UC12: Executar os processos de codificação, manutenção e documentação de aplicativos computacionais para desktop

CARGA HORÁRIA: **96 HORAS**

Indicadores

1. Configura o ambiente de desenvolvimento conforme as funcionalidades e características do aplicativo computacional para desktop.
2. Desenvolve softwares de acordo com as melhores práticas da linguagem de programação selecionada.
3. Elabora código conforme as funcionalidades e características do aplicativo computacional para desktops.
4. Realiza a compilação e depuração do código desenvolvido de acordo com orientações técnicas da IDE utilizada.
5. Utiliza comandos de integração dos objetos de bancos de dados com o código construído para desktop de acordo com premissas do sistema operacional (servidor) de rede.
6. Elabora o manual de utilização do projeto de software desenvolvido conforme a orientação técnica.

Elementos da competência

CONHECIMENTOS

- Análise de sistemas orientada a objetos – Tipos de dados Definição do projeto de sistema. Análise de requisitos do sistema. Definição de UML. Principais diagramas UML. Diagrama de caso de uso. Diagrama de classe.
- Estrutura de dados – Pilhas e filas. Listas e árvores. Ordenação de dados. Pesquisa de dados. Recursividade.
- Ferramentas de desenvolvimento de programas para desktop - Ferramentas de desenvolvimento colaborativo. Ferramentas de modelagem de software. Linguagens de programação. Ambientes de programação (IDE).
- Linguagem de programação orientada a eventos – Formulários. Uso de controles (eventos e propriedades). Módulos, funções e procedimentos. Conexão e manipulação do banco de dados. Vínculo do sistema com o formulário principal. Elaboração de listagens. Distribuição do aplicativo.

- Controle de versão de software– Conceito. Segurança da informação. Instalação e configuração.
- Política de recuperação de dados em programação – Conceito. Segurança da informação. Análise periódica. Procedimentos de backup e restore.
- Linguagem de Programação Orientada a Objetos – Conceitos e aplicações. Palavras reservadas. Application Program Interface (API). Plataforma de desenvolvimento: desktop. Tipos de dados. Variáveis e constantes. Coleções: lista, conjunto e mapa. Operadores. Comandos condicionais. Comandos de Repetição. Objetos, classes, interfaces, atributos, modificadores de acesso, métodos e propriedades. Herança, polimorfismo, encapsulamento e agregação. Tratamento de erros e exceções. Distribuição do aplicativo. Defeitos e falhas em programas de computador. Documentação de programas de computador.

HABILIDADES

- Resolver problemas lógicos e aritméticos.
- Representar expressões lógicas e matemáticas.
- Interpretar textos técnicos.
- Comunicar-se de maneira assertiva.
- Mediar conflitos nas situações de trabalho.
- Selecionar informações necessárias ao desenvolvimento do seu trabalho.
- Analisar as etapas do processo de trabalho.

ATITUDES/VALORES

- Zelo na apresentação pessoal e postura profissional.
- Sigilo no tratamento de dados e informações.
- Zelo pela segurança e pela integridade dos dados.
- Proatividade na resolução de problemas.
- Colaboração no desenvolvimento do trabalho em equipe.
- Cordialidade no trato com as pessoas.
- Responsabilidade no uso dos recursos organizacionais e no descarte de lixo eletrônico.

1. ANÁLISE DE SISTEMAS ORIENTADA A OBJETOS

1.1. Tipos de dados

Os tipos de dados nada mais são que uma forma de informar o que nossa variável irá guardar de informação.

Pensemos em algo do mundo real que queremos representar computacionalmente.

1.1.1 Tipo de dados numéricos

Os tipos de dados numéricos podem ser divididos em duas classes: inteiros e reais.

Entendemos como tipo de dados numéricos inteiros os que não possuem casas decimais ou fracionários, podendo ser positivos ou negativos, em outras palavras, é o conjunto dos números naturais (N) e inteiros (Z).

Os tipos de dados reais são números que podem possuir componentes decimais ou fracionários e podem ser positivos ou negativos.

1.1.2 Tipo de dados literais

Podemos dizer que são textos. É constituído por uma sequência de caracteres contendo letras, dígitos ou símbolos especiais. Normalmente, praticamente em todas as linguagens de programação, é denominado de String.

Em programação, quando usamos um tipo de dados literal, marcamos seu início e fim utilizando as aspas (“).

O comprimento de um dado literal é dado pela quantidade de caracteres nele contido, por exemplo, na variável com valor “informática” temos 11 caracteres, logo é um dado de tamanho 11.

1.1.3 Tipo de dados lógicos (booleanos)

Os tipos de dados lógicos são bem simples, e são usados para representar dois únicos valores lógicos: Verdadeiro e Falso. Podemos encontrar essas variáveis como 1 e 0, true e false ou sim e não, mas todas sempre irão representar apenas dois valores sendo um afirmativo e um negativo.

1.1.4 Tabela com tamanho de cada tipo de dados

Tipo de dado	Significado	Tamanho (em bytes)	Intervalo de valores aceitos
char	Caractere	1	0 a 255
string	Cadeia de caractetes	*	*
byte	Inteiro de 8bits	1	-127 a 127
short	Inteiro curto	2	-32 768 a +32 767
int	Inteiro	4	-2 147 483 648 a 2 147 483 647
long	Inteiro longo	8	-922337203685477808 a 922337203685477807
float	Ponto Flutuante (real)	4	$3.4*10^{-38}$ à $3.4*10^{38}$
double	Ponto Flutuante duplo	8	$1.7*10^{-308}$ a $1.7*10^{308}$
boolean	booleano	1	true ou false

Fonte: https://lipinf.com/introducao-a-logica-de-programacao-parte-2/

1.2. Definição do projeto de sistemas

A Análise e Projeto de Sistema é uma área da computação que descreve as funções de um sistema, verificar sua funcionalidade, elaborar e implementar soluções para diferentes problemas identificados durante a fase de análise, podemos dizer que consiste na construção da parte documental de um sistema a fim de garantir melhor performance, segurança e manutenção dos sistemas. Algumas normas ajudam na construção, desenvolvimento e regulamenta e define as etapas necessárias na construção de um sistema de software.

	Válida a partir de:	**Título:**	**Objetivos:**
ISO/IEC 14598-6:2001	29/11/2004	Engenharia de software - Avaliação de produto Parte 6: Documentação de módulos de avaliação	Esta parte da ABNT NBR ISO/IEC 14598 define a estrutura e o conteúdo da documentação a ser usada para descrever um Módulo de Avaliação (MA). Os módulos de avaliação serão usados no contexto das séries ISO/IEC 9126 e ABNT NBR ISO/IEC 14598.
ISO/IEC 25030:2007	22/10/2008	Engenharia de software - Requisitos e Avaliação da Qualidade de Produto de Software (SQuaRE) - Requisitos de qualidade	Esta Norma fornece os requisitos e recomendações para especificação de requisitos de qualidade de produto de software.
ISO/IEC 15939:2007	15/02/2009	Engenharia de sistemas e de software - Processo de medição	Esta Norma identifica as atividades e tarefas necessárias para identificar, definir, selecionar, aplicar e melhorar, de forma bem-sucedida, medição em um projeto ou em uma estrutura de medição organizacional. Também fornece definições de termos de medição geralmente utilizados pelas indústrias de software e de sistema.
ISO/IEC 25020:2007	20/04/2009	Engenharia de software - Requisitos e avaliação da qualidade de produto de software (SQuaRE) - Guia e modelo de referência para medição	O escopo desta Norma é a seleção e construção de medidas de qualidade de produto de software, especialmente, com relação a seu uso em conjunto com outros documentos da série SQuaRE.
ISO/IEC 25001:2007	20/04/2009	Engenharia de software - Requisitos e avaliação da qualidade de produto de software (SQuaRE) - Planejamento e gestão	Esta Norma fornece requisitos e recomendações para uma organização responsável por implementar e gerenciar a especificação dos requisitos de qualidade do produto de software e pelas atividades de avaliação da qualidade de software provendo tecnologia, ferramentas, experiências e habilidades de gestão. A função do grupo de avaliação inclui motivar as pessoas e treiná-las para as atividades de especificação dos requisitos e as de avaliação, preparar os documentos apropriados, identificar ou desenvolver os métodos necessários e responder questões relacionadas às tecnologias que sejam relevantes.

ISO/IEC TR 24774:2007	09/03/2010	Engenharia de software e de sistemas — Gerenciamento de ciclo de vida — Orientações para descrição de processos	Este Relatório Técnico provê orientações para a descrição de processos mediante a identificação descritiva de atributos e de regras para sua criação. São caracterizados os seguintes atributos de descrição de processos.
ISO/IEC 25062:2006	27/05/2011	Engenharia de software — Requisitos e avaliação da qualidade de produto de software (SQuaRE) — Formato comum da indústria (FCI) para relatórios de teste de usabilidade	Esta Norma destina-se à elaboração de relatório das medidas obtidas em um teste de usabilidade, conforme definidas na ISO 9241-11: eficácia, eficiência e satisfação em um contexto de uso especificado.
ISO/IEC/IEEE 16326:2009	13/10/2012	Engenharia de sistemas e de software — Processos de ciclo de vida — Gerenciamento de projeto	Esta Norma destina-se a auxiliar os gerentes de projeto a concluírem com sucesso os projetos que tratam de sistemas intensivos em software e produtos de software.
ISO/IEC/IEEE 29119-1:2013	04/07/2014	Engenharia de software e sistemas — Teste de software Parte 1: Conceitos e definições	Esta parte da ABNT NBR ISO/IEC/IEEE 29119 especifica definições e conceitos sobre teste de software. Ela fornece definições de termos de teste e discussão de conceitos-chave para a compreensão da série de normas de teste de software ISO/IEC/IEEE 29119.

	Data de Publicação:	**Título:**	**Objetivos:**
ISO/IEC 29110-4-1:2018	19/03/2020	Engenharia de software e sistemas - Perfis de ciclo de vida para micro-organizações (VSE) Parte 4-1: Engenharia de software - Especificações de perfil: Grupo de perfil genérico	A série ISO/IEC 29110 aplica-se às micro-organizações (VSE). As VSE são empresas, organizações, departamentos ou projetos com até 25 pessoas. Os processos de ciclo de vida descritos na ISO/IEC 29110 não se destinam a impedir ou desencorajar seu uso em organizações maiores que as VSE.
ISO/IEC/IEEE 12207:2017	24/08/2021	Engenharia de sistemas e software - Processos de ciclo de vida de software	Este documento estabelece uma estrutura comum para processos de ciclo de vida de software, com terminologia bem definida, que pode ser referenciada pela indústria de software. Ele contém processos, atividades e tarefas que são aplicáveis durante a aquisição, fornecimento, desenvolvimento, operação, manutenção ou desativação de sistemas, produtos e serviços de software. Estes processos de ciclo de vida são executados com sucesso por meio do envolvimento de stakeholders, com o objetivo final de alcançar a satisfação do cliente.

ISO/IEC 29110-4-2:2021	28/06/2022	Engenharia de software e sistemas - Perfis de ciclo de vida para micro-organizações (VSE) Parte 4-2: Engenharia de software: Especificações de perfil: Grupo de perfil de gestão organizacional	Este documento fornece uma especificação de perfil para o perfil de gestão organizacional. O perfil de gestão organizacional se aplica às VSE envolvidas em atividades de engenharia de sistemas e/ou engenharia de software.

1.3. Análise e requisito do sistema

Análise e requisitos do sistema é considerada uma das principais partes para o desenvolvimento de um software. É a primeira etapa de uma documentação ou até mesmo de um desenvolvimento, onde, através de coletas realizadas com o cliente, pode-se definir quais as etapas devem ser realizadas para a construção do sistema.

Essa análise tem o objetivo de dividir o todo do sistema em partes, estas partes são transformadas em funcionalidades, ou seja, dividida em ações que alguém vai desenvolver / interagir no programa desenvolvido.

Essa análise para levantamento de requisitos é de grande importância para o início do desenvolvimento, Smerville (2003) propõe um processo genérico de levantamento e análise que contém as seguintes atividades:

- **Compreensão do domínio:** Os analistas devem desenvolver sua compreensão do domínio da aplicação;
- **Coleta de requisitos:** É o processo de interagir com os stakeholders do sistema para descobrir seus requisitos. A compreensão do domínio se desenvolve mais durante essa atividade;
- **Classificação:** Essa atividade considera o conjunto não estruturado dos requisitos e os organiza em grupos coerentes;
- **Resolução de conflitos:** Quando múltiplos stakeholders estão envolvidos, os requisitos apresentarão conflitos. Essa atividade tem por objetivo solucionar esses conflitos;
- **Definição das prioridades:** Em qualquer conjunto de requisitos, alguns serão mais importantes do que outros. Esse estágio envolve interação com os stakeholders para a definição dos requisitos mais importantes;
- **Verificação de requisitos:** Os requisitos são verificados para descobrir se estão completos e consistentes e se estão em concordância com o que os stakeholders desejam do sistema.

Algumas técnicas para realizar o levantamento dos requisitos ajudam e muito na realização desta etapa do processo garantindo que seja a mais assertiva possível.

As técnicas de levantamento de requisitos têm por objetivo superar as dificuldades relativas a esta fase. Todas as técnicas possuem um conceito próprio e suas respectivas vantagens e desvantagens, que podem ser utilizadas em conjunto pelo analista.

1.3.1. Levantamento orientado a pontos de vista

Para qualquer sistema, de tamanho médio ou grande, normalmente há diferentes tipos de usuário final. Muitos stakeholders (qualquer pessoa ou organização que tenha interesse, ou seja afetado pelo projeto), tem alguma interação com o sistema, porém são muitos os pontos de vista ou usabilidade de cada um dentro do sistema. Cada usuário tem uma perceptiva diferente, que “aponta’ somente as tarefas que ele irá executar e todos os pontos de vista devem ser considerados. A forma como cada usuário “enxerga” os problemas não são inteiramente independentes, mas em geral apresentam alguma duplicidade, de modo que apresentam requisitos comuns.

As abordagens orientadas a ponto de vista, na engenharia de requisitos, reconhecem esses diferentes pontos de vista e os utilizam para estruturar e organizar o processo de

levantamento e os próprios requisitos. Uma importante capacidade da análise orientada a pontos de vista é que ela reconhece a existência de várias perspectivas e oferece um framework para descobrir conflitos nos requisitos propostos por diferentes stakeholders.

O método VORD (viewpoint-oriented requirements definition – definição de requisitos orientada a ponto de vista) foi projetado como um framework orientado a serviço para o levantamento e análise de requisitos.

Este framework possui algumas etapas:

- A **primeira etapa** da análise de ponto de vista é identificar os possíveis pontos de vista. Nessa etapa os analistas se reúnem com os stakeholders e utilizam a abordagem de brainstorming para identificar as ações realizadas no sistema e quem são as pessoas que executam estas ações.
- A s**egunda etapa** é a estruturação de pontos de vista, que envolve agrupar pontos de vista relacionados, segundo uma hierarquia. Serviços comuns estão localizados nos níveis mais altos da hierarquia e herdados por pontos de vista de nível inferior.
- A **etapa de documentação** do ponto de vista tem por objetivo refinar a descrição dos pontos de vista e serviços identificados.
- O **mapeamento de sistema** conforme ponto de vista envolve identificar objetos em um projeto orientado a objetos, utilizando as informações de serviço que estão encapsuladas nos pontos de vista.

1.3.2. Etnografia

A etnografia é uma técnica de observação que pode ser utilizada para compreender os requisitos sociais e organizacionais, ou seja, entender a política organizacional bem como a cultura de trabalho com objetivo de familiarizar-se com o sistema e sua história.

Nesta técnica, o analista se insere no ambiente de trabalho em que o sistema será utilizado. O trabalho diário é observado e são anotadas as tarefas reais em que o sistema será utilizado. O principal objetivo da etnografia é que ela ajuda a descobrir requisitos de sistema implícitos, que refletem os processos reais, em vez de os processos formais, onde as pessoas estão envolvidas, assim não se obtém os requisitos pelo que o cliente passa e sim pela análise do dia a dia dos funcionários para entender o fluxo do processo e a real necessidade de casa usuário no sistema.

Etnografia é particularmente eficaz na descoberta de dois tipos de requisitos:

- Os **requisitos derivados da maneira como as pessoas realmente trabalham**, em vez da maneira pelas quais as definições de processo dizem como elas deveriam trabalhar;
- Os **requisitos derivados da cooperação e conscientização das atividades de outras pessoas**.

Alguns itens importantes que devem ser executados antes, durante e depois do estudo de observação:

- **Antes**, é necessário identificar as áreas de usuário a serem observadas; obter a aprovação das gerências apropriadas para executar as observações; obter os nomes e funções das pessoas chave que estão envolvidas no estudo de observação; e explicar a finalidade do estudo;
- **Durante**, é necessário familiarizar-se com o local de trabalho que está sendo observado. Para isso é preciso observar os agrupamentos organizacionais atuais; as facilidades manuais e automatizadas; coletar amostras de documentos e procedimentos escritos que são usados em cada processo específico que está sendo observado; e acumular informações estatísticas a respeito das tarefas, como: frequência que ocorrem, estimativas de volumes, tempo de duração para cada pessoa que

está sendo observada. Além de observar as operações normais de negócios acima é importante observar as exceções;

- **Depois**, é necessário documentar as descobertas resultantes das observações feitas. Para consolidar o resultado é preciso rever os resultados com as pessoas observadas e/ou com seus superiores.

A análise de observação tem algumas desvantagens como, consumir bastante tempo e o analista ser induzido a erros em suas observações. Mas em geral a técnica de observação é muito útil e frequentemente usada para complementar descobertas obtidas por outras técnicas.

1.3.3. Workshops

Trata-se de uma técnica em grupo usada em uma reunião estruturada. Devem fazer parte do grupo uma equipe de analistas e uma seleção dos stakeholders que melhor representam a organização e o contexto em que o sistema será usado, obtendo assim um conjunto de requisitos bem definidos.

Ao contrário das reuniões, em que existe pouca interação entre todos os elementos presentes, o workshop tem o objetivo de acionar o trabalho em equipe. Há um facilitador neutro cujo papel é conduzir a workshop e promover a discussão entre os vários mediadores. As tomadas de decisão são baseadas em processos bem definidos e com o objetivo de obter um processo de negociação, mediado pelo facilitador.

Uma técnica utilizada em workshops é o **brainstorming**. Após os workshops serão produzidas documentações que refletem os requisitos e decisões tomadas sobre o sistema a ser desenvolvido.

Alguns aspectos importantes a serem considerados: a postura do condutor do seminário deve ser de mediador e observador; a convocação deve possuir dia, hora, local, horário de início e de término; assunto a ser discutido e a documentação do seminário.

1.3.4. Prototipagem

Protótipo tem por objetivo explorar aspectos críticos dos requisitos de um produto, implementando de forma rápida um pequeno subconjunto de funcionalidades deste produto. O protótipo é indicado para estudar as alternativas de interface do usuário; problemas de comunicação com outros produtos; e a viabilidade de atendimento dos requisitos de desempenho. As técnicas utilizadas na elaboração do protótipo são várias: interface de usuário, relatórios textuais, relatórios gráficos, entre outras.

Alguns dos benefícios do protótipo são as reduções dos riscos na construção do sistema, pois o usuário chave já verificou o que o analista captou nos requisitos do produto. Para ter sucesso na elaboração dos protótipos é necessária a escolha do ambiente de prototipagem, o entendimento dos objetivos do protótipo por parte de todos os interessados no projeto, a focalização em áreas menos compreendidas e a rapidez na construção.

1.3.5. Entrevistas

A entrevista é uma das técnicas tradicionais mais simples de utilizar e que produz bons resultados na fase inicial de obtenção de dados. Convém que o entrevistador dê margem ao entrevistado para expor as suas ideias. É necessário ter um plano de entrevista para que não haja dispersão do assunto principal e a entrevista fique longa, deixando o entrevistado cansado e não produzindo bons resultados.

As seguintes diretrizes podem ser de grande auxílio na direção de entrevistas bem-sucedidas com o usuário: **desenvolver um plano geral de entrevistas, certificar-se da autorização para falar com os usuários, planejar a entrevista para fazer uso eficiente do tempo, utilizar ferramentas automatizadas que sejam adequadas, tentar descobrir que informação o usuário está mais interessado e usar um estilo adequado ao entrevistar.**

Para planejar a entrevista é necessário que antes dela sejam coletados e estudados todos os dados pertinentes à discussão, como formulários, relatórios, documentos e outros. Dessa forma, o analista estará bem contextualizado e terá mais produtividade nos assuntos a serem discutidos na entrevista.

É importante determinar um escopo relativamente limitado, focalizando uma pequena parte do sistema para que a reunião não se estenda por mais de uma hora. O usuário tem dificuldade de concentração em reuniões muito longas, por isso é importante focalizar a reunião no escopo definido.

Após a entrevista é necessário validar se o que foi documentado pelo analista está de acordo com a necessidade do usuário, que o usuário não mudou de opinião e que o usuário entende a notação ou representação gráfica de suas informações.

A atitude do analista em relação à entrevista é determinar seu fracasso ou sucesso. Uma entrevista não é uma competição, deve-se evitar o uso excessivo de termos técnicos e não conduzir a entrevista em uma tentativa de persuasão. O modo como o analista fala não deve ser muito alto, nem muito baixo, tampouco indiretamente, ou seja, utilizar os termos: ele disse isso ou aquilo na reunião para o outro entrevistado. O modo melhor para agir seria, por exemplo, dizer: O João vê a solução para o projeto dessa forma. E o senhor André, qual é a sua opinião? Em uma entrevista o analista nunca deve criticar a credibilidade do entrevistado. O analista deve ter em mente que o entrevistado é o perito no assunto e fornecerá as informações necessárias ao sistema.

Para elaborar perguntas detalhadas é necessário solicitar que o usuário:

- Explique o relacionamento entre o que está em discussão e as demais partes do sistema;
- Descreva o ponto de vista de outros usuários em relação ao item que esteja sendo discutido;
- Descreva informalmente a narrativa do item em que o analista deseja obter informações;
- Perguntar ao usuário se o item em discussão depende para a sua existência de alguma outra coisa, para assim poder juntar os requisitos comuns do sistema, formando assim um escopo conciso.

Pode-se utilizar a confirmação, para tanto o analista deve dizer ao usuário o que acha que ouviu ele dizer. Neste caso, o analista deve utilizar as suas próprias palavras em lugar das do entrevistado e solicitar ao entrevistado confirmação do que foi dito.

1.3.6. Questionários

O uso de questionário é indicado, por exemplo, quando há diversos grupos de usuários que podem estar em diversos locais diferentes do país. Neste caso, elaboram-se pesquisas específicas de acompanhamento com usuários selecionados, que a contribuição em potencial pareça mais importante, pois não seria prático entrevistar todas as pessoas em todos os locais.

Existem vários tipos de questionários que podem ser utilizados. Entre estes podemos listar: múltipla escolha, lista de verificação e questões com espaços em branco. O questionário deve ser desenvolvido de forma a minimizar o tempo gasto em sua resposta.

Na fase de preparação do questionário deve ser indicado o tipo de informação que se deseja obter. Assim que os requisitos forem definidos o analista deve elaborar o questionário com questões de forma simples, clara e concisa, deixar espaço suficiente para as repostas que forem descritivas e agrupar as questões de tópicos específicos em um conjunto com um título especial. O questionário deve ser acompanhado por uma carta explicativa, redigida por um alto executivo, para enfatizar a importância dessa pesquisa para a organização.

Deve ser desenvolvido um controle que identifique todas as pessoas que receberão os questionários. A distribuição deve ocorrer junto com instruções detalhadas sobre como preenchê-lo e ser indicado claramente o prazo para devolução do questionário. Ao

analisar as respostas dos participantes é feito uma consolidação das informações fornecidas no questionário, documentando as principais descobertas e enviando uma cópia com estas informações para o participante como forma de consideração pelo tempo dedicado a pesquisa.

