**ANÁLISE ORIENTADA A OBJETO**

**RESUMO**

**CONCEITOS**

* **Objeto**: instância de uma classe; classes se relacionam entre si através relacionamentos de herança;
* **Programação Orientada a Objeto**: interação entre diversos objetos para gerar um programa;
* **Análise Orientada a Objeto**: método que examina os requisitos pela perspectiva das classes e objetos encontrados dentro do problema a ser resolvido/do que foi requisitado;
  + Seus resultados servem como modelos para se criar um projeto Orientado a Objeto e, assim, implementar um sistema através da Programação Orientada a Objeto.
* **Herança**: permite que uma classe herde os atributos e ações/métodos de outra
  + Utilizado quando mais de uma classe possuem comportamentos semelhantes ou que dependem de atributos semelhantes.
* **Polimorfismo**: capacidade de um mesmo método ser implementado de maneira diferente, mas com o mesmo objetivo, de acordo com o objeto;
* **Abstração**: foca no comportamento observável do objeto, permite definir o estado (ações) e comportamentos(métodos) para objetos que herdam a classe abstrata; classes abstratas não podem ser instanciadas diretamente;

**DIAGRAMA DE CLASSES**

* **Objetivo**: descrever os tipos de objetos e os relacionamentos presentes no sistema;
* **Componentes**:
  + **Classes**: representam os objetos existentes no modelo;
  + **Classes abstratas**: possuem o nome em itálico;
  + **Característica**: propriedades e operações de uma classe;
  + **Atributos**: descreve uma propriedade da classe;
    - **Visibilidade**: (+) public / (-) private / (#) protected;
    - **Tipo**: tipo do atributo;
    - **Multiplicidade**: (1), (0..1), (\*);
    - **Valor por omissão**: valor que o atributo recebe quando é criado; mantido até ser atualizado;
    - **Lista de propriedades**: adiciona propriedades adicionais ao atributo;
    - **Exemplo**: - nome: String [1] = “Sem nome” { readonly}
  + **Associações**: representa uma propriedade quando ela é do tipo de outra classe;
    - **Representação**: linha cheia entre duas classes, direcionada da classe de origem para a classe de destino; o nome da propriedade fica no destino final da associação, junto com a multiplicidade.
    - **Agregação**: tipo de associação onde um pode existir independente do outro;
    - **Composição**: tipo de associação onde um não pode existir independentemente do outro.
  + **Operações**: ações que uma classe pode realizar;
    - Operações de get e set, normalmente, não são mostradas;
    - **Visibilidade**: (+) public / (-) private / (#) protected
    - **Nome da operação**;
    - **Lista de parâmetros**: parâmetros que a operação precisa;
    - **Tipo**: tipo do valor retornado ou void;
    - **Exemplo**: +escreverNome (nome: String): void
  + **Generalização**: quando uma classe se divide em subclasses;
    - **Representação**: uma seta direcionada das subclasses para a superclasse, o que representa a herança;
  + **Interface:** classe que não possui implementação, todas as suas propriedades e métodos são abstratos; obriga as classes que a herdam a ter certos comportamentos;
    - **Representação**: seta pontilhada;

**GRASP**

* Conjunto de padrões que auxiliam a projetar sistemas orientados a objetos;
* Ajudam a identificar as responsabilidades de cada classe;
* **Responsabilidades**: obrigações de um objeto em relação ao seu comportamento; dividido em dois tipos;
  + **Fazer**: fazer algo por ele mesmo / iniciar ações em outros objetos / controlar e coordenar atividades em outros objetos; ex: uma venda é responsável por criar os ItensDeVenda.
  + **Conhecer**: conhecer sobre dados privados encapsulados / conhecer objetos relacionados / conhecer coisar em que esse objeto pode derivar ou calcular; ex: uma venda é responsável por conhecer o seu TotalDeItens.
  + Responsabilidades surgem através da implementação de métodos que agem sozinho ou em conjunto com outros (um conjunto de métodos que em conjunto fazem uma ação geram uma responsabilidade).
* **GRASP**: **G**eneral **R**esponsability **A**ssignment **S**ofwate **P**atterns  
  Padrões são pares de soluções que sistematizam bons conselhos e princípios utilizados na atribuição de responsabilidade
  + **Information Expert ou Expert**:
    - **Solução**: atribua a responsabilidade à classe que possui a informação necessária para suprir a necessidade;
    - **Contraindicação** há situações em que utilizar este padrão não é desejado, como por exemplo quando temos que salvar algum dado em um banco de dados;
    - **Benefícios**: o encapsulamento das informações é mantido, já que o objeto usa suas próprias informações para cumprir tarefas.
  + **Creator**:
    - **Solução**: a classe B deve ser responsável por criar uma instancia da classe A nos seguintes casos:
      * B agrega objetos do tipo A;
      * B contem objetos do tipo A;
      * B grava objetos do tipo A;
      * B usa de forma próxima e frequente objetos do tipo A;
      * B possui dados de inicialização que serão passados para o construtor de A no momento de sua instanciação.
    - **Contraindicação**: se a instanciação de classes exigir uma complexidade maior, como depender de propriedades externas, não é desejado utilizar o padrão Creator;
    - **Benefícios**: baixa dependência de manutenção e grande oportunidades de reuso são suportadas pelo baixo acoplamento;
  + **Low Coupling**:
    - **Solução**: atribuir responsabilidades de modo que o acoplamento permaneça baixo;
    - **Contraindicação**: alto acomplamento;
    - **Beneficios**: aumenta a chance de elementos não serem afetados por mudanças em outros elementos; simplifica o entendimento de partes isoladas, é conveniente quando pensa-se em reuso.
  + **High Cohesion**:
    - **Solução**: atribuir responsabilidades deixando a coesão alta;
    - **Benefícios**: facilidade de compreensão do projeto, melhora a manutenção, apoia o baixo acoplamento e aumenta o reuso;