



MODELAGEM DE SOFTWARE ORIENTADO A OBJETOS

BRASÍLIA-DF.

Elaboração

Jorge Umberto Scatolin Marques

Produção

Equipe Técnica de Avaliação, Revisão Linguística e Editoração

Sumário

APRESENTAÇÃO.....	5
ORGANIZAÇÃO DO CADERNO DE ESTUDOS E PESQUISA	6
INTRODUÇÃO.....	8
UNIDADE I	
MODELAGEM DE SOFTWARE	9
CAPÍTULO 1	
PROCESSO DE MODELAGEM	9
CAPÍTULO 2	
PARADIGMA ORIENTADO A OBJETOS.....	13
CAPÍTULO 3	
UML	16
CAPÍTULO 4	
CASCATA, RUP, XP	28
UNIDADE II	
MODELAGEM ORIENTADA A OBJETOS.....	33
CAPÍTULO 1	
CLASSES, OBJETOS, ATRIBUTOS E MÉTODOS	33
CAPÍTULO 2	
HERANÇA, ENCAPSULAMENTO, POLIMORFISMO	40
CAPÍTULO 3	
COESÃO, ACOPLAMENTO, ABSTRAÇÃO, INTERFACE	46
CAPÍTULO 4	
RELACIONAMENTO ENTRE OBJETOS: ASSOCIAÇÃO, AGREGAÇÃO, COMPOSIÇÃO.....	49
UNIDADE III	
DESIGN DE SISTEMAS E PADRÃO GOF	53
CAPÍTULO 1	
PADRÕES DE DESIGN E ARQUITETURAS.....	53
CAPÍTULO 2	
MODELO DE DESIGN.....	56

CAPÍTULO 3	
DOCUMENTAÇÃO DA ARQUITETURA	60
CAPÍTULO 4	
PADRÃO GOF	63
UNIDADE IV	
PADRÕES DE ARQUITETURA E DESIGN	71
CAPÍTULO 1	
PADRÃO JAVA	71
CAPÍTULO 2	
PADRÃO .NET	79
CAPÍTULO 3	
PADRÃO PHP	83
REFERÊNCIAS	87

Apresentação

Caro aluno

A proposta editorial deste Caderno de Estudos e Pesquisa reúne elementos que se entendem necessários para o desenvolvimento do estudo com segurança e qualidade. Caracteriza-se pela atualidade, dinâmica e pertinência de seu conteúdo, bem como pela interatividade e modernidade de sua estrutura formal, adequadas à metodologia da Educação a Distância – EaD.

Pretende-se, com este material, levá-lo à reflexão e à compreensão da pluralidade dos conhecimentos a serem oferecidos, possibilitando-lhe ampliar conceitos específicos da área e atuar de forma competente e conscienciosa, como convém ao profissional que busca a formação continuada para vencer os desafios que a evolução científico-tecnológica impõe ao mundo contemporâneo.

Elaborou-se a presente publicação com a intenção de torná-la subsídio valioso, de modo a facilitar sua caminhada na trajetória a ser percorrida tanto na vida pessoal quanto na profissional. Utilize-a como instrumento para seu sucesso na carreira.

Conselho Editorial

Organização do Caderno de Estudos e Pesquisa

Para facilitar seu estudo, os conteúdos são organizados em unidades, subdivididas em capítulos, de forma didática, objetiva e coerente. Eles serão abordados por meio de textos básicos, com questões para reflexão, entre outros recursos editoriais que visam tornar sua leitura mais agradável. Ao final, serão indicadas, também, fontes de consulta para aprofundar seus estudos com leituras e pesquisas complementares.

A seguir, apresentamos uma breve descrição dos ícones utilizados na organização dos Cadernos de Estudos e Pesquisa.



Provocação

Textos que buscam instigar o aluno a refletir sobre determinado assunto antes mesmo de iniciar sua leitura ou após algum trecho pertinente para o autor conteudista.



Para refletir

Questões inseridas no decorrer do estudo a fim de que o aluno faça uma pausa e reflita sobre o conteúdo estudado ou temas que o ajudem em seu raciocínio. É importante que ele verifique seus conhecimentos, suas experiências e seus sentimentos. As reflexões são o ponto de partida para a construção de suas conclusões.



Sugestão de estudo complementar

Sugestões de leituras adicionais, filmes e sites para aprofundamento do estudo, discussões em fóruns ou encontros presenciais quando for o caso.



Atenção

Chamadas para alertar detalhes/tópicos importantes que contribuam para a síntese/conclusão do assunto abordado.

**Saiba mais**

Informações complementares para elucidar a construção das sínteses/conclusões sobre o assunto abordado.

**Sintetizando**

Trecho que busca resumir informações relevantes do conteúdo, facilitando o entendimento pelo aluno sobre trechos mais complexos.

**Para (não) finalizar**

Texto integrador, ao final do módulo, que motiva o aluno a continuar a aprendizagem ou estimula ponderações complementares sobre o módulo estudado.

Introdução

Vivemos em uma era em que a demanda por dispositivos computacionais tende a ultrapassar muitas barreiras em relação ao número de dispositivos por pessoa e aos paradigmas de desenvolvimento de aplicativos. Isso significa que cada pessoa no planeta possuirá, em média, cinco dispositivos que necessitam de software.