1.3.7. Brainstorming

Brainstorming é uma técnica para geração de ideias. Ela consiste em uma ou várias reuniões que permitem que as pessoas sugiram e explorem ideias.

As principais etapas necessárias para conduzir uma sessão de brainstorming são:

- **Seleção dos participantes:** Os participantes devem ser selecionados em função das contribuições diretas que possam dar durante a sessão. A presença de pessoas bem-informadas, vindas de diferentes grupos garantirá uma boa representação;
- **Explicar a técnica e as regras a serem seguidas:** O líder da sessão explica os conceitos básicos de brainstorming e as regras a serem seguidas durante a sessão;
- **Produzir uma boa quantidade de ideias:** Os participantes geram tantas ideias quantas forem exigidas pelos tópicos que estão sendo o objeto do brainstorming. Os participantes são convidados, um por vez, a dar uma única ideia. Se alguém tiver problema, passa a vez e espera a próxima rodada.

No brainstorming as ideias que a princípio pareçam não convencionais, são encorajadas, pois elas frequentemente estimulam os participantes, o que pode levar a soluções criativas para o problema. O número de ideias geradas deve ser bem grande, pois quanto mais ideias forem propostas, maior será a chance de aparecerem boas ideias. Os participantes também devem ser encorajados a combinar ou enriquecer as ideias de outros e, para isso, é necessário que todas as ideias permaneçam visíveis a todos os participantes.

Nesta técnica é designada uma pessoa para registrar todas as ideias em uma lousa branca ou em papel. À medida que cada folha de papel é preenchida, ela é colocada de forma que todos os participantes possam vê-la.

Analisar as ideias é a fase final do brainstorming. Nessa fase é realizada uma revisão das ideias, uma de cada vez. As consideradas valiosas pelo grupo são mantidas e classificadas em ordem de prioridade.

1.3.8. JAD

JAD (Joint Application Design) é uma técnica para promover cooperação, entendimento e trabalho em grupo entre os usuários desenvolvedores.

O JAD facilita a criação de uma visão compartilhada do que o produto de software deve ser. Através da sua utilização os desenvolvedores ajudam os usuários a formular problemas e explorar soluções. Dessa forma, os usuários ganham um sentimento de envolvimento, posse e responsabilidade com o sucesso do produto.

A técnica JAD tem quatro princípios básicos:

- **Dinâmica de grupo:** são realizadas reuniões com um líder experiente, analista, usuários e gerentes, para despertar a força e criatividade dos participantes. O resultado será a determinação dos objetivos e requisitos do sistema;
- **Uso de técnicas visuais:** para aumentar a comunicação e o entendimento;
- **Manutenção do processo organizado e racional:** o JAD emprega a análise top down e atividades bem definidas. Possibilita assim, a garantia

de uma análise completa reduzindo as chances de falhas ou lacunas no projeto e cada nível de detalhe recebe a devida atenção;

- **Utilização de documentação padrão:** preenchida e assinada por todos os participantes. Este documento garante a qualidade esperada do projeto e promove a confiança dos participantes.

A técnica JAD é composta de duas etapas principais: **planejamento**, que tem por objetivo elicitar e especificar os requisitos; e **projeto**, em que se lida com o projeto de software.

Cada etapa consiste em três fases: **adaptação, sessão e finalização.**

A fase de **adaptação** consiste na preparação para a sessão, ou seja, organizar a equipe, adaptar o processo JAD ao produto a ser construído e preparar o material. Na fase de **sessão** é realizado um ou mais encontros estruturados, envolvendo desenvolvedores e usuários onde os requisitos são desenvolvidos e documentados. A fase de **finalização** tem por objetivo converter a informação da fase de sessão em sua forma final (um documento de especificação de requisitos).

Há seis tipos de participantes, embora nem todos participem de todas as fases:

- **Líder da sessão:** é responsável pelo sucesso do esforço, sendo o facilitador dos encontros. Deve ser competente, com bom relacionamento pessoal e qualidades gerenciais de liderança;
- **Engenheiro de requisitos:** é o participante diretamente responsável pela produção dos documentos de saída das sessões JAD. Deve ser um desenvolvedor experiente para entender as questões técnicas e detalhes que são discutidos durante as sessões e ter habilidade de organizar ideias e expressá-las com clareza;
- **Executor:** é o responsável pelo produto sendo construído. Tem que fornecer aos participantes uma visão geral dos pontos estratégicos do produto de software a ser construído e tomar as decisões executivas, tais como alocação de recursos, que podem afetar os requisitos e o projeto do novo produto;
- **Representantes dos usuários:** são as pessoas na empresa que irão utilizar o produto de software. Durante a extração de requisitos, os representantes são frequentemente gerentes ou peças-chaves dentro da empresa que tem uma visão melhor do todo e de como ele será usado;
- **Representantes de produtos de software:** são pessoas que estão bastante familiarizadas com as capacidades dos produtos de software. Seu papel é ajudar os usuários a entender o que é razoável ou possível que o novo produto faça;
- **Especialista:** é a pessoa que pode fornecer informações detalhadas sobre um tópico específico.

O conceito do JAD de abordagem e dinâmica de grupo poderá ser utilizado para diversas finalidades, como: planejamento de atividades técnicas para um grande projeto, discussão do escopo e objetivos de um projeto e estimativa da quantidade de horas necessárias para desenvolver sistemas grandes e complexos.

A maioria das técnicas JAD funciona melhor em projetos pequenos ou médios. Para um sistema grande e complexo podem ser usadas múltiplas sessões JAD para acelerar a definição dos requisitos do sistema.

Requisitos de sistema

Podemos entender como requisitos de um sistema as descrições (ações) que o sistema deve fazer, os serviços que ele poderá oferecer e as restrições de sistema para seu devido funcionamento, sempre levando em consideração as necessidades solicitadas

pelo cliente. Efetuar login, realizar cadastro, emitir relatório, controlar um pedido, exibir informações, cores do Software, segurança... todos são exemplos de requisitos que podemos encontrar nos sistemas. O processo de realizar esse levantamento descobrindo, analisando e documentando essas ações, serviços e restrições é denominado de "Engenharia de Requisitos". O levantamento incorreto das informações do sistema poderá acarretar no desenvolvimento incorreto do sistema.

Para que possamos ter uma melhor compreensão das funcionalidades, podemos separar as ações dos requisitos entre requisitos de sistemas e requisitos de usuários, onde **requisitos de sistemas** trata das funções e de serviços que o próprio sistema deverá realizar sem a ação do usuário a parte operacional e documental e **requisitos de usuário** são as ações propriamente ditas de cada usuário (ator) e suas restrições.

Os requisitos são localizados com sua nomenclatura de requisitos funcionais e não funcionais. De acordo com *Sommerville*:

- **Requisitos funcionais.** São declarações de serviços que o sistema deve fornecer, de como o sistema deve reagir a entradas específicas e de como o sistema deve se comportar em determinadas situações. Em alguns casos, os requisitos funcionais também podem explicitar o que o sistema não deve fazer.
- **Requisitos não funcionais.** São restrições aos serviços ou funções oferecidas pelo sistema. Incluem restrições de timing, restrições no processo de desenvolvimento e restrições impostas pelas normas. Ao contrário das características individuais ou serviços do sistema, os requisitos não funcionais, muitas vezes, aplicam-se ao sistema como um todo.

A seguir, entenderemos um pouco mais sobre ambos os conceitos.

Requisitos Funcionais

Os **Requisitos Funcionais** descrevem o que um determinado sistema deverá fazer, de fato suas funcionalidades. Ele pode ser simples a ponto de cada usuário entender apenas visualizando seu diagrama como também completo, descrevendo os detalhes das funções de sistema, suas entradas, saídas, exceções etc.

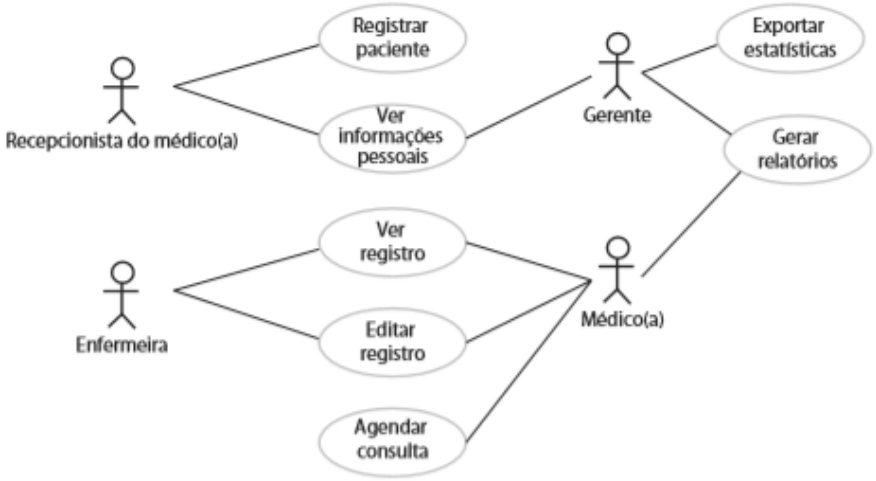
O levantamento incorreto dos requisitos funcionais pode desencadear outros problemas futuros no desenvolvimento do software, tendo que todo o projeto voltar a fase inicial e refazer toda a engenharia do software realizando novo levantamento de requisitos a fim de que de fato as funções estejam de acordo com o solicitado pelo cliente. Podemos realizar esse levantamento através de estudo de caso, pesquisa de campo, entrevistas com o cliente e com os funcionários da empresa etc.

A especificação dos requisitos funcionais deve ser o mais completa e consistente possível, colocando de fato todos os serviços (funções) solicitadas pelo cliente e não deve ter definições contraditórias.

Ao finalizar o levantamento dos requisitos funcionais, podemos estabelecer o documento de requisitos de software. Parte deste documento trata do detalhamento dos requisitos levantados. Para dar sequência neste documento é preciso analisar quem serão os atores do sistema, ou seja, quem são as pessoas ou sistemas que irão interagir com o software.

Após identificado os atores, os requisitos funcionais são atribuídos de acordo com a ação que cada ator tem no sistema. Relacionando ator com seus respectivos requisitos, podemos criar o diagrama de caso de uso.

O diagrama de caso de uso busca apresentar as iterações individuais entre o sistema e seus usuários ou outros sistemas. Cada caso de uso deve ser documentado com uma descrição textual o que chamamos de especificação de caso de uso.



Fonte: Engenharia de Software Sommerville

Na imagem acima podemos dar como exemplo o caso de uso “Agendar consulta”, onde podemos especificá-lo da seguinte maneira:

> *Agendar a consulta permite que dois ou mais médicos de consultórios diferentes possam ler o mesmo registro ao mesmo tempo. Um médico deve escolher, em um menu de lista de médicos on-line, as pessoas envolvidas. O prontuário do paciente é então exibido em suas telas, mas apenas o primeiro médico pode editar o registro. Além disso, uma janela de mensagens de texto é criada para ajudar a coordenar as ações. Supõe-se que uma conferência telefônica para comunicação por voz será estabelecida separadamente.*

Cenários e casos de uso são técnicas eficazes para deixar claro as ações que os stakeholders que vão interagir diretamente com o sistema. Cada tipo de interação pode ser representado como um caso de uso. No entanto, devido a seu foco nas interações com o sistema, eles não são tão eficazes para demonstrar os requisitos não funcionais do sistema.

Requisitos não funcionais

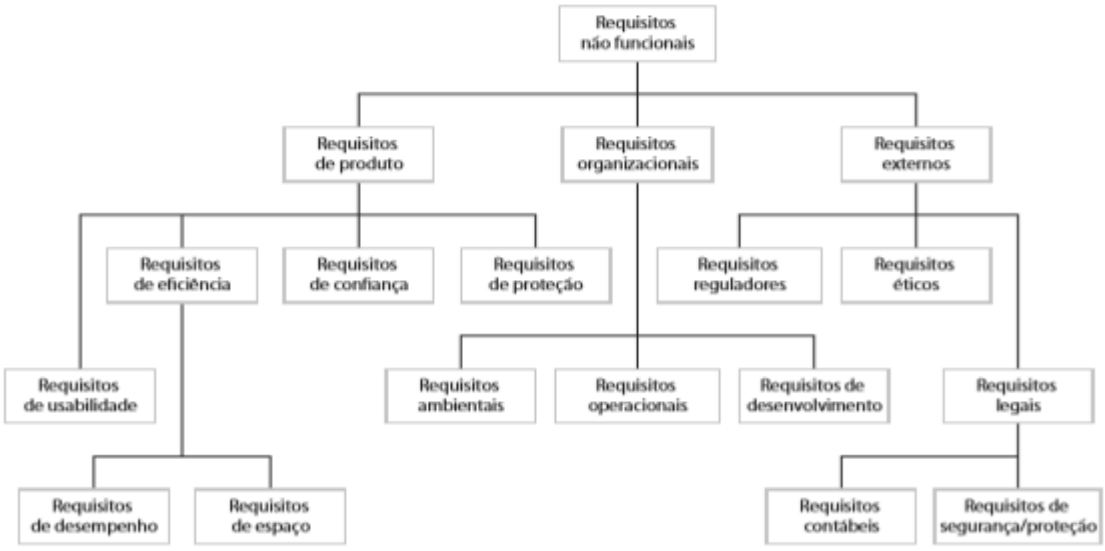
Requisitos não-funcionais são os requisitos relacionados ao uso da aplicação em termos de desempenho, usabilidade, confiabilidade, segurança, disponibilidade, manutenibilidade e tecnologias envolvidas. Não é preciso que o cliente diga ou peça tais requisitos, pois eles são características mínimas de um software de qualidade.

Os requisitos não funcionais:

- Demonstram qualidade acerca dos serviços ou funções disponibilizadas pelo sistema. Ex.: tempo, o processo de desenvolvimento, padrões etc.
- Surgem conforme a necessidade dos usuários, em razão de orçamento e outros fatores.
- Podem estar relacionados à confiabilidade, tempo de resposta e espaço nas mídias de armazenamento disponíveis.
- Caso ocorra falha do não atendimento a um requisito não funcional, poderá tornar todo o sistema ineficaz. Ex.: requisito confiabilidade em um sistema de controle de voos.

Classificação dos requisitos não funcionais:

- **Requisitos de produtos:** Requisitos que especificam o comportamento do produto.
  Ex. portabilidade; tempo na execução; confiabilidade, mobilidade etc.
- **Requisitos da organização:** Requisitos decorrentes de políticas e procedimentos corporativos.
  Ex. padrões, infraestrutura etc.
- **Requisitos externos:** Requisitos decorrentes de fatores externos ao sistema e ao processo de desenvolvimento.
  Ex. requisitos de interoperabilidade, legislação, localização geográfica etc.
- **Requisitos de facilidade de uso:**
  Ex. usuários deverão operar o sistema após um determinado tempo de treinamento.
- **Requisitos de eficiência:**
  Ex. o sistema deverá processar n requisições por um determinado tempo.
- **Requisitos de confiabilidade:**
  Ex. o sistema deverá ter alta disponibilidade, por exemplo, 99% do tempo.
- **Requisitos de portabilidade:**
  Ex. o sistema deverá rodar em qualquer plataforma.
- **Requisitos de entrega:**
  Ex. um relatório de acompanhamento deverá ser fornecido toda segunda-feira.
- **Requisitos de implementação:**
  Ex. o sistema deverá ser desenvolvido na linguagem Java.
- **Requisitos de padrões:**
  Ex. uso de programação orientada a objeto sob a plataforma A.
- **Requisitos de interoperabilidade:**
  Ex. o sistema deverá se comunicar com o SQL Server.
- **Requisitos éticos:**
  Ex. o sistema não apresentará aos usuários quaisquer dados de cunho privativo.
- **Requisitos legais:**
  Ex. o sistema deverá atender às normas legais, tais como padrões, leis etc.
- **Requisitos de Integração:**
  Ex. o sistema integra com outra aplicação.



Fonte:
https://www.inf.ufpr.br/lmperes/2019_2/ci162/material_aulas/slides/aula6_req_nao_funcionais.pdf

Propriedade	Medida
Velocidade	Transações processadas/segundo Tempo de resposta de usuário/evento Tempo de atualização de tela
Tamanho	Megabytes Número de chips de memória ROM
Facilidade de uso	Tempo de treinamento Número de *frames* de ajuda
Confiabilidade	Tempo médio para falha Probabilidade de indisponibilidade Taxa de ocorrência de falhas Disponibilidade
Robustez	Tempo de reinício após falha Percentual de eventos que causam falhas Probabilidade de corrupção de dados em caso de falha
Portabilidade	Percentual de declarações dependentes do sistema-alvo Número de sistemas-alvo

Fonte: https://www.inf.ufpr.br/lmperes/2019_2/ci162/material_aulas/slides/aula6_req_nao_funcionais.pdf

Exercício de fixação.

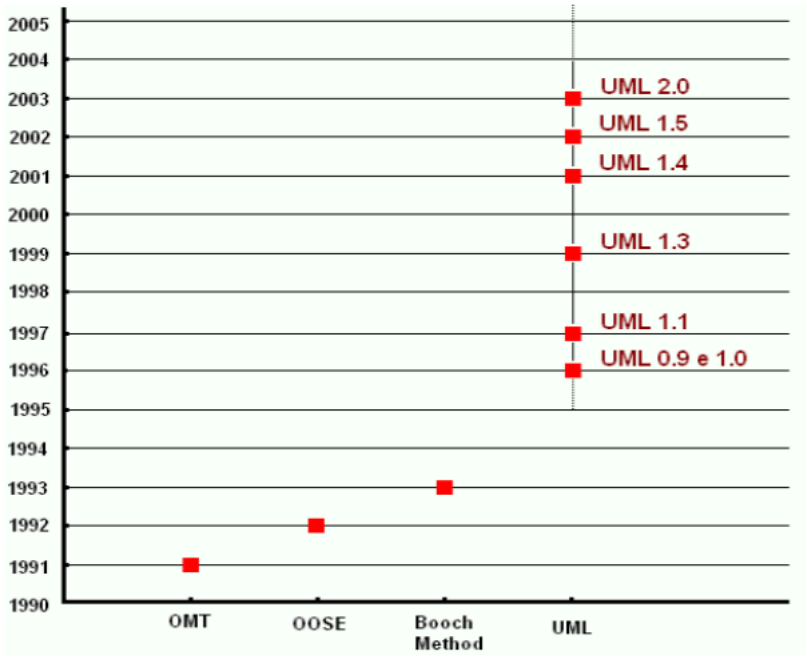
Pensando em um sistema de Pizzaria, elaborem a regra de negócio, descrevendo com funcionaria essa pizzaria, definindo os usuários, definam os requisitos funcionais e não funcionais da mesma. Ao final devem apresentar a solução idealizada.

1.4. Definição UML

No início da utilização do paradigma de orientação a objetos, diversos métodos foram apresentados para a comunidade. Chegaram a mais de cinquenta entre os anos de 1989 até 1994, porém a maioria deles cometeu o erro de tentar estender os métodos estruturados da época. Com isso os maiores prejudicados foram os usuários que não conseguiam encontrar uma maneira satisfatória de modelar seus sistemas. Foi a partir da década de 90 que começaram a surgir teorias que procuravam trabalhar de forma mais ativa com o paradigma da orientação a objetos. Diversos autores famosos contribuíram com publicações de seus respectivos métodos.

Por volta de 1993 existiam três métodos que mais cresciam no mercado, eram eles: Booch'93 de Grady Booch, OMT-2 de James Rumbaugh e OOSE de Ivar Jacobson. Cada um deles possuía pontos fortes em algum aspecto. O OOSE possuía foco em casos de uso (use cases), OMT-2 se destaca na fase de análise de sistemas de informação e Booch'93 era mais forte na fase de projeto. O sucesso desses métodos foi, principalmente, devido ao fato de não terem tentado estender os métodos já existentes. Seus métodos já convergiam de maneira independente, então seria mais produtivo continuar de forma conjunta (SAMPAIO, 2007).

Em outubro de 1994, começaram os esforços para unificação dos métodos. Já em outubro de 1995, Booch e Rumbaugh lançaram um rascunho do "Método Unificado" unificando o Booch'93 e o OMT-2. Após isso, Jacobson se juntou a equipe do projeto e o "Método Unificado" passou a incorporar o OOSE. Em junho de 1996, os três amigos, como já eram conhecidos, lançaram a primeira versão com os três métodos - a **versão 0.9** que foi batizada como **UML (FOWLER, 2003).** Posteriormente, foram lançadas várias novas versões na qual podemos acompanhar através do gráfico a seguir.

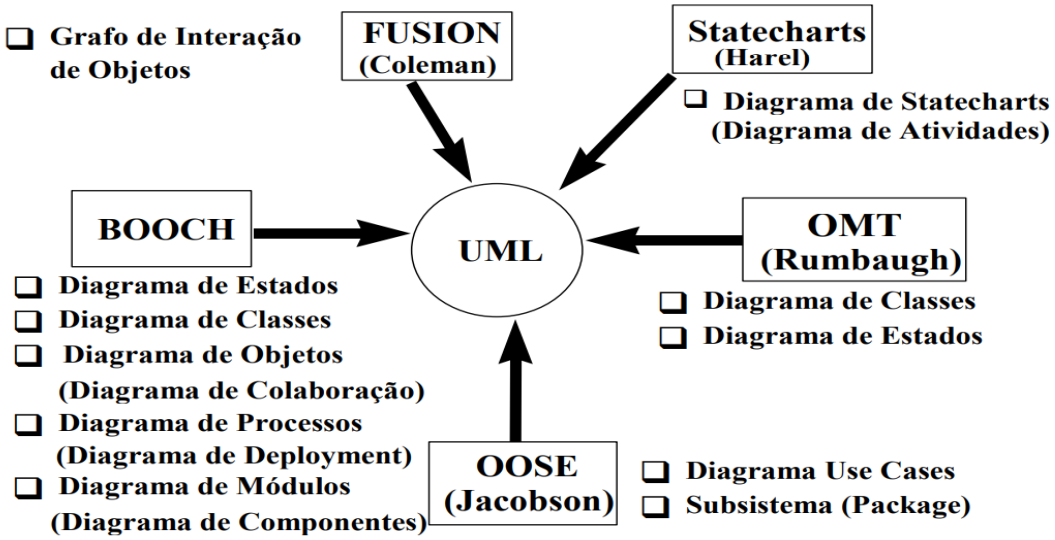


Fonte: https://web.archive.org/web/20101214121819/http://projetos.inf.ufsc.br:80/arquivos_projetos/projeto_721/artigo.tcc.pdf

OMG3 lançou uma RFP (Request for Proposals) para que outras empresas pudessem contribuir com a evolução da UML, chegando à versão 1.1. Após alcançar esta versão, a OMG3 passou a adotá-la como padrão e a se responsabilizar (através da RTF – Revision Task Force) pelas revisões. Essas revisões são, de certa forma, "controladas" a não provocar uma grande mudança no escopo original. Se observarmos as diferenças entre as versões atualmente, veremos que de uma para a outra não houve grande impacto, o que facilitou sua disseminação pelo mundo.

As últimas mudanças realizadas com relação aos diagramas foram realizadas em 2009 com a versão UML 2.2, onde uma das alterações foi a inclusão do diagrama de perfil.

Atualmente estamos na versão 2.4.1 lançada em 2011, mas ainda a mais utilizada é a versão 2.2.



Fonte: https://edisciplinas.usp.br/pluginfile.php/58064/mod_resource/content/1/Aula10_Visao_Geral_do_UML.pdf?forcedownload=1

A especificação de UML é composta por quatro documentos:

- Infraestrutura de UML(OMG, 2006),
- Superestrutura de UML (OMG, 2005c),
- Object Constraint Language (OCL) (OMG, 2005a) e
- Intercâmbio de Diagramas (OMG, 2005b).

Infraestrutura de UML: O conjunto de diagramas de UML constitui uma linguagem definida a partir de outra linguagem que define os elementos construtivos fundamentais. Esta linguagem que suporta a definição dos diagramas é apresentada no documento infraestrutura de UML.

