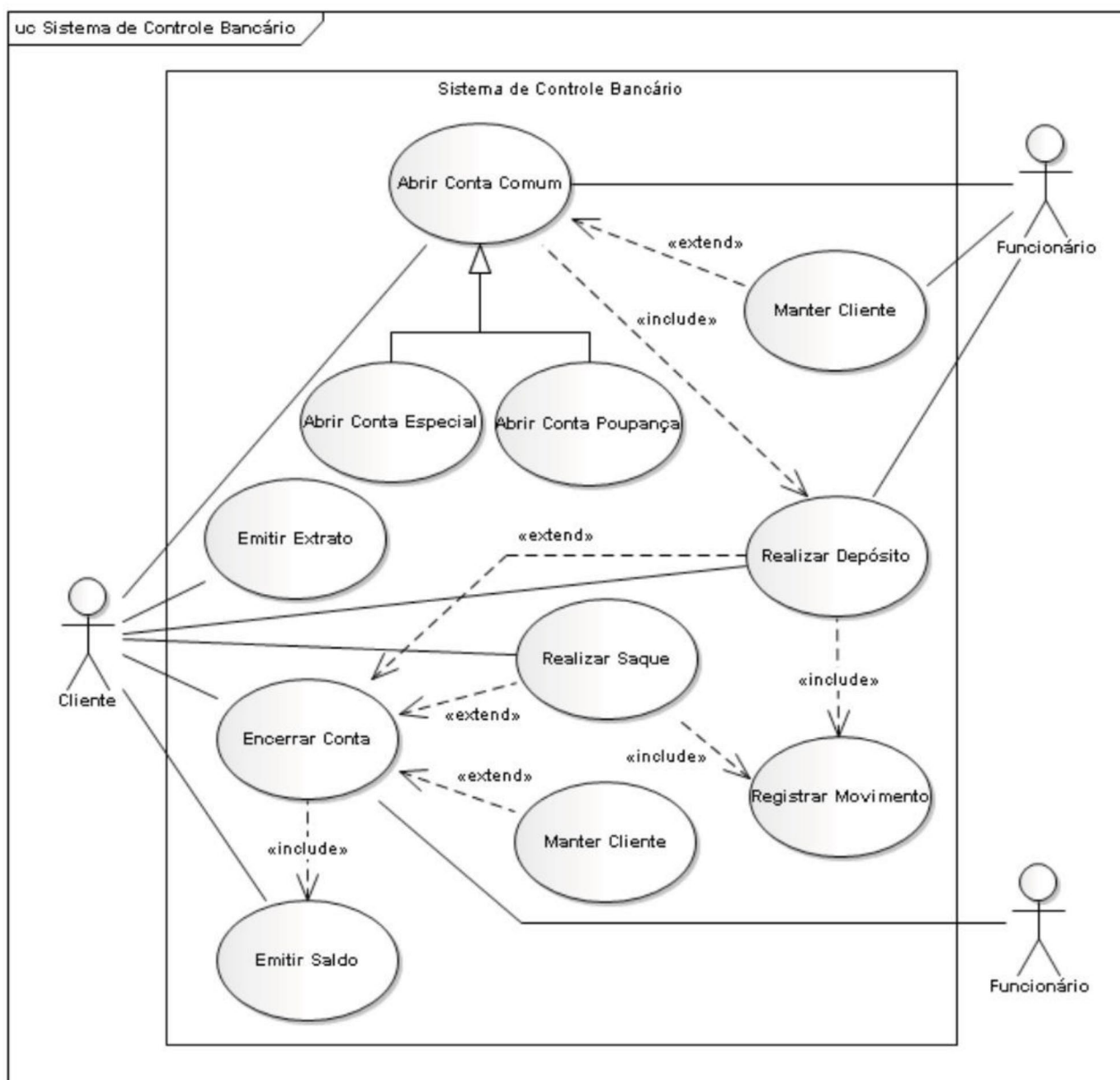
Apresenta uma linguagem simples e de fácil compreensão para que os usuários possam ter uma ideia geral de como o sistema irá se comportar. Procura identificar os atores (usuários, outros sistemas ou até mesmo algum hardware especial) que utilizarão de alguma forma o software, bem como os serviços, ou seja, as funcionalidades que o sistema disponibilizará aos atores, conhecidas nesse diagrama como casos de uso.

Geralmente, como sua criação ocorre na fase de análise de requisitos, os requisitos funcionais e não funcionais que irão auxiliar em seu processo de construção. Enquanto o primeiro expressa o comportamento da aplicação, ou seja, as informações de entrada, processamento e saída; o segundo se preocupa em estruturar aspectos qualitativos do software, como performance, segurança, perspectiva do usuário e usabilidade.

Por possuir uma linguagem simples e acessível, ou seja, não restrita a termos técnicos e ao setor de programação, ela serve como contrato


entre a empresa fornecedora e o cliente, além de auxiliar na validação daquilo que usuário deseja para o software, conforme pode ser visto na Figura 2.

