PONTIFÍCIA UNIVERSIDADE CATÓLICA

ENGENHARIA DE SOFTWARE

RAFAEL TROQUETE

PROGRAMAÇÃO ORIENTADA A OBJETOS: UMA VISÃO CONCEITUAL DOS ELEMENTOS DE MODELAGEM

SÃO PAULO

2019

PONTIFÍCIA UNIVERSIDADE CATÓLICA

ENGENHARIA DE SOFTWARE

RAFAEL TROQUETE

PROGRAMAÇÃO ORIENTADA A OBJETOS: UMA VISÃO CONCEITUAL DOS ELEMENTOS DE MODELAGEM

Monografia apresentada ao Curso de Engenharia de Software da Pontifícia Universidade Católica de São Paulo como um dos pré-requisitos para obtenção do título de Especialista em Engenharia de Software, sob orientação do Prof. Dr. Daniel Couto Gatti.

SÃO PAULO

2019

AGRADECIMENTOS

“Cada um que passa em nossa vida, passa sozinho, pois cada pessoa é única e nenhuma substitui outra. Cada um que passa em nossa vida, passa sozinho, mas não vai só nem nos deixa sós. Leva um pouco de nós mesmos, deixa um pouco de si mesmo. Há os que levam muito, mas há os que não levam nada. Essa é a maior responsabilidade de nossa vida, e a prova de que duas almas não se encontram ao acaso.”

Antoine de Saint-Exupéry

Agradeço a todos os colegas de profissão e amigos que me incentivaram na realização deste trabalho. A todos os professores que passaram pela minha vida e serviram como inspiração para meus constantes estudos.

Ao meu orientador, Prof. Dr. Daniel Couto Gatti por contribuir com suas preocupações sobre o objeto de estudo e mostrando sempre um grande interesse na discussão sobre o assunto.

Minha querida esposa, Paloma. Muito obrigado por sua compreensão e paciência com o marido.

Obrigado pai, mãe e irmão por contribuírem na formação da pessoa que sou atualmente.

RESUMO

O objetivo dessa pesquisa é apresentar uma visão mais conceitual sobre o design e modelagem de objetos, uma vez que os materiais encontrados atualmente dão um foco maior para uma tecnologia e menos importância para o entendimento do paradigma. Essa pesquisa foi idealizada pois a orientação a objetos tem sido amplamente utilizada pela indústria de software e cada vez mais a busca por profissionais com conhecimentos nessa área tem sido mais valorizada. No entanto, ainda é possível perceber uma grande defasagem de conhecimento sobre questões elementares do paradigma. Essa dificuldade pode ser causada pois não há uma preocupação em conhecer conceitualmente o que é cada elemento e no que ele contribui para a tecnologia de objetos. Considerando isto, este trabalho propõe uma sequência de estudo que leva ao leitor a ter uma melhor definição sobre os elementos da modelagem de objetos e visa fornecer insumos para elaborar modelos e designs ainda mais consistentes. A sequência proposta apresenta um conjunto de tópicos que passam desde a história, elencando a motivação para a estruturação do objeto de estudo até a evolução e concepção de técnicas avançadas.

Palavras chave: orientação a objetos, paradigmas de programação, modelagem

LISTA DE ILUSTRAÇÕES

Figura 1 - Representação de uma classe pessoa 17
Figura 2 - Representação de composição entre pessoa e família 18
Figura 3 - Representação de agregação entre casa e família 19
Figura 4 - Representação de implementação de um imposto 19
Figura 5 - Atributo alíquota da classe ICMS 20
Figura 6 - Operações de uma calculadora simples 21
Figura 7 - Diagrama de objetos de um modelo de empresa 23
Figura 8 - Abstração de um livro como modelo de leitura 25
Figura 9 - Abstração de um livro como objeto de venda 25
Figura 10 - Modelo utilizando herança 28
Figura 11 - Polimorfismo para o cálculo de imposto 29
Figura 12 - Exemplo de classe sem coesão 34
Figura 13 - Modelo de venda com classes coesas 34
Figura 14 - Cálculo de ISS em uma venda 36
Figura 15 - Classe venda aberta a extensão de novos impostos 37
Figura 16 - Criação de interfaces especializadas evoluindo o modelo de aves da seção de herança aplicando o princípio de segregação de interfaces 40
Figura 17 - Acoplamento forte entre duas classes 42
Figura 18 - Aplicando inversão de dependência para a conexão 43

LISTA DE SIGLAS

OO	Orientação a objetos
POO	Programação orientada a objetos
UML	*Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO 9
1.1 MOTIVAÇÃO 9
1.2 JUSTIFICATIVA 10
1.3 OBJETIVO 10
1.4 CONTRIBUIÇÕES 11
1.5 MÉTODO DE TRABALHO 11
1.6 ORGANIZAÇÃO DO TEXTO 12
2 PROGRAMAÇÃO ORIENTADA A OBJETOS 13
2.1 HISTÓRIA 13
2.2 SIMULA 13
2.3 SMALLTALK 14
2.4 DEFINIÇÃO 15
3 ESTRUTURA 17
3.1 CLASSE 17
3.1.1 ASSOCIAÇÃO 18
3.2 INTERFACE 19
3.3 ATRIBUTO 20
3.4 OPERAÇÃO 21
3.4.1 MÉTODO 22
3.5 OBJETO 22
4 FUNDAMENTOS 24
4.1.1 ABSTRAÇÃO 24
4.1.2 ENCAPSULAMENTO 25
4.1.3 HERANÇA 27
4.1.3.1 POLIMORFISMO 29
4.2 TÉCNICAS 30
4.2.1 REUSO 30
4.2.2 BAIXO ACOPLAMENTO 31
4.2.3 ALTA COESÃO 32
4.3 SOLID 33
4.3.1 PRINCÍPIO DE RESPONSABILIDADE ÚNICA (SRP) 33
4.3.2 PRINCÍPIO DO ABERTO FECHADO (OCP) 35
4.3.3 PRINCÍPIO DE SUBSTITUIÇÃO DE LISKOV (LSP) 37
4.3.4 PRINCÍPIO DE SEGREGRAÇÃO DE INTERFACE (ISP) 39
4.3.5 PRINCÍPIO DE INVERSÃO DE DEPENDÊNCIA (DIP) 41
5 PADRÕES DE PROJETO 43
5.1 CRIACIONAIS 44
5.2 ESTRUTURAIS 45

5.3 COMPORTAMENTAIS 47
6 CONSIDERAÇÕES FINAIS 49
7 BIBLIOGRAFIA 51

1 INTRODUÇÃO

Neste capítulo são apresentados os elementos principais que definem a pesquisa: motivação, justificativa, objetivo, contribuição, método de trabalho e organização do texto.

1.1 MOTIVAÇÃO

No momento atual da indústria de software há uma crescente preocupação em criar sistemas que possam ser cada vez mais adaptáveis as constantes mudanças que ocorrem no mercado tornando um negócio cada vez mais competitivo.

Diferentemente de outros paradigmas de programação, a orientação a objetos ajuda na construção de sistemas mais flexíveis e que tendem a ser mais suscetíveis a mudanças dado a sua capacidade de adaptação dos modelos.

Os estudos sobre orientação a objetos têm ganhado força dado sua capacidade de abstrair elementos do mundo real para o meio computacional. Esta presente característica de abstração de modelos permite que um sistema seja construído com foco em problemas reais. Apesar de ser datada da década de 60, sua popularização aconteceu por volta da década de 90 com o surgimento de linguagens mais sofisticadas (CARVALHO, 2018).

Até para pessoas com bastante experiência na área de programação, modelagem e arquitetura de sistemas, ainda é possível encontrar bastante dificuldade na compreensão dos fundamentos e da estrutura do paradigma de objetos, e um profissional com conhecimentos profundos sobre essa tecnologia passou a ser cada vez mais requisitado no mercado (JANKE, BRUNE e WAGNER, 2015). Para os menos experientes ainda há a dificuldade de compreender sua aplicação tendo em vista que o primeiro contato de alguém com programação de computadores é em um cenário com um paradigma estruturado (CARVALHO, 2018).

1.2 JUSTIFICATIVA

No momento atual da indústria de software, em um modo geral, já não é possível mais imaginar o desenvolvimento de sistemas complexos sem o uso do paradigma de objetos. Apesar de não ser um modelo novo, somente a partir dos anos 90 que essa técnica ganhou mais notoriedade e começou a ser objeto de estudo ao redor do mundo.

A orientação a objetos deu seus primeiros passos quando começaram a levar para o universo da programação o conceito de simulação de eventos, assim surgiu o SIMULA e o SMALLTALK, que deram o pontapé inicial para a criação das primeiras linguagens de alto nível com suporte a esse paradigma, dando vida ao que é conhecido atualmente como programação orientada a objetos.

Apesar de ser objeto de estudo desde a década de 60, muitos dos conceitos que ajudaram a tornar essa técnica indispensável nos modelos de software atuais ainda carecem de conhecimento por grande parte dos profissionais de programação (CARVALHO, 2018). É fundamental para uma boa modelagem e principalmente para obter-se um bom design que se tenha um profundo conhecimento dos elementos que constituem o paradigma.

1.3 OBJETIVO

Este trabalho tem como objetivo geral abordar as características do paradigma orientado a objetos, buscando principalmente uma definição mais clara dos elementos de modelagem que o compõe.

Para alcançar o objetivo de conceituar os elementos da orientação a objetos, é necessário começar da história e as motivações que deram origem ao que conhecemos hoje como paradigma de objetos. Após essa breve volta ao passado, será apresentado como os conceitos evoluíram e foram sendo aperfeiçoados ao longo do tempo.

1.4 CONTRIBUIÇÕES

Com este trabalho se busca refinar os conhecimentos sobre os elementos de modelagem do paradigma de objetos, bem como apresentar de maneira objetiva tanto para os mais experientes quanto para os menos experientes o conceito de cada parte que o compõem. Dado esse cenário espera-se ao final da leitura uma melhor compreensão para uma aplicação mais consistente e segura do paradigma de objetos.