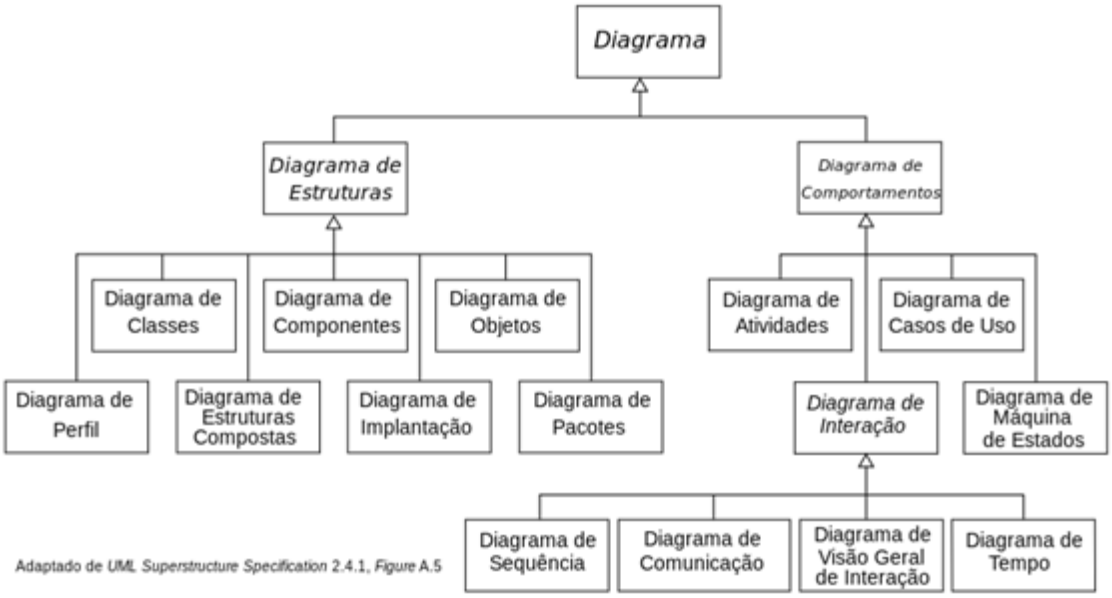
Superestrutura de UML: Documento que complementa o documento de infraestrutura e que define os elementos da linguagem no nível do usuário.

Linguagem para Restrições de Objetos (OCL): Documento que apresenta a linguagem usada para descrever expressões em modelos UML, com pré-condições, pós-condições e invariantes.

Intercâmbio de diagramas de UML: Apresenta uma extensão do meta-modelo voltado a informações gráficas. A extensão permite a geração de uma descrição no estilo XML orientada a aspectos gráficos que, em conjunto com o XML original permite produzir representações portáveis de especificações UML.

1.5. Principais diagramas UML

A linguagem UML 2.x é composta por quatorze diagramas, classificados em diagramas estruturais e diagramas de comportamento. A figura a seguir apresenta a estrutura das categorias utilizando a notação de diagramas de classes (OMG, 2006).



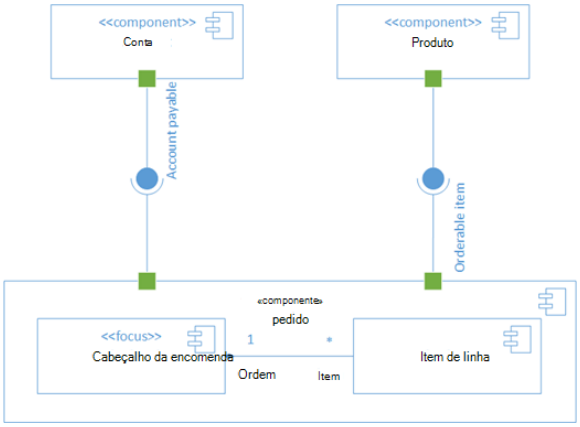
Fonte: https://blog.grancursosonline.com.br/uml/

A seguir, falaremos brevemente sobre cada diagrama, a princípio com exceção do diagrama de classes e do diagrama de caso de uso, pois deles falaremos logo em seguida estudando mais a fundo seus conceitos.

Diagrama de Estruturas

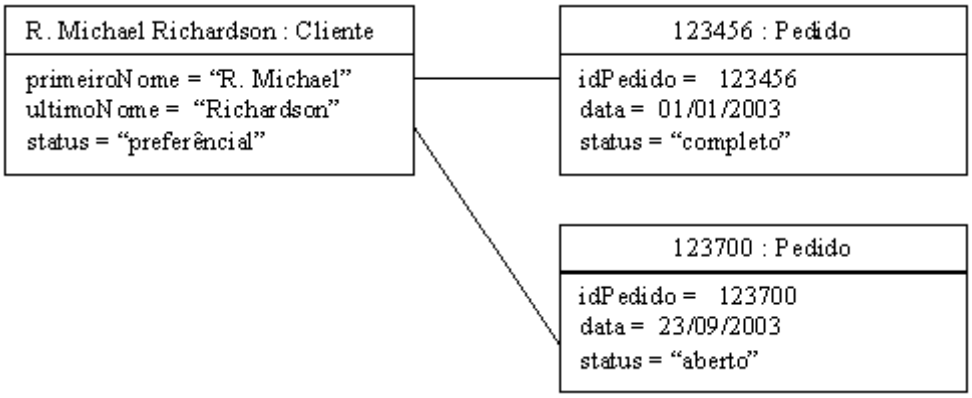
Diagrama de Componentes: O diagrama de componentes é um dos dois diagramas de UML voltados a modelar software baseado em componentes. Tem por finalidade indicar os componentes do software e seus relacionamentos. Este diagrama mostra os artefatos de que os componentes são feitos, como arquivos de código fonte,

bibliotecas de programação ou tabelas de bancos de dados. As interfaces é que possibilitam as associações entre os componentes.



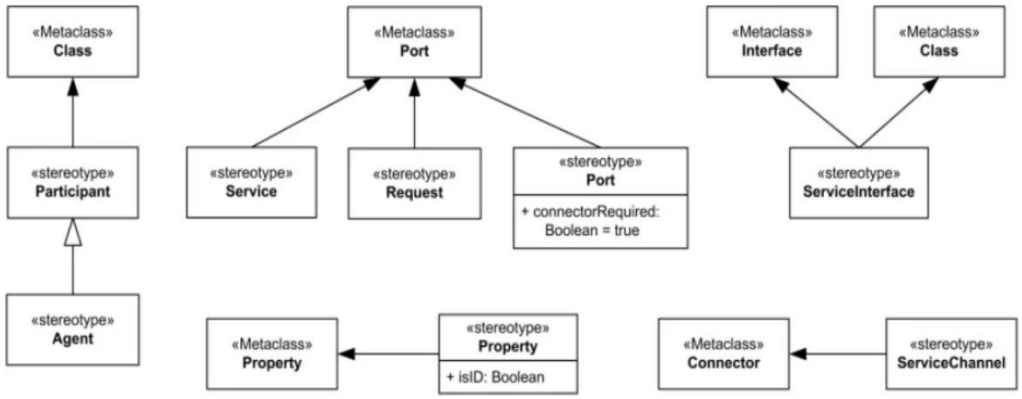
Fonte: https://support.microsoft.com/pt-pt/office/criar-um-diagrama-de-componente-uml-aa924ecb-e4d2-4172-976e-a78fa157b074

Diagrama de Objetos: O diagrama de objetos consiste em uma variação do diagrama de classes em que, em vez de classes, são representadas instâncias e ligações entre instâncias. A finalidade é descrever um conjunto de objetos e seus relacionamentos em um ponto no tempo. Contém objetos e vínculos e são usados para fazer a modelagem da visão de projeto estática de um sistema a partir da perspectiva de instâncias reais ou prototípicas.



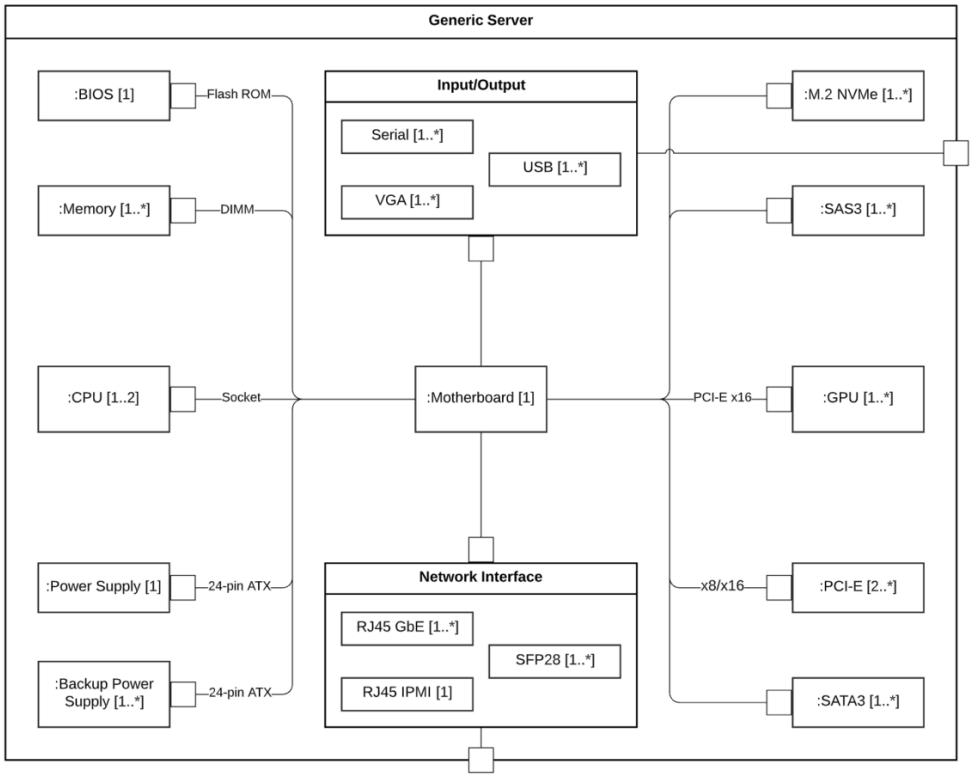
Fonte: https://homepages.dcc.ufmg.br/~amendes/GlossarioUML/glossario/conteudo/objetos/diagrama_de_objetos.htm

Diagrama de Perfil: Em UML, um diagrama de perfil é um diagrama de estrutura que permite o uso de perfis para um determinado meta-modelo. Apresentado com UML 2.2, este diagrama fornece uma representação dos conceitos usados na definição de perfis (pacotes, estereótipos, aplicação de perfis etc.).



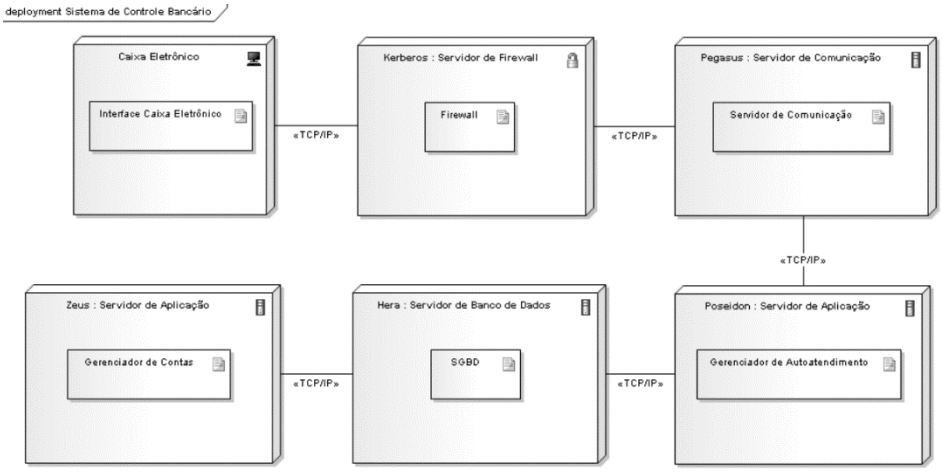
Fonte: https://pt.slideshare.net/natanaelsimoes/introduo-engenharia-de-software-e-uml

Diagrama de Estrutura Composta: O diagrama de estrutura composta fornece meios de definir a estrutura de um elemento e de focalizá-la no detalhe, na construção e em relacionamentos internos. É um dos novos diagramas propostos na segunda versão de UML, voltado a detalhar elementos de modelagem estrutural, como classes, pacotes e componentes, descrevendo sua estrutura interna. O diagrama de estrutura composta introduz a noção de "porto", um ponto de conexão do elemento modelado, a quem podem ser associadas interfaces. Também utiliza a noção de "colaboração", que consiste em um conjunto de elementos interligados através de seus portos para a execução de uma funcionalidade específica – recurso útil para a modelagem de padrões de projeto (SILVA, 2007).



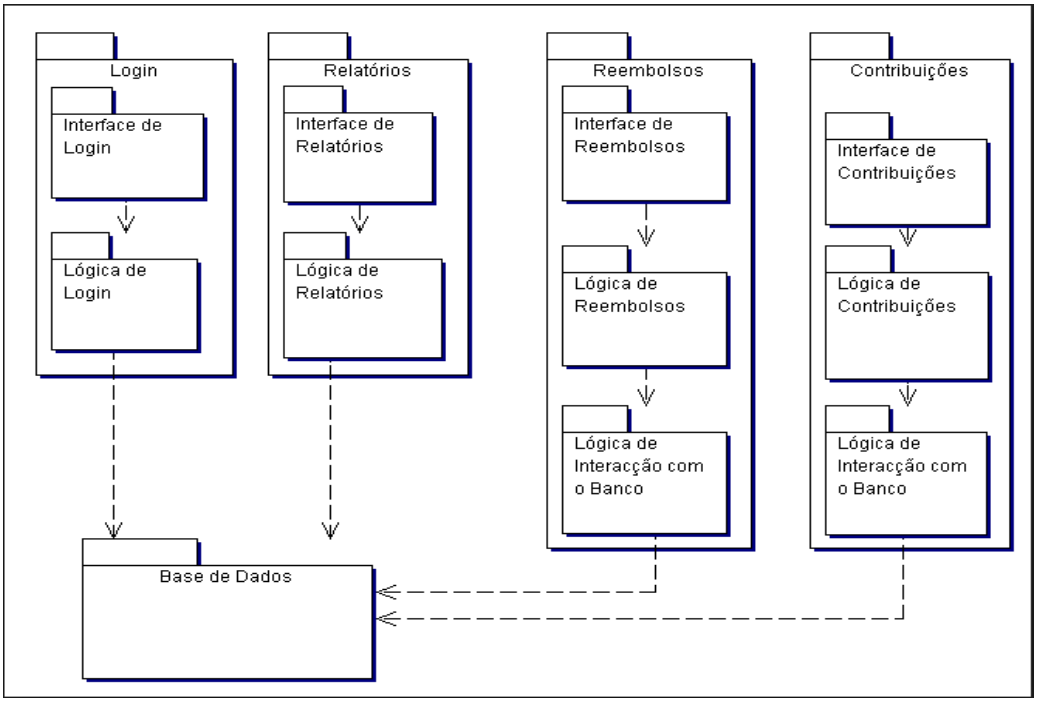
Fonte: https://www.lucidchart.com/pages/pt/diagrama-de-estrutura-composta-uml

Diagrama de Implantação: O diagrama de utilização, também denominado diagrama de implantação, consiste na organização do conjunto de elementos de um sistema para a sua execução. O principal elemento deste diagrama é o nodo, que representa um recurso computacional. Podem ser representados em um diagrama tantos os nodos como instâncias de nodos. O diagrama de implantação é útil em projetos onde há muita interdependência entre pedaços de hardware e software.



Fonte: http://micreiros.com/diagrama-de-implantacao/

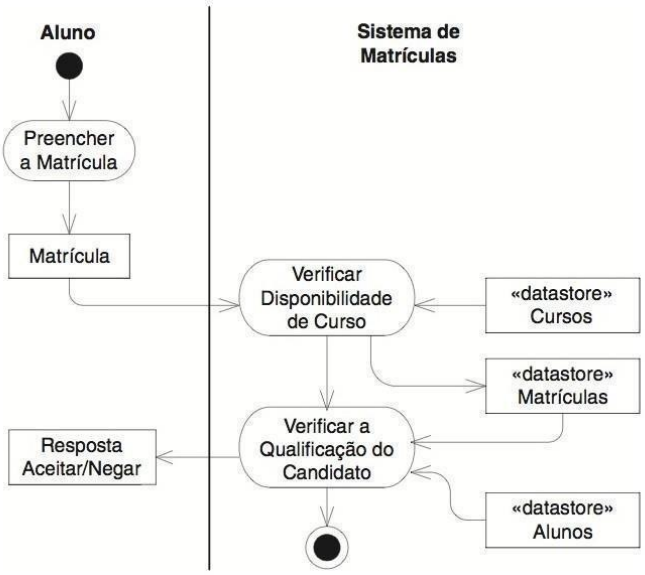
Diagrama de Pacotes: O pacote é um elemento sintático voltado a conter elementos sintáticos de uma especificação orientada a objetos. Esse elemento foi definido na primeira versão de UML para ser usado nos diagramas então existentes, como diagrama de classes, por exemplo. Na segunda versão da linguagem, foi introduzido um novo diagrama, o diagrama de pacotes, voltado a conter exclusivamente pacotes e relacionamentos entre pacotes (SILVA, 2007). Sua finalidade é tratar a modelagem estrutural do sistema dividindo o modelo em divisões lógicas e descrevendo as interações entre ele em alto nível.



Fonte: https://micreiros.com/diagrama-de-pacotes/

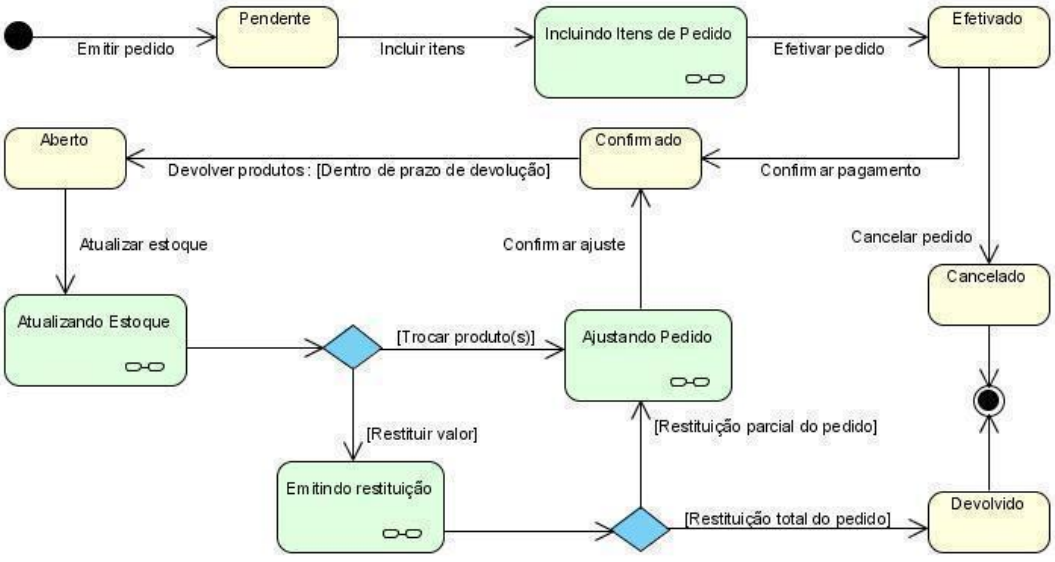
Diagrama de Comportamentos

Diagrama de Atividades: O diagrama de atividades representa a execução das ações e as transições que são acionadas pela conclusão de outras ações ou atividades. Uma atividade pode ser descrita como um conjunto de ações e um conjunto de atividades. A diferença básica entre os dois conceitos que descrevem comportamento é que a ação é atômica, admitindo particionamento, o que não se aplica a atividade, que pode ser detalhada em atividades e ações (SILVA, 2007).



Fonte:https://www.researchgate.net/figure/Figura-3-Diagrama-de-Atividades_fig2_284032090

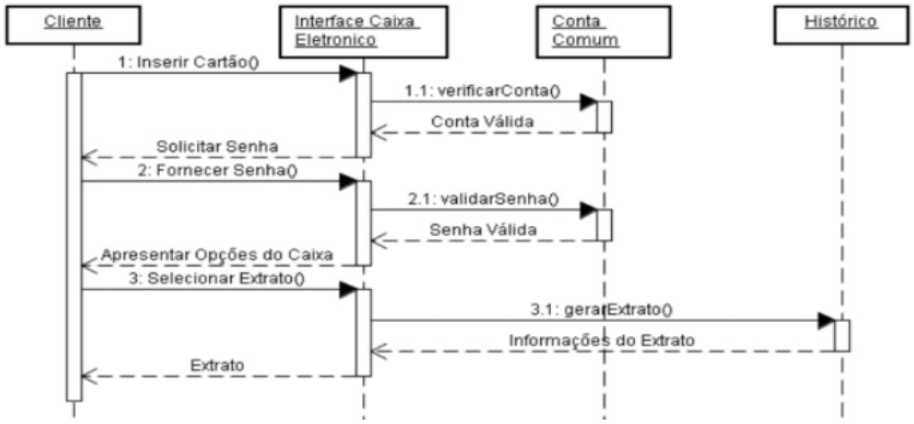
Diagrama de Máquina de Estados: O diagrama de máquina de estados tem como elementos principais o estado, que modela uma situação em que o elemento modelado pode estar ao longo de sua existência, e a transição, que leva o elemento modelado de um estado para o outro. O diagrama de máquina de estados vê os objetos como máquinas de estados ou autômatos (opera de maneira automática, imitando movimentos humanos), finitos que poderão estar em um estado pertencente a uma lista de estados finita e que poderão mudar o seu estado através de um estímulo pertencente a um conjunto finito de estímulos.



Fonte:https://dtic.tjpr.jus.br/wiki/-/wiki/Governan%C3%A7a-TIC/Modelo+de+M%C3%A1quina+de+Estados/pop_up

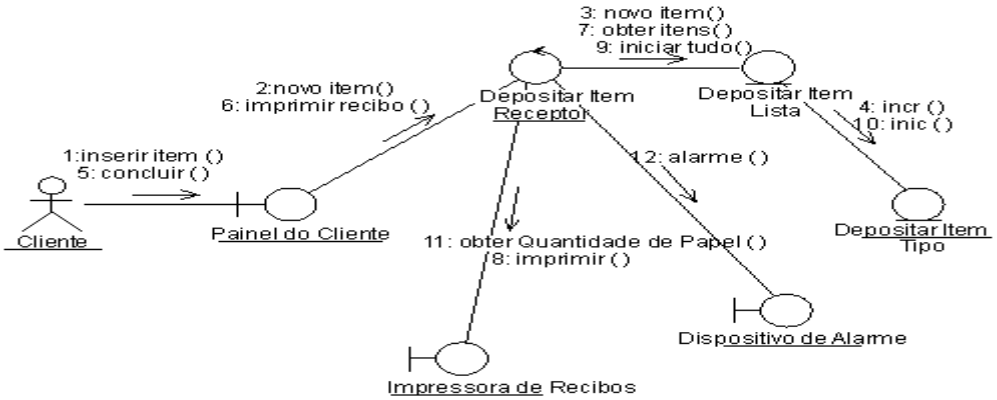
Diagrama de Interação

Diagrama de Sequência: O diagrama de sequência mostra a troca de mensagens entre diversos objetos, em uma situação específica e delimitada no tempo. Coloca ênfase especial na ordem e nos momentos nos quais mensagens para os objetos são enviadas. Em diagramas de sequência, objetos são representados através de linhas verticais tracejadas (denominadas como linha de existência), com o nome do objeto no topo. O eixo do tempo é também vertical, aumentando para baixo, de modo que as mensagens são enviadas de um objeto para outro na forma de setas com a operação e os nomes dos parâmetros.