Por isso, para atender a essa demanda, a indústria de software vem se atualizando e se reinventando num período de tempo cada vez mais curto, ou seja, em poucos anos vários paradigmas foram criados e muitos outros deixaram de ser seguidos.

O desenvolvimento de software baseado em tecnologias antigas acabou se tornando moroso, devido à necessidade de se fabricar aplicativos para diversos públicos e plataformas. As equipes eram grandes, normalmente localizadas no mesmo prédio, e cada uma tinha seu papel bem especificado, jamais saíam das regras, que eram mais rígidas. Entretanto, as grandes dificuldades estavam a ponto de estragar o produto final, pois, muitas vezes, a própria legislação era mudada, e o desenvolvimento do software não conseguia acompanhar.

Com as metodologias ágeis, que são mais modernas e atualizadas, muitos dos problemas encontrados antigamente foram solucionados. Hoje, projetar a arquitetura de um software baseando-se nas técnicas atuais significa entregar um produto mais rapidamente, com eficiência na geração das documentações e na escolha da tecnologia.

Neste módulo, serão explicadas as principais tecnologias de modelagem de software orientado a objetos.

Objetivos

- » Compreender melhor as principais técnicas de modelagem de arquitetura de software de mercado.
- » Estudar as principais técnicas do paradigma Orientado a Objetos.
- » Compreender os padrões de projetos Orientados a Objetos.

CAPÍTULO 1

Processo de modelagem

Modelar software significa criar modelos. Segundo Guedes (2001, p. 21), um modelo de software captura uma visão de um sistema físico; é uma abstração do sistema com certo propósito, como descrever aspectos estruturais ou comportamentais do software. Esse propósito determina o que deve ser incluído no modelo e o que deve ser considerado irrelevante.

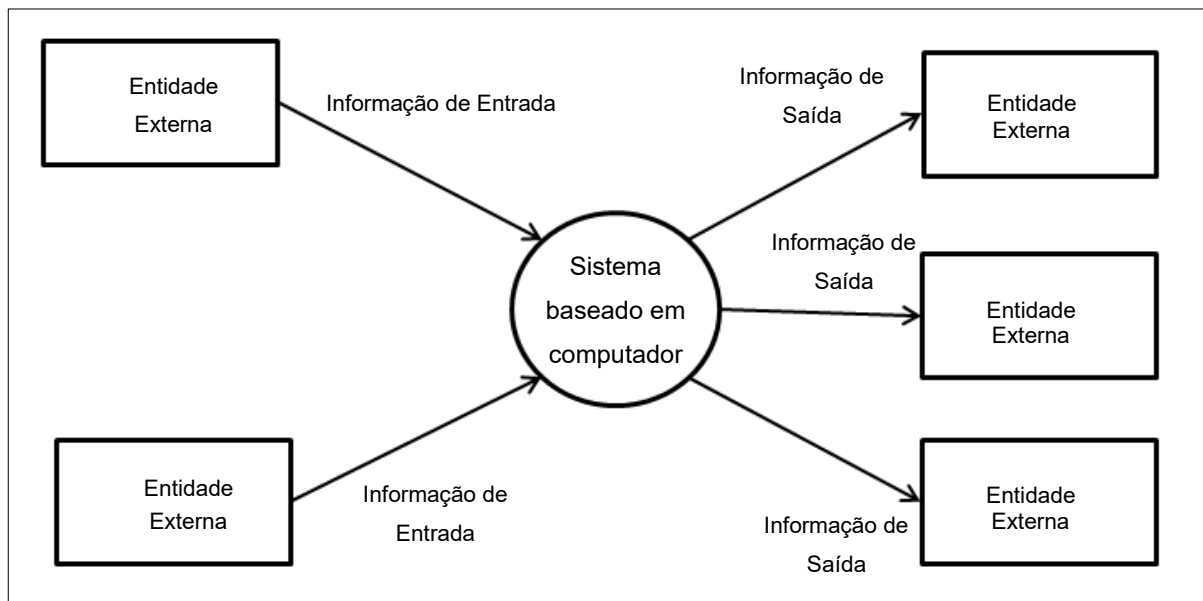
Modelagem estruturada

O foco deste material é voltado inteiramente para conceitos modernos de arquitetura orientada a objeto. Entretanto, é muito importante que o aluno saiba de onde esses padrões surgiram e que tudo começou com a modelagem estruturada. A modelagem estruturada não surgiu de uma documentação específica ou um padrão criado para tal finalidade. DeMarco (1979), citado por Pressman, (1995, p. 227), foi quem introduziu e nomeou termos e símbolos gráficos que possibilitaram ao analista criar modelos de fluxo de dados, heurísticas para o uso desses símbolos, sugeriu um dicionário de dados e narrativa de processamentos que pudesse ser usado como complemento aos modelos de fluxo de informações e apresentou numerosos exemplos que ilustraram o uso desse novo modelo.

Entretanto, à medida que o tempo passava, muitas deficiências foram encontradas em sua execução. Muitas extensões foram implementadas por diversos engenheiros, que tornaram o método um pouco mais robusto, podendo ser aplicado aos problemas de engenharia da época. Essas extensões baseavam-se no fluxo das informações.