1.5 MÉTODO DE TRABALHO

Para atingir o objetivo proposto, foram executadas as seguintes atividades:

1 – Pesquisa bibliográfica: leitura de publicações dos principais autores que contribuíram para a concepção e entendimentos do paradigma;

2 – Elaboração/organização da apresentação conceitual: foram conhecidos os elementos que compõem o paradigma para apresentar desde a história até a elaboração e evolução dos conceitos de objetos;

3 - Elaboração da estratégia: dado o estudo e compreensão do material bibliográfico foram observados os elementos estruturais que compõem o paradigma e que necessitam de uma explicação objetivo sobre eles;

4 – Apresentação fundamental: foram observados que alguns conceitos se tornaram fundamentais para a utilização da orientação a objetos, como abstração e encapsulamento, e foi apresentado uma definição sobre esses elementos;

5 – Apresentação evolutiva: foram verificados como se deu a evolução da orientação a objetos e elencadas as principais técnicas;

6 – Redação da monografia: consiste na digitação e normalização do trabalho que será executada durante toda a extensão da pesquisa.

1.6 ORGANIZAÇÃO DO TEXTO

No capítulo 2, Programação Orientada a Objetos, será apresentado uma breve história de como esse conceito surgiu e as motivações que levaram a sua concepção. Também é apresentado uma definição sobre o que consiste esse paradigma.

No capítulo 3, Estrutura, será apresentado os modelos que consistem na estrutura do paradigma, ou seja, que dão forma aos elementos que podem ser aplicados para a modelagem de objetos.

No capítulo 4, Fundamentos, será abordado o surgimento de pilares e técnicas que se tornaram indispensáveis no paradigma de objetos tendo em vista todo o estudo sobre os elementos que o compõem.

No capítulo 5, Padrões de Projeto, será apresentado uma técnica de padrões que são estudos que usam todo o potencial do paradigma para a construção de soluções reutilizáveis.

No capítulo 6, "Considerações Finais", é apresentado um breve resumos das motivações e a conclusão sobre a importância de um entendimento conceitual dos elementos que constituem o paradigma de objetos.

2 PROGRAMAÇÃO ORIENTADA A OBJETOS

2.1 HISTÓRIA

Muitos dão como o início da programação orientada a objetos por volta de 1970 com a chegada do *Smalltalk.* Não se pode negar que essa linguagem tenha sido responsável por popularizar o paradigma, mas é possível chegar um pouco mais fundo e entender como ela teve sua origem e quais foram as bases para sua concepção.

2.2 SIMULA

Entre a década de 50 e 60 os estudos sobre simulação de eventos na área computacional começou a ganhar mais força, e em 1957 Kristen Nygaard iniciou seus trabalhos para escrever uma linguagem onde fosse possível trabalhar com a abstração de eventos reais. Já em 1962, com a parceria de Ole-Johan Dahl, foi apresentada à comunidade o SIMULA I. Essa linguagem nasceu com o objetivo de trazer para a computação a simulação de eventos discretos e que fosse uma linguagem com foco no problema.

Nessa primeira versão já foram constituídos alguns dos conceitos que serviram como base para o que conhecemos hoje como classes e objetos. Com o SIMULA I foi possível criar módulos que englobam uma definição de estrutura e comportamento.

No decorrer dos anos grandes incentivos foram feitos em cima do SIMULA I, Nygaard e Dhal passaram por vários países divulgando esse trabalho e lecionando sobre essa nova tecnologia. Já no ano de 1967, as primeiras definições formais do SIMULA começam a surgir, o que foi um grande marco para o que é conhecido hoje de programação de objetos.

Essa nova linguagem foi chamada de SIMULA 67 e incorporou os conceitos de classes, objetos e herança. Com isso temos a primeira versão de uma linguagem construída para que um novo paradigma de programação pudesse criar forma. No ano

seguinte, durante uma reunião do *SIMULA Standards Group,* todas as definições do SIMULA 67 foram padronizadas e registradas (Simula Standards, 1968).

O SIMULA foi o grande responsável pela concepção da programação orientada a objetos, uma citação que demonstra o quão importante essa tecnologia foi para isso está em um artigo publicado como *A history of discrete events simulation programming languages* (NANCE, 1993), onde foi muito bem elucidado por Thiago Carvalho em seu livro, Orientação a Objetos: Aprenda seus conceitos e suas aplicabilidades de forma efetiva (CARVALHO, 2018):

“As contribuições técnicas do SIMULA I são impressionantes, quase incríveis. Dahl e Nygaard, em sua tentativa de criar uma linguagem onde os objetos do mundo real seriam de forma precisa e naturalmente descritos, apresentou avanços conceituais que se tornariam realidade somente quase duas décadas mais tarde: tipo abstrato de dados, o conceito de classe, herança, o conceito de *corotina* (método), [...] a criação, exclusão e operações de manipulação em objetos são apenas um exemplo.”

2.3 SMALLTALK

Conforme observado, o desenvolvimento do SIMULA foi o pontapé inicial para a construção da programação orientada a objetos, mas a concepção do termo e sua popularização aconteceu devido aos esforços de Alan Kay e sua equipe na Xerox com o desenvolvimento do *Smalltalk.*

Durante a década de 70, e inspirado pela linguagem de programação LOGO (desenvolvida inicialmente para ensinar crianças sobre computadores), Alan Kay inicia uma pesquisa junto com sua equipe na Xerox para o desenvolvimento de uma tecnologia que fosse focada na passagem de mensagens. Parte dessa inspiração surgiu da ideia de pensar em objetos como se fossem células biológicas, e/ou computadores em uma rede, que se comunicam através dessas mensagens.

A primeira versão dessa tecnologia ficou conhecida como Smalltalk-71 que abstraiu em sua essência um modelo para a troca de mensagens entre objetos. O que inicialmente diferencia o Smalltalk do SIMULA é que literalmente tudo no Smalltalk é um objeto, incluindo blocos e números, não havendo somente abstrações para valores literais, afinal, o Smalltalk já surge com a premissa de trabalhar inteiramente com objetos.

O termo “orientado a objetos” surgiu no mesmo período, com esse nome criado por Alan Kay, dando início a um novo paradigma para a indústria de software, e com as novas versões do Smalltalk, principalmente o Smalltalk-80, esse conceito começou a se popularizar.

Desde então já temos os conceitos que formam a base para a POO, que de acordo com Alan Kay se sustenta em alguns pilares como a troca de mensagens e encapsulamento.

"POO para mim significa apenas mensagens, retenção local e proteção e ocultação de processos de estado e extrema ligação tardia de todas as coisas. Isso pode ser feito em Smalltalk e LISP. Possivelmente existe outros sistemas nos quais isso é possível, mas eu não os conheço." (KAY)

2.4 DEFINIÇÃO

Muito material e muitos estudos sobre orientação a objetos surgiram após a concepção do paradigma nos anos 70, entre eles a concepção da UML, que abraçou os conceitos para trazer uma nova linguagem que fosse capaz de ajudar na modelagem de software, em especial, trazendo aspectos do paradigma de objetos para essa linguagem.

Como visto anteriormente, o conceito de OO surgiu da ideia da simulação de eventos e da troca de mensagens na criação de uma nova linguagem de programação

que fornecesse esses suportes, mais tarde outras denominações e fundamentos foram definidos como encapsulamento, herança e classes.

Não é possível somente através de fundamentos descrever uma definição formal sobre qualquer tema, porém, no caso de OO, podemos elencar o pensamento de alguns autores e assim traçar um paralelo para uma definição satisfatória sobre orientação a objetos.

Uma frase de Alan Kay ficou bastante famosa quando questionado sobre ao que ele pensou quando cunhou o termo:

"Quando eu inventei o termo orientado a objetos, eu posso te garantir que o C++ não era o que eu tinha em mente."

O C++ teve grande responsabilidade por popularizar o paradigma de objetos, e assim seguido pelos avanços adotados na linguagem Java, que por sua vez teve bastante influência no C++. Muitas vezes essas duas linguagens ainda são utilizadas como exemplo de suporte a OO, mas elas não abrangem totalmente os conceitos, pois dentro delas nem tudo são objetos, há valores literais que podem ser utilizados para compor o programa.

É importante ressaltar que a concepção do termo foi algo totalmente informal, surgiu de uma simples pergunta sobre o que Kay estaria fazendo naquele momento, e a resposta simplesmente foi: programação orientada a objetos.

Pode-se após a análise da história e de como foi a criação do termo, tratar a orientação a objetos como um paradigma de programação, modelagem e análise de software (CARVALHO, 2018). Esse paradigma está intrinsicamente ligado com as ideias fundamentais de sua concepção, que são a troca de mensagens e a interação entre as unidades de software, sendo capaz de abstrair para a área da computação elementos reais que serão construídos com o objetivo de simular comportamentos para um determinado fim.

3 ESTRUTURA

Na Orientação a Objetos existem alguns modelos que servem de base para qualquer tipo de desenvolvimento utilizando esse paradigma, pode-se dizer que eles são modelos estruturais.

Esses modelos têm como objetivo criar toda a estrutura que será responsável por abstrair os conceitos do mundo real que serão levados para o meio computacional.

3.1 CLASSE

Tanto na modelagem quanto na programação orientada a objetos é sempre pensando em como criar uma estrutura que seja mais adequada ao problema que será resolvido, e para isso existe o conceito de classes, que abstraem as características e comportamentos de um elemento do mundo real para um modelo computacional. Em cenários anteriores a POO era bastante comum o conceito de módulos, que são estruturas bastante semelhantes ao que é conhecido atualmente como classes e que são utilizadas também para representar um modelo.

Uma classe tem por objetivo descrever o comportamento de suas instâncias (objetos) através de atributos e operações, e essas definições tendem a ser o mais próximas possíveis de um objeto do mundo real, seja ela uma entidade física (ave, livro) ou conceitual (imposto, empréstimo).

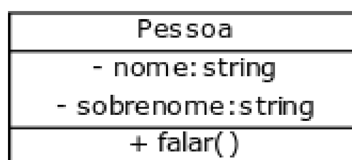

Figura 1 - Representação de uma classe pessoa

Uma dúvida bastante frequente, é do porquê deve-se ter uma preocupação com classes se o foco do paradigma são os objetos, e para entender esse cenário a noção de abstração acaba sendo o principal fator, pois ao agrupar os objetos por uma

noção de classes, está sendo abstraído um problema, ou seja, abstraindo os conceitos daquele elemento.

As classes além de serem compostas por atributos e operações, também podem estabelecer um tipo de relacionamento entre elas. Um relacionamento representa as dependências e o nível de acoplamento que eles possuem entre si.

3.1.1 ASSOCIAÇÃO

Uma associação, ou relacionamento, permite que os objetos criados a partir de uma classe detenham vínculos entre outros objetos, esses vínculos podem ser descritos como um grupo com uma estrutura e semânticas comuns.

