UniFil
EDUCAÇÃO A DISTÂNCIA

Unidade 01

INTRODUÇÃO À ORIENTAÇÃO A OBJETOS E UML

Abertura

Apresentação

A UML tem uma relevância crucial na implementação de sistemas, ela permite aumentar a qualidade do processo do desenvolvimento do *software*. Com a UML será possível termos uma visualização, especificação, a construção e a documentação de artefatos que façam uso de sistemas complexos de *software*. Na unidade 1, apresentaremos os principais conceitos utilizados na orientação a objetos e a modelagem de requisitos com o diagrama de **Casos de Uso.** Na unidade 2, faremos a modelagem dos diagramas de **Atividades**, **Estados** e **Implantação**. Na unidade 3, trabalharemos com o diagrama de **Classe**. Finalmente, na unidade 4, serão utilizados os diagramas de **Sequência**, **Comunicação** e **Componentes**. Serão mostradas as aplicabilidades dos diagramas de **Visão Geral**, **Tempo**, **Pacotes** e **Perfil.**

Apresentação do Professor

- Meu nome é Simone Sawasaki Tanaka;
- Fiz o meu mestrado em Ciência da Computação na Universidade Estadual de Londrina;
- Conclui a Especialização em Engenharia de *Software* na Unopar;
- Fiz Especialização em Educação a Distância no Senac;
- Sou graduada em Tecnologia em Processamento de Dados no Cesulon (o Cesulon é a antiga UniFil);
- Sou docente na UniFil desde 2003, atualmente ministro aulas nos cursos de Ciência da Computação, Engenharia de *Software*, Análise e Desenvolvimento de Sistemas. Estou como coordenadora de Estágio em todos os cursos citados e coordeno o Projeto Aluno Tutor Google.

Objetivos

- Conhecer metodologias para o desenvolvimento de *software* orientado a objetos;
- Abstrair e modelar requisitos de *software*;
- Elaborar os diagramas com rastreabilidade e integração entre todos os diagramas;

- Interpretar um modelo UML e gerar os códigos utilizando uma ferramenta CASE.

Quer assistir às videoaulas em seu celular? Basta apontar a câmera para os **QR Codes** distribuídos neste conteúdo.

Caso necessário, instale um aplicativo de leitura QR Code no celular e efetue o login na sua conta Gmail.

Videoaula Apresentação da Disciplina

Utilize o QR Code para assistir!

Assista!

Videoaula Minicurrículo

Utilize o QR Code para assistir!

Assista!

Introdução da Unidade

Olá, amigo(a) discente! Seja bem-vindo(a)!

Nesta unidade, serão apresentados os principais conceitos utilizados na orientação a objeto e sua aplicabilidade. A Linguagem de Modelagem Unificada (UML) será utilizada como padrão na aplicação dos projetos. Vamos aprender a modelar os requisitos de *software* com o diagrama de caso utilizando o estudo de caso de Locação de Veículos.

Objetivos

- Contextualizar sobre os principais conceitos da orientação a objetos;
- Introduzir os conceitos da UML;
- Entender o que é um requisito de *software*;
- Modelar os requisitos de *software* utilizando o diagrama de caso de uso.

Conteúdo Programático

Aula 01 – Introdução a Orientação a Objetos e UML.

Aula 02 – Diagrama de Caso de Uso.

INTRODUÇÃO A ORIENTAÇÃO A OBJETOS E UML

Nesta aula, vamos abordar sobre os principais conceitos da Orientação a Objetos. Esses conceitos serão essenciais para o entendimento do restante da nossa disciplina.

Orientação a objetos pode ser considerada uma tecnologia que define os sistemas como uma coleção de objetos e suas funcionalidades. Esta tecnologia permitiu colocar em prática o conceito de reusabilidade no desenvolvimento de *software* (TACLA, 2010).

Representar fielmente as situações do mundo real, é a proposta da Orientação a Objetos. Por este motivo, o conceito não vê um sistema computacional como um conjunto de processos, mas como uma coleção de objetos que interagem entre si.

Logo, os sistemas Orientados a Objetos têm uma estrutura diferente. São disponibilizados em módulos que contêm estado e suas operações, e permitem a reutilização do código, por meio da herança. Polimorfismo também é importante, uma vez que permite escolher funcionalidades que um determinado programa poderá assumir assim que for executado.

A Linguagem Simula 67, foi a primeira a possibilitar a implementação do conceito de objeto. Em seguida, os conceitos foram refinados na linguagem Smalltalk 80, que ainda hoje auxilia sendo protótipo de implementação dos modelos orientados a objetos. A linguagem mais popular hoje em dia é Java, porém, muitos adeptos consideram que a Smalltalk 80 é a única linguagem de programação integralmente orientada a objetos (VIDEIRA, 2008).

Esse novo paradigma se baseia principalmente em dois conceitos chave: **classes e objetos.**

Classes

Classe pode ser definida como uma entidade que representa um conjunto de objetos com características semelhantes. Uma classe define o comportamento dos objetos, por meio de métodos, e quais características ela possui, por meio de atributos. Exemplo de classe: Computador, Pessoa, Automóvel, Animal etc.

Objetos

Um objeto é a instância de uma classe. Dentro de um objeto podemos armazenar características por meio de seus atributos e ações por meio dos seus métodos, além de realizar comunicação com outros objetos, mediante o envio de mensagens. Exemplo de objetos:

- Pessoas: Simone, Sérgio, Wesley, José, Ana, Matheus, Elizabete, Maryane, Ghetter etc.
- Animais: Cachorro, gato, boi, vaca etc.
- Veículos: Ferrari, Fusca, Mercedes, Gol, Parati etc.
- Objetos: Geladeira, micro-ondas, telefone, livro etc.