O Diagrama de Fluxo de Dados é uma técnica gráfica que descreve o fluxo de informação e as transformações que são aplicadas à medida que os dados se movimentam da entrada para a saída (PRESSMAN, 1995, p. 279).

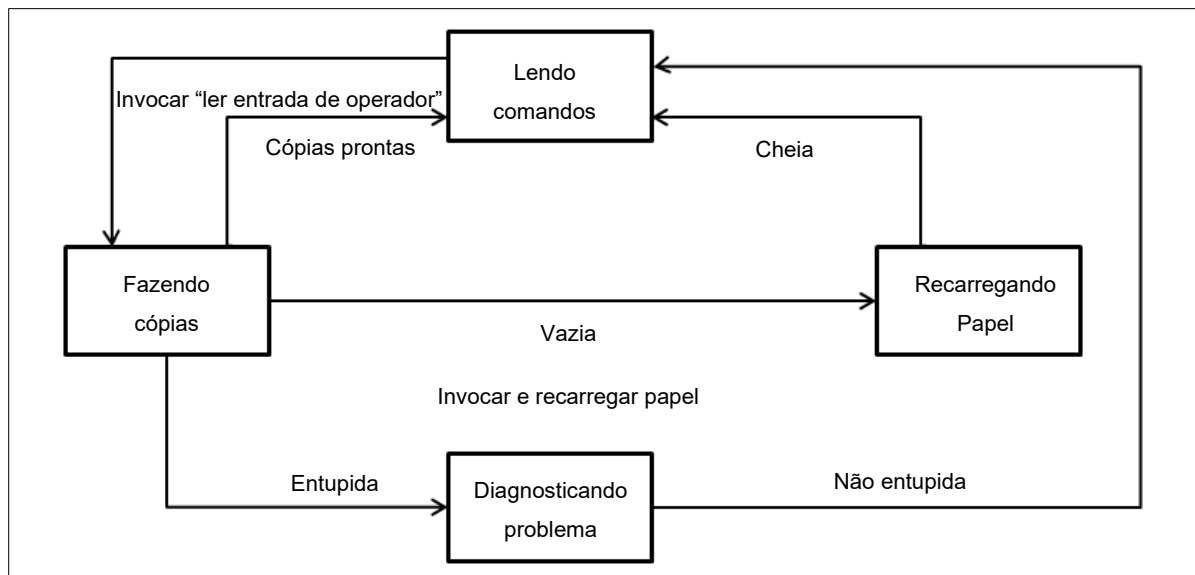
Figura 1. Diagrama de Fluxo de Dados (DFD).



Fonte: Adaptada de Pressman (1995, p. 279).

Além do DFD, utiliza-se a Modelagem Comportamental, estendida da análise estruturada. Essa modelagem gera um Diagrama de Transição de Estados e descreve cada mudança de eventos e estados que ocorrem no sistema.

Figura 2. Diagrama de Transição de Estado simplificando um software de copiadora.



Fonte: Adaptada de Pressman (1995, p. 293).

Uma das ferramentas produzidas pela modelagem estruturada foi o Dicionário de Requisitos (não confundir com Análise de Requisitos) ou Dicionário de Dados. Segundo Pressman (1995, p. 307), ele foi proposto como uma gramática quase formal

para descrever o conteúdo de objetos definidos durante a análise estruturada. Como o processo de descrever os significados dos dados se tornou tedioso, foram criadas notações que simplificavam o dicionário. As principais são:

- » = é composto de
- » + e
- » () opcional
- » {} iteração
- » [] escolha das alternativas
- » | separa as alternativas de []
- » ** comentário
- » @identificador de um depósito

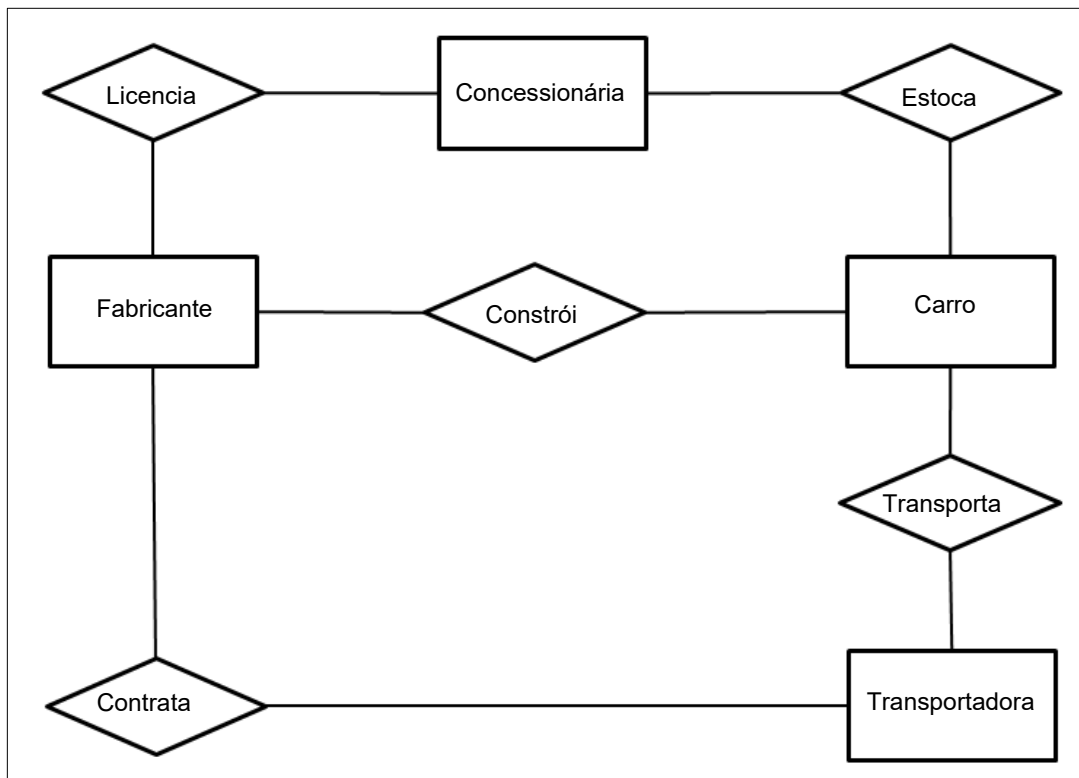
Exemplo da composição de um número telefônico:

- » **número telefônico** = [extensão local | número local]
- » **extensão local** = [19]
- » **número local** = prefixo + número de acesso
- » **prefixo** = [015, 019, 021]

Na utilização em sistemas complexos, o dicionário de dados pode crescer exponencialmente, tornando inviável sua utilização.

Para finalizar a modelagem estruturada e dar ênfase à modelagem orientada a objetos, os repositórios de dados foram representados por meio do Diagrama Entidade-relacionamento. Esse diagrama ajudará, de forma gráfica, identificar os objetos e seus relacionamentos, além de se relacionar com o DFD em seus processos existentes. O DER foi uma transição, muito utilizada até hoje, entre a análise estruturada e a análise orientada a objetos, por mencionar objetos e relacionamentos.

Figura 3. Diagrama Entidade-Relacionamento.



Fonte: Adaptada de Pressman (1995, p. 347).

Em meados de 1980, a análise, a programação e a modelagem começaram a perder força para um novo conceito que estava surgindo: a orientação a objetos.

CAPÍTULO 2

Paradigma orientado a objetos

Como pudemos perceber, o conceito estruturado, seja no planejamento, seja na programação, foi entrando em declínio, dando espaço a uma nova forma de visualizar as “coisas”. Esse conceito surgiu com a necessidade cada vez mais dinâmica de se desenvolver software complexo, multifuncional, multiplataformas etc. Quando um software precisa crescer, toda sua estrutura, como documentação, programação, testes, validações, controle de qualidade, deve acompanhar na mesma velocidade. E é por isso que, atualmente, a orientação a objetos está como um padrão no desenvolvimento de sistemas. Porém, foi preciso que houvesse muitas mudanças de paradigmas para que a *Oriented Object Programming* (OOP) fosse aceita por Engenheiros, Arquitetos e Desenvolvedores. Tudo passou a ser pensado em objetos. A Orientação a Objetos trata de uma modelagem dos objetos do mundo real, estudando-os e criando classes a partir de suas características e comportamentos.

Segundo Rumbaugh (1994, p. 2), pensar baseado em objetos significa que o software será organizado como uma coleção de objetos separados que incorporam tanto a estrutura quanto o comportamento dos dados. Os objetos, portanto, serão os componentes que terão suas características e comportamentos abstraídos do mundo real para o mundo digital, ou seja, um componente concreto ou um conceito se tornará um componente de software. O objeto é um tipo abstrato que contém os dados e os procedimentos que manipulam esses dados. Segundo o dicionário, objeto é tudo que é manipulável e/ou manufaturável, tudo que é perceptível por qualquer um dos sentidos; o que é conhecido, pensado ou representado, em oposição ao ato de conhecer, pensar ou representar. Portanto, objeto é a representação de elementos físicos do mundo real.

O conceito de objetos também requer entendimento para que algumas características, como métodos e atributos, sejam agrupadas, gerando uma classificação. Usando a abstração, podemos classificar alguns objetos de acordo com seus comportamentos e atributos. A esse agrupamento, damos o nome de Classes. Uma classe é composta pela sua descrição, que identifica as propriedades da classe (atributos) e os métodos (comportamento). Assim, por exemplo, em um dado contexto, Esfera, Cubo, Círculo, Quadrado etc. podem se tornar a classe Forma Geométrica. Segundo Rumbaugh (1994, p. 3), cada classe descreve um conjunto infinito de objetos individuais. Cada objeto é dito ser uma instância de sua classe. Cada instância tem seu próprio valor para cada atributo, porém compartilha os nomes de atributos e operações com outras instâncias da mesma classe. As figuras a seguir mostram a classe Veículos, seus atributos e métodos.

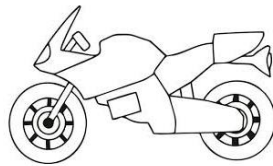
Classes

Figura 4. Classe Veículos.