Para representar uma associação podemos dizer que uma pessoa pertence a uma família, e essas associações ligam os objetos entre as mesmas classes e ajudam a modelar uma aplicação de uma maneira ainda mais fiel com os modelos obtidos de um contexto do mundo real.

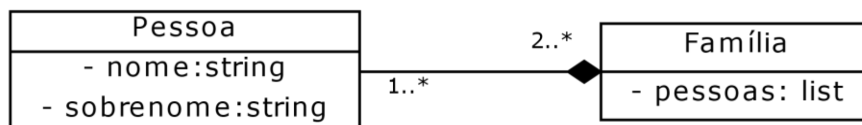

Figura 2 - Representação de composição entre pessoa e família

Vale destacar dois tipos que representam uma associação, a composição e agregação. Cada uma delas remete a um grau diferente de relacionamento que um objeto mantém com outro. A composição dá o entendimento que um objeto não pode existir sem outro, por exemplo, uma família não existe sem pessoas. A agregação é um pouco mais flexível, que descreve que um objeto pode existir sem outro que o componha, uma casa pode existir sem moradores.

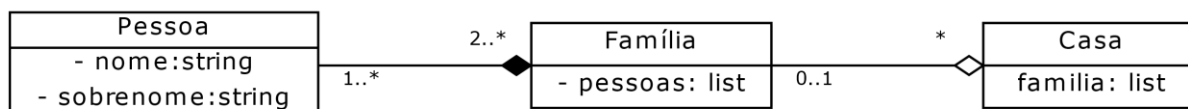

Figura 3 - Representação de agregação entre casa e família

3.2 INTERFACE

Durante a modelagem de um sistema utilizando o paradigma de objetos, pode haver a necessidade de criar um conceito mais abstrato e que não contenha nenhum detalhe de implementação, funcionando apenas para representar um tipo genérico dentro do sistema.

Para esse fim existe o conceito de interface, que designa um contrato de implementação de uma classe. Para a criação de uma interface pouco importa como o objetivo será atingido, o que é realmente esperado é que a implementação respeite o que está presente nesse contrato.

Seja os argumentos de uma operação, atributos herdados ou o retorno de tal operação, as classes que implementam uma interface ficam condicionadas a obedecer ao que descreve um contrato.

Um exemplo bastante prático pode ser de uma classe para calcular algum tipo de imposto, pouco importa que cálculo será utilizado, mas o que realmente é esperado é que seja passado um valor para a operação e seja devolvido um novo valor com o imposto aplicado.

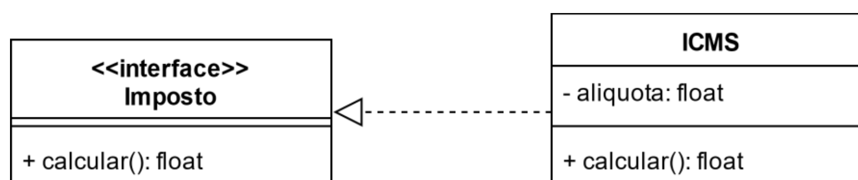

Figura 4 - Representação de implementação de um imposto

3.3 ATRIBUTO

A definição formal de atributo é descrita como algo próprio, peculiar ou característico de algo, na visão de modelagem de objetos, um atributo representa um valor que é mantido por um objeto.

Para representar um modelo por meio de uma classe, é possível determinar as características que serão utilizadas nessa representação, tais características são chamadas de atributos. Os atributos são elementos fundamentais para determinar os estados que um objeto pode assumir, já que são responsáveis por assegurar um valor para o objeto.

É importante ressaltar que entender o problema que será resolvido é fundamental para criar uma classe que realmente atenda a um objetivo. Ao modelar uma classe que representa uma cadeira, deve-se levar em conta se esse objeto será tratado como sendo um produto de venda ou para cumprir sua função principal para acomodar uma pessoa.

Para cada um desses casos citados, existirão características totalmente distintas para representar um modelo que melhor atende ao problema a ser resolvido. Por fim, um objeto deve ser capaz de mudar de estado, e os atributos são os elementos que tornam esse processo possível.

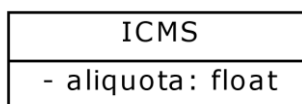

Figura 5 - Atributo alíquota da classe ICMS

3.4 OPERAÇÃO

Após a definição de uma classe e seus atributos, há ainda a necessidade de manipular esses valores, e para esse fim a modelagem de objetos provê a criação de operações para que os objetos possam interagir e definir um comportamento.

Uma operação pode ser classificada como uma função, ação ou transformação que será realizada por um objeto, tendo um tempo finito para execução e que consiste em uma mudança de estado. A execução dessa tarefa majoritariamente deve durar até o fim da transação de estado ou até que um evento interrompa esse processo prematuramente.

Essa mudança de estado é esperada ao fim de uma tarefa realizada com sucesso. Uma operação pode ou não ser composta por valores de entrada, e podem ser também representadas da seguinte forma:

$$f(x) = x$$

Essa função exemplifica uma operação que dado um valor de entrada X devolve X. Dentro da modelagem de objetos o conceito é o mesmo, é possível criar operações que representam um comportamento do mundo real. Para criar uma operação em uma classe deve-se definir um nome que representará a ação que deve ocorrer, definir os valores de entrada e o tipo de dado que devolverá ao fim da operação.

Calculadora

+ somar(num1 : float, num2 : float): float
+ subtrair(num1 : float, num2 : float): float
+ multiplicar(num1 : float, num2 : float): float
+ dividir(num1 : float, num2 : float): float

Figura 6 - Operações de uma calculadora simples

É importante frisar que todos os objetos que são instâncias de uma determinada classe compartilharam as mesmas operações, ou seja, todos os objetos do mesmo tipo podem realizar as mesmas operações.

3.4.1 MÉTODO

Uma mesma operação pode ser aplicável em diferentes classes e o que definirá o comportamento de tal operação é em qual modelo ele está inserido, dessa forma um objeto conhece sua classe e consequentemente a correta implementação da operação em questão.

A operação calcular pode estar inserida em um contexto de calculadora como em uma classe que representa um imposto, apesar de serem uma operação com igual identificação, elas possuem métodos para executar determinada ação de maneiras completamente distintas.

Dessa forma é possível afirmar que um método é a implementação da operação de uma classe satisfazendo o propósito da qual está inserido.

3.5 OBJETO

Embora a maior parte do trabalho durante o uso do paradigma de objetos seja na modelagem de classes e seus atributos, operações e relacionamentos, o que realmente interessa são os objetos e as trocas de mensagens que serão realizadas.

A maior parte do trabalho está em modelar as classes de uma forma que elas consigam descrever claramente os objetos e prover os recursos necessários para que as mudanças de estado possam ocorrer, sendo assim, toda essa modelagem deve fazer algum sentido para o contexto ao qual está inserido.

"*Nós definimos um objeto como um conceito, abstração ou algo com limites e significados nítidos para o problema em questão. Objetos servem dois propósitos:*

Promovem entendimento do mundo real e provê uma base prática para uma implementação computacional. Decomposição de um problema em objetos depende do julgamento e da natureza do problema.“ (RUMBAUGH, BLAHA, *et al.*, 1991)

Todos os objetos possuem uma identidade e podem ser distinguidos, apesar de dois objetos poderem ter valores idênticos para seus atributos, ainda assim eles representam elementos distintos.

Um objeto tem a capacidade de manipular os elementos (atributos e operações) que fazem parte da declaração do seu modelo (classe). É importante deixar claro que uma classe representa a abstração dos conceitos enquanto um objeto representa a entidade concreta desse modelo que foi abstraído.

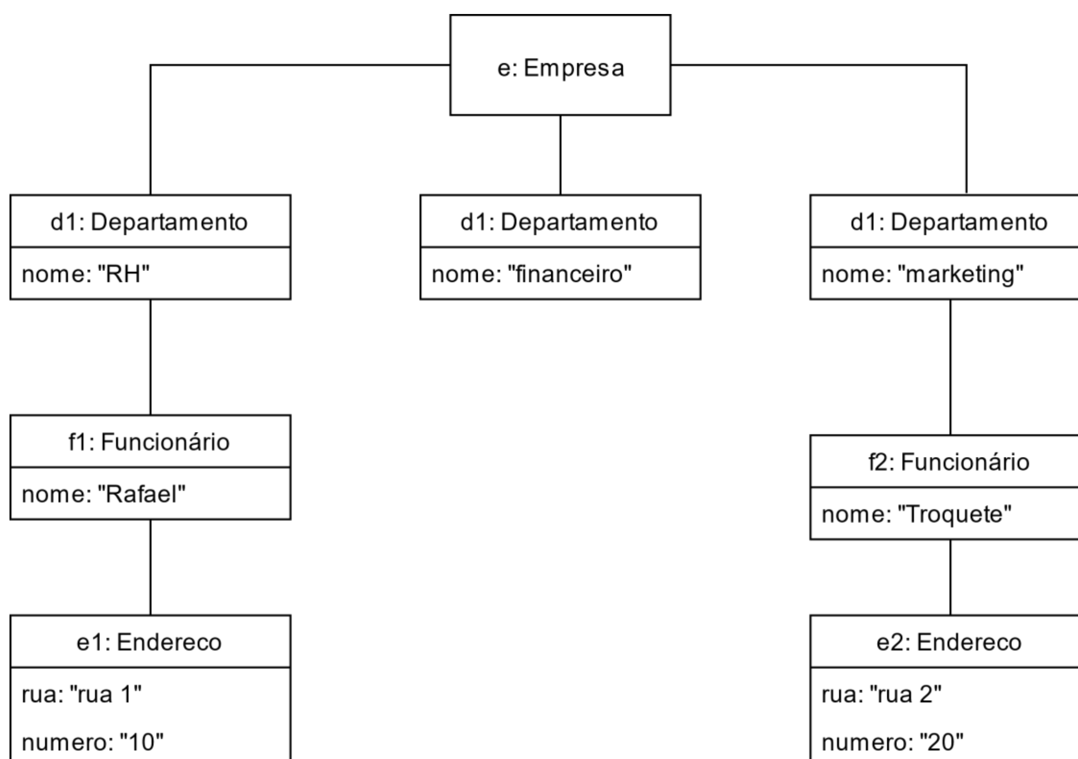

Figura 7 - Diagrama de objetos de um modelo de empresa

4 FUNDAMENTOS

Um fundamento é caracterizado como base para algo, no caso de POO podemos tratar como os pilares que dão forma e trazem sentido para o paradigma.