Interfaces

Uma interface pode ser definida como uma classe que possui um conjunto de métodos padronizados, sem implementação. Por meio das interfaces é possível estabelecer uma forma de programação uniforme, pois, ela possui o esqueleto para o desenvolvimento de uma determinada classe ou componente.

Como já foi dito, na interface possuímos apenas a assinatura do método, essa assinatura será transmitida e implementada por todas as classes que utilizarem a instrução "*implements*" seguida pelo nome da interface em sua assinatura de classe. Por exemplo, se criarmos uma classe chamada Cliente, e uma interface chamada InterfaceCliente para que a classe cliente possua e implemente os métodos da interface, a sintaxe para o comando seria a seguinte:

- public class Cliente implements InterfaceCliente, então, a assinatura de todos os métodos declarados na interface seria adicionada a classe cliente.

Atributos

Na programação orientada a objetos, os atributos podem ser definidos como características que uma determinada classe ou objeto possui, apresentando assim a estrutura dessa classe. Eles também são chamados de variáveis de classe e podem ser classificados em atributos de instância e atributos de classe, de acordo com seu nível de acesso.

Atributos de instância são os responsáveis por determinar o estado de um determinado objeto e suas características. Já os atributos de classe são os responsáveis por controlar informações de uma classe ou de um conjunto de classes, pois, contém informações que são compartilhadas por todos os objetos de uma determinada classe; a esse tipo de atributo, dá-se também o nome de atributos estáticos ou constantes.

Quando uma mensagem é enviada de um objeto para outro (invocação de um método, um objeto pode ter um ou mais de seus atributos alterados, o que poderá alterar também o seu estado.

Métodos

Definem os comportamentos dos objetos. Ferrari é uma instância da classe carro, portanto tem habilidade para ligar o motor, implementada por meio do método ligarMotor().

Ao declarar um método em uma classe, algumas regras devem ser seguidas:

- Pode ser definido um tipo de acesso para ele;
- *public*: método pode ser acessado por qualquer parte do programa, por meio da instância do objeto;
- *private*: somente a classe que possui o método poderá acessá-lo;
- *protected*: somente subclasses poderão acessar o método;
- *static*: método pode ser acessado por qualquer parte do programa, sem necessidade de instanciar o objeto.
- Pode ser definido o tipo de retorno do método.
    - *String*: conjunto de caracteres;

- *int*: números inteiros, positivos e negativos;
- *float* e *double*: números com ponto flutuante;
- *boolean*: verdadeiro ou falso;
- *Date*: data e hora;
- *void*: não retorna nada, somente executa as ações do método.

- Pode ser definido um nome para o método.
  - O nome sempre começa com a letra minúscula;
  - Nomes compostos devem iniciar com as letras maiúsculas;
  - Não devem ser utilizados caracteres especiais;
  - O nome deve ser claro em relação ao objetivo da classe;
  - Nomes de classes não devem possuir acentos.
- Caso haja parâmetros, eles podem ser informados.
  - Pode ser especificado o tipo do parâmetro (*int*, *string*, *float*, *double*, *boolean*, *date*).
- Pode ser especificado o nome do parâmetro:
  - O nome sempre começa com a letra minúscula;
  - Não devem ser utilizados caracteres especiais;
  - O nome deve ser claro em relação ao objetivo da classe;
  - Nomes de classes não devem possuir acentos.
  - Nomes compostos devem iniciar com as letras maiúsculas, precedido de letras minúsculas.

Estrutura da assinatura de um método:

<<tipo de acesso>> <<tipo de retorno>> <<nome>> (<<tipo_parametro nome_parametro1>>).

Exemplos:

public void andar(int velocidade) { }

Para a nossa Ferrari

public void ligar(){ }

public void desligar(){ }

Abstração

Abstração, como o próprio nome diz se refere à capacidade de abstrair algo, ou seja, para o desenvolvimento de *software*, a abstração representa a habilidade de pensar em coisas do

mundo (pessoas, carros, clientes, fornecedores, produtos, notas fiscais, relatórios) e ser capaz de gerar partes de um sistema, visualizando a forma pela qual esses se relacionam.

Ela é um processo mental no qual as ideias estão distanciadas dos objetos e é por meio dela que podemos imaginar resultados para uma determinada decisão ou ação, sem recorrer a mecanismos físicos ou mecânicos de resolução. Mesmo que a abstração esteja sempre vinculada à criação, podemos perceber que a criação também está vinculada à abstração, portanto, dificilmente uma existirá sem a outra.

Encapsulamento

Encapsulamento tem a finalidade de auxiliar na manutenção do *software*, tornando-o mais "flexível" a mudanças e inclusão de recursos. Para isso, no desenvolvimento do *software*, é necessário que o código seja analisado minuciosamente, a fim de abstrair as partes que podem ser reutilizadas.

Por exemplo, como sabemos, sistemas são usados diariamente por pessoas, para realização de suas atividades, como forma de organização, para realizar vendas, enfim, uma série de finalidades é atribuída a um sistema computacional. Nem sempre o usuário tem necessidade de conhecimento técnico para a sua utilização, tal qual, quando acessamos a internet e realizamos uma compra.