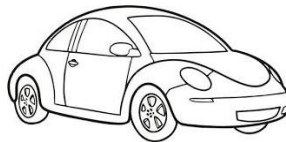
Fonte: <https://brstatic.guiainfantil.com/pictures//2315-dibujos-de-una-lancha-para-colorear-dibujos-de-barcos-para-ninos.jpg>. Acesso em: 27/9/2019.

Figura 5. Classe Veículos.



Fonte: <https://www.artesanatopassoapassoja.com.br/wp-content/uploads/2019/02/moto.jpg>. Acesso em: 27/9/2019.

Figura 6. Classe Veículos.



Fonte: <https://www.artesanatopassoapassoja.com.br/wp-content/uploads/2019/02/fusca-novo.jpg>. Acesso em: 27/9/2019.

Atributos:

- » Tamanho.
- » Local de locomoção.
- » Combustível.
- » Cor.
- » Tamanho.
- » Peso.
- » Cilindradas.

Comportamento:

- » Acelerar.
- » Parar.
- » Mudar marcha.
- » Estacionar.

Os métodos ou operações são procedimentos residentes nos objetos e determinam como eles irão atuar ao receber as mensagens. Métodos podem receber parâmetros e retornar valores.

Outras características, como Polimorfismo, Herança, Encapsulamento, são conceitos fundamentais para que a modelagem possa ser realmente orientada a objetos e serão fontes de estudos em capítulos específicos.

Um ponto importante a se observar e que gera muita discussão é quando a programação orientada a objetos passa a se misturar com o desenvolvimento orientado a objetos. Embora o resultado seja, de fato, um software desenvolvido em uma linguagem de programação orientada a objetos, o desenvolvimento está focado na fase inicial do ciclo de vida do software: análise, projeto e implementação. Segundo Rumbaugh (1994, p. 5), o desenvolvimento baseado em objetos é um processo conceitual independente de uma linguagem de programação até as etapas finais. As linguagens de programação baseadas em objetos só são úteis para remover as restrições devidas à inflexibilidade das linguagens de programação estruturadas.

A abordagem de desenvolvimento baseados em objetos encoraja os desenvolvedores de software a trabalharem e pensarem em termos do domínio da aplicação durante a maior parte do ciclo de vida da engenharia (RUMBAUGH, 1994, p.5). Com isso, é possível apresentar algumas metodologias para a construção de modelos de domínio e para posterior adição de detalhes de implementação. Segundo Rumbaugh (1994, p. 6), as etapas da metodologia podem ser divididas em:

- » Análise: parte do enunciado do problema do mundo real, mostrando propriedades relevantes.
- » Projeto do sistema: o projetista deverá decidir quais características de desempenho devem ser otimizadas, escolher a estratégia de ataque ao problema e realizar alocações experimentais de recursos.
- » Projeto dos objetos: o projetista constrói um modelo de projeto baseado no modelo da análise, mas que contenha detalhes da implementação. O enfoque é a estrutura de dados e os algoritmos para implementação das classes.
- » Implementação: as classes de objetos são traduzidas para uma determinada linguagem de programação, porém essa tradução deve possuir uma relevância menor, pois todas as decisões difíceis devem ser tomadas durante o projeto.

CAPÍTULO 3

UML

Introdução

Quando se trata de modelagem de software orientado, vários recursos foram criados para agilizar todo o ciclo, desde sua ideia, concepção e implantação. Conforme as fronteiras sociogeográficas foram se rompendo, as equipes de desenvolvimento foram se tornando cada vez mais diversificadas em relação a idioma, cultura etc. Isso aconteceu com as linguagens de programação também. Inúmeras tecnologias surgiram, cada uma com suas particularidades, vantagens e desvantagens. Nesse momento, alguma tecnologia deveria surgir para que as pessoas envolvidas pudessem criar projetos de modelagem sem depender de plataforma de desenvolvimento e que todos pudessem falar a mesma língua: foi então que surgiu *Unified Modeling Language* (UML) ou Linguagem de Modelagem Unificada.

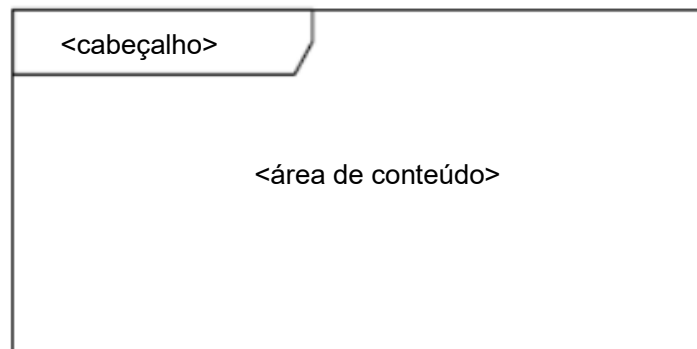
Segundo Guedes (2011, p. 19), a UML é uma linguagem visual, utilizada para modelar softwares baseados no paradigma de orientação a objetos. É uma linguagem de modelagem de propósito geral que pode ser aplicada a todos os domínios da aplicação.

A UML, apesar de o nome conter a palavra linguagem, não é uma linguagem de programação. Ela é uma linguagem visual, que define todas as características do sistema, mesmo antes do software começar a ser programado em uma linguagem de programação.