Com a evolução natural da OO e os constantes estudos feitos na área da programação, com o tempo foram surgindo algumas técnicas que se tornaram fundamentais para o uso desse paradigma.

Esses fundamentos serviram de base para novas técnicas e padrões, possibilitando ainda mais que as ideias iniciais advindas desde o SIMULA fizessem cada vez mais sentido e se tornassem comuns para qualquer software moderno.

Uma das grandes vantagens desse paradigma é construir modelos que sejam bastante adaptáveis e suscetíveis a mudanças. Dentro de um contexto de sistemas computacionais, o que mais interfere e gera desconforto são as mudanças constantes que devem acontecer dentro de um sistema.

A modelagem de objetos veio para tentar diminuir esse problema e ajudar na estruturação de um modelo computacional, e como um de seus objetivos centrais é o de resolver problemas reais, estar preparado para mudanças é um grande diferencial quando abordamos a criação de programas de computadores.

Podemos nos basear em alguns pilares, que servem de base para a construção de sistemas mais fiéis a uma abordagem do mundo real: abstração, encapsulamento e polimorfismo.

4.1.1 ABSTRAÇÃO

Na orientação a objetos, o foco é criar modelos computacionais que assumem características e comportamentos de modelos reais, dessa forma, obtém-se um programa que atende a um problema de fato. Com essa premissa em mente é possível

elaborar uma estrutura de dados baseada em atributos e operações para que seja possível comportar esse modelo dentro de um programa.

A abstração é a técnica onde é isolada as características de um objeto dado determinado contexto. Um livro pode ser representado de várias maneiras, seja ele como um objeto do tipo leitura, venda, empréstimo etc. Em cada um desses cenários, a abstração deve ser realizada pensando nos comportamentos que serão necessários dado o contexto no qual a representação de livro será inserida.

Alguns exemplos podem ser elucidados para mostrar a diferença entre um livro como objeto de leitura e como objeto de venda, e assim apontar as principais diferenças entre os modelos representados dessa forma.

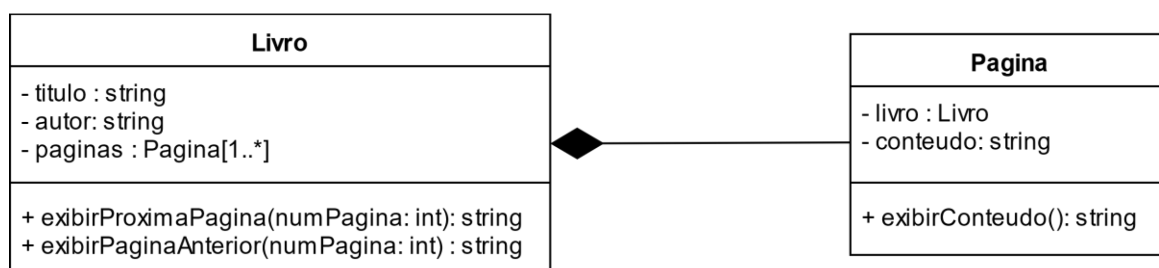

Figura 8 - Abstração de um livro como modelo de leitura

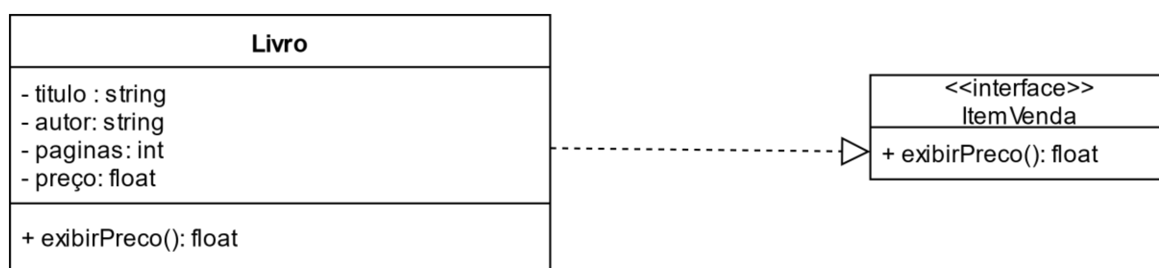

Figura 9 - Abstração de um livro como objeto de venda

4.1.2 ENCAPSULAMENTO

O encapsulamento remete a ideia de que um objeto deve ser responsável por realizar seus comportamentos e trabalhar com sua troca de estado sem que detalhes de sua implementação sejam expostos.

Esse fundamento dá sequência ao trabalho que é feito durante a abstração, ao levar para o modelo as operações e atributos que uma classe deve ter, deve-se analisar se detalhes de como as mudanças de estado dentro desse objeto irão acontecer estão ocultas o suficiente para que um comportamento ocorra sem necessidade de conhecimento prévio de determinada ação.

Ao realizar um saque em um caixa eletrônico, somente é escolhido o valor a ser retirado e em alguns instantes o dinheiro estará disponível. Muita coisa acontece no meio do caminho, é verificado se há saldo suficiente em conta, o valor é subtraído, a máquina faz a contagem das notas e libera o dinheiro. Para o cliente foi somente apertar um botão, e toda a complexidade acerca de como o processo aconteceu não faz a menor diferença para ele, o que importa é somente o objetivo final, o saque.

Na programação orientada a objetos esse cenário elucida claramente qual deve ser o trabalho do encapsulamento, se um objeto se propõe a realizar um saque, ele deve realizar todo o procedimento para essa ação, e assim não deixar que nenhuma outra parte importante para essa tarefa fique de fora. Se outro objeto conhecer algum detalhe dessa implementação significa que o comportamento não está encapsulado o bastante e com certeza estará sujeito a falhas no processo.

Um objeto conhecer algum detalhe de implementação significa que ele precisa realizar alguma ação para a conclusão do processo, quando isso ocorre é uma evidência bastante clara que determinado comportamento não está bem encapsulado.

O encapsulamento tem por objetivo esconder toda a complexidade por trás de uma tarefa, na analogia do caixa eletrônico todo o processo está escondido em um botão só, não importa em qual ordem as ações estão sendo realizadas desde que o objetivo final seja atingido.

Pode-se perceber que quanto mais informações ficarem ocultas sobre o processo melhor, todos os processos internos estão encapsulados, ou seja, blindados de interferências externas deixando todo o processo mais seguro para a realização de uma atividade.

4.1.3 HERANÇA

Há na orientação a objetos a definição de herança, que é uma característica que permite que abstrações sejam definidas em mais um nível.

Com esse conceito é possível criar classes que sejam especialistas em determinada função, e ao mesmo tempo, essas abstrações podem ser ainda mais refinadas dentro do modelo gerando funções ainda mais especializadas. Para esse refinamento pode-se sobrescrever os comportamentos pré-existentes para torná-las mais condizente com o modelo.

Ao utilizar deste recurso estamos compartilhando todos os atributos e operações que compõe uma classe, criando um esquema de hierarquia entre eles, esse compartilhamento ainda pode ser restrito se aplicarmos diferentes tipos de visualizações para os atributos e operações ali inseridos.

Podemos então separar as classes em dois tipos:

- Tipo Base – Classe que tem as características principais daquele modelo
- Tipo Derivado – Classes que herdam as características visíveis da classe base

Quando se pensa em um tipo base, temos um conjunto de classes que pode ser generalizado por uma que representa um conceito mais genérico e abstrato daquele modelo, e ao utilizar a herança não herdamos somente os atributos e operações, herdamos também os contratos que o tipo base mantém.

Para exemplificar esse modelo, pode-se criar uma interface do tipo Ave, onde é definido que toda ave deve comer e voar, em sequência, é criado classes concretas para respeitar esse contrato. Resultando então dois tipos genéricos:

- Voadores
- Não Voadores

Com essas estruturas, podemos criar modelos que representam mais fielmente os tipos de aves existentes os classificando nesses dois tipos genéricos, onde herdamos as implementações presentes do tipo base, consequentemente, respeitando o contrato pré-estabelecido.

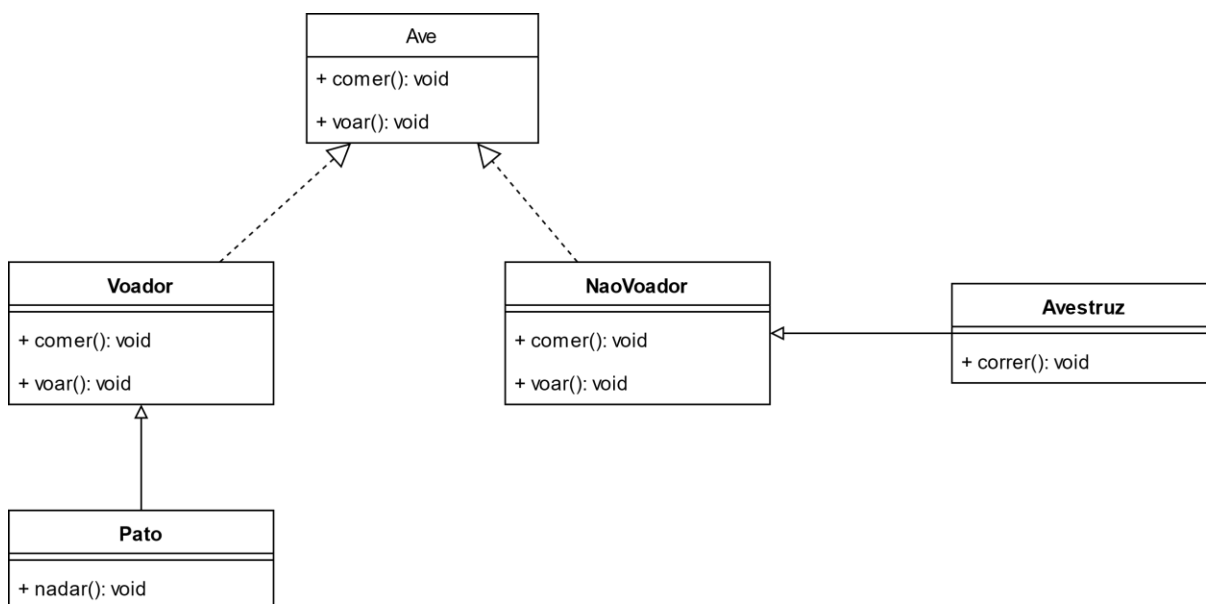

Figura 10 - Modelo utilizando herança

O modelo apresentado tem um problema muito evidente, pois apesar de ser possível criar uma representação dos tipos de aves existentes, a operação voar para os objetos não voadores apresentará um comportamento que pode não ter sido previsto na interface idealizada inicialmente.