Fonte:https://profandreagarcia.wordpress.com/2019/11/19/diagrama-de-sequencia-uml/

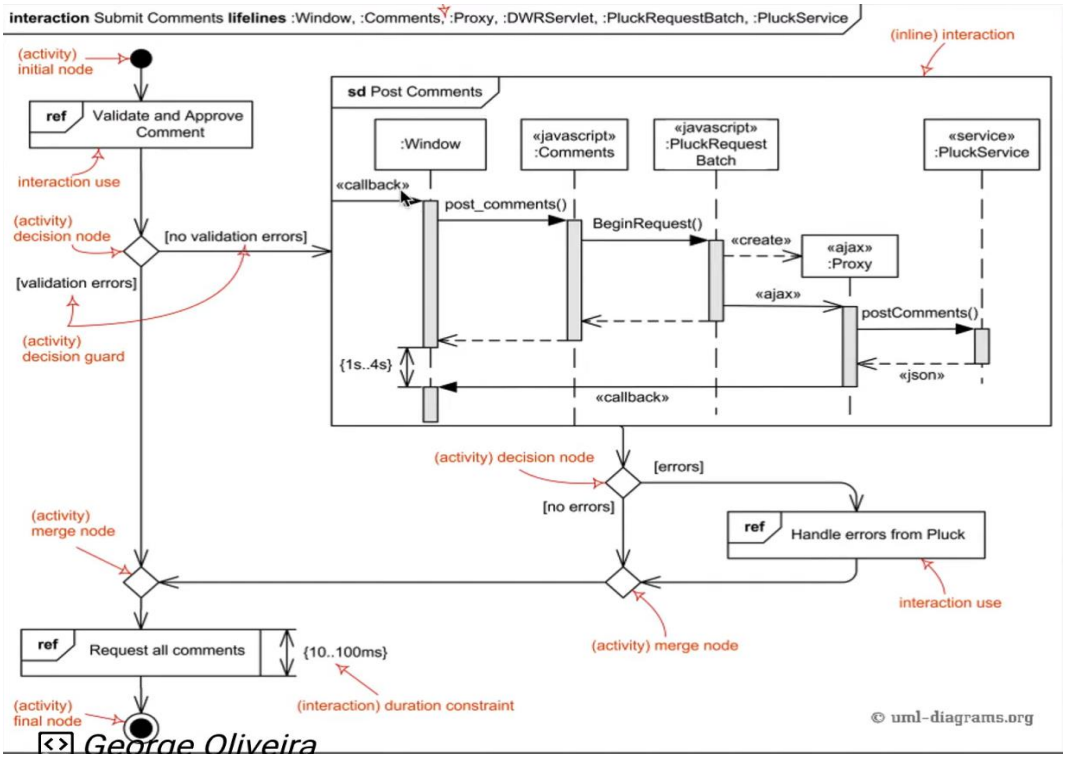
Diagrama de Comunicação: Os elementos de um sistema trabalham em conjunto para cumprir os objetos do sistema e uma linguagem de modelagem precisa poder representar esta característica. O diagrama de comunicação é voltado a descrever objetos interagindo e seus principais elementos sintáticos são “objeto” e “mensagem”. Corresponde a um formato alternativo para descrever interação entre objetos. Ao contrário do diagrama de sequência, o tempo não é modelado explicitamente, uma vez que a ordem das mensagens é definida através de enumeração. Vale ressaltar que tanto o diagrama de comunicação como o diagrama de sequência são diagramas de interação.



Fonte:https://www.cin.ufpe.br/~gta/rup-vc/core.base_rup/guidances/guidelines/communication_diagram_FFFEA1B5.html

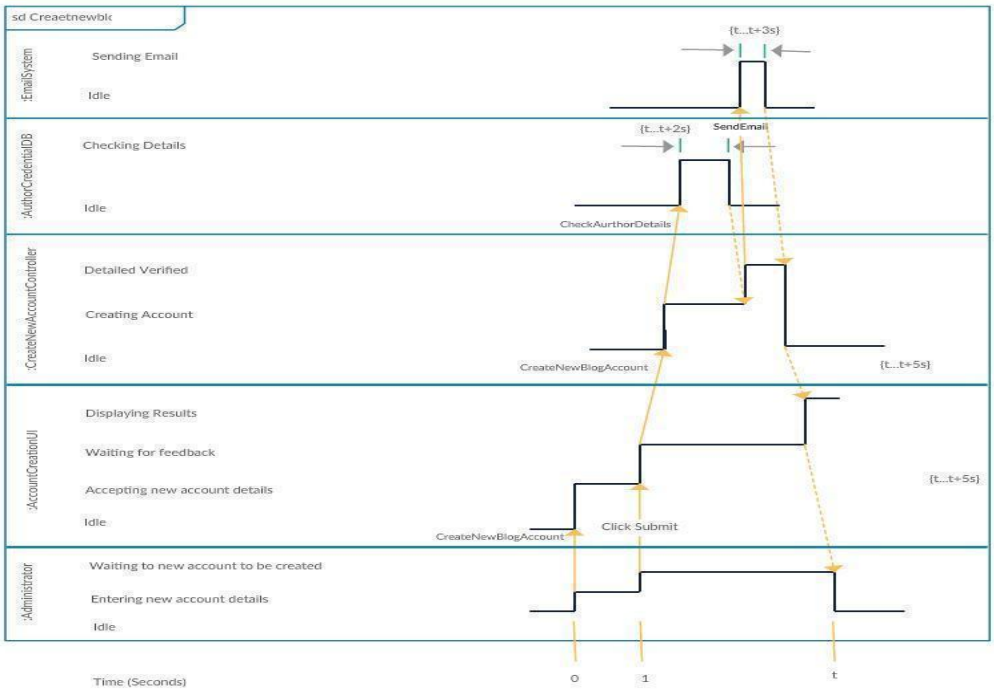
Diagrama de Visão Geral de Integração: O diagrama de visão geral de interação é uma variação do diagrama de atividades, proposto na versão atual de UML. Seus elementos sintáticos são os mesmos do diagrama de atividades. As interações que fazem

parte do diagrama de visão geral de interação podem ser referências a diagramas de interação existentes na especificação tratada.



Fonte: https://www.youtube.com/watch?v=AqNDWYpgOm8

Diagrama de Temporização: O diagrama de temporização consiste na modelagem de restrições temporais do sistema. É um diagrama introduzido na segunda versão de UML, classificado como diagrama de interação. Este diagrama modela interação e evolução de estados.



Fonte: https://creately.com/blog/pt/diagrama/guia-de-tipos-de-diagramas-uml-aprenda-sobre-todos-os-tipos-de-diagramas-uml-com-exemplos/

1.6. Diagrama de caso de uso

Na Linguagem de modelagem unificada (UML), o diagrama de caso de uso resume os detalhes dos usuários do seu sistema (também conhecidos como atores) e suas interações. Para criar um diagrama de caso de uso, usamos um conjunto de símbolos e conectores especiais. Um bom diagrama de caso de uso ajuda sua equipe a representar e discutir:

- Cenários em que o sistema ou aplicativo interage com pessoas, organizações ou sistemas externos;
- Metas que o sistema ou aplicativo ajuda essas entidades (conhecidas como atores) a atingir;
- O escopo do sistema.

Quando usar o diagrama de caso de uso

O diagrama de caso de uso não oferece muitos detalhes — não espere, por exemplo, que ele mostre a ordem em que os passos são executados. Em vez disso, um diagrama de caso de uso adequado dá uma visão geral do relacionamento entre casos de uso, atores e sistemas. Os especialistas recomendam usar o diagrama de caso de uso para complementar um caso de uso descrito em texto.

UML é o kit de ferramentas de modelagem para criar o diagrama. O caso de uso é representado por uma forma oval rotulada. Bonecos palito representam os atores no processo, e a participação do ator no sistema é modelada com uma linha entre o ator e o caso de uso. Para representar o limite do sistema, desenhe uma caixa em torno do próprio caso de uso.

O diagrama de caso de uso UML é ideal para:

- Representar as metas de interações entre sistemas e usuários;
- Definir e organizar requisitos funcionais no sistema;
- Especificar o contexto e os requisitos do sistema;
- Modelar o fluxo básico de eventos no caso de uso.

Elementos do diagrama de caso de uso

Para entender o que é um diagrama de caso de uso, é necessário primeiro saber como é ele feito. São componentes comuns:

- **Atores:** os usuários que interagem com o sistema. Ator pode ser uma pessoa, organização ou sistema externo que interage com seu aplicativo ou sistema. Eles devem ser objetos externos que produzam ou consumam dados, são representados por bonecos palito, e correspondem as pessoas que realmente implementam os casos de uso.
- **Sistema:** uma sequência específica de ações e interações entre os atores e o sistema. O sistema também pode ser chamado de cenário.
- **Metas**: o resultado da maioria dos casos de uso. Um diagrama criado corretamente deve descrever as atividades e variantes usadas para atingir a meta.
- A notação do diagrama de caso de uso é bastante objetiva e não envolve a mesma quantidade de símbolos de outros diagramas UML.
- **Caso de uso:** formato oval na horizontal e que representam os diferentes usos que um usuário pode ter.
- **Associações:** uma linha entre atores e casos de uso. Nos diagramas complexos, é importante saber quais atores estão associados a quais casos de uso.

Relacionamentos de Associação

Nos modelos UML, uma associação é um relacionamento entre dois classificadores, como classes ou casos de uso, que descreve as razões para o relacionamento e as regras que o regem.

Relacionamentos de Generalização

Na modelagem UML, um relacionamento de generalização é aquele no qual um elemento de modelo (o filho) tem como base outro elemento de modelo (o pai). Os relacionamentos de generalização são utilizados em diagramas de classe, componente, implementação e caso de uso para indicar que o filho recebe todos os atributos, operações e relacionamentos definidos no pai.

Relacionamentos de Inclusão

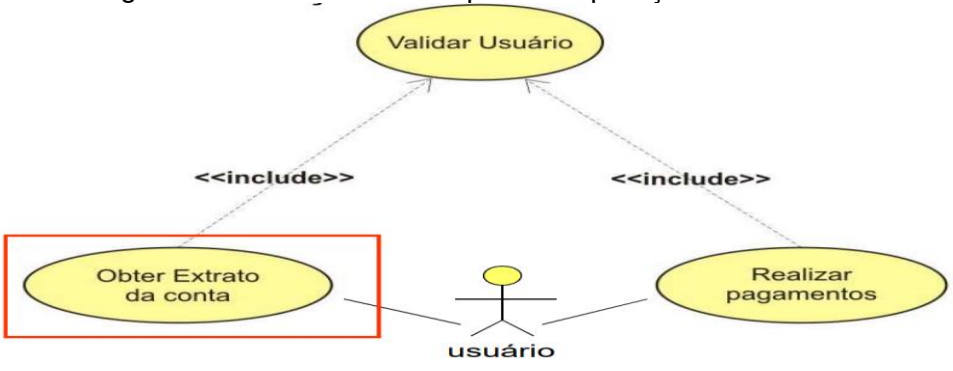
Na modelagem UML, um relacionamento de inclusão é aquele no qual um caso de uso (o caso de uso base) inclui a funcionalidade de outro caso de uso (o caso de uso de inclusão). O relacionamento de inclusão suporta a reutilização da funcionalidade em um modelo de caso de uso.

- A relação de inclusão («include») entre casos de uso corresponde a uma relação típica de delegação, significando que o caso base incorpora o comportamento do outro caso relacionado.
- Usa-se a relação de inclusão para evitar a descrição dos mesmos fluxos de ações inúmeras vezes. A relação de inclusão é representada por uma relação de dependência (seta tracejada) com o estereótipo «include».

Exemplo: Os casos de uso “Obter Extrato de Conta” ou “Realizar Pagamentos” exigem que seja realizada previamente uma validação do respectivo usuário. Para que essa funcionalidade não seja especificada mais que uma vez, os casos anteriores incorporam-na (como sua) ao estabelecerem uma relação de inclusão com o caso “Validar Usuário”.

Nome: Obter Extrato de Conta
Cenário Principal
Incluir caso de uso “Validar Usuário”.

- Obter e verificar o número da conta.
- Selecionar todas as linhas de movimentos realizados nos últimos 30 dias.
- Produzir uma lista resumo com esses movimentos, apresentando a data, o tipo de movimento (débito ou crédito), uma breve descrição e o valor do movimento.
- Produzir o saldo corrente da conta.
- Emitir um documento com essa informação, imprimindo no terminal do caixa eletrônico o referido documento.
- Apresentar mensagem no visor do terminal para o cliente retirar o extrato.
- Registrar na conta do cliente que esta operação foi realizada com sucesso.

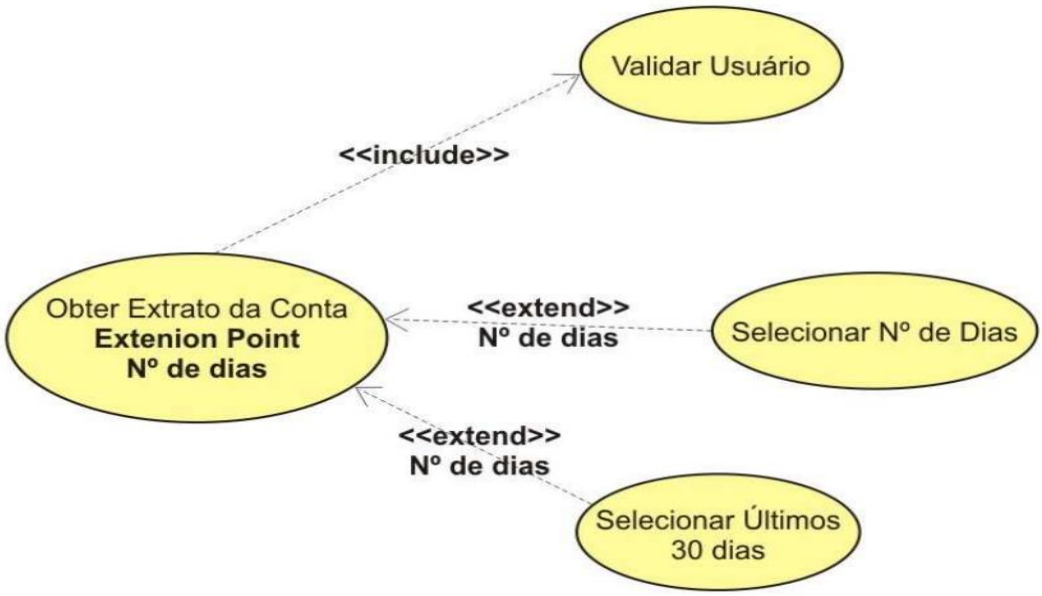


Relacionamento de Extensão

Na modelagem UML, é possível utilizar um relacionamento de extensão para especificar que um caso de uso (extensão) estende o comportamento de outro caso de uso (base). Esse tipo de relacionamento revela detalhes sobre um sistema ou aplicativo que normalmente estão ocultos em um caso de uso.

Uma relação de extensão («extend») entre casos de uso significa que o caso de uso base incorpora implicitamente o seu comportamento num local especificado indiretamente pelo caso de uso que é usado. Ou seja, o caso de uso destino pode ser estendido com o comportamento de outro(s) caso(s) de uso(s). Uma relação de extensão permite representar:

- A parte de um caso que um usuário vê como opcional, ou como existindo várias alternativas.
- Um sub-fluxo de ações que é executado apenas se determinadas condições se verificarem.
- Vários fluxos de ações que podem ser inseridos num determinado ponto de extensão, de forma controlada, através de uma interação explícita com um ator.
- O caso de uso destino é estendido num ou mais pontos, designados por pontos de extensão os quais são mecanismos de variabilidade. Ou seja, são pontos de entrada do caso de uso que lhe dá algum nível de configuração e versatilidade.



Exemplo:
Nome: Obter Extrato de Conta
Pontos de Extensão: N.º de dias
Cenário Principal

Incluir caso de uso “Validar Usuário”. Obter e verificar o número da conta. Selecionar o n.º de dias com base no qual se produz o extrato. (N.º de dias). Por omissão são selecionados os últimos 30 dias. Produzir uma lista resumo com esses movimentos, apresentando a data, o tipo de movimento (débito ou crédito), uma breve descrição e o valor do movimento. Produzir o saldo corrente da conta. Emitir um documento com essa informação produzida imprimindo no terminal do caixa eletrônico o referido documento. Apresentar mensagem no visor do terminal para o cliente retirar o extrato. Registrar na conta do cliente que esta operação foi realizada com sucesso.

Nome: Seleciona N.º de Dias
Tipo: Abstrato
Cenário Principal

É apresentado uma tela em que o usuário pode especificar o n.º de dias desejado, através da marcação em vários botões numéricos (de '0' a '9'). Há uma caixa de texto construída dinamicamente que vai apresentando o valor corrente. Por fim, o usuário aciona o botão "Confirmar" e o valor construído é retornado ao caso destino no seu respectivo ponto de extensão.

Cenário Alternativo 1

Idêntico ao cenário principal. Em qualquer momento o usuário pode marcar sobre o botão "Apagar" de modo a apagar o algarismo introduzido mais recentemente.

Cenário Alternativo 2

Idêntico ao cenário principal. Quando o usuário marca "Confirmar" e o valor introduzido for superior a 59 dias é apresentada uma mensagem de aviso que o número máximo é 59, e o caso é reiniciado.

Cenário Alternativo 3

Idêntico ao cenário principal. Em qualquer momento o usuário pode selecionar o botão "Cancelar" – O caso termina e é retornado o valor 1 (dia) por omissão.

1.7. Diagrama de classe

Diagramas de classes estão entre os tipos mais úteis de diagramas UML pois mapeiam de forma clara a estrutura de um determinado sistema ao modelar suas classes, seus atributos, operações e relações entre objetos.

Bastante usado por engenheiros de software para documentar arquiteturas de software, os diagramas de classes são um tipo de diagrama da estrutura porque descrevem o que deve estar presente no sistema a ser modelado.

A forma de classe em si consiste em um retângulo com três linhas. A linha superior contém o nome da classe, a linha do meio, os atributos da classe e a linha inferior expressa os métodos ou operações que a classe pode utilizar. Classes e subclasses são agrupadas juntas para mostrar a relação estática entre cada objeto.

1.7.1. Os benefícios de diagramas de classes

Diagramas de classes oferecem uma série de benefícios para qualquer organização. Use diagramas de classes UML para:

- Ilustrar modelos de dados para sistemas de informação, não importa quão simples ou complexo.
- Entender melhor a visão geral dos esquemas de uma aplicação.
- Expressar visualmente as necessidades específicas de um sistema e divulgar essas informações por toda a empresa.
- Criar gráficos detalhados que destacam qualquer código específico necessário para ser programado e implementado na estrutura descrita.
- Fornecer uma descrição independente de implementação de tipos utilizados em um sistema e passados posteriormente entre seus componentes.

1.7.2. Componentes básicos de um diagrama de classes

O diagrama de classes padrão é composto de três partes:

- **Parte superior:** contém o nome da classe. Esta parte é sempre necessária, seja falando do classificador ou de um objeto.

- **Parte do meio:** contém os atributos da classe. Use esta parte para descrever as qualidades da classe. É necessário somente quando se descreve uma instância específica de uma classe.
- **Parte inferior:** inclui as operações da classe (métodos). Exibido em formato de lista, cada operação ocupa sua própria linha. As operações descrevem como uma classe interage com dados.

1.7.3. Modificadores de acesso de membro

Todas as classes têm diferentes níveis de acesso, dependendo do modificador de acesso (visibilidade). Veja os níveis de acesso com seus símbolos correspondentes:

- Público (+)
- Privado (-)
- Protegido (#)
- Pacote (~)
- Derivado (/)
- Estático (sublinhado)

1.7.4. Escopos para membros

Há dois escopos para membros: classificadores e instâncias.

Classificadores são membros estáticos, enquanto **instâncias** são as instâncias específicas da classe.

1.7.5. Componentes adicionais de diagramas de classes

Dependendo do contexto, as classes em um diagrama de classes podem representar os principais objetos, interações no aplicativo ou classes a serem programadas. Para responder à pergunta “o que é um diagrama de classes em UML?”, você deve primeiro entender sua composição básica.

1.7.5.1. Classes

Um *template* para a criação de objetos e implementação de comportamento em um sistema. Em UML, uma classe representa um objeto ou um conjunto de objetos que compartilham uma estrutura e comportamento comum. São representadas por um retângulo que inclui linhas do nome da classe, seus atributos e suas operações. Ao desenhar uma classe em um diagrama de classes, somente a primeira linha deve ser preenchida — as outras são opcionais caso queira fornecer mais detalhes.

- **Nome:** a primeira linha em uma forma de classe.
- **Atributos:** a segunda linha em uma forma de classe. Cada atributo da classe é exibido em uma linha separada.
- **Métodos:** a terceira linha em uma forma de classe. Também conhecidos como operações, métodos são exibidos em formato de lista, com cada operação representada em sua própria linha.

Sinais: símbolos que representam comunicações unidirecionais e assíncronas entre objetos ativos.

Tipos de dados: classificadores que definem valores de dados. Os tipos de dados podem modelar tipos primitivos e enumerações.

Pacotes: formas projetadas para organizar classificadores relacionados em um diagrama. São simbolizados por uma grande forma de retângulo com abas.

Interfaces: uma coleção de assinaturas de operações e/ou definições de atributos que definem um conjunto coeso de comportamentos. Interfaces são semelhantes às classes, exceto que uma classe pode ter uma instância de seu tipo, e uma interface deve ter pelo menos uma classe para implementá-la.

Enumerações: representações de tipos de dados definidos pelo usuário. Uma enumeração inclui grupos de identificadores que representam os valores da enumeração.

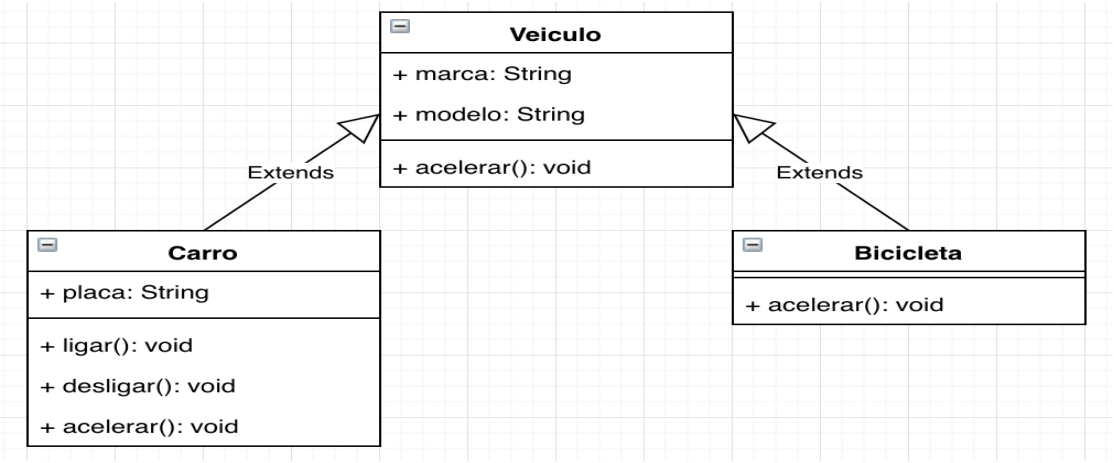
Objetos: instâncias de uma classe ou classes. Objetos podem ser adicionados a um diagrama de classes para representar instâncias concretas ou prototípicas.

Artefatos: elementos de modelo que representam as entidades concretas em um sistema de software, tais como documentos, bancos de dados, arquivos executáveis, componentes de software etc.