De acordo com Guedes (2011, p. 19), a UML surgiu da união de três métodos de modelagem: o método de *Booch*, o método OMT (*Object Modeling Technique*), de Jacobson, e o método OOSE (*Object Oriented Software Engineering*), de *Rumbaugh*. Em 1996, o trabalho de Booch, Jacobson e Rumbaugh resultou na primeira versão da UML, e a versão 2.0 foi lançada em 2005. Veja a documentação e histórico em <https://www.omg.org/spec/UML/2.5.1/PDF>.

Segundo OMG (2017, p. 683), um modelo UML consiste em elementos como pacotes, classes e associações. Os diagramas UML correspondentes são representações gráficas de partes do modelo UML e contêm elementos gráficos que representam elementos no modelo UML. Cada diagrama tem uma área de conteúdo. Como opção, pode ter um quadro e um cabeçalho:

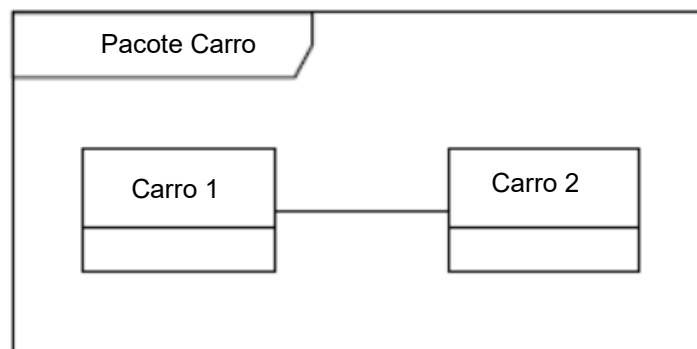
Figura 7. Diagrama UML.



Fonte: OMG (2017, p. 683).

O cabeçalho do diagrama representa o **tipo**, o **nome** e os parâmetros do **espaço de nome incluído** ou o elemento do modelo que possui elementos, representados por símbolos, na área de conteúdo.

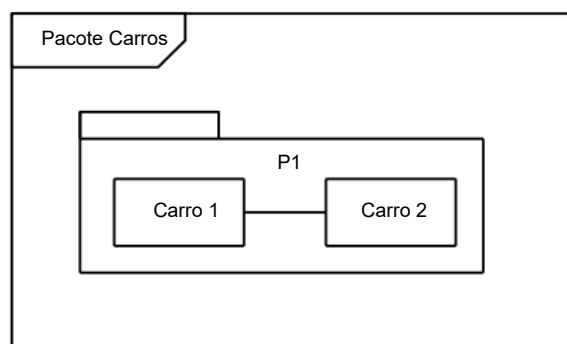
Figura 8. Diagrama de Classe do Pacote P1.



Fonte: OMG (2017, p. 683).

Alguns diagramas representam as propriedades contidas em outro modelo. Sendo assim, serão definidos no espaço de nomes ou *namespace*. Como exemplo, duas classes associadas definidas em um pacote, em um diagrama para o pacote, serão representadas por dois símbolos de classe e um caminho de associação conectando essas duas classes símbolos (OMG 2017, p. 683).

Figura 9. Dois diagramas em um pacote.



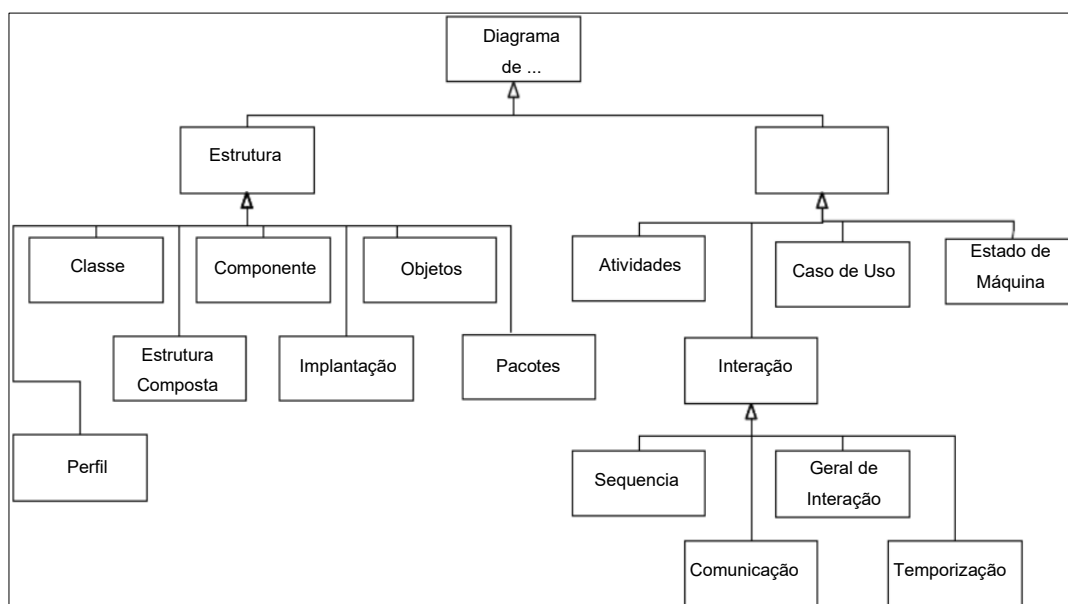
Fonte: OMG (2017, p. 683).