A linguagem de programação utilizada para o desenvolvimento do *e-commerce*, a forma de integração com o banco e as atividades que compõem o processo de venda não são de preocupação do cliente, e ele não precisa conhecê-los, o que importa para ele, é que haja uma interface na qual, em poucos cliques ele selecione um produto desejado, efetue o pagamento e o receba em sua casa.

Uma das vantagens do encapsulamento é que toda parte encapsulada pode ter suas características alteradas sem afetar outros objetos ou partes do sistema que as utilizam. No exemplo citado acima, sobre *o e-commerce*, um desenvolvedor poderia mudar a linguagem de programação, ou a forma de integração com os bancos, sem alterar o *layout* da página e o usuário final nem perceberia, pois o que importa para ele é o simples fato de realizar a compra clicando sobre um botão.

A finalidade do encapsulamento é proteger o acesso direto (referência) aos atributos de uma classe, vinculando seu acesso às instâncias de um objeto dessa classe. Essa proteção é realizada de forma simples, na qual, os modificadores de acesso dos atributos de cada classe são declarados como privados. Sendo assim, essas propriedades só podem ser alteradas pela própria classe que possui os atributos, então, são criados métodos para acessar indiretamente esses atributos pela instância da classe, o que garante que o novo valor daquele atributo seja atribuído somente para o objeto que instanciou, evitando assim "efeitos colaterais" no *software*.

Por padrões, utilizamos dois métodos para manipular os atributos de um objeto que estão encapsulados, são eles o *get* e o *set*. O método *get* é responsável por informar o valor de um dos atributos que o objeto possui já o método *set*, é responsável por "setar" ou atribuir um valor para um atributo do objeto.

O que difere os dois, exceto pelo nome, é o fato de o método get não possuir parâmetros para sua execução, já o método *set* necessita de um parâmetro com o novo valor que será atribuído ao atributo selecionado do objeto. Abaixo é apresentado um exemplo de classe utilizada para encapsulamento com os atributos e os métodos *get* e *set*.

```
age oo;

/* Classe responsável por encapsular os campos. */
public class OoEncapsulamento {
    // Atributos que serão encapsulados
    private String nome;
    private int matricula;
    /**
     * Método responsável por informar o nome
     * @return atributo da classe: nome
     */
    public String getNome() {
        return nome;
    }
    /**
     * Atribui um novo valor ao atributo "nome" da classe.
     * @param Novo nome que será atribuído.
     */
    public void setNome(String novoNome) {
        this.nome = novoNome;
    }
    /**
     * Método responsável por informar o número de matrícula
     * @return atributo da classe: matrícula
     */
    public int getMatricula() {
        return matricula;
    }
    /**
```

```
     * Atribui um novo valor ao atributo "matricula" da classe.
     * @param Novo número de matrícula.
     */
    public void setMatricula(int novaMatricula) {
        this.matricula = novaMatricula;
    }
}
```

Um exemplo prático: alterando o nome do aluno:

```
OoEncapsulamento aluno = new OoEncapsulamento();
aluno.setNome("Novo nome do aluno");
```

Para visualizarmos o nome:

```
System.out.println("Nome do aluno: " + aluno.getNome());
```

Herança

Um dos princípios mais importantes da programação orientada a objetos, a herança possibilita que classes compartilhem seus atributos, métodos por intermédio de um relacionamento chamado generalização.

Ao utilizar a herança estabelecemos uma relação entre uma classe chamada superclasse, essa superclasse pode ser considerada como uma classe Mãe, que irá possuir todas as operações e atributos que as classes filhas (também chamadas subclasses) irão possuir. Ao realizar a herança, assume-se que as subclasses se especializem, ou seja, além das características herdadas da classe mãe, elas ainda possuem seus métodos e atributos específicos. Uma das responsabilidades da herança é evitar a redundância de atributos e métodos em um sistema de *software*.

Na linguagem Java, existe uma palavra reservada responsável por realizar a herança entre classes, chamada *extends*, ela é apresentada na linha onde é declarada a classe, logo após o nome da classe. Em seguida, depois de utilizarmos a palavra *extends* devemos informar qual será a classe pai que a classe filha irá generalizar.

Observação Importante:

- É necessário realizar o *import* da classePai caso ela não esteja no mesmo *package* da classeFilha.

Abaixo será apresentado um exemplo de herança utilizando duas classes (ClassePai e ClasseFilha) na linguagem de programação Java. Por meio dela, podemos perceber que a classe filha herda o método getInformacoesPai(); que está contido na classe pai.

Classe 1

```
package oo.heranca;
```

```
public class ClasseFilha extends ClassePai {


 public ClasseFilha() {
   // Método da ClassePai, atribuído a ClasseFilha por Herança.
   getInformacoesPai();
  }
}
```

Classe 2

```
package oo.heranca;


public class ClassePai {


  /** Construtor */
  public ClassePai(){ }


   /** Método da classe pai que será acessado na classe filha */
   public void getInformacoesPai(){
     System.out.println("Informações contidas na ClassePai.java!!!");
   }
}
```

Observem que o método getInformacoesPai() existe somente na classe pai, e o mesmo foi utilizado na classe filha, que herdou todas as suas características. Nesse exemplo foi apresentado somente a herança de um único método, porém todos os métodos contidos em uma superclasse são atribuídos à classe filha no ato da herança, e não só os métodos, mas também os atributos que cada classe ou objeto possuir.