1.7.5.2. Interações

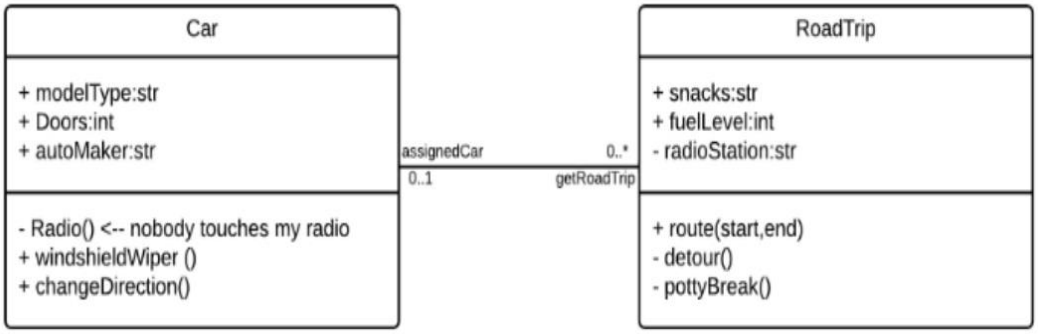
O termo “interações” refere-se às diversas relações e ligações que podem existir em diagramas de classes e objetos. Algumas das interações mais comuns incluem:

- **Hereditariedade:** também conhecida como generalização, é o processo de um secundário, ou subclasse, assumindo a funcionalidade de um primário, ou superclasse. É simbolizada por uma linha conectada reta com uma ponta de seta fechada apontando para a superclasse.



Fonte: https://www.treinaweb.com.br/blog/os-pilares-da-orientacao-a-objetos

- **Associação bidirecional:** a relação padrão entre duas classes. Ambas as classes estão cientes uma da outra e de sua relação entre si. Essa associação é representada por uma linha reta entre duas classes.

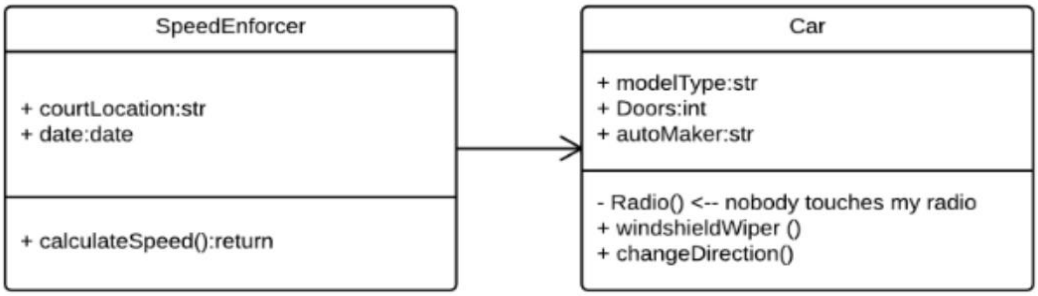


Fonte: https://www.lucidchart.com/pages/pt/o-que-e-diagrama-de-classe-uml

No exemplo acima, a classe Carro e a classe RoadTrip (passeio) estão inter-relacionadas. Em uma extremidade da linha, o Carro assume a associação de “assignedCar” (carro atribuído) com o valor multiplicidade de 0..1, portanto, quando a instância de RoadTrip existe, pode ter uma instância de Carro associada a ela ou nenhum Carro associado a ela. Neste caso, uma classe separada de Caravana, com valor multiplicidade de 0..* é necessária para demonstrar que uma RoadTrip pode ter múltiplas instâncias de Carros associadas a ela. Já que uma instância Carro pode ter múltiplas

associações “getRoadTrip” (obter passeio) — ou seja, um carro pode fazer diversos passeios —, o valor multiplicidade é 0..*

- **Associação unidirecional:** uma relação um pouco menos comum entre duas classes. Uma classe tem conhecimento da outra e interage com ela. A associação unidirecional é modelada por uma linha reta de ligação com uma ponta de seta aberta da classe conhecimento à classe conhecida.



Fonte: https://www.lucidchart.com/pages/pt/o-que-e-diagrama-de-classe-uml

Por exemplo, em seu passeio pelo estado do Arizona você pode passar por um radar de velocidade onde uma câmera registra sua atividade de condução, mas você só saberá quando receber a notificação pelo correio. Não é desenhado na imagem mas, neste caso, o valor multiplicidade seria 0..*, dependendo de quantas vezes você conduz o carro pela câmera de velocidade.

1.8. Exercício de fixação.

De acordo com o sistema de pizzaria idealizado por vocês, criem o diagrama de classes do mesmo.

2. Por que Orientação a Objetos?

Inicialmente vamos olhar para os precursores da programação orientada a objetos (POO). Atualmente, quando usa um computador, você está tirando proveito de mais de 50 anos de refinamento. Antigamente, a programação era engenhosa: os programadores introduziam os programas diretamente na memória principal do computador. Os programadores escreviam seus programas em linguagem binária, tal programação era extremamente propensa a erros e, a falta de estrutura, tornou a manutenção do código praticamente impossível. Além disso, o código da linguagem binária não era muito acessível.

Quando os computadores se tornaram mais comuns, linguagens de nível mais alto e procedurais começaram; a primeira foi FORTRAN. Entretanto, linguagens procedurais posteriores como ALGOL, tiveram mais influência sobre a orientação a objetos (OO). As linguagens procedurais permitem ao programador reduzir um programa em procedimentos refinados para processar dados. Esses procedimentos geram a execução de um programa procedural. O programa termina quando acaba de chamar sua lista de procedimentos. Esse paradigma apresentou diversas melhorias em relação à linguagem binária, incluindo a adição de uma estrutura de apoio: o procedimento. As funções menores não são apenas mais fáceis de entender, mas também são mais fáceis de depurar.

Por outro lado, a programação procedural limita a reutilização de código. Isso leva os programadores, com muita frequência, a produzir “código de espaguete” – código cujo caminho de execução se assemelhava a uma tigela de espaguete. Dessa forma, a natureza voltada aos dados da programação procedural causou alguns problemas próprios.

A programação modular, com uma linguagem como MODULA2, tenta melhorar algumas das deficiências encontradas na programação procedural. A programação modular divide os programas em vários componentes ou módulos constituintes. Ao contrário da programação procedural, que separa dados e procedimentos, os módulos combinam os dois.

A programação modular também é um híbrido procedural que ainda divide um programa em vários procedimentos. Porém, em vez de atuar em dados brutos, esses procedimentos manipulam módulos.

A Programação Orientada a Objetos (POO) dá o próximo passo lógico após a programação modular, adicionando herança e polimorfismo ao módulo. A POO estrutura um programa, dividindo-o em vários objetos de alto nível. Cada objeto modela algum aspecto do problema a ser resolvido. Escrever listas sequenciais de chamadas de procedimento para dirigir o fluxo do programa não é mais o foco da programação sob a OO. Em vez disso, os objetos interagem entre si, para orientar o fluxo global do programa. De certa forma, um programa OO se torna uma simulação viva do problema.

Os objetos nos obrigam verificar, em nível conceitual, tudo o que eles fazem, ou seja, nos fazem listar todos os seus comportamentos. Ver um objeto a partir do nível conceitual não é definir sua *implementação* (em termos de programação, implementação é o código). Essa mentalidade nos obriga a pensar em programas em termos naturais e reais. Ou seja, os objetos permitem que você modele programas nos substantivos, verbos e adjetivos do *domínio* de seu problema. Sendo o domínio, o espaço onde um problema reside, o conjunto de conceitos que representam os aspectos importantes do problema.

Quando recua e pensa nos termos do problema que está resolvendo, você evita se emaranhar nos detalhes da implementação. Provavelmente alguns de seus objetos de alto nível precisarão interagir com o computador. Entretanto, o objeto isolará essas interações do restante do sistema.

Vamos pensar em um objeto “carrinho de compras”. Ocultação da implementação significa que o caixa não vê dados brutos ao totalizar um pedido. O caixa não sabe procurar, em certas posições de memória, números de item e outra variável para um cupom. Em vez disso, o caixa interage com objetos item. Ele sabe perguntar quanto custa o item.

A computação atualmente é parte do dia a dia da Sociedade. Em termos de hardware e software a evolução é constante e tem atingido altos níveis. Pode-se dizer que para boa parte dos problemas do mundo real que precisam ser mapeados em sistemas computacionais, mais precisamente em softwares, já possuem as tecnologias de hardware e software necessários.

Assim o problema neste caso deixa de ser onde o sistema executar e passa a ser como resolver o problema de maneira mais eficiente, ou seja, como representar o problema do mundo real para o mundo computacional capturando todas as características necessárias e que execute a melhor solução possível. E é quando se começa a falar em metodologias, tecnologias e técnicas de desenvolvimento de software. O que está dominando atualmente o mercado é justamente a Orientação a Objetos (OO).

A técnica da orientação a objeto está entre as melhores maneiras conhecidas para se desenvolver Sistemas de Informação Automatizados grandes ou complexos.

Aprender a raciocinar orientado a objetos, bem como a utilizar corretamente as suas técnicas não é fácil, mas pode-se afirmar que é essencial para qualquer profissional da área atualmente.

Portanto, podemos concluir que a Orientação a Objeto é um paradigma de análise, projeto e programação de sistemas de informação automatizados baseado na composição e interação entre diversas unidades de software chamadas objetos.

3. Plataforma JAVA

As aulas de nossa UC serão baseadas na linguagem Java, mas antes de colocarmos as “mãos na massa”, uma breve introdução à mesma.

A história começa em 1991, quando um grupo de empregados da Sun Microsystems iniciaram um projeto para pequenos dispositivos eletrônicos de consumo, tais como o PDA (*Personal Digital Assistant*), o projeto recebeu o nome de Projeto *Green*, e James Gosling assumiu sua coordenação.

A ideia era possibilitar a criação de programas portáveis que pudessem ser executados em diversos dispositivos. Mas a equipe teria que desenvolver programas específicos para cada tipo de dispositivo, dai surgiu a ideia de desenvolver um sistema operacional que permitiria a utilização de seus programas pelos mais diversos tipos de equipamento. A nova linguagem foi batizada de Oak (carvalho), uma referência ao carvalho que James Gosling visualizava a partir de seu escritório. O sistema operacional que foi desenvolvido, foi chamado de GreenOS, e junto com ele foi construída uma interface gráfica padronizada.

Após ter um sistema operacional e uma interface gráfica, a equipe desenvolveu um avançado PDA chamado de Star7, a Sun Microsystems participou de uma competição pública para o desenvolvimento de uma tecnologia para TV a Cabo interativa, onde seria aplicado o Star7, no entanto ela perdeu essa competição, mesmo sendo um produto de alta qualidade o mercado ainda não estava preparado para o Star7. Perto de cortar o financiamento do projeto, a Sun decidiu abandonar a ênfase nos dispositivos eletrônicos e se voltar para a internet que já começava a crescer.

O nome da linguagem desenvolvida pelo projeto Green foi mudada de Oak para Java, que foi uma homenagem à uma ilha da Indonésia de onde os Norte-Americanos importavam o café que era consumido pela equipe de James Gosling. Até 1994, não havia uma aplicação definida para o Java. Foi quando Jonathan Payne e Patrick Naughton criaram um novo navegador para Web que podia executar programas escritos em Java (*applets*), batizado de *Web Runner*. E em 1996, em uma iniciativa inédita, a Sun Microsystems resolveu disponibilizar gratuitamente um kit de desenvolvimento de software para a comunidade, que ficou conhecido como Java *Developer`s* Kit (JDK). Desde então a aceitação da tecnologia Java cresceu rapidamente entre empresas e desenvolvedores. A Sun Microsystems lançou o JDK 1.1 com melhorias significativas para o desenvolvimento de aplicações gráficas e distribuídas. Depois disso, a empresa continuou lançando novas versões gratuitas com novas melhorias e recursos.

Em abril de 2009, a Oracle ofereceu US$ 7,4 bilhões pela aquisição da Sun Microsystems e a proposta foi aceita. Essa aquisição deu à Oracle a propriedade de vários produtos, incluindo o Java e o sistema operacional Solaris. Em comunicado, a Oracle afirmou que o Java foi o software mais importante adquirido ao longo de sua história. Muitas especulações foram feitas a cerca do futuro do Java depois de passar a ser propriedade da Oracle. Mas com certeza essa aquisição contribuiu muito para que o Java tivesse um salto qualitativo.

Sua principal característica é ser multiplataforma, o que quer dizer que ela é uma aplicação que pode ser executada em qualquer plataforma que seja suportada por esta linguagem, sempre de acordo com as limitações impostas pelas plataformas que forem utilizadas.

Mesmo hoje pertencendo a Oracle, ela não perdeu a sua marca que é ser um software de código aberto e que é mantida pela comunidade de usuários e empresas Java além da própria Oracle.

Na plataforma Java devemos salientar que os conceitos de orientação a objetos que serão vistos poderão ser aplicados também na plataforma .NET ou quaisquer linguagens que suporte Orientação a Objetos.

No primeiro momento, os dois elementos mais importantes da plataforma Java são:

- A linguagem de programação Java.
- O ambiente de execução Java.

A linguagem de programação Java permite que os conceitos de orientação a objetos sejam utilizados no desenvolvimento de uma aplicação.

O ambiente de execução Java permite que uma aplicação Java seja executada em sistemas operacionais diferentes.

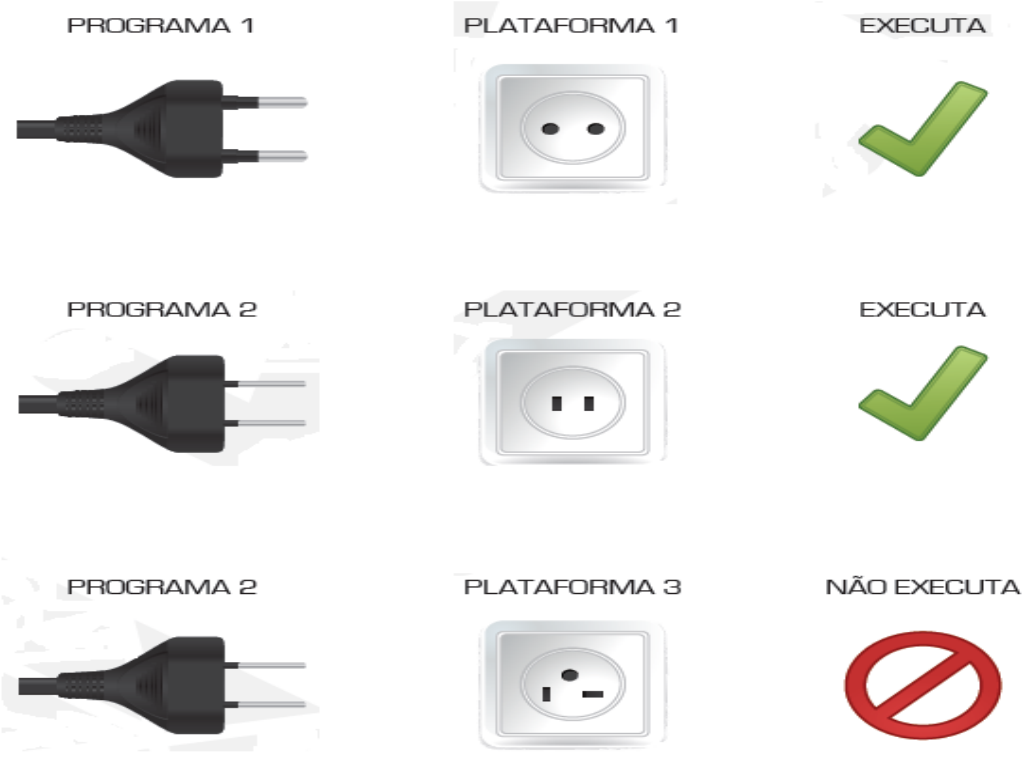
Para entendermos melhor o que está sendo explanado vamos fazer uma associação ao mundo real com pessoas que precisam se comunicar através de línguas diferentes, os computadores podem se comunicar através de linguagens de máquina diferentes. A linguagem de máquina de um computador é definida pela **arquitetura do processador** desse computador. Há diversas arquiteturas diferentes (Intel, ARM, PowerPC, etc) e cada uma delas define uma linguagem de máquina diferente.

Em outras palavras, um programa pode não executar em computadores com processadores de arquiteturas diferentes.

Os computadores são controlados por um **sistema operacional** que oferece diversas bibliotecas necessárias para o desenvolvimento das aplicações que podem ser executadas através dele. Sistemas operacionais diferentes (Windows, Linux,Mac OS X, etc) possuem bibliotecas diferentes.

Portanto, para determinar se um código em linguagem de máquina pode ou não ser executada por um computador, devemos considerar a arquitetura do processador e o sistema operacional desse computador.

Algumas bibliotecas específicas de sistema operacional são chamadas diretamente pelas instruções em linguagem de programação. Dessa forma, geralmente, o código fonte está “amarrado” a uma plataforma (sistema operacional + arquitetura de processador). Abaixo temos um exemplo:

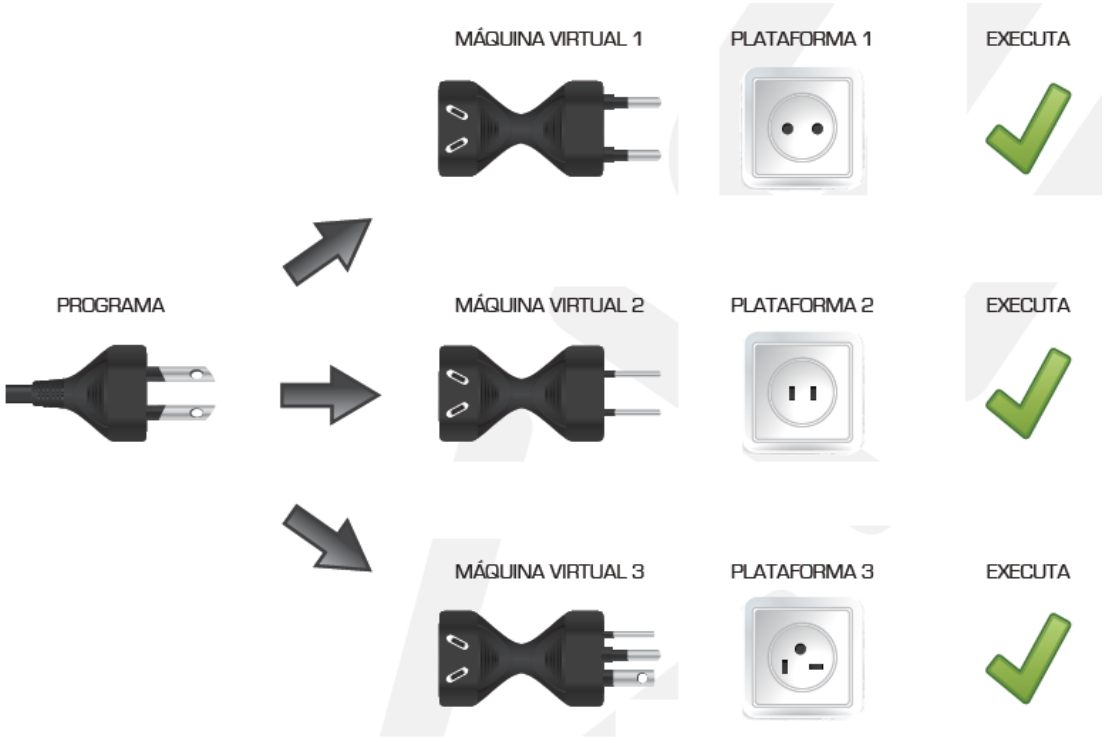


Uma empresa que deseja ter a sua aplicação disponível para diversos sistemas operacionais (Windows, Linux, Mac OS X, etc), e diversas arquiteturas de processador (Intel, ARM, PowerPC, etc), terá que desenvolver versões diferentes do código fonte para cada plataforma (sistema operacional + arquitetura de processador). Isso pode causar um impacto financeiro nessa empresa que inviabiliza o negócio.

Para tentar resolver o problema do desenvolvimento de aplicações multiplataforma, surgiu o conceito de *máquina virtual*.

Uma máquina virtual funciona como uma camada a mais entre o código compilado e a plataforma.

Quando compilamos um código fonte, estamos criando um executável que a máquina virtual saberá interpretar e ela é quem deverá traduzir as instruções do seu programa para a plataforma.



Tudo parece estar perfeito agora. Porém, olhando atentamente a figura acima, percebemos que existe a necessidade de uma máquina virtual para cada plataforma. Alguém poderia dizer que, de fato, o problema não foi resolvido, apenas mudou de lugar.

A diferença é que implementar a máquina virtual não é tarefa do programador que desenvolve as aplicações que serão executadas nela. A implementação da máquina virtual é responsabilidade de terceiros, que geralmente são empresas bem conceituadas ou projetos de código aberto que envolvem programadores do mundo inteiro. Como maiores exemplos podemos citar a Oracle JVM (Java Virtual Machine) e OpenJDK JVM.

Uma desvantagem em utilizar uma máquina virtual para executar um programa é a diminuição de performance, já que a própria máquina virtual consome recursos do computador. Além disso, as instruções do programa são processadas primeiro pela máquina virtual e depois pelo computador.

Por outro lado, as máquinas virtuais podem aplicar otimizações que aumentam a performance da execução de um programa. Inclusive, essas otimizações podem considerar informações geradas durante a execução. São exemplos de informações geradas durante a execução: a quantidade de uso da memória RAM e do processador do

computador, a quantidade de acessos ao disco rígido, a quantidade de chamadas de rede e a frequência de execução de um determinado trecho do programa.

Algumas máquinas virtuais identificamos trechos do programa que estão sendo mais chamados em um determinado momento da execução para traduzi-los para a linguagem de máquina do computador.

A partir daí, esses trechos podem ser executados diretamente no processador sem passar pela máquina virtual. Essa análise da máquina virtual é realizada durante toda a execução.

Com essas otimizações que consideram várias informações geradas durante a execução, um programa executado com máquina virtual pode até ser mais eficiente em alguns casos do que um programa executado diretamente no sistema operacional.

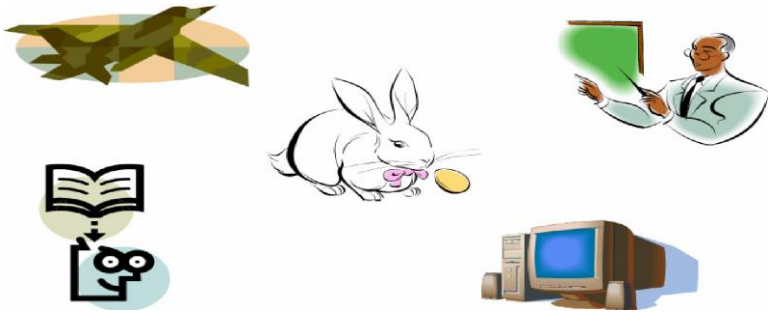
4. Plataforma JAVA VS Orientação a Objeto

Do ponto de vista do aprendizado, é interessante tentar definir o que é mais importante: a plataforma Java ou a orientação a objetos. Consideramos que a orientação a objetos é mais importante, pois ela é aplicada em muitas outras plataformas, a orientação a objetos é um conceito de programação e não uma linguagem de programação.

5. Objetos

O Que São Objetos?

De acordo com o dicionário: - Objeto: “1. Tudo que se oferece aos nossos sentidos ou à nossa alma. 2. Coisa material: Havia na estante vários objetos. 3. Tudo que constitui a matéria de ciências ou artes. 4. Assunto, matéria. 5. Fim a que se mira ou que se tem em vista”.



A figura destaca uma série de objetos. Objetos podem ser não só coisas concretas como também coisas inanimadas, como por exemplo uma matrícula, as disciplinas de um curso, os horários de aula.

Na programação orientada a objetos, implementa-se um conjunto de classes que definem os objetos presentes no *software*. Cada classe possui um comportamento (definidos pelos métodos) e estados possíveis (valores dos atributos) de seus objetos, assim como o relacionamento com outros objetos.

Há alguns anos, Alan Kay, um dos pais do paradigma da orientação a objetos, formulou a chamada "analogia biológica". Nessa analogia ele imaginou como seria um sistema de software que funcionasse como um ser vivo, no qual cada "célula" interagiria com outras células através do envio de mensagens para realização do objetivo comum.