Diagramas

A finalidade da UML é ser uma linguagem de representação gráfica, ou seja, não existe nenhuma tecnologia específica, como uma linguagem de programação qualquer. Por isso, ao longo de sua evolução, vários diagramas foram sendo criados para atender diversos propósitos. É como se o sistema fosse modelado em camadas, sendo que alguns diagramas enfocam o sistema de forma geral, apresentando uma visão externa, como os Diagramas de Casos de Uso. A utilização de diversos diagramas permite que falhas sejam descobertas, diminuindo a possibilidade da ocorrência de erros futuros (GUEDES, 2001, p. 30).

Os diagramas podem ser divididos em Diagrama de Estrutura e Diagrama de Comportamento.

Figura 10. A taxonomia de diagramas de estrutura e comportamento.



Fonte: OMG (2017, p. 685).

Diagrama de casos de uso

Um caso de uso é uma lista de ações ou etapas de eventos que normalmente definem as interações entre uma função de um ator e um sistema para atingir um objetivo. Um caso de uso é uma técnica útil para identificar, esclarecer e organizar os requisitos do sistema. Um caso de uso é composto de um conjunto de possíveis sequências de interações entre sistemas e usuários que define os recursos a serem implementados e a resolução de quaisquer erros que possam ser encontrados. Os diagramas de casos de uso da UML 2 fornecem uma visão geral dos requisitos de uso de um sistema.

Eles são úteis para apresentações aos envolvidos na construção do sistema, mas, para o desenvolvimento real, você descobrirá que os casos de uso fornecem um valor significativamente maior, porque descrevem os requisitos reais.

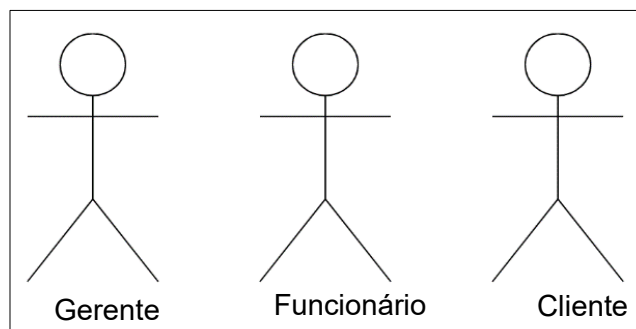
Segundo Guedes (2009, p. 30), um diagrama de caso de uso é o mais geral e informal na UML, utilizado normalmente nas fases de levantamento de requisitos do sistema. Apresenta uma linguagem simples e de fácil compreensão para que os usuários possam ter uma ideia geral de como o sistema irá se comportar.

O diagrama de uso define interações entre atores externos e o sistema, para atingir objetivos específicos. Um diagrama de casos de uso contém alguns componentes principais, como os atores e caso de uso, e alguns não obrigatórios, como as caixas de limites e os pacotes.

Ator

Ator é uma pessoa, organização ou sistema externo que desempenha um papel em uma ou mais interações com seu sistema. Segundo Guedes (2009, p. 53), os atores, além dos usuários do sistema, podem representar, eventualmente, hardware ou um software que interaja com o sistema; são representados por figuras de “bonecos magros” contendo breve descrição logo abaixo do seu símbolo que identifica que papel o ator assume dentro do diagrama.

Figura 11. Representação de atores.



Fonte: Elaborada pelo autor.

Caso de uso ou use case

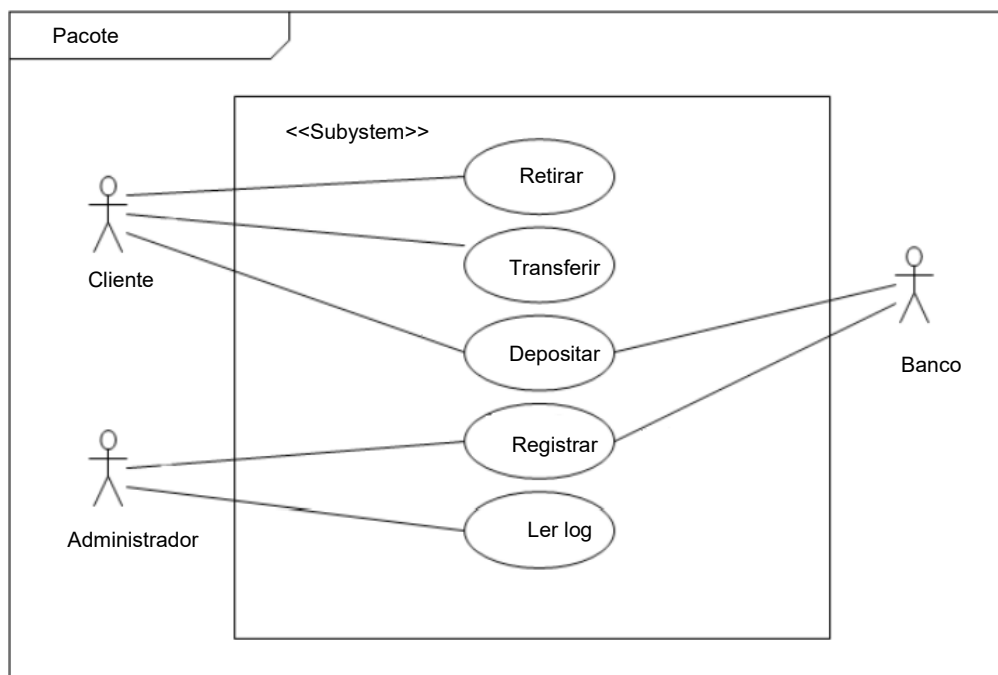
Segundo a própria documentação da UML, os *UseCases* são um meio para capturar os requisitos dos sistemas, ou seja, quais sistemas devem ser feitos. Os principais conceitos especificados nessa cláusula são **Atores**, **UseCase** e **Assuntos**. Cada sujeito de *UseCase* representa um sistema sob consideração a que o *UseCase* se