Figura 2 – Exemplo diagrama casos de uso – sistema de controle bancário



Fonte: Guedes (2011, p. 31).

Quanto a seus componentes, os quais auxiliam na descrição do cenário para mostrar as funcionalidades do sistema do ponto de vista do usuário, pode-se destacar:

- 
1. Atores: representado por um boneco, ele simboliza um usuário do sistema, podendo ser humano ou outra interação computacional.
 2. Casos de uso: define uma grande função sistêmica, podendo ser estruturado em outras funções e, por isso, pode ser estruturado. Desse modo, sua representação gráfica é uma elipse.
 3. Relacionamentos: determinam o direcionamento do sistema, sendo:
 - a. Associações entre atores e casos de uso: define uma funcionalidade do sistema do ponto de vista do usuário.
 - b. Generalizações entre atores: os casos de uso do ator A, também são casos de uso do ator B, mesmo este segundo possuindo seus próprios casos de uso.
 - c. *Include* (entre casos de uso): entre um caso de uso A para um caso de uso B, indica que B é necessário para o comportamento de A.
 - d. *Extend* (entre casos de uso): entre um caso de uso B para um caso de uso A, o caso de uso B pode ser acrescentado para descrever o comportamento de A.
 - e. Generalização ou especialização: é um relacionamento entre um caso de uso genérico para um mais específico, herdando todas as características de seu pai. Um caso de uso B é um tipo específico do caso de uso A, podendo A ser uma generalização de B, ou B de A.

1.4.2 Diagrama de classes

Esse diagrama é considerado um modelo fundamental da especificação orientada a objetos. Segundo Silva (2007), ele cria a descrição mais próxima da estrutura do código de um programa, ou seja, apresenta o conjunto de classes com seus atributos e métodos e os relacionamentos entre classes. Nesse sentido, como elementos básicos temos as classes e os relacionamentos.

De acordo com Guedes (2011, p. 32):

O diagrama de classes é provavelmente o mais utilizado e é um dos mais importantes da UML. Serve de apoio para a maioria dos demais diagramas. Como o próprio nome diz, define a estrutura das classes utilizadas pelo sistema, determinando os atributos e métodos que cada classe tem, além de estabelecer como as classes se relacionam e trocam informações entre si.

Além disso, o diagrama de classes apresenta similaridade com o modelo de entidade-relacionamento, o MER, o qual é utilizado em banco de dados relacional. Entretanto o seu nível de abstração está em um nível mais alto, além de representar o comportamento da classe por meio de suas operações ou métodos.

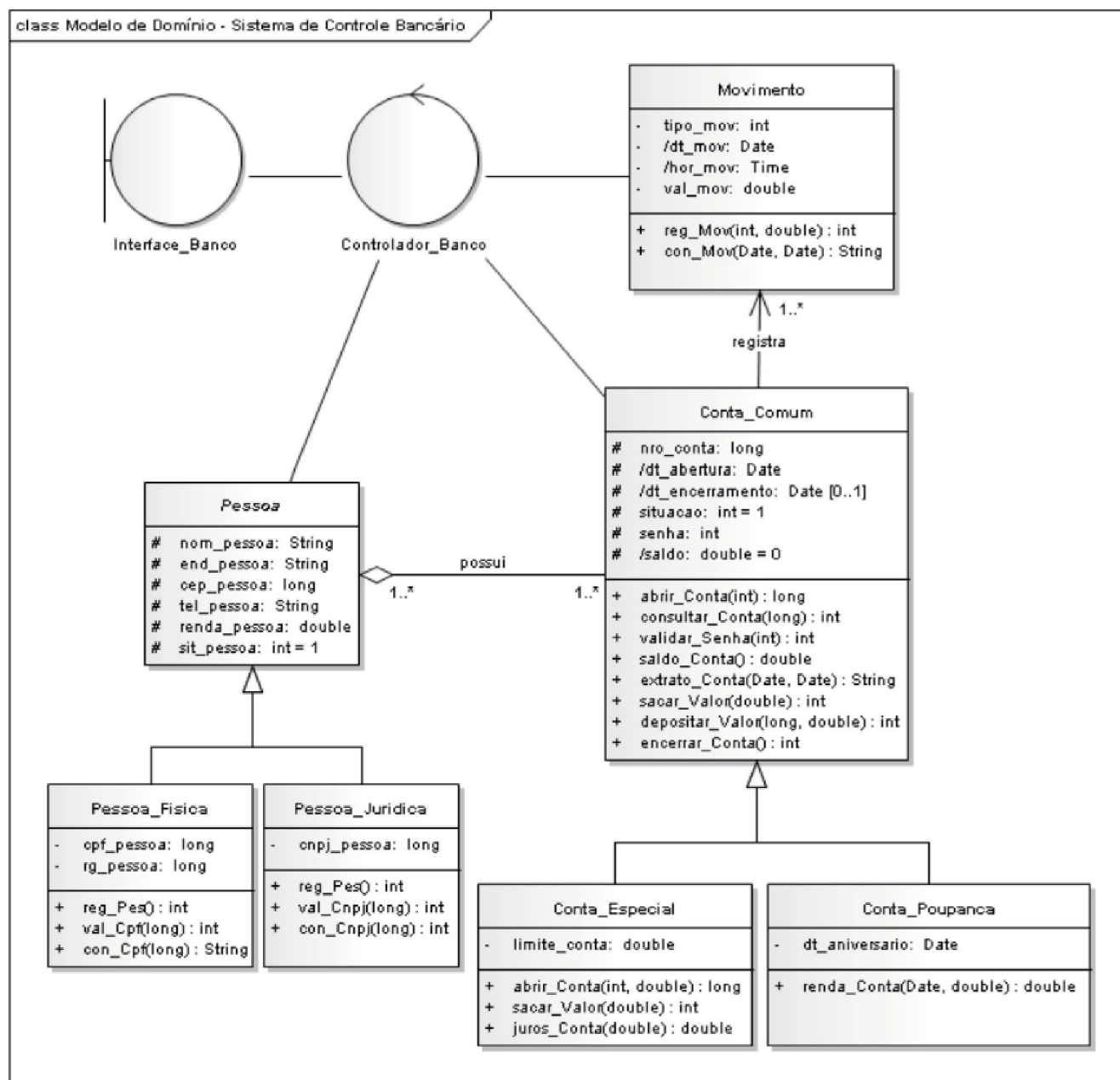
Neste diagrama temos a persistência como uma importante característica, já que, algumas classes, podem representar, em seu projeto, entidades físicas do mundo real, e que serão implementadas futuramente no banco de dados.