De uma forma mais geral, Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagem entre si. Ele então estabeleceu os seguintes princípios da orientação a objetos:

(a) Qualquer coisa é um objeto;

(b) Objetos realizam tarefas através da requisição de serviços a outros objetos;

(c) Cada objeto pertence a uma determinada classe. Uma classe agrupa objetos similares;

(d) A classe é um repositório para comportamento associado ao objeto;
(e) Classes são organizadas em hierarquias.

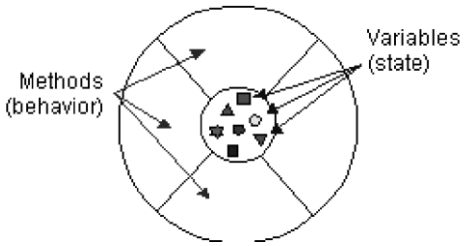
Quando temos um problema e queremos resolvê-lo usando um computador, necessariamente temos que fazer um programa. Este nada mais é do que uma série de instruções que indicam ao computador como proceder em determinadas situações. Assim, o grande desafio do programador é estabelecer a associação entre o modelo que o computador usa e o modelo que o problema lhe apresenta. Isso geralmente leva o programador a modelar o problema apresentado para o modelo utilizado em alguma linguagem. Se a linguagem escolhida for LISP, o problema será traduzido como listas encadeadas. Se for Prolog, o problema será uma cadeia de decisões. Assim, a representação da solução para o problema é característica da linguagem usada, tornando a escrita difícil.

A orientação a objetos tenta solucionar esse problema. A orientação a objetos é geral o suficiente para não impor um modelo de linguagem ao programador, permitindo a ele escolher uma estratégia, representando os aspectos do problema em objetos. Assim, quando se "lê" um programa orientado a objeto, podemos ver não apenas a solução, mas também a descrição do problema em termos do próprio problema. O programador consegue "quebrar" o grande problema em pequenas partes que juntas fornecem a solução.

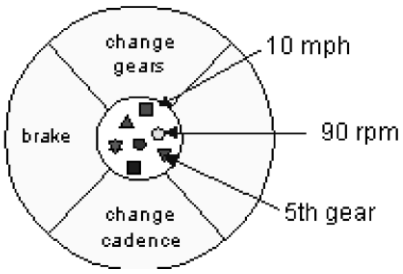
Olhando à sua volta é possível perceber muitos exemplos de objetos do mundo real: seu cachorro, sua mesa, sua televisão, sua bicicleta. Esses objetos têm duas características básicas, que são o estado e o comportamento. Por exemplo, os cachorros tem nome, cor, raça (estados) e eles balançam o rabo, comem, e latem (comportamento). Uma bicicleta tem 18 marchas, duas rodas, banco (estado) e elas brecam, aceleram e mudam de marcha (comportamento).

De maneira geral, definimos objetos como um conjunto de variáveis e métodos, que representam seus estados e comportamentos.

Veja a ilustração:



Temos aqui representada (de maneira lógica) a ideia que as linguagens orientadas a objeto utilizam. Tudo que um objeto sabe (estados ou variáveis) e tudo que ele podem fazer (comportamento ou métodos) está contido no próprio objeto. No exemplo da bicicleta, poderemos representar o objeto como no exemplo a seguir:



Temos métodos para mudar a marcha, a cadência das pedaladas, e para brecar. A utilização desses métodos altera os valores dos estados. Ao brecar, a velocidade diminui. Ao mudar a marcha, a cadência é alterada. Note que os diagramas mostram que as variáveis do objeto estão no centro do mesmo. Os métodos cercam esses valores e os “escondem” de outros objetos. Deste modo, só é possível ter acesso a essas variáveis

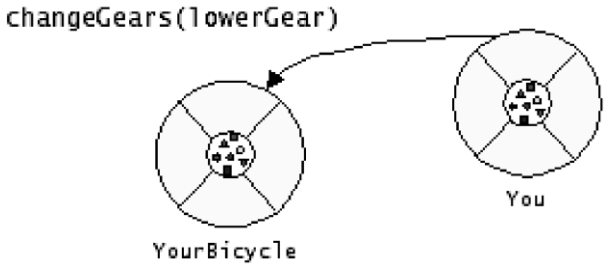
através dos métodos. Esse tipo de construção é chamada de encapsulamento, e é a construção ideal para objetos.

Mas um objeto sozinho geralmente não é muito útil. Ao contrário, em um programa orientado a objetos temos muitos objetos se relacionando entre si. Uma bicicleta encostada na garagem nada mais é que um pedaço de ferro e borracha. Por si mesma, ela não tem capacidade de realizar nenhuma tarefa. Apenas com outro objeto (você) utilizando a bicicleta, ela é capaz de realizar algum trabalho.

A interação entre objetos é feita através de mensagens. Um objeto "chama" os métodos de outro, passando parâmetros quando necessário. Quando você passa a mensagem "mude de marcha" para o objeto bicicleta você precisa dizer qual marcha você deseja.

A figura a seguir mostra os três componentes que fazem parte de uma mensagem:

- Objeto para o qual a mensagem é dirigida (bicicleta)
- Nome do método a ser executado (muda marcha)
- Os parâmetros necessários para a execução do método.



A troca de mensagens nos fornece ainda um benefício importante. Como todas as interações são feitas através delas, não importa se o objeto faz parte do mesmo programa; ele pode estar até em outro computador. Existem maneiras de enviar essas mensagens através da rede, permitindo a comunicação remota entre objetos.

Assim, um programa orientado a objetos nada mais é do que um punhado de objetos dizendo um ao outro o que fazer. Quando você quer que um objeto faça alguma coisa, você envia a ele uma "mensagem" informando o que quer fazer, e o objeto faz. Se ele precisar de outro objeto que o auxiliar a realizar o "trabalho", ele mesmo vai cuidar de enviar mensagem para esse outro objeto. Deste modo, um programa pode realizar atividades muito complexas baseadas apenas em uma mensagem inicial.

Você pode definir vários tipos de objetos diferentes para seu programa. Cada objeto terá suas próprias características, e saberá como interpretar certos pedidos. A parte interessante é que objetos de mesmo tipo se comportam de maneira semelhante. Assim, você pode ter funções genéricas que funcionam para vários tipos diferentes, economizando código.

Vale lembrar que métodos são diferentes de procedimentos e funções das linguagens procedurais, e é muito fácil confundir os conceitos. Para evitar a confusão, temos que entender o que são classes.

6. O que é uma classe?

Assim como os objetos do mundo real, o mundo da Programação Orientada a Objetos (POO) agrupa os objetos pelos comportamentos e atributos comuns.

A biologia classifica todos os cães, gatos, elefantes e seres humanos como mamíferos. Características compartilhadas dão a essas criaturas separadas um senso de comunidade. No mundo do software, as classes agrupam objetos relacionados da mesma maneira.

Uma classe define todas as **características comuns a um tipo de objeto**. Especificamente, a classe **define todos os atributos e comportamentos** expostos pelo objeto. A classe define às quais mensagens o seu objeto responde. Quando um objeto

quer exercer o comportamento de outro objeto, ele não faz isso diretamente, mas pede ao outro objeto para que se mude, normalmente baseado em alguma informação adicional. Frequentemente, isso é referido como “envio de mensagem”.

Uma *classe* define os atributos e comportamentos comuns compartilhados por um tipo de objeto. Imagine a classe como a forma do objeto. Os objetos de certo tipo ou classificação compartilham os mesmos comportamentos e atributos. As classes atuam de forma muito parecida com um cortador de molde ou biscoito, no sentido de que você usa uma classe para criar ou *instanciar* objetos.

Os *atributos* são as características de uma classe visíveis externamente. A cor dos olhos e a cor dos cabelos são exemplos de atributos. Um objeto pode expor um atributo fornecendo um link direto a alguma variável interna ou retornando o valor através de um método.

Comportamento é uma ação executada por um objeto quando passada uma mensagem ou em resposta a uma mudança de estado: é algo que um objeto faz. Um objeto pode executar o *comportamento* de outro, executando uma operação sobre esse objeto. Você pode ver os termos: *chamada de método, chamada de função* ou *passar uma mensagem*; usados em vez de executar uma *operação*. O que é importante é que cada uma dessas ações omite o comportamento de um objeto.

A definição de classes e seus inter-relacionamentos é o principal resultado da etapa de projeto de software. Em geral, esse resultado é expresso em termos de alguma linguagem de modelagem, tal como UML.

Uma classe é um gabarito para a definição de objetos. Através da definição de uma classe, descreve-se que propriedades — ou atributos — o objeto terá.

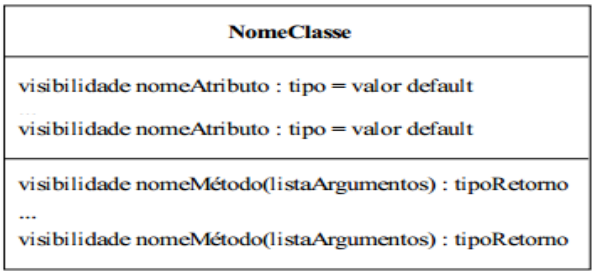
Além da especificação de atributos, a definição de uma classe descreve também qual o comportamento de objetos da classe, ou seja, que funcionalidades podem ser aplicadas a objetos da classe.

Essas funcionalidades são descritas através de métodos. Um método nada mais é que o equivalente a um procedimento ou função, com a restrição que ele manipula apenas suas variáveis locais e os atributos que foram definidos para a classe.

Uma vez que estejam definidas quais serão as classes que irão compor uma aplicação, assim como qual deve ser sua estrutura interna e comportamento, é possível criar essas classes em Java.

Na *Unified Modeling Language* (UML), a representação para uma classe no diagrama de classes é tipicamente expressa na forma gráfica, como mostrado na Figura a seguir.

Como se observa nessa figura, a especificação de uma classe é composta por três regiões: o nome da classe, o conjunto de atributos da classe e o conjunto de métodos da classe.



O nome da classe é um identificador para a classe, que permite referenciá-la posteriormente, por exemplo, no momento da criação de um objeto.

O conjunto de atributos descreve as propriedades da classe. Cada atributo é identificado por um nome e tem um tipo associado. Em uma linguagem de programação orientada a objetos pura, o tipo é o nome de uma classe. Na prática, a maior parte das linguagens de programação orientada a objetos oferecem um grupo de tipos primitivos,

como inteiro, real e caráter, que podem ser usados na descrição de atributos. O atributo pode ainda ter um valor_default opcional, que especifica um valor inicial para o atributo.

Os métodos definem as funcionalidades da classe, ou seja, o que será possível fazer com objetos dessa classe. Cada método é especificado por uma assinatura, composta por um identificador para o método (o nome do método), o tipo para o valor de retorno e sua lista de argumentos, sendo cada argumento identificado por seu tipo e nome.

Através do mecanismo de sobrecarga (*overloading*), dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método. Nesse caso, diz-se que ocorre a ligação prematura (early binding) para o método correto.

O modificador de visibilidade pode estar presente tanto para atributos como para métodos. Em princípio, três categorias de visibilidade podem ser definidas:

- público, denotado em UML pelo símbolo +: nesse caso, o atributo ou método de um objeto dessa classe pode ser acessado por qualquer outro objeto (visibilidade externa total);
- privativo, denotado em UML pelo símbolo -: nesse caso, o atributo ou método de um objeto dessa classe não pode ser acessado por nenhum outro objeto (nenhuma visibilidade externa);
- protegido, denotado em UML pelo símbolo #: nesse caso, o atributo ou método de um objeto dessa classe poderá ser acessado apenas por objetos de classes que sejam derivadas dessa através do mecanismo de herança.

7. Domínio e Aplicação

Um domínio é composto pelas entidades, informações e processos relacionados a um determinado contexto. Uma aplicação pode ser desenvolvida para automatizar ou tornar factível as tarefas de um domínio. Portanto, uma aplicação é basicamente o "reflexo" de um domínio.

Para exemplificar, suponha que estamos interessados em desenvolver uma aplicação para facilitar as tarefas do cotidiano de um banco. Podemos identificar clientes, funcionários, agências e contas como entidades desse domínio. Assim como podemos identificar as informações e os processos relacionados a essas entidades.

DOMÍNIO BANCÁRIO

Observação:

A identificação dos elementos de um domínio é uma tarefa difícil, pois depende fortemente do conhecimento das entidades, informações e processos que o compõem. Em geral, as pessoas que possuem esse conhecimento ou parte dele estão em contato constante com o domínio e não possuem conhecimentos técnicos para desenvolver uma aplicação.

Desenvolvedores de software buscam constantemente mecanismos para tornar mais eficiente o entendimento dos domínios para os quais eles devem desenvolver aplicações.

8. Objetos, Atributos e Métodos

As entidades identificadas no domínio devem ser representadas de alguma forma dentro da aplicação correspondente. Nas aplicações orientadas a objetos, as entidades são representadas por objetos.

• Uma aplicação orientada a objetos é composta por objetos.
• Em geral, um objeto representa uma entidade do domínio.

Para exemplificar, suponha que no domínio de um determinado banco exista um cliente chamado João. Dentro de uma aplicação orientada a objetos correspondente a esse domínio, deve existir um objeto para representar esse cliente.

Suponha que algumas informações do cliente João como nome, data de nascimento e sexo são importantes para o banco. Já que esses dados são relevantes para o domínio, o objeto que representa esse cliente deve possuir essas informações. Esses dados são armazenados nos atributos do objeto que representa o João.

• Um atributo é uma variável que pertence a um objeto.
• Os dados de um objeto são armazenados nos seus atributos.

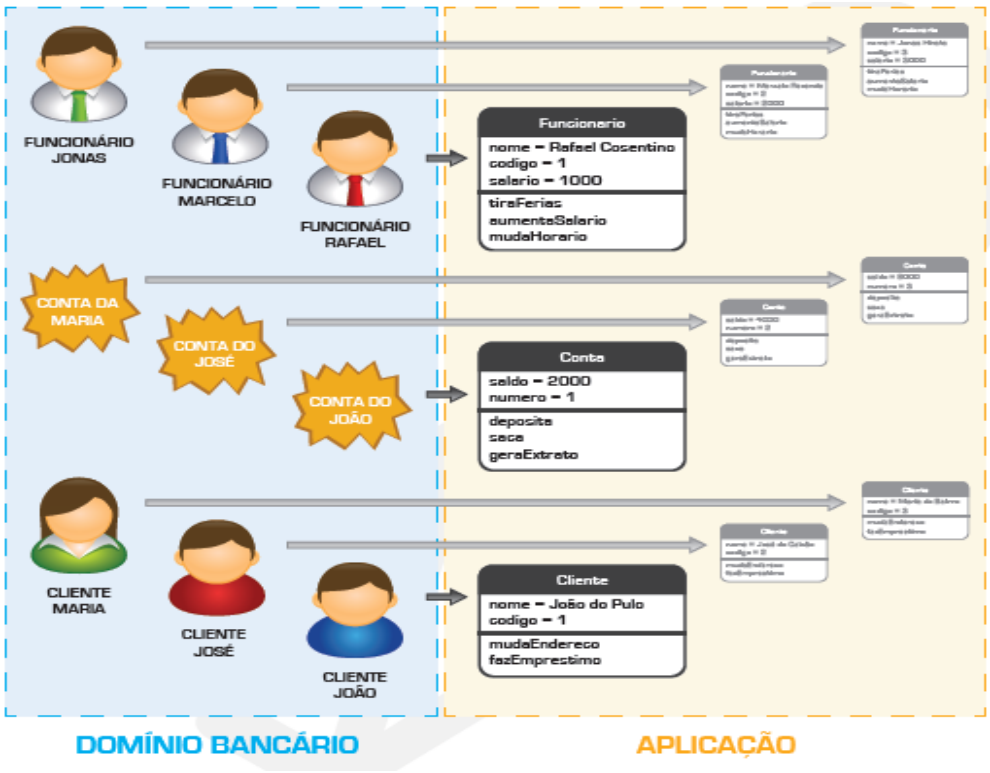
O próprio objeto deve realizar operações de consulta ou alteração dos valores de seus atributos.

Essas operações são definidas nos métodos do objeto.

Os métodos também são utilizados para possibilitar interações entre os objetos de uma aplicação.

Por exemplo, quando um cliente requisita um saque através de um caixa eletrônico do banco, o objeto que representa o caixa eletrônico deve interagir com o objeto que representa a conta do cliente.

• As tarefas que um objeto pode realizar são definidas pelos seus métodos.
• Um objeto é composto por atributos e métodos.



Observações:

- Em geral, não é adequado utilizar o objeto que representa um determinado cliente para representar outro cliente do banco, pois os dados dos clientes podem ser diferentes.

Dessa forma, para cada cliente do banco, deve existir um objeto dentro do sistema para representá-lo.

- Os objetos não representam apenas coisas concretas como os clientes do banco. Eles também devem ser utilizados para representar coisas abstratas como uma conta de um cliente ou um serviço que o banco ofereça.

8.1. Classes em Java

O conceito de classe apresentado anteriormente é genérico e pode ser aplicado em diversas linguagens de programação. Mostraremos como a classe Conta poderia ser escrita utilizando a linguagem Java. Inicialmente, discutiremos apenas sobre os atributos. Os métodos serão abordados posteriormente.

A classe Java Conta é declarada utilizando a palavra reservada **class**. No corpo dessa classe, são declaradas três variáveis que são os atributos que os objetos possuirão. Como a linguagem Java é estaticamente tipada, os tipos dos atributos são definidos no código. Os atributos saldo e limite são do tipo double, que permite armazenar números com casas decimais, e o atributo número é do tipo int, que permite armazenar números inteiros.

Por convenção, os nomes das classes na linguagem Java devem seguir o padrão “Pascal Case”.

```
1  class Conta {
2    double saldo;
3    double limite;
4    int numero;
5  }
```

8.2. Criando objetos em Java

Após definir a classe Conta, podemos criar objetos a partir dela. Esses objetos devem ser alocados na memória RAM do computador. Felizmente, todo o processo de alocação do objeto na memória é gerenciado pela máquina virtual. O gerenciamento da memória é um dos recursos mais importantes oferecidos pela máquina virtual.

Do ponto de vista da aplicação, basta utilizar um comando especial para criar objetos e a máquina virtual se encarrega do resto. O comando para criar objetos é o **new**.

```
1  class TestaConta {
2    public static void main(String[] args) {
3      // criando um objeto
4      new Conta();
5    }
6  }
```

A linha com o comando new poderia ser repetida cada vez que desejássemos criar (instanciar) um objeto da classe Conta. A classe TestaConta serve apenas para colocarmos o método main, que é o ponto de partida da aplicação.

```
1  class TestaConta {
2    public static void main(String[] args) {
3      // criando três objetos
4      new Conta();
5      new Conta();
6      new Conta();
7    }
8  }
```

Chamar o comando new passando uma classe Java é como se estivéssemos contratando uma construtora passando a planta da casa que queremos construir. A construtora se encarrega de construir a casa para nós de acordo com a planta. Assim como a máquina virtual se encarrega de construir o objeto na memória do computador.

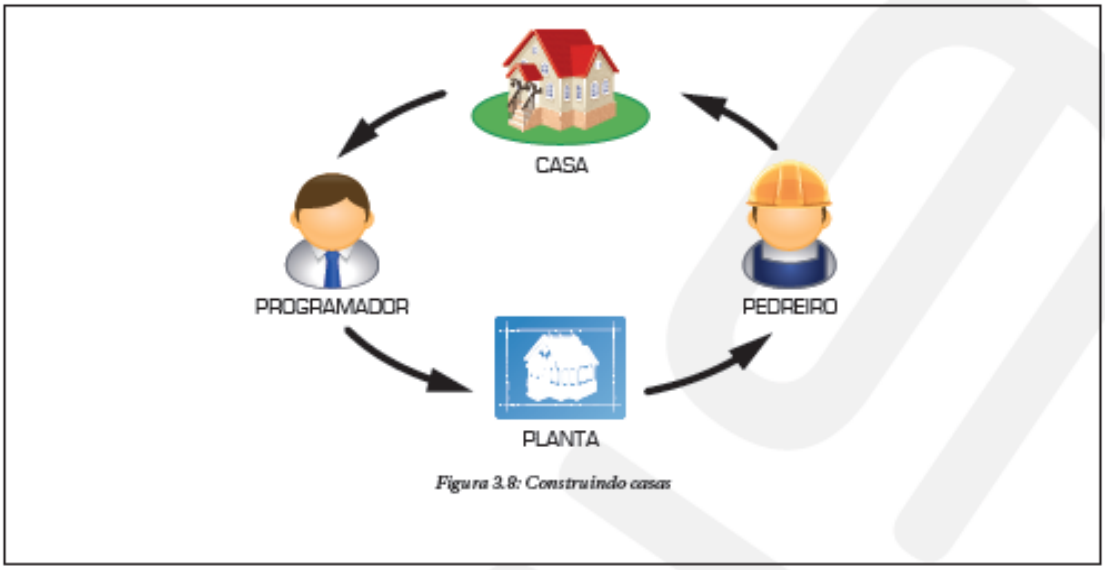


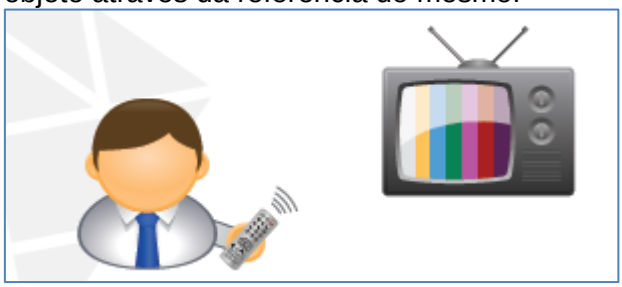
Figura 3.8: Construindo casas

8.3. Referências

Todo objeto possui uma referência. A referência de um objeto é a única maneira de acessar os seus atributos e métodos. Dessa forma, devemos guardar as referências dos objetos que desejamos utilizar.

A princípio, podemos comparar a referência de um objeto com o endereço de memória desse objeto. De fato, essa comparação simplifica o aprendizado. Contudo, o conceito de referência é mais amplo. Uma referência é o elemento que permite que um determinado objeto seja acessado. Uma referência está para um objeto assim como um controle remoto está para um aparelho de TV. Através do controle remoto de uma TV

você pode aumentar o volume ou trocar de canal. Analogamente, podemos controlar um objeto através da referência do mesmo.



8.3.1. Referências em Java

Ao utilizar o comando new, um objeto é alocado em algum lugar da memória. Para que possamos acessar esse objeto, precisamos de sua referência. O comando new devolve a referência do objeto que foi criado.

Para guardar as referências devolvidas pelo comando new, devemos utilizar variáveis não primitivas.

```
1  Conta referecia = new Conta();
```

No código Java acima, a variável **referencia** receberá a referência do objeto criado pelo comando new. Essa variável é do tipo Conta. Isso significa que ela só pode armazenar referências de objetos do tipo Conta.

8.4. Manipulando Atributos

Podemos alterar ou acessar os valores guardados nos atributos de um objeto se tivermos a referência a esse objeto. Os atributos são acessados pelo nome. No caso específico da linguagem Java, a sintaxe para acessar um atributo utiliza o operador "**.**".

```
1  Conta referecia = new Conta();
2
3  referecia.saldo = 1000.0;
4  referecia.limite = 500.0;
5  referecia.numero = 1;
6
7  System.out.println(referecia.saldo);
8  System.out.println(referecia.limite);
9  System.out.println(referecia.numero);
```

No código acima, o atributo saldo recebe o valor 1000.0. O atributo limite recebe o valor 500 e o numero recebe o valor 1. Depois, os valores são impressos na tela através do comando System.out.println.