A herança pode ser considerada um dos recursos mais poderosos que na orientação a objetos, consequentemente, acaba sendo um dos mais desafiadores, já que um dos grandes desafios da modelagem seria identificar onde essas abstrações mais genéricas podem ser realizadas de forma efetiva para obter um modelo simples e rico para a solução abordada.

4.1.3.1 POLIMORFISMO

O termo polimorfismo significa uma qualidade ou estado em que algo é capaz de assumir diferentes formas, e na modelagem de objetos é exatamente essa qualidade que é fundamental para a criação de um modelo rico e de qualidade.

É possível caracterizar o polimorfismo como uma consequência do uso de herança, já que não haveria necessidade de criarmos tipos genéricos se eles não pudessem assumir formas e comportamentos distintos dentro de um modelo.

Pode-se dizer que essa é uma das técnicas de maior potencial quando é utilizado a orientação a objetos, já que ela possibilita que um modelo seja mais adaptável a mudanças, uma vez que através de um contrato, ou um tipo mais abstrato, é possível estender ou até mesmo reescrever ações de maneira mais simples.

Se é esperado que um objeto do tipo imposto realize uma ação, deve ser possível que os tipos mais especializados sejam repassados para a realização de tal ação, assim independentemente do nível de especialização do objeto, a ação será realizada pois respeita todos os contratos anteriormente estabelecidos.

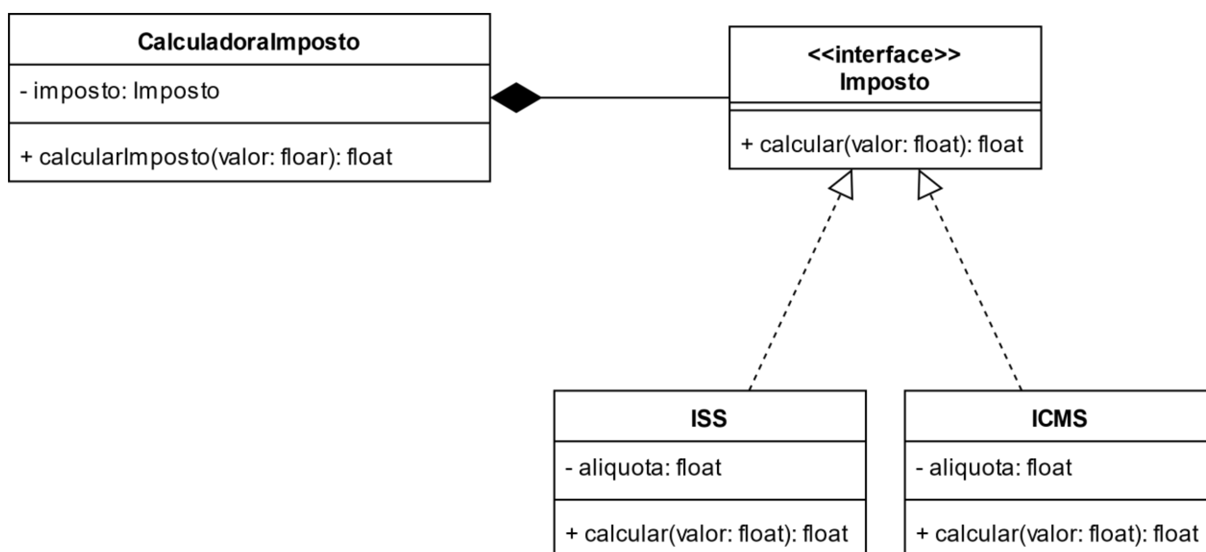

Figura 11 - Polimorfismo para o cálculo de imposto

4.2 TÉCNICAS

Caracterizamos como técnica a maneira de realizar uma ação ou um conjunto delas, como em um jogo de futebol, um tipo de drible bem executado é uma técnica para ajudar um jogador a chegar no seu objetivo, sendo assim, realizando um fundamento (drible) com maestria.

Dentro do paradigma de objetos existem algumas técnicas que nos ajudam no design e na modelagem de classes, como por exemplo, a alta coesão, que nos dá uma noção muito boa se a abstração de determinado elemento está condizente com seu propósito dentro do sistema.

4.2.1 REUSO

Quando trabalhamos com sistemas complexos podemos nos deparar com trechos de códigos que muitas vezes têm um comportamento que pode ser reaproveitado em diversos momentos do sistema.

Se esse código comum está espalhado em vários trechos de uma aplicação, corremos o risco de criamos implementações diferentes para o mesmo cenário e assim tornando a manutenção cada vez mais custosa e complexa.

Esse não é um recurso que OO proporciona nativamente, é necessário também que os fundamentos de polimorfismo, abstração e encapsulamento estejam bastante claros a ponto de podermos construir algumas estruturas que podem ser reutilizadas no sistema.

Antes do reuso tornar-se uma palavra chave dentro do paradigma de objetos, no mundo da computação já existia o conceito de DRY – Don’t Repeat Yourself (não repita a si mesmo), que representa muito bem o que queremos com reuso, que cada conhecimento do sistema possua apenas uma única representação, e não esteja espalhado pelo projeto.

A modelagem de objetos ajuda a ter uma visibilidade maior de onde encaixar novas regras ou somente definições de contratos a serem estabelecidos para que essas unidades sejam reutilizadas e facilitem que mudanças aconteçam somente uma vez e com um impacto bastante reduzido.

4.2.2 BAIXO ACOPLAMENTO

É comum que um objeto seja auxiliado ou troque mensagens com outros para a realização de uma tarefa, esse é inclusive um dos propósitos do paradigma, a troca de mensagens.

O quanto um objeto depende de outro para funcionar remete ao nível de acoplamento que eles possuem, e quanto maior for essa dependência significa que estão fortemente acopladas. Também é necessário verificar o quanto uma classe conhece sobre a outra, e se elas se restringem à uma interface, temos um baixo acoplamento.

Não há uma forma garantida de estruturar um modelo sem que haja um nível de acoplamento, em determinado momento será necessário uma comunicação ou o apoio de outros objetos para a realização de uma tarefa, portanto é importante sempre buscar que tais acoplamentos sejam sempre feitos com representações que são mais estáveis.

Quando uma classe se relaciona com um tipo concreto, há uma evidência que ali há um forte acoplamento, pois quaisquer mudanças podem afetar inteiramente o funcionamento das classes. O ideal é sempre buscar manter o relacionamento com classes que são mais estáveis e menos propensas a mudanças. Na orientação a objetos, é recomendado sempre criar modelos que sejam voltados as interfaces, pois esses modelos mudam pouco e tendem a trazer uma confiabilidade maior para o sistema ao qual está inserido.

4.2.3 ALTA COESÃO

Esse é um aspecto que atualmente pode ser considerado um dos mais conhecidos quando tratamos de modelagem de objetos. Dizer que algo é coeso significa que há um objetivo, uma harmonia bastante clara sobre algo, e ao sugerir os elementos que farão parte de um modelo, levar coesão para tais elementos significa que há uma responsabilidade bastante clara em suas definições.

Criar classes coesas é fundamental para a modelagem de objetos, elas são mais simples de ser mantidas pois contam com uma definição e responsabilidade bastante clara dentro do contexto ao qual está inserida.

O termo alta coesão refere-se ao nível de objetividade que uma classe possui, quanto mais centrado é o objetivo de uma classe é certo dizer que ela é mais coesa já que representa somente o propósito ao qual foi concebida.

Uma classe que representa uma conta corrente deve ser responsável somente por operações relacionadas a conta (transferir, sacar, depositar), não faz parte de seu objetivo, por exemplo, tratar valores de uma conta poupança.

Para tais objetivos são criadas classes que representam cada elemento de um determinado problema e eles exercem um nível específico de acoplamento entre si, uma conta corrente transfere dinheiro para uma conta poupança, enquanto uma conta poupança tem seus fundos resgatados para uma conta corrente, portanto elas podem manter um nível de acoplamento para que tais funções sejam estabelecidas.

Encontrar um nível certo entre acoplamento e coesão torna-se um dos maiores desafios desse tipo de modelagem, um baixo acoplamento e uma alta coesão devem ser buscados constantemente para um melhor design do modelo.

4.3 SOLID

Com a evolução dos estudos sobre orientação a objetos alguns princípios se tornaram fundamentais para esse modelo, e em 1995, Robert Cecil Martin inicia um esboço do que seriam esses princípios, inicialmente idealizados como “os 10 mandamentos da programação orientada a objetos”.

“Esses princípios expõem os aspectos de gerenciamento de dependência do design orientado a objetos em oposição aos aspectos de conceituação e modelagem. Isso não quer dizer que a OO seja uma ferramenta ruim para a conceituação do espaço do problema ou que não seja um bom local para a criação de modelos. Certamente muitas pessoas obtêm valor desses aspectos da OO. Os princípios, no entanto, concentram-se fortemente no gerenciamento de dependências.” (MARTIN, 2005)

Alguns desses princípios ficaram conhecido pelo acrônimo de SOLID que são designados principalmente ao design de classes.

4.3.1 PRINCÍPIO DE RESPONSABILIDADE ÚNICA (SRP)

“Uma classe deve ter somente uma razão para mudar”

Um dos grandes erros no design de classes e da modelagem orientada a objetos é quando uma classe é criada com muitas responsabilidades dentro de um sistema. Uma grande quantidade de operações e atributos desconexos começam a fazer parte dessa classe e assim deixá-la muito complexa e com elevado custo de manutenção.

Basicamente o que precisamos é que os objetos do nosso sistema tenham apenas uma única responsabilidade, ou seja, esses mesmos objetos devem se concentrar em exercer apenas uma única responsabilidade dentro do sistema e não ser utilizado para serviços das quais ele não tenha algum sentido semântico.

Quando percebemos esse tipo de cenário, é possível perceber que um dos fundamentos da POO não está sendo adequadamente seguido, pois estamos trabalhando com classes e objetos pouco coesos.

Se um objeto foi criado com o objetivo de executar uma ordem de compra, ele deve trabalhar somente com esse propósito, inserir funções para emissão de nota fiscal e calcular tributos devem ser abstraídas em outros objetos para que essas responsabilidades sejam encapsuladas e não haja necessidade de um objeto ter um nível maior de conhecimento sobre outro, bastando somente a troca de mensagens.