Polimorfismo

Esse é um dos recursos muito interessantes da programação orientada a objetos, pois, ele permite que um mesmo método com a mesma assinatura tenha comportamentos diferentes de acordo com suas necessidades. O termo polimorfismo tem origem grega e significa "muitas formas" (*poli* = muitas, *morphos* = formas).

O Polimorfismo fornece uma diretriz na qual torna possível aumentar o nível de abstração de uma aplicação visando reaproveitamento de código, facilidade na manutenção e padronização no desenvolvimento, separando a semântica de uma interface da implementação ou das implementações que a representam.

Videoaula 1

Utilize o QR Code para assistir!

Neste momento, vamos assistir ao vídeo sobre os principais conceitos de Orientação a Objetos.

Introdução a UML

É por meio da linguagem que as pessoas se comunicam e procuram se entender. A UML é uma linguagem, que fornece um vocabulário e regras, assim como uma linguagem verbal ou não verbal (BOOCH; RUMBAUGH; JACOBSON, 2006).

A UML é uma linguagem gráfica para visualizar, especificar, construir e documentar os artefatos de um sistema de *software*. Por meio de seus diagramas, é possível representar sistemas de *software* sob diversas perspectivas de visualização, facilitando a comunicação de todas as pessoas envolvidas no processo de desenvolvimento de um sistema - gerentes, coordenadores, analistas, desenvolvedores, por apresentar um vocabulário de fácil entendimento (BOOCH; RUMBAUGH; JACOBSON, 2006).

A importância da modelagem para um bom desenvolvimento e entendimento de um sistema de *software* torna a UML indispensável, proporcionando assim melhor comunicação entre todas as pessoas que estão envolvidas no projeto de desenvolvimento do *software* (TANAKA, 2011).

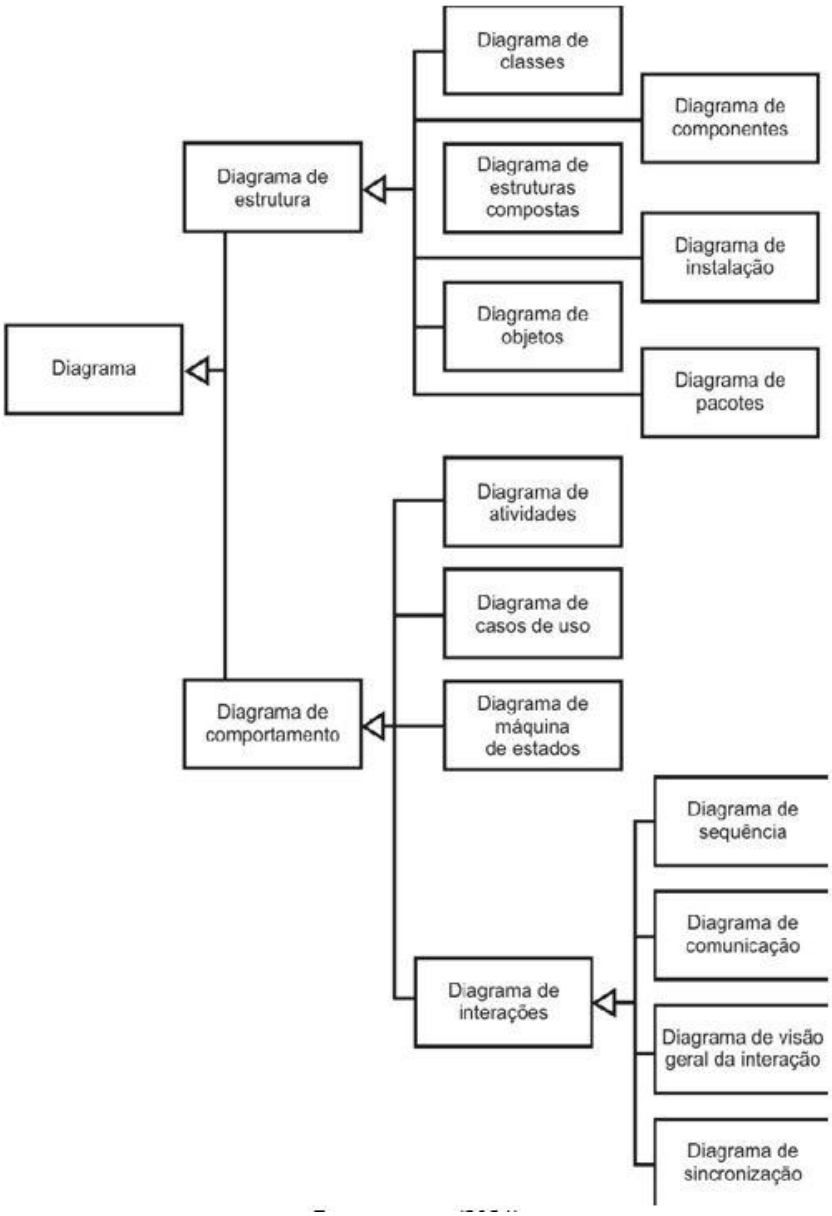
A UML começou a ser definida a partir de uma tentativa de Rumbaugh e Grady Booch de combinar dois métodos populares de modelagem orientada a objeto: *Booch* e *Object Modeling Language* (OMT). Mais tarde, Ivar Jacobson, o criador do método *Objectory*, uniu-se aos dois, para a concepção da primeira versão da linguagem *Unified Modeling Language* (UML).