8.5. Valores Padrão

Poderíamos instanciar um objeto e utilizar seus atributos sem inicializá-los explicitamente, pois os atributos são inicializados com valores padrão. Os atributos de tipos numéricos são inicializados com 0, os atributos do tipo *boolean* são inicializados com false e os demais atributos com *null* (referência vazia).

```
1  class Conta {
2    double limite;
3  }
```

```
1  class TestaConta {
2    public static void main(String[] args) {
3      Conta conta = new Conta();
4
5      // imprime 0
6      System.out.println(conta.limite);
7    }
8  }
```

A inicialização dos atributos com os valores padrão ocorre na instanciação, ou seja, quando o comando new é utilizado. Dessa forma, todo objeto "nasce" com os valores padrão. Em alguns casos, é necessário trocar esses valores. Para trocar o valor padrão de um atributo, devemos inicializá-lo na declaração. Por exemplo, suponha que o limite padrão das contas de um banco seja R$ 500. Nesse caso, seria interessante definir esse valor como padrão para o atributo limite.

```
1  class Conta {
2    double limite = 500;
3  }
```

```
1  class TestaConta {
2    public static void main(String[] args) {
3      Conta conta = new Conta();
4
5      // imprime 500
6      System.out.println(conta.limite);
7    }
8  }
```

8.6. Exercícios de Fixação

1 – Iremos fazer juntos esse projeto, para isso crie um novo projeto chamado **orientacao-a-objetos** para os arquivos desenvolvidos neste exercício.

2 - Implemente uma classe para definir os objetos que representarão os clientes de um banco. Essa classe deve declarar dois atributos: um para os nomes e outro para os códigos dos clientes. Adicione o seguinte arquivo na pasta orientacao-a-objetos.

```
1  class Cliente {
2    String nome;
3    int codigo;
4  }
```

3 - Faça um teste criando dois objetos da classe Cliente. Adicione o seguinte arquivo na pasta **orientacao-a-objetos**.

```
class TestaCliente {
  public static void main(String[] args) {
    Cliente c1 = new Cliente();
    c1.nome = "Rafael Cosentino";
    c1.codigo = 1;

    Cliente c2 = new Cliente();
    c2.nome = "Jonas Hirata";
    c2.codigo = 2;

    System.out.println(c1.nome);
    System.out.println(c1.codigo);

    System.out.println(c2.nome);
    System.out.println(c2.codigo);
  }
}
```

Execute a classe TestaCliente.

4 - Os bancos oferecem aos clientes a possibilidade de obter um cartão de crédito que pode ser utilizados para fazer compras. Um cartão de crédito possui um número e uma data de validade. Crie uma classe para modelar os objetos que representarão os cartões de crédito. Adicione o seguinte arquivo na pasta **orientacao-a-objetos**.

```
class CartaoDeCredito {
  int numero;
  String dataDeValidade;
}
```

5 - Faça um teste criando dois objetos da classe CartaoDeCredito. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta **orientacao-a-objetos**.

```
class TestaCartaoDeCredito {
  public static void main(String[] args) {
    CartaoDeCredito cdc1 = new CartaoDeCredito();
    cdc1.numero = 111111;
    cdc1.dataDeValidade = "01/01/2013";

    CartaoDeCredito cdc2 = new CartaoDeCredito();
    cdc2.numero = 222222;
    cdc2.dataDeValidade = "01/01/2014";

    System.out.println(cdc1.numero);
    System.out.println(cdc1.dataDeValidade);

    System.out.println(cdc2.numero);
    System.out.println(cdc2.dataDeValidade);
  }
}
```

Execute a classe TestaCartaoDeCredito.

6 - As agências do banco possuem número. Crie uma classe para definir os objetos que representarão as agências.

```
class Agencia {
  int numero;
}
```

7 - Faça um teste criando dois objetos da classe Agencia. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta orientacao-a-objetos.

```
class TestaAgencia {
  public static void main(String[] args) {
    Agencia a1 = new Agencia();
    a1.numero = 1234;

    Agencia a2 = new Agencia();
    a2.numero = 5678;

    System.out.println(a1.numero);

    System.out.println(a2.numero);
  }
}
```

Execute a classe TestaAgencia.

8 - As contas do banco possuem número, saldo e limite. Crie uma classe para definir os objetos que representarão as contas.

```
class Conta {
  int numero;
  double saldo;
  double limite;
}
```

9 - Faça um teste criando dois objetos da classe Conta. Altere e imprima os atributos desses objetos. Adicione o seguinte arquivo na pasta orientacao-a-objetos.

```
class TestaConta {
  public static void main(String[] args) {
    Conta c1 = new Conta();
    c1.numero = 1234;
    c1.saldo = 1000;
    c1.limite = 500;

    Conta c2 = new Conta();
    c2.numero = 5678;
    c2.saldo = 2000;
    c2.limite = 250;

    System.out.println(c1.numero);
    System.out.println(c1.saldo);
    System.out.println(c1.limite);

    System.out.println(c2.numero);
    System.out.println(c2.saldo);
    System.out.println(c2.limite);
  }
}
```

Execute a classe TestaConta.

10 - Faça um teste que imprima os atributos de um objeto da classe Conta logo após a sua criação.

```
class TestaValoresPadrao {
  public static void main(String[] args) {
    Conta c = new Conta();

    System.out.println(c.numero);
    System.out.println(c.saldo);
    System.out.println(c.limite);
  }
}
```

Execute a classe TestaValoresPadrao

11 -. **Altere** a classe Conta para que todos os objetos criados a partir dessa classe possuam R$ 100 de limite inicial.

```
1  class Conta {
2    int numero;
3    double saldo;
4    double limite = 100;
5  }
```

Execute a classe TestaValoresPadrao.

8.7. Métodos

No banco, é possível realizar diversas operações em uma conta: depósito, saque, transferência, consultas e etc. Essas operações podem modificar ou apenas acessar os valores dos atributos dos objetos que representam as contas.

Essas operações são realizadas em métodos definidos na própria classe Conta. Por exemplo, para realizar a operação de depósito, podemos acrescentar o seguinte método na classe Conta.

```
1  void deposita(double valor) {
2    // implementação
3  }
```

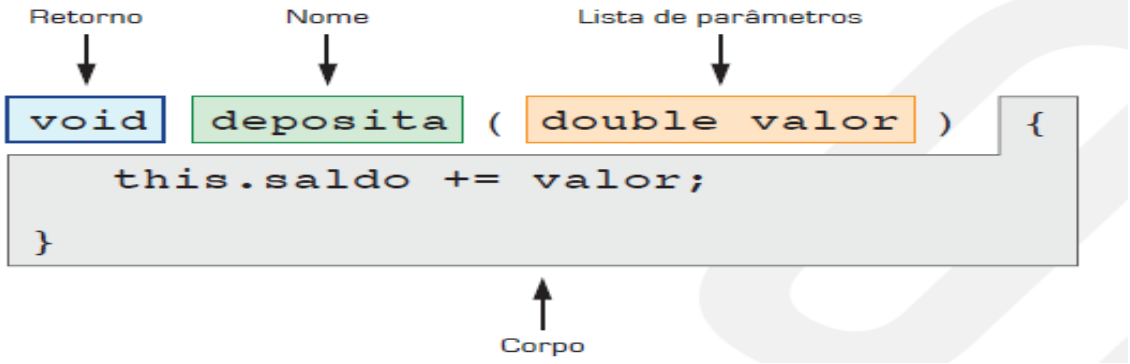
Podemos dividir um método em quatro partes:

Nome: É utilizado para chamar o método. Na linguagem Java, é uma boa prática definir os nomes dos métodos utilizando a convenção “Camel Case” com a primeira letra minúscula.

Lista de Parâmetros: Define os valores que o método deve receber. Métodos que não devem receber nenhum valor possuem a lista de parâmetros vazia.

Corpo: Define o que acontecerá quando o método for chamado.

Retorno: A resposta que será devolvida ao final do processamento do método. Quando um método não devolve nenhuma resposta, ele deve ser marcado com a palavra reservada void.



Para realizar um depósito, devemos chamar o método deposita() através da referência do objeto que representa a conta que terá o dinheiro creditado.

```
1  // Referência de um objeto
2  Conta c = new Conta();
3
4  // Chamando o método deposita()
5  c.deposita(1000);
```

Normalmente, os métodos acessam ou alteram os valores armazenados nos atributos dos objetos.

Por exemplo, na execução do método deposita(), é necessário alterar o valor do atributo saldo do objeto que foi escolhido para realizar a operação.

Dentro de um método, para acessar os atributos do objeto que está processando o método, devemos utilizar a palavra reservada *this*.

```
void deposita(double valor) {
  this.saldo += valor;
}
```

O método deposita() não possui nenhum retorno lógico. Por isso, foi marcado com *void*. Mas, para outros métodos, pode ser necessário definir um tipo de retorno específico.

Considere, por exemplo, um método para realizar a operação que consulta o saldo disponível das contas. Suponha também que o saldo disponível é igual a soma do saldo e do limite. Então, esse método deve somar os atributos saldo e limite e devolver o resultado. Por outro lado, esse método não deve receber nenhum valor, pois todas as informações necessárias para realizar a operação estão nos atributos dos objetos que representam as contas.

```
double consultaSaldoDisponivel() {
  return this.saldo + this.limite;
}
```

Ao chamar o método consultaSaldoDisponivel() a resposta pode ser armazenada em uma variável do tipo *double*.

```
Conta c = new Conta();
c.deposita(1000);

// Armazenando a resposta de um método em uma variável
double saldoDisponivel = c.consultaSaldoDisponivel();

System.out.println("Saldo Disponível: " + this.saldoDisponivel);
```

8.8. Exercícios de Fixação

1-) Acrescente alguns métodos na classe Conta para realizar as operações de deposito, saque, impressão de extrato e consulta do saldo disponível.

```
class Conta {
  int numero;
  double saldo;
  double limite = 100;
  Agencia agencia;

  // ADICIONE OS MÉTODOS ABAIXO
  void deposita(double valor) {
    this.saldo += valor;
  }

  void saca(double valor) {
    this.saldo -= valor;
  }

  void imprimeExtrato() {
    System.out.println("SALDO: " + this.saldo);
  }

  double consultaSaldoDisponivel() {
    return this.saldo + this.limite;
  }
}
```

2-) Teste os métodos da classe Conta.

```
class TestaMetodosConta {
  public static void main(String[] args) {
    Conta c = new Conta();

    c.deposita(1000);
    c.imprimeExtrato();

    c.saca(100);
    c.imprimeExtrato();

    double saldoDisponivel = c.consultaSaldoDisponivel();
    System.out.println("SALDO DISPONÍVEL: " + saldoDisponivel);
  }
}
```

8.8.1. Exercícios Complementares

1-) Em uma empresa, temos os funcionários, que precisam ser representados em nossa aplicação. Então implemente uma classe chamada Funcionario que contenha dois atributos: o primeiro para o nome e o segundo para o salário dos funcionários.

2-) Faça uma classe chamada TestaFuncionario e crie dois objetos da classe Funcionario atribuindo valores a eles. Mostre na tela as informações desses objetos.

3-) Adicione na classe Funcionario dois métodos: um para aumentar o salário e outro para consultar os dados dos funcionários.

4-) Na classe TestaFuncionario teste novamente os métodos de um objeto da classe Funcionario.

Atenção: Desenvolva os exercícios propostos e verificaremos em aula.

9. Sobrecarga (*Overloading*)

Os clientes dos bancos costumam consultar periodicamente informações relativas às suas contas.

Geralmente, essas informações são obtidas através de extratos. No sistema do banco, os extratos podem ser gerados por métodos da classe Conta.

```
1  class Conta {
2    double saldo;
3    double limite;
4
5    void imprimeExtrato(int dias){
6      // extrato
7    }
8  }
```

O método imprimeExtrato() recebe a quantidade de dias que deve ser considerada para gerar o extrato da conta. Por exemplo, se esse método receber o valor 30 então ele deve gerar um extrato com as movimentações dos últimos 30 dias.

Em geral, extratos dos últimos 15 dias atendem as necessidades dos clientes. Dessa forma, poderíamos acrescentar um método na classe Conta para gerar extratos com essa quantidade fixa de dias.

```
1  class Conta {
2    double saldo;
3    double limite;
4
5    void imprimeExtrato(){
6      // extrato dos últimos 15 dias
7    }
8
9    void imprimeExtrato(int dias){
10     // extrato
11   }
12 }
```

O primeiro método não recebe parâmetros pois ele utilizará uma quantidade de dias padrão definida pelo banco para gerar os extratos (15 dias).

O segundo recebe um valor inteiro como parâmetro e deve considerar essa quantidade de dias para gerar os extratos.

Os dois métodos possuem o mesmo nome e lista de parâmetros diferentes. Quando dois ou mais métodos são definidos na mesma classe com o mesmo nome, dizemos que houve uma **sobrecarga** de métodos. Uma sobrecarga de métodos só é válida se as listas de parâmetros dos métodos são diferentes entre si.

No caso dos dois métodos que geram extratos, poderíamos evitar repetição de código fazendo um método chamar o outro.

```
class Conta {

  void imprimeExtrato(){
    this.imprimeExtrato(15);
  }

  void imprimeExtrato(int dias){
    // extrato
  }
}
```

9.1. Exercícios de Fixação

1-) Crie uma classe chamada **Gerente** para definir os objetos que representarão os gerentes do banco. Defina dois métodos de aumento salarial nessa classe. O primeiro deve aumentar o salário com uma taxa fixa de 10%. O segundo deve aumentar o salário com uma taxa variável.

```
class Gerente {
  String nome;
  double salario;

  void aumentaSalario() {
    this.aumentaSalario(0.1);
  }

  void aumentaSalario(double taxa) {
    this.salario += this.salario * taxa;
  }
}
```

Teste os métodos de aumento salarial definidos na classe Gerente.

```
class TestaGerente {
  public static void main(String[] args){
    Gerente g = new Gerente();
    g.salario = 1000;

    System.out.println("Salário: " + g.salario);

    System.out.println("Aumentando o salário em 10% ");
    g.aumentaSalario();

    System.out.println("Salário: " + g.salario);

    System.out.println("Aumentando o salário em 30% ");
    g.aumentaSalario(0.3);

    System.out.println("Salário: " + g.salario);
  }
}
```

Compile e execute a classe TestaGerente.

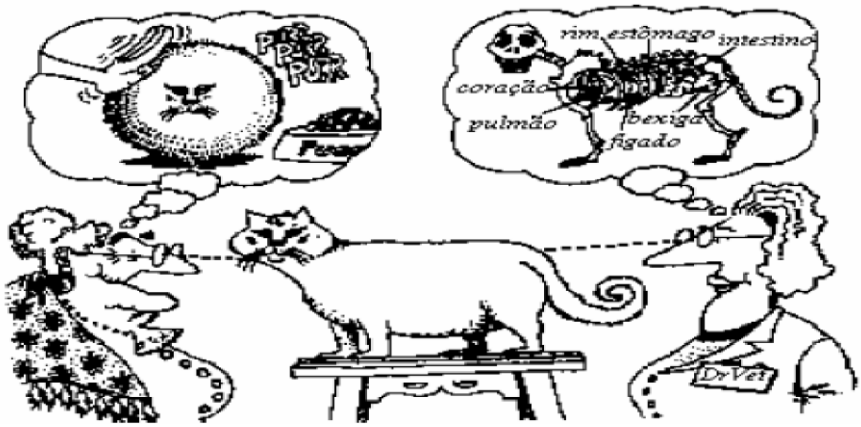
10. Abstração

Uma das principais formas do ser humano lidar com a complexidade é através do uso de abstrações. As pessoas tipicamente tentam compreender o mundo, construindo modelos mentais de partes dele. Tais modelos são uma visão simplificada de algo, onde apenas elementos relevantes são considerados. Modelos mentais, portanto, são mais simples do que os complexos sistemas que eles modelam.

Consideremos, por exemplo, um mapa como um modelo do território que ele representa. Um mapa é útil porque abstrai apenas aquelas características do território que se deseja modelar. Se um mapa incluísse todos os detalhes do território, provavelmente teria o mesmo tamanho do território e, portanto, não serviria a seu propósito.

Da mesma forma que um mapa precisa ser significativamente menor que o território que mapeia, incluindo apenas informações cuidadosamente selecionadas, um modelo mental abstrai apenas as características relevantes de um sistema para seu entendimento. Assim, podemos definir abstração como sendo o princípio de ignorar aspectos não relevantes de um assunto, segundo a perspectiva de um observador, tornando possível uma concentração maior nos aspectos principais do mesmo. De fato, a abstração consiste na seleção que um observador faz de alguns aspectos de um assunto, em detrimento de outros que não demonstram ser relevantes para o propósito em questão, isto é, a abstração é aplicada de acordo com o interesse do observador, e por isso de um mesmo objeto pode-se ter diferentes visões, como demonstra a figura 1 abaixo.

Só devem ser mapeados os objetos que são relevantes ao problema, bem como as características (propriedades e comportamento) desses objetos que forem necessários.



No que tange ao desenvolvimento de software, duas formas adicionais de abstração têm grande importância: a abstração de dados e a abstração de procedimentos.

11. Atributos Privados

No sistema do banco, cada objeto da classe Funcionario possui um atributo para guardar o salário do funcionário que ele representa.

```
class Funcionario {
  double salario;
}
```

O atributo salario pode ser acessado ou modificado por código escrito em qualquer classe que esteja no mesmo diretório que a classe Funcionario. Portanto, o controle desse atributo é descentralizado.

Para identificar algum erro relacionado a manipulação dos salários dos funcionários, é necessário verificar o código de todos os arquivos da pasta onde a classe Funcionario está definida. Quanto maior o número de arquivos, menos eficiente será a manutenção da aplicação.

Podemos obter um controle centralizado tornando o atributo salario **privado** e definindo métodos para implementar todas as lógicas que utilizam ou modificam o valor desse atributo.

```
class Funcionario {
  private double salario;

  void aumentaSalario(double aumento) {
    // lógica para aumentar o salário
  }
}
```

Um atributo privado só pode ser acessado ou alterado por código escrito dentro da classe na qual ele foi definido. Se algum código fora da classe Funcionario tentar acessar ou alterar o valor do atributo privado salario, um erro de compilação será gerado.

Definir todos os atributos como privado e métodos para implementar as lógicas de acesso e alteração é quase uma regra da orientação a objetos. O intuito é ter sempre um controle centralizado dos dados dos objetos para facilitar a manutenção do sistema e a detecção de erros.

11.1. Métodos Privados

O papel de alguns métodos pode ser o de auxiliar outros métodos da mesma classe. E muitas vezes, não é correto chamar esses métodos auxiliares de fora da sua classe diretamente.

No exemplo abaixo, o método desconta Tarifa() é um método auxiliar dos métodos deposita() e saca(). Além disso, ele não deve ser chamado diretamente, pois a tarifa só deve ser descontada quando ocorre um depósito ou um saque.

```
class Conta {
  private double saldo;

  void deposita(double valor) {
    this.saldo += valor;
    this.descontaTarifa();
  }

  void saca(double valor) {
    this.saldo -= valor;
    this.descontaTarifa();
  }

  void descontaTarifa() {
    this.saldo -= 0.1;
  }
}
```

Para garantir que métodos auxiliares não sejam chamados por código escrito fora da classe na qual eles foram definidos, podemos torná-los privados, acrescentando o modificador private.

```
private void descontaTarifa() {
  this.saldo -= 0.1;
}
```

Qualquer chamada ao método descontaTarifa() realizada fora da classe Conta gera um erro de compilação.

11.2. Métodos Públicos

Os métodos que devem ser chamados a partir de qualquer parte do sistema devem possuir o modificador de visibilidade public.

```
class Conta {
  private double saldo;

  public void deposita(double valor) {
    this.saldo += valor;
    this.descontaTarifa();
  }

  public void saca(double valor) {
    this.saldo -= valor;
    this.descontaTarifa();
  }

  private descontaTarifa(){
    this.saldo -= 0.1;
  }
}
```

11.3. Por que encapsular?

Uma das ideias mais importantes da orientação a objetos é o encapsulamento. Encapsular significa esconder a implementação dos objetos. O encapsulamento favorece principalmente dois aspectos de um sistema: a manutenção e o desenvolvimento.

A manutenção é favorecida pois, uma vez aplicado o encapsulamento, quando o funcionamento de um objeto deve ser alterado, em geral, basta modificar a classe do mesmo.

O desenvolvimento é favorecido pois, uma vez aplicado o encapsulamento, conseguimos determinar precisamente as responsabilidades de cada classe da aplicação.

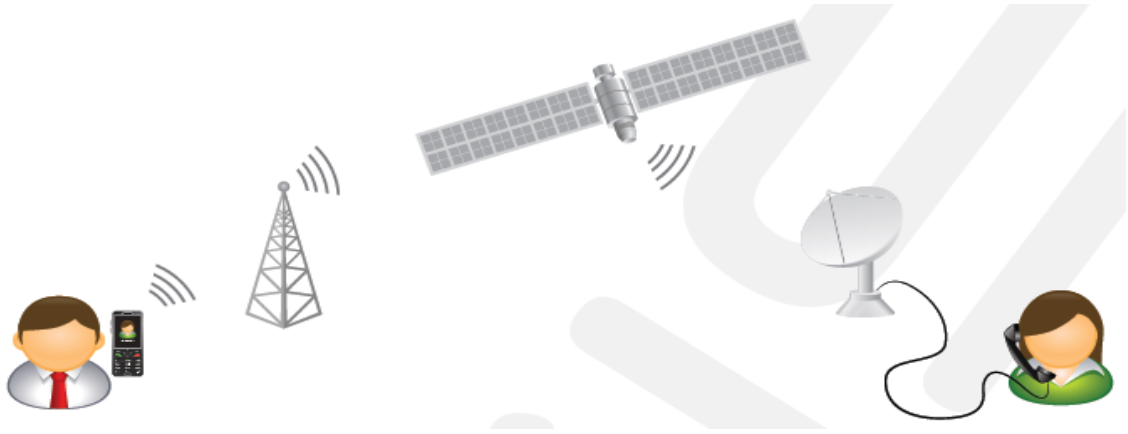
O conceito de encapsulamento pode ser identificado em diversos exemplos do cotidiano. Mostraremos alguns desses exemplos para esclarecer melhor a ideia.

11.3.1. Celular - Escondendo a complexidade

Hoje em dia, as pessoas estão acostumadas com os celulares. Os botões, a tela e os menus de um celular formam a **interface de uso** do mesmo. Em outras palavras, o usuário interage com esses aparelhos através dos botões, da tela e dos menus. Os dispositivos internos de um celular e os processos que transformam o som capturado pelo microfone em ondas que podem ser transmitidas para uma antena da operadora de telefonia móvel constituem a **implementação** do celular.

Do ponto de vista do usuário de um celular, para fazer uma ligação, basta digitar o número do telefone desejado e clicar no botão que efetua a ligação. Porém, diversos processos complexos são realizados pelo aparelho para que as pessoas possam conversar através dele. Se os usuários tivessem que possuir conhecimento de todo o funcionamento interno dos celulares, certamente a maioria das pessoas não os utilizariam.

No contexto da orientação a objetos, aplicamos o encapsulamento para criar objetos mais simples de serem utilizados em qualquer parte do sistema.



11.3.2. Carro - Evitando efeitos colaterais

A interface de uso de um carro é composta pelos dispositivos que permitem que o motorista conduza o veículo (volante, pedais, alavanca do câmbio, etc).