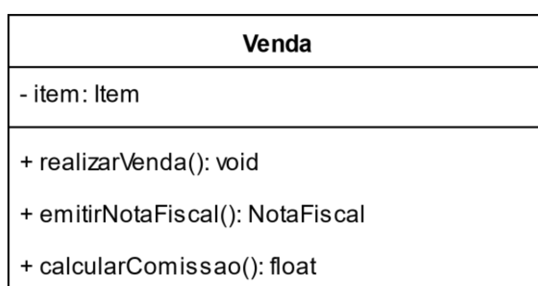

Figura 12 - Exemplo de classe sem coesão

Essa separação de responsabilidade visa principalmente fazer com que um objetivo não interfira no outro, se mudanças forem necessárias em qualquer uma das partes relacionadas, haverá somente uma razão para tal modificação, a de alterar um comportamento específico, e com isso o impacto tende a ser bastante reduzido já que a mudança ocorre em um escopo menor. A dependência entre as classes continua existindo, mas cada uma com seu propósito dentro do modelo.

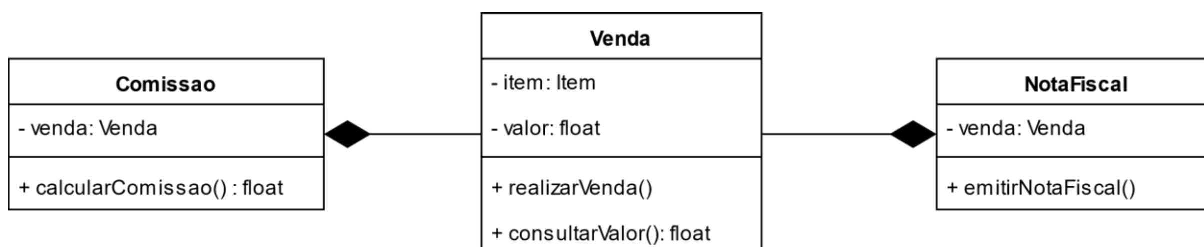

Figura 13 - Modelo de venda com classes coesas

Uma outra vantagem dessa individualização de responsabilidade, é a possibilidade de reuso, é bastante comum que uma operação precise aparecer em

mais de um ponto dentro do sistema. Quando essa necessidade é observada, fica bastante claro que é necessário encapsular determinado comportamento e ter um objeto que saiba realizar somente essa função.

4.3.2 PRINCÍPIO DO ABERTO FECHADO (OCP)

> *“Entidades de software (classes, módulos, funções etc.) devem ser abertas para extensão e fechadas para modificação”*

Umas das características de um programa de computador é que ele tende a mudar com bastante frequência, isso acaba sendo um problema na hora de conceber o design de um sistema dado que ele deve ter a capacidade de adaptar-se a constantes mudanças.

Ivar Jacobson elucidou muito bem isso em uma frase: *“Todos os sistemas tendem a mudar durante seu ciclo de vida. Isso deve ser sempre lembrado ao se desenvolver sistemas com expectativa de durar mais do que sua primeira versão”.*

O grande desafio acaba sendo em como criar um design que continua sendo estável mesmo após as modificações que serão incluídas em versões futuras. Para minimizar o impacto de uma mudança, ou com fim de facilitar esse trabalho, foi elaborado o princípio do aberto fechado.

Essa técnica foi introduzida por Bertand Meyer em seu livro *Object-Oriented Software Construction,* de 1988. O princípio aberto-fechado atende a duas premissas iniciais:

1. Um módulo é considerado aberto se ele está disponível para extensão.
2. Um módulo é considerado fechado se ele está disponível para ser utilizado por outros módulos.

Esse princípio entrega um grande valor para a modelagem de objetos, dado suas características de criar abstrações e objetos polimórficos, podem existir

entidades que têm seus comportamentos alterados sem a necessidade de modificações em cascata.

Uma entidade de software considerada aberta é aquela que conseguimos mudar ou gerar novos comportamentos sem que haja impacto nos comportamentos já existentes. É dito então que tal artefato está disposto a ter uma extensão já que sua estrutura suporta que novos comportamentos sejam adicionados com facilidade.

Para atender a esse objetivo o uso de abstrações é imprescindível, pois quando é criado modelos mais abstratos, é possível obter um número ilimitado de comportamentos. Essas abstrações são constituídas por um tipo base, fazendo uso de interfaces ou qualquer outro modelo que represente uma abstração.

Para a realização de uma venda, devemos calcular algum tipo de imposto, dado esse cenário é possível obter o seguinte modelo:

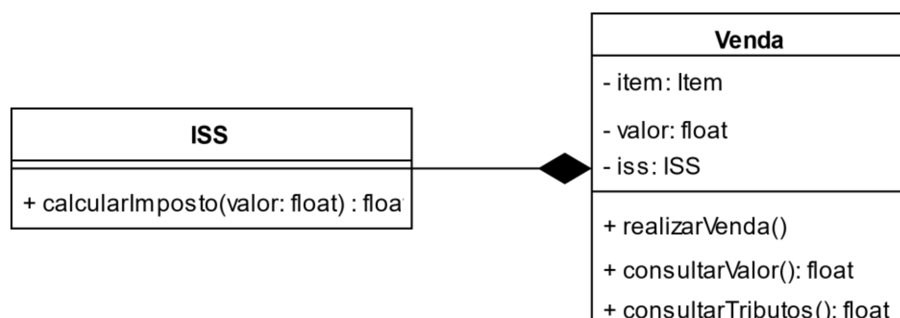

Figura 14 - Cálculo de ISS em uma venda

É possível observar que a classe imposto tem uma associação com a classe compra, e há uma operação que realiza o cálculo de um imposto qualquer dado o valor da compra. Portanto, se alguma mudança precisar acontecer na forma de calcular o imposto, será necessário sempre alterar o método de cálculo.

Apesar da alteração precisar ser feita somente na classe imposto, e a classe compra não precisar sofrer nenhuma modificação, ainda é possível observar outro problema, se for necessário o cálculo de outro tipo de imposto pela compra.

Visando aperfeiçoar esse modelo, podemos aplicar o princípio do aberto fechado e fazer com que a classe compra dependa de uma abstração de imposto, dessa forma, poderíamos obter um comportamento diferente para cada necessidade que surja a partir desse ponto.

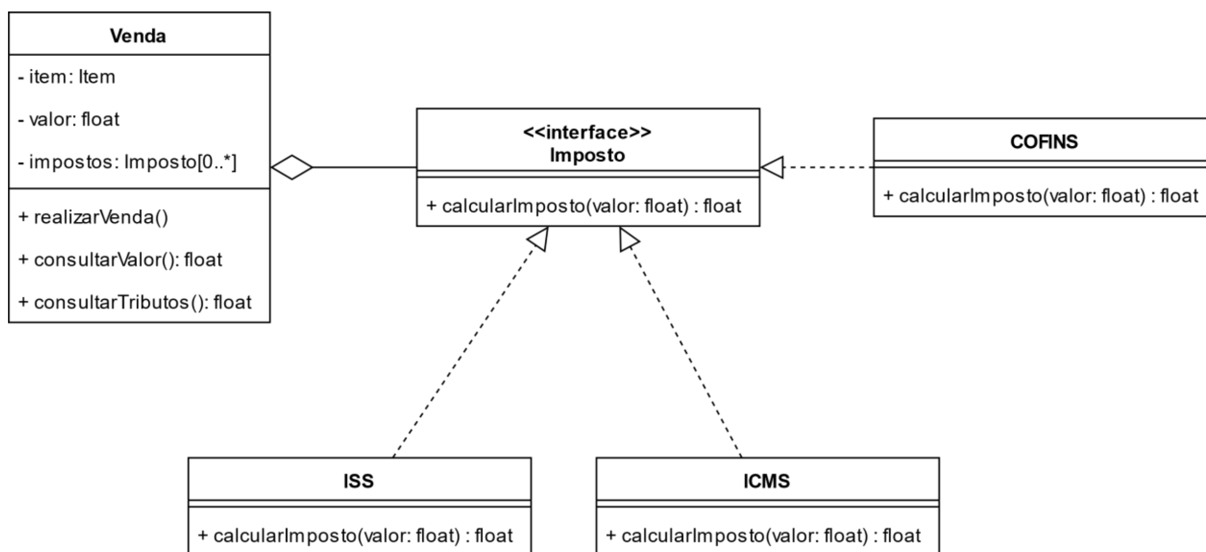

Figura 15 - Classe venda aberta a extensão de novos impostos

Quando consideramos um módulo fechado significa que ele deve ser inviolável e é esperado que ele tenha sido bem definido e contenha uma descrição estável. Se tais premissas forem impostas, este módulo poderá ser utilizado por outras partes do sistema sem a necessidade de modificações, podendo ter seu comportamento estendido de acordo com a necessidade do sistema.

4.3.3 PRINCÍPIO DE SUBSTITUIÇÃO DE LISKOV (LSP)

> "Funções que utilizam referências para as classes base devem poder utilizar os objetos da classe derivada sem conhecê-los"

Para compreender esse princípio é necessário primeiramente entender as motivações por trás do princípio do aberto fechado. Uma das formas de estender um comportamento é pela utilização de herança, um tipo base pode ser criado com fim

de abstrair um conceito qualquer e em sequência ser utilizado por outro objeto para a realização de uma tarefa.

Quando a herança é utilizada pode acontecer de determinada generalização não necessariamente ter um comportamento que o tipo base impõe, podendo levar algum tipo de inconsistência para a operação a qual faz uso de um tipo de abstração base.

O princípio de substituição de Liskov retrata o seguinte cenário:

“Se para cada objeto o1 do tipo S existir um objeto o2 do tipo T de forma que, para todos os programas P definidos em termos de T, o comportamento de P é inalterado quando o1 é substituído por o2 então S é um subtipo de T” (LISKOV e WING, 1994)

Colocando em poucas palavras, se objetos compartilham o mesmo tipo dentro de um programa, eles podem ser substituídos por seus subtipos sem que haja nenhuma inconsistência dentro do programa.

Esse princípio foi amplamente baseado em um conceito de design por contrato, exposto por Bertrand Meyer no livro *Object-Oriented Software Construction*. Esse modelo de design define como as regras de pré e pós condições devem ser obedecidas, e tais definições foram incorporadas no princípio de Liskov e elencadas as definições de substituição por subtipos na utilização de herança.