Os esforços para a criação da UML tiveram início em outubro de 1994, quando Rumbaugh se juntou a Booch na *Rational*. Com o objetivo de unificar os métodos *Booch* e *OMT*, foi lançado, em outubro de 1995, o esboço da versão 0.8 do *Unified Process* - Processo Unificado (como era conhecido). Naquela mesma época, Jacobson se associou à *Rational*, e o escopo do projeto da UML foi expandido para incorporar o método *Object Oriented Software Engineering* (OOSE). Nasceu então, em junho de 1996, a “versão 0.9 da UML”.

Diagramas da UML

Atualmente, a UML encontra-se na versão 2.5 e possui 14 diagramas conforme apresentado na Figura abaixo: diagrama de objetos, diagrama de atividades, diagrama de pacotes, diagrama de sequência, diagrama de colaboração, diagrama de estados, diagrama de componentes, diagrama de implantação, diagrama de tempo, diagrama de classes, diagrama de casos de uso, diagrama de visão geral, diagrama de estrutura composta e diagrama de perfil.

Figura 1 – Tipos de diagramas UML



Fonte: o autor (2021)

Videoaula 2

Utilize o QR Code para assistir!

Neste momento, vamos assistir ao vídeo sobre o histórico da UML e sua aplicabilidade.

O fato da UML ser composta por vários diagramas se dá em virtude de fornecer múltiplas visões do sistema a ser modelado, analisando-o e modelando-o sob diversos aspectos, procurando-se, assim, atingir a completude das modelagens, permitindo que cada diagrama complemente os outros (GUEDES, 2009).

Diagrama de atividades

Tem o objetivo de especificar o processo de trabalho da empresa, relacionando as atividades executadas no processo aos papéis responsáveis pela sua execução. Ele apresenta um fluxo de dependência de atividades, pois as atividades são sequenciais e mutuamente dependentes para serem executadas.

Diagrama de pacotes

É um diagrama estrutural, que tem por objetivo representar os subsistemas ou submódulos englobados por um sistema, de forma a determinar as partes que o compõem. Pode ser utilizado também para auxiliar e demonstrar a arquitetura de uma linguagem, como ocorre com a própria UML, ou ainda para definir as camadas de um *software* ou processo de desenvolvimento (GUEDES, 2009).

Diagrama de sequência

É utilizado para ilustrar a interação entre os objetos. Eles modelam os objetos e as mensagens entre os objetos (PENDER, 2004). Costumam identificar o evento gerador do processo modelado, bem como o ator responsável por esse evento (GUEDES, 2009).

Diagrama de comunicação

Era conhecido como de colaboração até a versão 1.5 da UML, tendo seu nome modificado para diagrama de comunicação, a partir da versão 2.0. Está amplamente associado ao diagrama de sequência. O diagrama de comunicação não se preocupa com a temporalidade do processo, concentrando-se em como os elementos do diagrama estão vinculados e quais mensagens trocam entre si durante o processo (GUEDES, 2009).

Diagrama de estados

Demonstra o comportamento por meio de um conjunto finito de transições de estado, ou seja, uma máquina de estado (GUEDES, 2009).

Diagrama de componentes

Está associado à linguagem de programação que será utilizada para desenvolver o sistema modelado. Representa os componentes do sistema em termos de módulos de código-fonte, bibliotecas, formulários, entre outros (GUEDES, 2009).

Diagrama de implantação

Mostra a configuração dos nós de processamento em tempo de execução e os componentes neles existentes (BOOCH, 2005).

Diagrama de tempo

Ou diagrama de sincronismo é um diagrama de interação que mostra os tempos reais em diferentes objetos ou papéis, em vez da sequência de mensagens relativas. É um híbrido do diagrama de atividades e um diagrama de sequência (BOOCH, 2005).

Diagrama de classe

Mostra um conjunto de classes, interfaces, colaborações e seus relacionamentos. Este diagrama é importante não só para a visualização, a especificação e a documentação de modelos estruturais, mas também para a construção de sistemas executáveis por intermédio de engenharia de produção e reversa (BOOCH, 2005).

Diagrama de caso de uso

Exibe um conjunto de casos de uso e atores, e seus relacionamentos. São importantes principalmente para a organização e a modelagem de comportamento do sistema (BOOCH, 2005).

Diagrama de visão geral

É uma variação do diagrama de atividade que fornece uma visão geral dentro de um sistema ou processo de negócio (GUEDES, 2009).

Diagrama de estrutura composta

Descreve a estrutura de um classificador, como uma classe ou componente, detalhando as partes internas que o compõem, como estas se comunicam e colaboram entre si (GUEDES, 2009).

Leitura Obrigatória

Leia o livro *Fundamentos do Desenho Orientado a Objeto com UML*.

Acesse o livro no *link* abaixo:

Disponível em: https://plataforma.bvirtual.com.br/Leitor/Publicacao/33/pdf/0. Acesso em: 5 set. 2021.

Videoaula 3

Utilize o QR Code para assistir!

Neste momento, vamos assistir ao vídeo sobre uma abordagem geral dos Diagramas da UML.

Diagrama de Caso de Uso

DIAGRAMA DE CASO DE USO

Esse diagrama tem a finalidade de auxiliar a comunicação entre os envolvidos no projeto, sendo que, utilizando para isso a descrição de cada um dos cenários que irá compor o *software*. É por meio deste diagrama que o cliente percebe a relação entre as funcionalidades do sistema e os papéis que as executam.