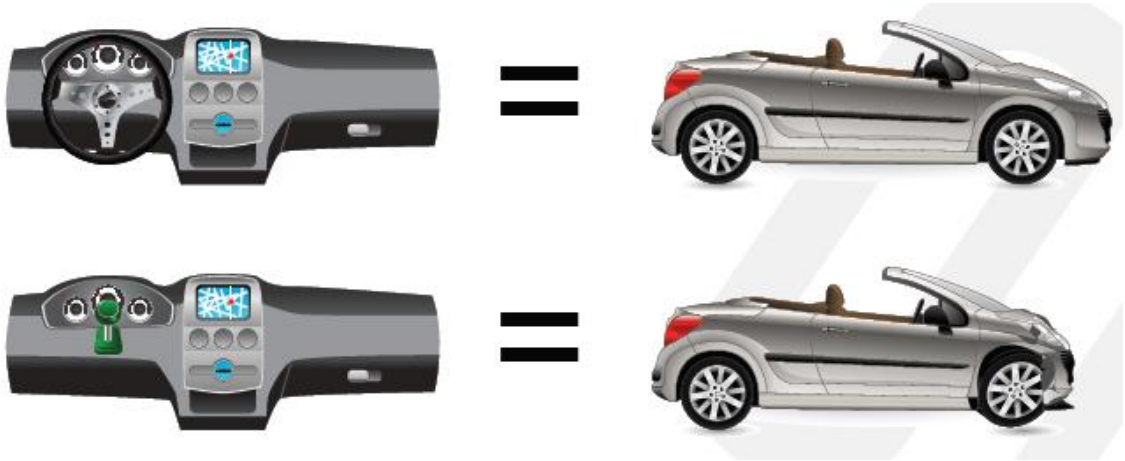
A implementação do carro é composta pelos dispositivos internos (motor, caixa de câmbio, radiador, sistema de injeção eletrônica ou carburador, etc) e pelos processos realizados internamente por esses dispositivos.

Nos carros mais antigos, o dispositivo interno que leva o combustível para o motor é o carburador. Nos carros mais novos, o carburador foi substituído pelo sistema de injeção eletrônica. Inclusive, algumas oficinas especializadas substituem o carburador pelo sistema de injeção eletrônica.

Essa alteração na implementação do carro não afeta a maneira que o motorista dirige. Todo mundo que sabe dirigir um carro com carburador também sabe dirigir um carro com injeção eletrônica. Hoje em dia, as montadoras fabricam veículos com câmbio mecânico ou automático. O motorista acostumado a dirigir carros com câmbio mecânico pode ter dificuldade para dirigir carros com câmbio automático e vice-versa. Quando a interface de uso do carro é alterada, a maneira de dirigir é afetada, fazendo com que as pessoas que sabem dirigir tenham que se adaptar.

No contexto da orientação a objetos, aplicando o conceito do encapsulamento, as implementações dos objetos ficam “escondidas”. Dessa forma, podemos modificá-las sem afetar a maneira de utilizar esses objetos. Por outro lado, se alterarmos a interface de uso que está exposta, afetaremos a maneira de usar os objetos.

Considere, por exemplo, a mudança do nome de um método público. Todas as chamadas a esse método devem ser alteradas, o que pode causar diversos efeitos colaterais nas classes da aplicação.



11.3.3. Máquinas de Porcarias - Aumentando o controle

Estamos acostumados a utilizar máquinas de refrigerantes, de salgadinhos, de doces, de café, etc. Em geral, essas máquinas oferecem uma interface de uso composta por:

- Entradas para moedas ou cédulas.
- Botões para escolher o produto desejado.
- Saída do produto.
- Saída para o troco.

Normalmente, essas máquinas são extremamente protegidas. Elas garantem que nenhum usuário mal intencionado (ou não) tente alterar a implementação da máquina, ou seja, tente alterar como a máquina funciona por dentro.

Levando essa ideia para um sistema orientado a objetos, um objeto deve ser bem protegido para que outros objetos não prejudiquem o seu funcionamento interno.

11.3.4. Acessando ou modificando atributos

Aplicando a ideia do encapsulamento, os atributos deveriam ser todos privados. Consequentemente, os atributos não podem ser acessados ou modificados por código escrito fora da classe na qual eles foram definidos.

Porém, muitas vezes, as informações armazenadas nos atributos precisam ser consultadas de qualquer lugar do sistema. Nesse caso, podemos disponibilizar métodos para consultar os valores dos atributos.

```
class Cliente {
  private String nome;

  public String consultaNome() {
    return this.nome;
  }
}
```

Da mesma forma, eventualmente, é necessário modificar o valor de um atributo a partir de qualquer lugar do sistema. Nesse caso, também poderíamos criar um método para essa tarefa.

```
class Cliente {
  private String nome;

  public void alteraNome(String nome){
    this.nome = nome;
  }
}
```

Muitas vezes, é necessário consultar e alterar o valor de um atributo a partir de qualquer lugar do sistema. Nessa situação, podemos definir os dois métodos discutidos anteriormente. Mas, o que é melhor? Criar os dois métodos (um de leitura e outro de escrita) ou deixar o atributo público?

Quando queremos consultar a quantidade de combustível de um automóvel, olhamos o painel ou abrimos o tanque de combustível?

Quando queremos alterar o toque da campainha de um celular, utilizamos os menus do celular ou desmontamos o aparelho?

Acessar ou modificar as propriedades de um objeto manipulando diretamente os seus atributos é uma abordagem que normalmente gera problemas. Por isso, é mais seguro para a integridade dos objetos e, consequentemente, para a integridade da aplicação, que esse acesso ou essa modificação sejas realizados através de métodos do objeto. Utilizando métodos, podemos controlar como as alterações e as consultas são realizadas. Ou seja, temos um controle maior.

12. Getters e Setters

Na linguagem Java, há uma convenção de nomenclatura para os métodos que têm como finalidade acessar ou alterar as propriedades de um objeto.

Segundo essa convenção, os nomes dos métodos que permitem a consulta das propriedades de um objeto devem possuir o prefixo **get**. Analogamente, os nomes dos métodos que permitem a alteração das propriedades de um objeto devem possuir o prefixo **set**.

Na maioria dos casos, é muito conveniente seguir essa convenção, pois os desenvolvedores Java já estão acostumados com essas regras de nomenclatura e o funcionamento de muitas bibliotecas do Java depende fortemente desse padrão.

```
class Cliente {
  private String nome;

  public String getNome() {
    return this.nome;
  }

  public void setNome(String nome) {
    this.nome = nome;
  }
}
```

12.1. Exercícios de Fixação

1-) Crie um projeto no NetBeans chamado **Encapsulamento**.

2-) Defina uma classe para representar os funcionários do banco com um atributo para guardar os salários e outro para os nomes.

```
public class Funcionario {
  double salario;
  String nome;
}
```

3-) Teste a classe Funcionario criando um objeto e manipulando diretamente os seus atributos.

```
class Teste {
  public static void main(String[] args) {
    Funcionario f = new Funcionario();

    f.nome = "Rafael Cosentino";
    f.salario = 2000;

    System.out.println(f.nome);
    System.out.println(f.salario);
  }
}
```

4-) Compile a classe Teste e perceba que ela pode acessar ou modificar os valores dos atributos de um objeto da classe Funcionario. Execute o teste e observe o console.

5-) Aplique a ideia do encapsulamento tornando os atributos definidos na classe Funcionario privados.

```
class Funcionario {
  private double salario;
  private String nome;
}
```

6-) Tente compilar novamente a classe Teste. Observe os erros de compilação. Lembre-se que um atributo privado só pode ser acessado por código escrito na própria classe do atributo.

7-) Crie métodos de acesso com nomes padronizados para os atributos definidos na classe Funcionario.

```
class Funcionario {
  private double salario;
  private String nome;

  public double getSalario() {
    return this.salario;
  }

  public String getNome() {
    return this.nome;
  }

  public void setSalario(double salario) {
    this.salario = salario;
  }

  public void setNome(String nome) {
    this.nome = nome;
  }
}
```

8-) Altere a classe Teste para que ela utilize os métodos de acesso ao invés de manipular os atributos do objeto da classe Funcionario diretamente.

```
class Teste {
  public static void main(String[] args) {
    Funcionario f = new Funcionario();

    f.setNome("Rafael Cosentino");
    f.setSalario(2000);

    System.out.println(f.getNome());
    System.out.println(f.getSalario());
  }
}
```

Compile e execute o teste!

13. HERANÇA

13.1. Reutilização de Código

Um banco oferece diversos serviços que podem ser contratados individualmente pelos clientes. Quando um serviço é contratado, o sistema do banco deve registrar quem foi o cliente que contratou o serviço, quem foi o funcionário responsável pelo atendimento ao cliente e a data de contratação.

Com o intuito de ser produtivo, a modelagem dos serviços do banco deve diminuir a repetição de código. A ideia é reaproveitar o máximo do código já criado. Essa ideia está diretamente relacionada ao conceito Don't Repeat Yourself. Em outras palavras, devemos minimizar ao máximo a utilização do "copiar e colar". O aumento da produtividade e a diminuição do custo de manutenção são as principais motivações do DRY.

Em seguida, vamos discutir algumas modelagens possíveis para os serviços do banco. Buscaremos seguir a ideia do DRY na criação dessas modelagens.

Uma classe para todos os serviços

Poderíamos definir apenas uma classe para modelar todos os tipos de serviços que o banco oferece.

```
1  class Servico {
2    private Cliente contratante;
3    private Funcionario responsavel;
4    private String dataDeContratacao;
5
6    // métodos
7  }
```

13.1. Empréstimo

O empréstimo é um dos serviços que o banco oferece. Quando um cliente contrata esse serviço, são definidos o valor e a taxa de juros mensal do empréstimo. Devemos acrescentar dois atributos na classe Servico: um para o valor e outro para a taxa de juros do serviço de empréstimo.

```
1  class Servico {
2    // GERAL
3    private Cliente contratante;
4    private Funcionario responsavel;
5    private String dataDeContratacao;
6
7    // EMPRÉSTIMO
8    private double valor;
9    private double taxa;
10
11   // métodos
12 }
```

13.2. Seguro de veículos

Outro serviço oferecido pelo banco é o seguro de veículos. Para esse serviço devem ser definidas as seguintes informações: veículo segurado, valor do seguro e a franquia. Devemos adicionar três atributos na classe Servico.

```
class Servico {
  // GERAL
  private Cliente contratante;
  private Funcionario responsavel;
  private String dataDeContratacao;

  // EMPRÉSTIMO
  private double valor;
  private double taxa;

  // SEGURO DE VEICULO
  private Veiculo veiculo;
  private double valorDoSeguroDeVeiculo;
  private double franquia;

  // métodos
}
```

Apesar de seguir a ideia do DRY, modelar todos os serviços com apenas uma classe pode dificultar o desenvolvimento. Supondo que dois ou mais desenvolvedores são responsáveis pela implementação dos serviços, eles provavelmente modificariam a mesma classe concorrentemente. Além disso, os desenvolvedores, principalmente os recém chegados no projeto do banco, ficariam confusos com o código extenso da classe Servico.

Outro problema é que um objeto da classe Servico possui atributos para todos os serviços que o banco oferece. Na verdade, ele deveria possuir apenas os atributos relacionados a um serviço. Do ponto de vista de performance, essa abordagem causaria um consumo desnecessário de memória.

13.3. Uma classe para cada serviço

Para modelar melhor os serviços, evitando uma quantidade grande de atributos e métodos desnecessários, criaremos uma classe para cada serviço.

```
class SeguroDeVeiculo {
  // GERAL
  private Cliente contratante;
  private Funcionario responsavel;
  private String dataDeContratacao;

  // SEGURO DE VEICULO
  private Veiculo veiculo;
  private double valorDoSeguroDeVeiculo;
  private double franquia;

  // métodos
}
```

```
class Emprestimo {
  // GERAL
  private Cliente contratante;
  private Funcionario responsavel;
  private String dataDeContratacao;

  // EMPRÉSTIMO
  private double valor;
  private double taxa;

  // métodos
}
```

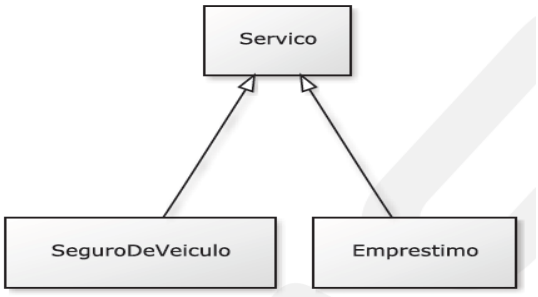
Criar uma classe para cada serviço torna o sistema mais flexível, pois qualquer alteração em um determinado serviço não causará efeitos colaterais nos outros. Mas, por outro lado, essas classes teriam bastante código repetido, contrariando a ideia do DRY. Além disso, qualquer alteração que deva ser realizada em todos os serviços precisa ser implementada em cada uma das classes.

13.4. Uma classe genérica e várias específicas

Na modelagem dos serviços do banco, podemos aplicar um conceito de orientação a objetos chamado Herança. A ideia é reutilizar o código de uma determinada classe em outras classes.

Aplicando herança, teríamos a classe Servico com os atributos e métodos que todos os serviços devem ter e uma classe para cada serviço com os atributos e métodos específicos do determinado serviço.

As classes específicas seriam “ligadas” de alguma forma à classe Servico para reaproveitar o código nela definido. Esse relacionamento entre as classes é representado em UML pelo diagrama abaixo.



Os objetos das classes específicas Emprestimo e SeguroDeVeiculo possuiriam tanto os atributos e métodos definidos nessas classes quanto os definidos na classe Servico.

```
1  Emprestimo e = new Emprestimo();
2
3  // Chamando um método da classe Servico
4  e.setDataDeContratacao("10/10/2010");
5
6  // Chamando um método da classe Emprestimo
7  e.setValor(10000);
```

As classes específicas são vinculadas a classe genérica utilizando o comando extends. Não é necessário redefinir o conteúdo já declarado na classe genérica.

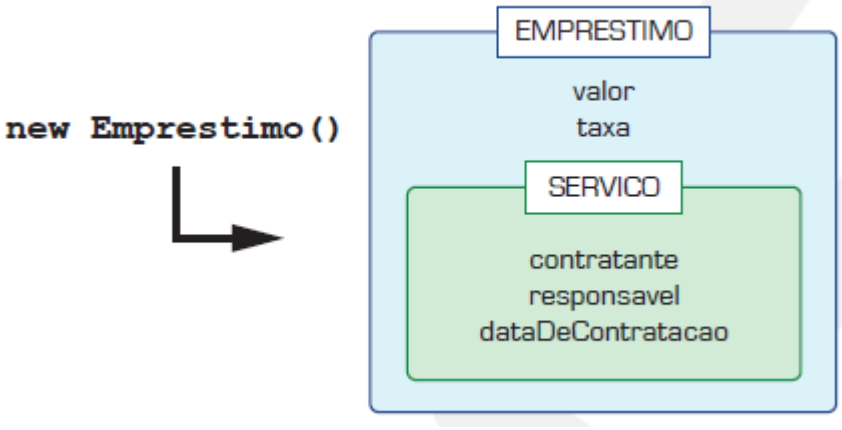
```
1  class Servico {
2    private Cliente contratante;
3    private Funcionario responsavel;
4    private String dataDeContratacao;
5  }
```

```
1  class Emprestimo extends Servico {
2    private double valor;
3    private double taxa;
4  }
```

```
1  class SeguroDeVeiculo extends Servico {
2    private Veiculo veiculo;
3    private double valorDoSeguroDeVeiculo;
4    private double franquia;
5  }
```

A classe genérica é denominada super classe, classe base ou classe mãe. As classes específicas são denominadas sub classes, classes derivadas ou classes filhas.

Quando o operador new é aplicado em uma sub classe, o objeto construído possuirá os atributos e métodos definidos na sub classe e na super classe.



Preço Fixo

Suponha que todo serviço do banco possui uma taxa administrativa que deve ser paga pelo cliente que contratar o serviço. Inicialmente, vamos considerar que o valor dessa taxa é igual para todos os serviços do banco.

Neste caso, poderíamos implementar um método na classe Servico para calcular o valor da taxa.

Este método será reaproveitado por todas as classes que herdam da classe Servico.

```
class Servico {
  // ATRIBUTOS

  public double calculaTaxa() {
    return 10;
  }
}
```

```
Emprestimo e = new Emprestimo();

SeguroDeVeiculo sdv = new SeguroDeVeiculo();

System.out.println("Emprestimo: " + e.calculaTaxa());

System.out.println("SeguroDeVeiculo: " + sdv.calculaTaxa());
```

13.5. Reescrita de Método

Suponha que o valor da taxa administrativa do serviço de empréstimo é diferente dos outros serviços, pois ele é calculado a partir do valor emprestado ao cliente. Como esta lógica é específica para o serviço de empréstimo, devemos acrescentar um método para implementar esse cálculo na classe Emprestimo.

```
class Emprestimo extends Servico {
  // ATRIBUTOS

  public double calculaTaxaDeEmprestimo() {
    return this.valor * 0.1;
  }
}
```

Os métodos das classes específicas têm prioridade sobre os métodos das classes genéricas. Em outras palavras, se o método chamado existe na classe filha ele será chamado, caso contrário o método será procurado na classe mãe.

Quando definimos um método com a mesma assinatura na classe base e em alguma classe derivada, estamos aplicando o conceito de Reescrita de Método.

Fixo + Específico

Suponha que o preço de um serviço é a soma de um valor fixo mais um valor que depende do tipo do serviço. Por exemplo, o preço do serviço de empréstimo é 5 reais mais uma porcentagem do valor emprestado ao cliente. O preço do serviço de seguro de veículo é 5 reais mais uma porcentagem do valor do veículo segurado. Em cada classe específica, podemos reescrever o método calculaTaxa().

```
class Emprestimo extends Servico {
  // ATRIBUTOS

  public double calculaTaxa() {
    return 5 + this.valor * 0.1;
  }
}
```

```
class SeguraDeVeiculo extends Servico {
  // ATRIBUTOS

  public double calculaTaxa() {
    return 5 + this.veiculo.getTaxa() * 0.05;
  }
}
```

Se o valor fixo dos serviços for atualizado, todas as classes específicas devem ser modificadas. Outra alternativa seria criar um método na classe Servico para calcular o valor fixo de todos os serviços e chamá-lo dos métodos reescritos nas classes específicas.

```
class  Servico {
  public double calculaTaxa() {
    return  5 ;
  }
}
```

```
class Emprestimo extends Servico {
  // ATRIBUTOS

  public double calculaTaxa() {
    return  super.calculaTaxa() + this.valor * 0.1;
    }
}
```

Dessa forma, quando o valor padrão do preço dos serviços é alterado, basta modificar o método na classe Servico.

13.6. Construtores e Herança

Quando temos uma hierarquia de classes, as chamadas dos construtores são mais complexas do que o normal. Pelo menos um construtor de cada classe de uma mesma seqüencia hierárquica deve ser chamado ao instanciar um objeto. Por exemplo, quando um objeto da classe Emprestimo é criado, pelo menos um construtor da própria classe Emprestimo um da classe Servico devem ser executados. Além disso, os construtores das classes mais genéricas são chamados antes dos construtores das classes específicas.

```
class  Servico {
  // ATRIBUTOS

  public Servico() {
    System.out.println("Servico");
  }
}
```

```
class Emprestimo extends Servico {
  // ATRIBUTOS

  public Emprestimo() {
    System.out.println("Emprestimo");
  }
}
```

Por padrão, todo construtor chama o construtor sem argumentos da classe mãe se não existir nenhuma chamada de construtor explícita.

13.7. Exercícios de Fixação

1) Crie um projeto chamado Heranca2.

2) Defina uma classe para modelar os funcionários do banco. Sabendo que todo funcionário possui nome e salário, inclua os getters e setters dos atributos.

```
1  class Funcionario {
2    private String nome;
3    private double salario;
4
5    // GETTERS AND SETTERS
6  }
```

3) Crie uma classe para cada tipo específico de funcionário herdando da classe Funcionario. Considere apenas três tipos específicos de funcionários: gerentes, telefonistas e secretarias. Os gerentes possuem um nome de usuário e uma senha para acessar o sistema do banco. As telefonistas possuem um código de estação de trabalho. As secretarias possuem um número de ramal.

```
1  class Gerente extends Funcionario {
2    private String usuario;
3    private String senha;
4
5    // GETTERS AND SETTERS
6  }
```

```
1  class Telefonista extends Funcionario {
2    private int estacaoDeTrabalho;
3
4    // GETTERS AND SETTERS
5  }
```

```
1  class Secretaria extends Funcionario {
2    private int ramal;
3
4    // GETTERS AND SETTERS
5  }
```

4) Teste o funcionamento dos três tipos de funcionários criando um objeto de cada uma das classes: Gerente, Telefonista e Secretaria.

```
class TestaFuncionarios {
  public static void main(String[] args) {
    Gerente g = new Gerente();
    g.setNome("Rafael Cosentino");
    g.setSalario(2000);
    g.setUsuario("rafael.cosentino");
    g.setSenha("12345");

    Telefonista t = new Telefonista();
    t.setNome("Carolina Mello");
    t.setSalario(1000);
    t.setEstacaoDeTrabalho(13);

    Secretaria s = new Secretaria();
    s.setNome("Tatiane Andrade");
    s.setSalario(1500);
    s.setRamal(198);

    System.out.println("GERENTE");
    System.out.println("Nome: " + g.getNome());
    System.out.println("Salário: " + g.getSalario());
    System.out.println("Usuário: " + g.getUsuario());
    System.out.println("Senha: " + g.getSenha());

    System.out.println("TELEFONISTA");
    System.out.println("Nome: " + t.getNome());
    System.out.println("Salário: " + t.getSalario());
    System.out.println("Estacao de trabalho: " + t.getEstacaoDeTrabalho());

    System.out.println("SECRETARIA");
    System.out.println("Nome: " + s.getNome());
    System.out.println("Salário: " + s.getSalario());
    System.out.println("Ramal: " + s.getRamal());
  }
}
```

Execute o teste!

5) Suponha que todos os funcionários recebam uma bonificação de 10% do salário. Acrescente um método na classe Funcionario para calcular essa bonificação.

```
class Funcionario {
  private String nome;
  private double salario;

  public double calculaBonificacao() {
    return this.salario * 0.1;
  }

  // GETTERS AND SETTERS
}
```

6) Altere a classe TestaFuncionarios para imprimir a bonificação de cada funcionário, além dos dados que já foram impressos. Depois, execute o teste novamente.

```
class TestaFuncionarios {
  public static void main(String[] args) {
    Gerente g = new Gerente();
    g.setNome("Rafael Cosentino");
    g.setSalario(2000);
    g.setUsuario("rafael.cosentino");
    g.setSenha("12345");

    Telefonista t = new Telefonista();
    t.setNome("Carolina Mello");
    t.setSalario(1000);
    t.setEstacaoDeTrabalho(13);

    Secretaria s = new Secretaria();
    s.setNome("Tatiane Andrade");
    s.setSalario(1500);
    s.setRamal(198);

    System.out.println("GERENTE");
    System.out.println("Nome: " + g.getNome());
    System.out.println("Salário: " + g.getSalario());
    System.out.println("Usuário: " + g.getUsuario());
    System.out.println("Senha: " + g.getSenha());
    System.out.println("Bonificação: " + g.calculaBonificacao());

    System.out.println("TELEFONISTA");
    System.out.println("Nome: " + t.getNome());
    System.out.println("Salário: " + t.getSalario());
    System.out.println("Estacao de trabalho: " + t.getEstacaoDeTrabalho());
    System.out.println("Bonificação: " + t.calculaBonificacao());

    System.out.println("SECRETARIA");
    System.out.println("Nome: " + s.getNome());
    System.out.println("Salário: " + s.getSalario());
    System.out.println("Ramal: " + s.getRamal());
    System.out.println("Bonificação: " + s.calculaBonificacao());
  }
}
```

7) Suponha que os gerentes recebam uma bonificação maior que os outros funcionários. Reescreva o método calculaBonificacao() na classe Gerente. Depois, compile e execute o teste novamente.

```
class Gerente extends Funcionario {
  private String usuario;
  private String senha;

  public double calculaBonificacao() {
    return this.getSalario() * 0.6 + 100;
  }

  // GETTERS AND SETTERS
}
```

13.8. Exercícios Complementares

1) Defina na classe Funcionario um método para imprimir na tela o nome, salário e bonificação dos funcionários.

2) Reescreva o método que imprime os dados dos funcionários nas classes Gerente, Telefonista e Secretaria para acrescentar a impressão dos dados específicos de cada tipo de funcionário.

3) Modifique a classe TestaFuncionarios para utilizar o método mostraDados().

13.9. Exercício final.

Implementem um sistema com os conceitos abordados em aula, criando, classes, atributos, métodos, herança, sobrecarga de método e reescrita de método, podem realizar em equipes.