As pré-condições são definições sobre o que um objeto espera e as restrições para que determinada operação nele descrita funcione corretamente. Para fazer com que um objeto possa continuar funcionando quando um subtipo é utilizado, há uma regra bastante importante, que é a de nunca reforçar uma pré-condição. Reforçar significa colocar uma restrição maior do que a esperada pelo tipo pai, no entanto podemos afrouxar essa pré-condição que isso não trará impacto. Se um objeto sabe trabalhar com números naturais, podemos expandir para que ele trabalhe com números reais, mas nunca o contrário, pois seria quebrado tal pré-condição e não seria mais válida a substituição desse objeto como subtipo.

As pós-condições remetem-se ao valor que uma operação entrega, as definições sobre um retorno devem ser bem descritas e não devolver valores inesperados. Isso vale para as classes derivadas, não deve ser permitido que valores que não fazem parte o tipo pai sejam devolvidos por uma operação na qual foi herdado um comportamento e alterado seu método.

Para a estratégia é não entregar valores que fazem parte de um contexto maior, logo só é possível que uma pós-condição seja estreitada. Se uma pós-condição devolve um número natural, não se pode entregar um conjunto de números reais, pois nesse caso haveria um afrouxamento dessa regra, e o objeto poderia não estar preparado para trabalhar com aqueles valores.

4.3.4 PRINCÍPIO DE SEGREGRAÇÃO DE INTERFACE (ISP)

> "Os clientes não devem ser forçados a depender de contratos que não usam"

O princípio de responsabilidade única (SRP) tem como objetivo tornar as classes coesas, mas quando um trabalho de modelagem começa a tornar forma deve-se estender esse princípio também para as interfaces do sistema.

Por muitas vezes é possível encontrar interfaces que possuem muitas operações que por sua vez acabam se tornando contratos extensos e que podem dar um significado maior do que o necessário dentro do design do sistema.

Se uma operação não é genérica o bastante para estar dentro de uma interface, é bem provável que ela pode ser abstraída para fazer parte de outra interface que possa apresentar um valor maior para o modelo.

No capítulo onde foi explicado sobre os conceitos de herança, um modelo sobre aves voadoras e não voadoras foi apresentado, é possível perceber que aquele

desenho tem um grande problema, foi definido na interface que toda ave voa, logo é necessário implementar essa operação em todos os subtipos que forem criados a partir desse modelo.

Dado esse fato pode-se perceber que a interface ave tem muito mais operações do que o necessário, e os clientes que implementam esse contrato como um todo podem estar herdando comportamentos que não são necessários para seu funcionamento.

Essa visão de separação de responsabilidade das operações das interfaces, tornando-as mais enxutas, é que diz respeito ao princípio de segregação de interfaces (ISP), pois dessa forma as classes devem estender somente os comportamentos que elas realmente necessitam.

Separando essas responsabilidades pode-se obter o seguinte desenho:

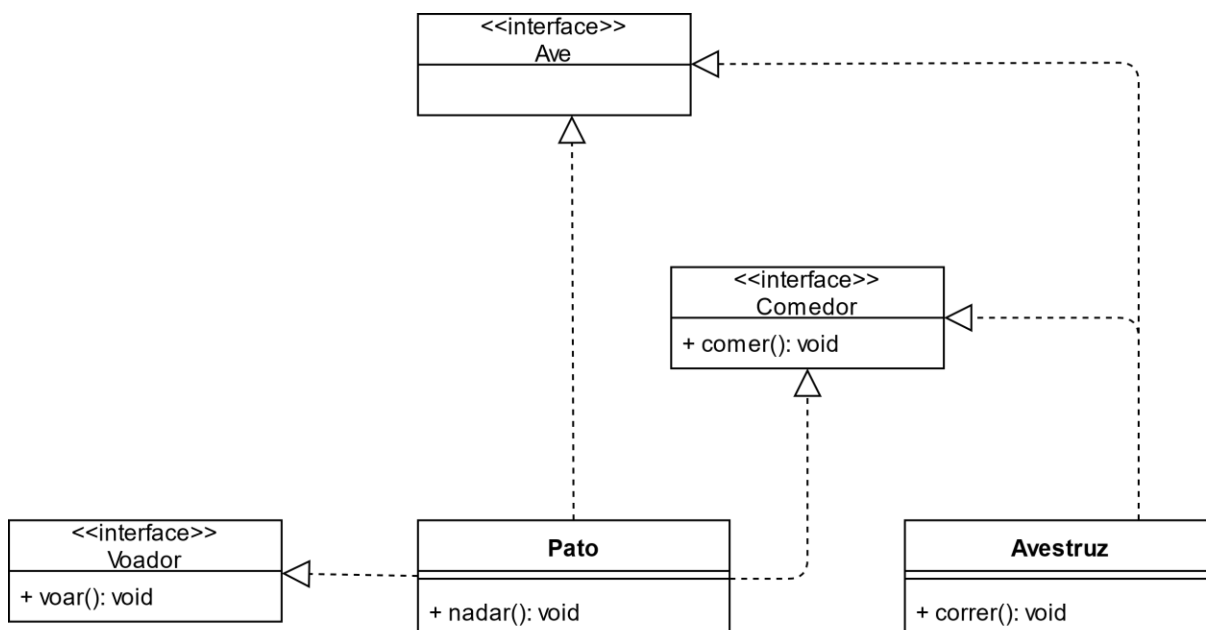

Figura 16 - Criação de interfaces especializadas evoluindo o modelo de aves da seção de herança aplicando o princípio de segregação de interfaces

Com esse conceito é possível criar um sistema mais estável, já que por meio da abstração, modelos mais enxutos e com responsabilidades menores estão sendo criados o que acaba criando unidades menores de manutenção e mudanças.

Quanto menor um modelo, menos tendência a mudar ele tem, e isso é uma verdade também para as interfaces, quanto menos contratos estão expostos, mais significado ela terá dentro do domínio e consequentemente gerando implementações mais coesas de seus contratos.

4.3.5 PRINCÍPIO DE INVERSÃO DE DEPENDÊNCIA (DIP)

“Dependa de abstrações, não de modelos concretos”

Um dos grandes desafios da modelagem de objetos é o de criar um design estável e maduro para atender a um problema específico. Esse design precisa ser o mais coeso e menos acoplado possível, e encontrar o meio termo entre acoplamento e coesão acaba sendo uma tarefa bastante complexa.

Quando existe a necessidade de criar algum tipo de acoplamento, deve-se ter em mente que tal acoplamento deve ser bastante estável para não levar nenhum comportamento indesejado para a classe da qual está fazendo uso de suas operações.

Dentro do mundo de objetos, as interfaces tendem a ser os modelos mais estáveis pois eles definem um contrato para seu uso e com isso pode-se esperar que ele atenderá as pré e pós condições a ele impostas. Adicionando o princípio de segregação de interfaces (ISP), levamos ainda mais estabilidade para as interfaces, já que estão sendo criado modelos ainda mais coesos.

É dito que implementações são menos estáveis por possuírem um comportamento já descrito para um contrato podendo levar consigo algumas particularidades que podem ser complexas de serem alteradas em momentos futuros, também pode haver operações a mais dentro dessa classe, o que pode dificultar uma mudança para outro subtipo devido ao uso dessas operações não declaradas na interface que a compõem.

Se uma classe necessita depender de outra para realizar determinada ação, então essa outra deve ser mais estável que ela mesma, no entanto um design deve sempre ser pensado de modo que sempre busquem elementos mais estáveis para realizar suas ações, em outras palavras, deve-se majoritariamente idealizar um acoplamento voltado a interfaces.

Esse termo foi cunhado pois em outros paradigmas é comum um módulo de alto nível depender de módulos de baixo nível diretamente, e isso caracteriza em um nível mais baixo de confiabilidade. No paradigma de objetos é possível “inverter” essa ideia e passar a fazer que um módulo de alto nível dependa de uma abstração.

Essa ideia leva a duas características fundamentais que descrevem o princípio de inversão de dependência:

1. Módulos de alto nível não devem depender dos módulos de baixo nível. Os dois devem ser baseados em abstrações.
2. Abstrações não devem ser baseadas em detalhes. Detalhes devem ser baseados em abstrações.

A figura abaixo ilustra um acoplamento entre duas implementações:

Figura 17 - Acoplamento forte entre duas classes

O problema nesse caso é que um módulo de alto nível depende de um componente de baixo nível, e dado esse cenário, uma abstração entre esses dois elementos torna o modelo mais desacoplado, pois a responsabilidade que a classe de alto nível mantém com a de baixo nível agora foi invertida para uma abstração.

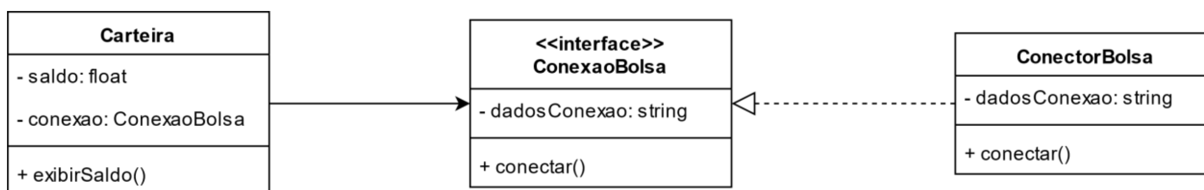

Figura 18 - Aplicando inversão de dependência para a conexão

Como o relacionamento entre as classes em algum momento vai ocorrer dentro do sistema, pode-se concluir que esse acoplamento sempre deve ocorrer com componentes considerados mais estáveis para que seja possível diminuir o grau e acoplamento entre os componentes do sistema.

5 PADRÕES DE PROJETO

Com a evolução das técnicas e do uso do constante da modelagem orientada a objetos, alguns padrões no desenvolvimento foram observados. Esses padrões são soluções que podem ser reutilizados para a resolução de diversos problemas.

Curiosamente os padrões de projeto têm origem das ideias do arquiteto de cidades Christopher Alexander, que propôs em alguns de seus livros, as características para que algo fosse considerado um padrão. Além das características Alexander também definiu o formato de um padrão, que deve ser composto por:

1. Nome
2. Exemplo

3. Contexto
4. Problema
5. Solução

A partir das ideias de Alexander, alguns programadores propuseram os primeiros padrões de projeto, estruturados a partir de elementos do paradigma de objetos na linguagem Smalltalk. O que tornou o termo padrões de projeto bastante conhecido dentro da indústria de software foi o livro *Design Pattern: Elements of Reusable Object Oriented Software.*

O livro publicado em 1995 apresenta padrões de projeto orientado a objetos que são utilizados atualmente, o que comprova que são soluções que podem ser adaptadas para os problemas mais modernos encontrados dentro da modelagem de objetos.