Para sua composição, são necessários três componentes: o ator (representado por um boneco, esse componente é definido como algo ou alguém que interage com o sistema), o caso de uso (representado por uma elipse, este componente define uma funcionalidade do sistema) e os relacionamentos (meio pelo qual o usuário interage com o caso de uso) entre eles.

Caso de Uso: um Caso de Uso descreve o que um sistema faz, mas não como ele faz (BOOCH, 2005). A simplicidade do diagrama de caso de uso produz pontos fortes e fracos. Um ponto forte é a falta de detalhamento, de tal forma que possamos visualizar o sistema pelo diagrama. Em relação ao ponto fraco, o caso de uso normalmente é acompanhado por uma descrição narrativa ou também chamado de especificação de caso de uso (Pender, 2004).

Figura 1 – Caso de Uso

Fonte: o autor (2021)

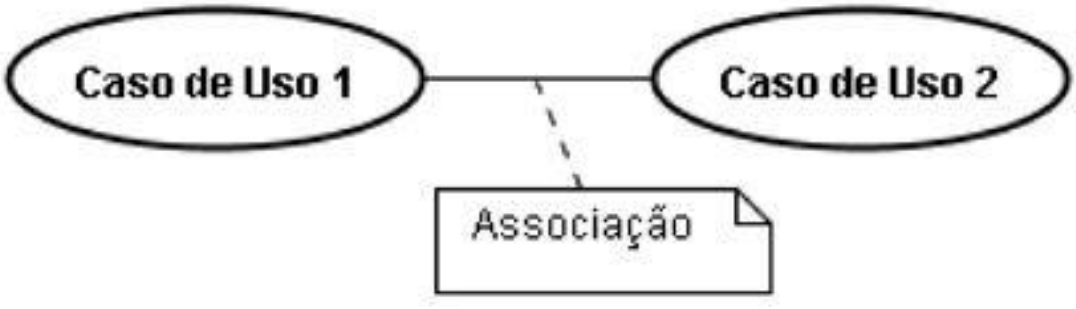
Ator: qualquer pessoa, departamento, sistema computacional e dispositivos que utilizam as funcionalidades do Sistema.

Figura 2 - Ator

Fonte: o autor (2021)

Associação: execução de uma funcionalidade do sistema por uma determinada característica pertencente a um outro componente

Figura 3 - Associação



Fonte: o autor (2021)

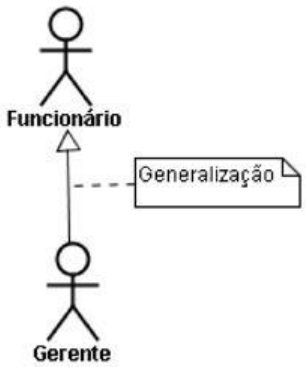
Videoaula 1

Utilize o QR Code para assistir!

Neste momento, vamos assistir ao vídeo sobre os conceitos básicos do diagrama de caso de uso.

- **Generalização**: capacidade de um ator adquirir comportamentos herdados de um outro ator ou de outro caso de uso. O relacionamento de generalização pode ser utilizado entre casos de uso, atores, classes. Na figura abaixo, o ator Gerente está herdando as funcionalidades do ator Funcionário, isto significa que o que o funcionário faz, o gerente também pode fazer, além disso o gerente pode executar outras atividades nas quais o funcionário não executará.

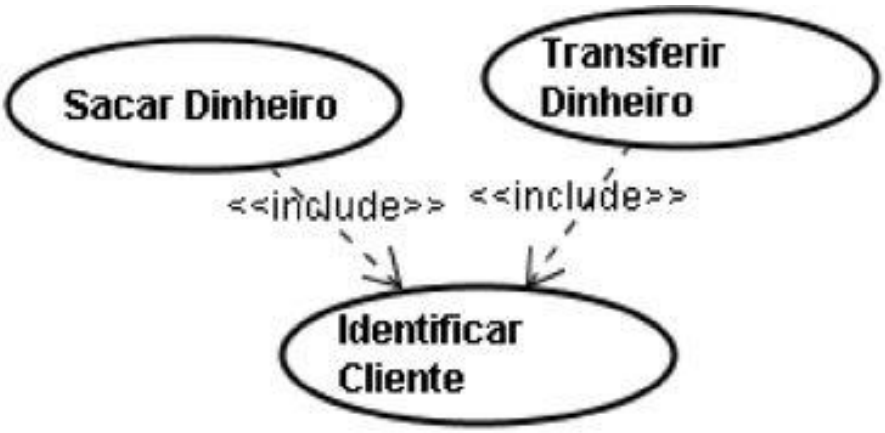
Figura 4 - Generalização



Fonte: o autor (2021)

Include: esse tipo de relacionamento, contido nos casos de uso indica uma "especialização entre casos de uso", que obriga a implementação do recurso no caso de uso "mais especializado". Conforme o exemplo, todas as vezes que ocorrer o Saque de dinheiro ou a Transferência do dinheiro, obrigatoriamente é necessário identificar o cliente.

Figura 5 - Include



Fonte: o autor (2021)

Extend: o relacionamento *extend* é utilizado para representar um comportamento opcional. Ex.: Considerando um Caso de Uso A e outro B. Levando em consideração que o caso de Uso A é um caso de uso Base e o B é um caso de uso estendido, o caso de uso B só será executado quando a condição de extensão for verdadeira. Ou seja, nem sempre o caso de uso B será executado. Na figura abaixo, ao identificar o cliente, o usuário poderá ou não adquirir o seguro (contra roubo do cartão). Por ser opcional, utilizamos o *extend*.