Para Guedes (2011, p. 101):

Basicamente, o diagrama de classes é composto por suas classes e pelas associações existentes entre elas, ou seja, os relacionamentos entre as classes. Alguns métodos de desenvolvimento de software, como o Processo Unificado, recomendam que se utilize o diagrama de classes ainda durante a fase de análise, produzindo-se um modelo conceitual a respeito das informações necessárias ao software.

A seguir, observe a figura, onde pode ser visto um exemplo de diagrama de classes.

Figura 3 – Exemplo diagrama de classes – sistema controle bancário



Fonte: Guedes (2011, p. 101).

A UML não se resume apenas nestes dois diagramas, bem como o detalhamento efetuado, mas em uma infinidade de diagramas e

particularidades, as quais podem ser mais bem exploradas em livros aqui já citados.

1.5 Ferramentas CASE UML

Com o objetivo de auxiliarem no processo de engenharia de software, as ferramentas CASE (*Computer-Aided Software Engineering* ou Engenharia de Software Auxiliada por Computador) foram criadas. Elas são softwares que auxiliam na construção dos diagramas da UML, o que facilita o processo de criação, análise e manutenção do referido diagrama.

Atualmente, o mercado oferece uma vasta opção em ferramentas CASE, cada qual com suas particularidades e objetivos. Porém, como não temos condições de mostrar todas as ferramentas, vamos nos basear em algumas indicações de Guedes (2011, p. 42), o qual faz apontamentos interessantes:

Visual Paradigm for UML ou VP-UML – a ferramenta pode ser encontrada no site www.visual-paradigm.com e oferece uma edição para a comunidade, ou seja, uma versão da ferramenta que pode ser baixada gratuitamente de sua página. Logicamente, a edição para a comunidade não suporta todos os serviços e opções disponíveis nas suas versões standard ou professional. No entanto, para quem deseja praticar a UML, a edição para a comunidade é uma boa alternativa, apresentando um ambiente amigável e de fácil compreensão. Além disso, a Visual-Paradigm oferece ainda uma cópia acadêmica da versão standard para instituições de ensino superior, que podem consegui-la solicitando-a na própria página da empresa. Tão logo a Visual-Paradigm comprovar a veracidade das informações fornecidas pela instituição, ela enviará uma licença de um ano para uso pelos professores e seus alunos. A licença precisa ser renovada anualmente.

Poseidon for UML – esta ferramenta também tem uma edição para a comunidade, apresentando bem menos restrições que a edição

para a comunidade da Visual-Paradigm, mas a interface da Poseidon é sensivelmente inferior à VP-UML, além de apresentar um desempenho um pouco inferior, embora ambas as ferramentas tenham sido desenvolvidas em Java. Uma cópia da Poseidon for UML pode ser adquirida no site www.gentleware.com.

Referências Bibliográficas

GUEDES, G. T. A. **UML 2: uma abordagem prática**. 2. ed. São Paulo: Novatec, 2011.

MEDEIROS, E. S. de. **Desenvolvendo software com UML 2.0**. São Paulo: Pearson Makron Books, 2004.

SILVA, R. P. **UML 2 em modelagem orientada a objetos**. Florianópolis: Visual Books, 2007.

VIEIRA, R. UML — Unified Modeling Language. **Medium**, 6 dez. 2015. Disponível em: <https://medium.com/operacionalti/uml-1f7b99dd15bb>. Acesso em: 12 nov. 2020.



Bons estudos!