Esse livro separa os padrões de projeto apresentando-os em três diferentes tipos, cada uma abordando soluções para um tipo específico de problema dentro de um contexto, esses tipos são comportamentais, estruturais e criacionais.

5.1 CRIACIONAIS

Os padrões criacionais têm como objetivo principal abstrair as complexidades por trás da instanciação de um objeto, ajudando a tornar o sistema independente de como essas criações funcionam. Em muitas situações criar um objeto pode ser um trabalho custoso e se esse comportamento for encapsulado em algum elemento que possa ser reutilizado, boa parte dessa complexidade pode ficar transparente para a aplicação.

Se alguma mudança ocorre na criação de um objeto complexo, não é necessário se preocupar com as partes que serão afetadas já que um padrão foi utilizado para abstrair essa criação.

É comum um componente de alto nível estar acoplado com um componente de baixo nível, o mais comum de se pensar é no uso de um banco de dados por uma aplicação, gerir os dados de conexão pode ser um trabalho complexo para se manter em cada ponto que a aplicação precisar se conectar com esse item de infraestrutura. Para esse fim existem os padrões criacionais, que podem gerir essas comunicações e encapsular a complexidade para a criação de um objeto.

Por fim, os padrões de projeto criacionais não são responsáveis pelo ciclo de vida do objeto, eles são responsáveis somente por prover uma instância e nada mais, qualquer outra demanda relacionada ao ciclo de vida não compete a esse tipo de padrão.

Os padrões criacionais são:

- Abstract Factory – Fornece uma interface para produzir objetor sem definir suas classes concretas.
- Builder – Provê uma interface para construir um objeto complexo de modo que seja possível criar diferentes representações.
- Factory Method – Provê uma interface para a criação de um objeto deixando para as subclasses decidirem a classe a ser instanciada.
- Prototype – Permite copiar um objeto existente sem tornar o código dependente de suas classes
- Singleton – Garante que uma classe tenha somente uma instância fornecendo um ponto de acesso global

5.2 ESTRUTURAIS

Os padrões estruturais consistem em um modelo de design onde há uma preocupação de como se dará a composição entre os objetos, e para facilitar a comunicação com estruturas maiores dentro do modelo. O objetivo por trás de

padrões estruturais é o de facilitar o design mantendo estruturas flexíveis e com maior capacidade de adaptação de novos elementos do sistema.

Por muitas vezes manter de forma simples um modelo grande pode ser uma tarefa bastante complexa, e os padrões estruturais ajudam a manter essas estruturas da maneira que tornem o design mais eficiente e flexível diminuindo a complexidade entre os relacionamentos.

Os modelos estruturais sugerem formas de criar adaptações, trabalhar em como decorar o comportamento de um objeto e até em modos de desacoplar o modelo sempre visando diminuir a complexidade de acoplamento.

Os padrões estruturais são:

- Adapter – Provê que objetos com interfaces incompatíveis possam colaborar um com o outro.
- Bridge – Separa uma abstração da implementação, fazendo com que elas possam variar independentes umas da outras.
- Composite – Compõe objetos em estruturas de árvores trabalhando com essa estrutura como se fossem objetos individuais.
- Decorator – Permite anexar a um objeto novos comportamentos dinamicamente.
- Façade – Fornece uma interface única para as funções de um sistema.
- Flyweight – Utiliza partes comuns do estado dos objetos criando um compartilhamento ao invés de armazenar todos os estados separadamente.
- Proxy – Fornece uma representação de um objeto permitindo adicionar um comportamento as operações do objeto original.

5.3 COMPORTAMENTAIS

Padrões comportamentais concentram-se em facilitar a comunicação e a atribuição de responsabilidades entre os objetos. Os padrões desse tipo trabalham definindo estratégias para que um comportamento seja executado de maneira mais simples e desacoplada.

Os padrões comportamentais baseiam-se bastante nos quesitos de polimorfismo e herança para distribuir o comportamento entre as classes, também através da composição descrevem a cooperação entre objetos para a execução de uma tarefa que um objeto sozinho não poderia realizar sozinho.

Uma característica bastante presente nesses padrões é a de cooperação entre os objetos, enquanto os padrões estruturais e criacionais concentram-se em tarefas menores, visando em sua maioria apenas um tipo de objeto, os padrões comportamentais entregam um dinamismo maior para o modelo, pois tratam de comunicação e estratégias para descrever um comportamento.

Os padrões comportamentais são:

- Chain of Responsability – Provê um modo de passar uma requisição ao longo de uma cadeia de manipulação, onde cada nó terá uma tratativa diferente antes de realizar uma função efetivamente.
- Command – Encapsula uma solicitação em um objeto independente que contém todas as informações sobre a solicitação.
- Iterator – Permite percorrer elementos de uma coleção sem expor detalhes de sua representação subjacente.
- Mediator – Reduz a forma de como um conjunto de objetos interagem restringindo sua comunicação direta, somente por um mediador.
- Memento – Permite salvar e restaurar estados de um objeto sem revelar detalhes de sua implementação.
- Observer – Define um mecanismo de inscrição para notificar todos os objetos inscritos sobre suas mudanças de estado

- State – Permite que um objeto mude seu comportamento quando seu estado interno muda.
- Strategy – Permite definir um conjunto de algoritmos, cada um em classes separadas tornando-os intercambiáveis.
- Template Method – Define o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses.
- Visitor – Permite definir uma nova operação sem mudar as classes dos elementos sobre os quais operam.

6 CONSIDERAÇÕES FINAIS

Durante a atividade de pesquisa bibliográfica foi possível observar que há muito material focado em ensinar sobre orientação a objetos voltada para uma tecnologia específica e explicando os conceitos de maneira superficial. É bastante raro encontrar publicações recentes que trabalham na contextualização dos elementos de modelagem desse paradigma.

Essa pesquisa reforçou ainda mais a tese de elaborar um trabalho que levasse ao leitor de maneira objetiva todo o histórico e as motivações para o surgimento e o uso do modelo orientado a objetos. Para esse propósito todo o texto foi pensado para ser apresentado de maneira evolutiva, onde fosse possível caminhar pela história, obter uma definição, entender a estrutura, fundamentos e exemplificar algumas características evolutivas que no cenário atual são indispensáveis, que é o caso dos padrões de projetos.

A ideia de Nygaard era criar uma linguagem para a simulação de eventos discretos, mas isso tornou-se um objeto de estudo muito maior que o proposto inicialmente, pois de uma linguagem surgiu um paradigma que até os dias de hoje vem ganhando cada vez mais força, pois possibilita que sistemas mais complexos sejam elaborados com foco em problemas reais. Essa foi uma das grandes vantagens observadas durante o curso deste trabalho, que é a capacidade de abstrair elementos para o meio computacional.

Foi possível notar durante a elaboração do trabalho que é fundamental conhecer os detalhes do que cada elemento representa dentro da orientação a objetos. O modelo e design orientado a objetos é um jeito de pensar sobre problemas usando modelos organizados em torno dos conceitos do mundo real (RUMBAUGH, BLAHA, *et al.*, 1991).

Com o intuito de facilitar a compreensão desse paradigma, esse trabalho propôs uma sequência de estudo para reforçar os conceitos que podem ser considerados como básicos, contemplando desde as definições de classes, atributos e operações, também conceitos fundamentais de abstração e encapsulamento até

conceitos mais avançados como padrões de projeto, para dessa forma apresentar a evolução sobre a modelagem de objetos.

Por meio deste trabalho, foi possível apresentar o potencial que o paradigma de objetos entrega quando se obtém uma visão mais conceitual dos elementos que a compõem, e isso pôde ser observado com a evolução dos diagramas, que permitiram uma melhor reflexão a respeito do design que estava sendo utilizado. Os diagramas foram propostos para servir de apoio para o entendimento do que estava sendo apresentado, e com isso foi possível mostrar como os elementos estruturais interagem e constituem alguns dos fundamentos e técnicas.

Por fim, para trabalhos futuros, seria importante introduzir os estudos sobre UML e outras linguagens para além de conceituar, ter insumos práticos para ajudar ainda mais na motivação de se conhecer os fundamentos e as técnicas de modelagem e design orientado a objetos.

7 BIBLIOGRAFIA

ANICHE, M. **Orientação a Objetos e SOLID para Ninjas**. [S.l.]: Casa do Código, 2015.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The Unified Modeling Language User Guide**. [S.l.]: [s.n.], 1998.

CARVALHO, T. L. E. **Orientação a Objetos Aprenda seus conceitos e suas aplicabilidades de forma efetiva**. São Paulo: Casa do Código, 2018.

FREEMAN, E.; FREEMAN, E. **Use a Cabeça! Padrões de Projetos**. [S.l.]: Alta Books, 2007.

GAMMA, E. et al. **Design Patterns:** Elements of Reusable Object-Oriented Software. [S.l.]: Addison-Wesley Professional, 1994.

GUERRA, E. **Design Patterns com Java:** Projeto orientado a objetos guiado por padrões. [S.l.]: Casa do Código, 2012.

JANKE, E.; BRUNE, P.; WAGNER, S. **Does Outside-In Teaching Improve the Learning of Object-Oriented Programming?** 37h International Conference on Software Engineering (ICSE'15). [S.l.]: [s.n.]. 2015.

KAY, A.. **HOPL II**. The Early History of Smalltalk. Cambridge, Massachusetts: [s.n.]. 1993.

KAY, A. Dr. Alan Kay on the Meaning of “Object-Oriented Programming”. Disponivel em: <http://www.purl.org/stefan_ram/pub/doc_kay_oop_en>.

LISKOV, B. H.; WING,. A Behavioral Notion of Subtyping. **ACM Transactions on Programming Languages and Systems**, p. 1811--1841, 1994.

MARTIN, R. C. Principles of OOD. **But Uncle Bob**, 11 Maio 2005. Disponivel em: <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod>.

MCLAUGHLIN, B.; POLLICE, G.; WEST, D. **Use a Cabeça! Análise e Projeto Orientado ao Objeto**. [S.l.]: Alta Books, 2007.

MEYER, B. **Object-Oriented Software Construction**. [S.l.]: Prentice Hall, 1988.

NANCE, R. E. A history of discrete events simulation programming languages. **Blacksburg: Department of Computer Science**, 1993.

RUMBAUGH, J. et al. **Object-Oriented Modeling And Design**. [S.l.]: [s.n.], 1991.

SIMULA Standards. **Simula Software Development**, 25 ago. 1968. Disponivel em: <http://s120.hbv.no/simula/Standard/index.html>.