Figura 6 - Extend

Fonte: o autor (2021)

O diagrama de Caso de Uso, quando focado para a área de desenvolvimento deve possuir alguns padrões que serão utilizados pelos programadores. São eles:

- Ao definir o nome de um caso de uso, não utilize espaços no nome;
- Se o nome for composto, a primeira letra de cada palavra deve ser maiúscula, com exceção da primeira palavra;
- Os nomes não podem possuir acento;
- O nome do caso de uso deve ter relação com as funcionalidades que ele desempenha.

Por exemplo, para o caso de uso manter histórico, o nome correto seria manterHistorico..

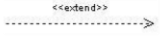
Videoaula 2

Utilize o QR Code para assistir!

Neste momento, vamos assistir ao vídeo sobre o diagrama de caso de uso e suas associações.

Segue abaixo um resumo dos conceitos e notações utilizadas no diagrama de caso de uso.

Quadro 1 - Conceitos e notações utilizadas no diagrama de caso de uso

Nome			Definição	Notação
Caso de Uso			São utilizados para capturar os requisitos do sistema, ou seja, referem-se aos serviços, tarefas ou funcionalidades.	
Ator			Representa os papéis desempenhados pelos diversos usuários que poderão utilizar, de alguma maneira, os serviços e funções do sistema.	
Relacionamento			Interação entre o ator e o Caso de Uso.	Não há.
Relacionamento	Associação		O relacionamento de associação representa a informação de quais atores estão associados a que casos de uso, podendo ser unidirecional ou bidirecional.	Não há.
Relacionamento	Associação	Associação unidirecional	Indica o sentido em que as informações trafegam.	
Relacionamento	Associação	Associação bidirecional	As informações são transmitidas nas duas direções.	
Relacionamento	Dependência		Este relacionamento, como o próprio nome diz, identifica certo grau de dependência de um Caso de Uso em relação à outra. O relacionamento de dependência é apresentado por uma linha tracejada entre Caso de Uso.	Não há.
Relacionamento	Dependência	*Include*	Costuma ser utilizada quando existe um cenário comum a mais de um Caso de Uso. Indica a obrigatoriedade da execução do Caso de Uso incluído.	
Relacionamento	Dependência	*Extend*	São utilizados para descrever cenários opcionais de um Caso de Uso. Representa eventos que não ocorrem sempre.	
	Generalização		Um relacionamento de um Caso de Uso-filho para Caso de Uso–pai.	
Estereótipo			Permite a extensibilidade aos componentes ou associação da UML. Como exemplo, temos o *include* e o *extend*.	«estereótipo»
Requisitos			São definidos como condição ou capacidade com a qual o sistema deve estar de acordo, podendo ser funcional ou não-funcional.	Não há.
Requisito	Requisito Funcional		É definido como uma condição ou capacidade com a qual o sistema deve estar de acordo.	Não há.
Requisito	Requisito não-funcional		Descreve os atributos do sistema ou atributos do ambiente de sistema.	Não há.
Especificação de Caso de Uso			Descreve com linguagem simples as informações referentes ao Caso de Uso, quais atores interagem, os passos a serem executados, entre outros.	Verificar o Anexo.

Fonte: o autor (2021)

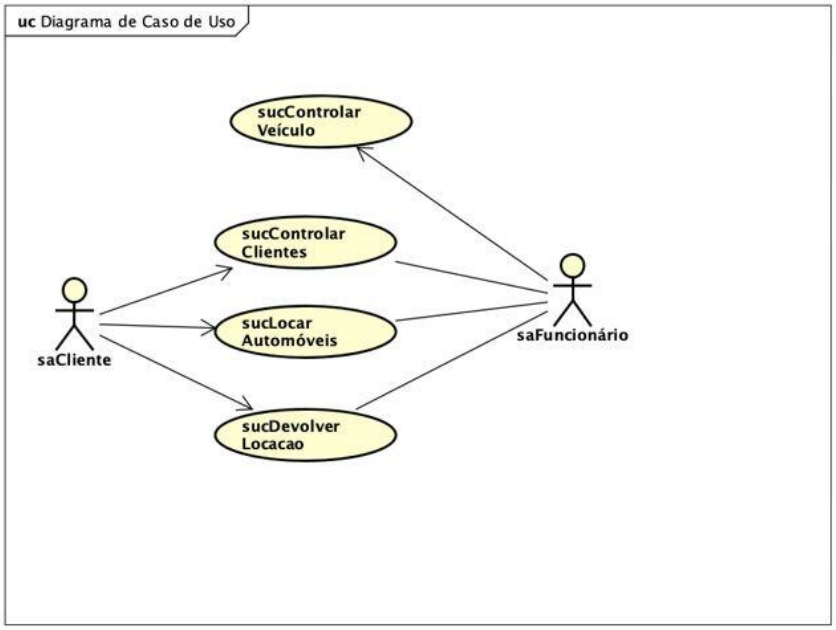
Exemplo um estudo de caso de Locação de Veículos

Desenvolva o diagrama de caso de uso para um sistema de controle de aluguel de veículos, levando em consideração os seguintes requisitos: (GUEDES, 2009).

- A empresa tem uma grande frota de carros de passeio, sendo que esses carros apresentam diferentes marcas e modelos. Eventualmente um carro pode ser retirado da frota devido a algum acidente grave ou simplesmente por ter sido considerado velho demais para o padrão da empresa e tenha sido vendido. Da mesma forma, a empresa eventualmente renova a frota, sendo necessário, portanto, estar sempre mantendo o cadastro de veículo da empresa;
- Os clientes dirigem-se à empresa e solicitam o aluguel de carros. No entanto, primeiramente é necessário cadastrá-los, caso ainda não possuam cadastro ou seus dados tenham sido alterados;
- Depois de ter se identificado/cadastrado, o cliente escolherá o carro que deseja alugar (o valor da locação varia de acordo com o ano, marca e modelo do automóvel). Durante o processo de locação, o cliente deve informar por quanto tempo utilizará o carro, para qual finalidade e por onde deseja trafegar, visto que essas informações também influenciam o preço da locação. Antes de liberar o veículo, a empresa exige que o cliente forneça um valor superior ao estabelecido na análise da locação, a título de caução. Caso o cliente não utilize todo o valor da caução até o momento da devolução do veículo, o valor restante lhe será devolvido;
- Quando o cliente devolve o carro deve-se definir o automóvel como devolvido, registar a data e hora da devolução e a quilometragem em que se encontra, bem como verificar se o automóvel se encontra nas mesmas condições em que foi alugado. Caso o cliente tenha ocupado o carro por mais tempo que o combinado, deve pagar o aluguel referente ao tempo excedente. Da mesma maneira, o cliente deverá pagar por qualquer dano sofrido pelo veículo quando este encontrava-se locado. Por outro lado, o cliente pode ser ressarcido de parte do valor que pagou caso o custo do tempo em que esteve em posse do veículo seja inferior ao valor previamente fornecido.

Observem que estamos utilizando algumas nomenclaturas antes dos nomes dos atores e casos de usos. Isso não é uma regra da UML, mas sim um padrão que você pode adotar na sua empresa para diferenciar atores e casos de usos de sistemas em relação a atores e casos de usos de negócios. Nesta aula foi abordado somente os atores e casos de uso de sistemas. O "sa" significa *system actor* e o "suc" *system use case.* O Ator saCliente tem uma associação unidirecional para os casos de uso sucControlarClientes, sucLocarAutomóveis e sucDevolverLocacao, isso significa que o Ator saCliente está inicializando a comunicação com os casos de uso. Já a comunicação destes três casos de uso com o Ator saFuncionário é bidirecional, ou seja, tanto o ator como o caso de uso podem inicializar a comunicação.

Figura 7 – Diagrama de caso de Uso



Fonte: adaptado Guedes (2009)

Videoaula 3

Utilize o QR Code para assistir!

Vamos assistir ao vídeo sobre como modelar um diagrama de caso de uso na prática.

Leitura Obrigatória

Leia o capítulo 1, 2, e 3 do livro do Medeiros. Desenvolvendo *Software* com UML 2.0.

Acesse o livro no *link* abaixo:

Disponível em: https://plataforma.bvirtual.com.br/Leitor/Publicacao/2921/pdf/0. Acesso em: 5 set. 2021.

Acesse o RUP no *link* abaixo:

Disponível em: http://walderson.com/IBM/RUP7/LargeProjects/#core.base_rup/customcategories/inception_D506BCB4.html. Acesso em: 5 set. 2021.

Videoaulas

Prezado(a) aluno(a)!
Você também poderá encontrar todas as videoaulas, clicando em "**Módulos**" no "**Menu Lateral**" e acessar a página de vídeos.

Encerramento

Nesta unidade foram abordados os principais conceitos de orientação a objetos e UML.

Iniciamos os conceitos e os elementos sobre o Diagrama de Caso de Uso, bem como um exemplo do estudo de caso de uma Locadora de Veículos.

Nos vemos na próxima unidade!

Bom trabalho!

Referências

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML – Guia do Usuário**. 2. ed. Rio de Janeiro: Editora Campus, 2006.

FOWLER, M. **UML Essencial**. 3. ed. Porto Alegre: Bookman, 2007.

GUEDES, G. T. A. **UML 2.0: uma abordagem prática.** São Paulo: Novatec, 2009.

IBM. **RUP – Rational Unified Process (Software**) Versão 7.0. USA: IBM Rational, 2006.

LARMAN, C. **Utilizando UML e Padrões**. 3. ed. Porto Alegre: Bookman, 2007.

MEDEIROS, E. **Desenvolvendo Software com UML 2.0**. São Paulo: Pearson, 2004.

OMG. OBJECT MANAGEMENT GROUP. UML 2.0. Disponível em: www.omg.org. 2020. Acesso em: 5 set. 2021.

PAGE-JONES, M. **Fundamentos do Desenho Orientado a objetos com UML**. São Paulo: Makron books, 2001.

PENDER, T. **UML 2.0 - A Bíblia**. Rio de Janeiro: Editora Campus, 2004.

TACLA, C. A. **Análise e projeto OO e UML 2.0. Departamento Acadêmico de Informática**. Universidade Tecnológica Federal do Paraná, 2010.

TANAKA, S. S**. O poder da tecnologia de workflow e dos mapas conceituais no processo de ensino e aprendizagem da UML**. Dissertação de Mestrado, UEL. Londrina, 2011.

VIDEIRA, C. A. E. **UML, metodologias e ferramentas case**. 2. ed. Lisboa: Centro Atlântico, 2008. 2 v.


UniFil
EDUCAÇÃO A DISTÂNCIA
UNIFIL.BR