aplica. Usuários e quaisquer outros sistemas que possam interagir com um assunto são representados como atores. Um *UseCase* é uma especificação de comportamento. Uma instância de um *UseCase* refere-se a uma ocorrência do comportamento emergente que está em conformidade com o *UseCase* correspondente. Tais instâncias são frequentemente descritas por Interações.

Segundo Guedes (2009, p. 54), os casos de uso podem ser considerados primários, que se refere a processos importantes e enfoca um dos requisitos funcionais do software, como realizar um saque ou emitir um extrato. Já o secundário se refere como um processo periférico, como manutenção de um cadastro.

Um *UseCase* é mostrado como uma elipse, contendo o nome do *UseCase*. Uma palavra-chave de estereótipo opcional pode ser colocada acima do nome. Um assunto para um conjunto de *UseCases* (às vezes, chamado de limite do sistema) pode ser mostrado como um retângulo com seu nome no canto superior esquerdo, com as elipses *UseCase* localizadas visualmente dentro desse retângulo.

Figura 12. Pacote que possui um conjunto de *UseCases*, atores e um subsistema.



Fonte: OMG (2017, p. 643).

Documentação do caso de uso

Por meio de linguagem simples, a documentação do caso de uso descreve informações do caso de uso, dos atores, de suas interações, execuções dos atores e sistema. Entretanto, não existe um formato específico ou padronizações.

Segundo sugestão de Guedes (2009, p. 56), em primeiro lugar deve-se fornecer uma descrição para o caso de uso que está sendo documentado. Também deve possuir a informação do *UseCase* Geral, pois, como vimos, pode haver Casos de Uso que herdam as suas características. O campo ator principal se refere ao ator que mais espera o resultado ou que mais interage com o sistema. Podemos também apresentar uma linha com restrições e validações.

Tabela 1. Modelo sugestivo de documentação de Caso de Uso.

Nome do Caso de Uso	Abrir Conta
Caso de uso Geral	-
Ator Principal	Cliente
Atores Secundários	Funcionário
Resumo	Esse Caso de Uso descreve as etapas percorridas por um cliente para abrir uma conta
Pré-condições	O pedido de abertura precisa ser previamente aprovado
Pós- condições	É necessário realizar um depósito inicial
Fluxo Principal	
Ações do Ator	Ações do Sistema
1. Solicitar Abertura de Conta	
	2. Consultar cliente por seu CPF ou CNPJ
3. Informar a senha da cota	
	4. Abrir conta
5. Fornecer valor a ser depositado	
	6. Registrar depósito
	7. Emitir cartão da conta
Restrições/Validações	1. Para abrir uma conta corrente é preciso ser maior de idade 2. O valor mínimo de depósito é R\$ 5,00 3. O cliente precisa fornecer algum comprovante de residência
Fluxo Alternativo – Manutenção no cadastro do Cliente	
Ações do Ator	Ações do Sistema
	Se for necessário, executar Caso de Uso Manter Cliente para gravar ou atualizar o cadastro do cliente
Fluxo de Exceção – Cliente menor de idade	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que este não possui a idade mínima 2. Recusar o pedido de abertura

Fonte: Guedes (2009, p. 56).

Estereótipos

Estereótipos são mecanismos de extensibilidade em UML que permitem aos projetistas estender o vocabulário da UML para criar novos elementos de modelo. Ao aplicar estereótipos apropriados em seu modelo, você pode tornar o modelo de especificação compreensível. Segundo Guedes (2009, p. 68), estereótipos podem atribuir funções extras ou diferentes a um componente, permitindo que este possa ser utilizado para modelar diferentes componentes e situações diferentes das quais foram originalmente projetados.

Existem vários elementos que podem ser aplicados aos componentes da UML e podem ser acessados em: <https://www.omg.org/spec/UML/2.5.1/PDF>, página 681.

Os estereótipos podem ser classificados em:

Classe

- » «**auxiliary**»: aplicado a uma classe que suporta outra classe, geralmente fornecendo mecanismos de controle. A classe suportada é uma classe de foco.
- » «**focus**»: aplicado a uma classe que especifica a lógica principal ou o controle com classes auxiliares que fornecem mecanismos subordinados.
- » «**implementationClass**»: é aplicado a uma implementação de uma classe em que a instância da classe não pode ter mais de uma classe.
- » «**metaclass**»: é aplicado a uma classe cujas instâncias são outras classes que estão em conformidade com a metaclasses.
- » «**type**»: é aplicado a uma classe que descreve o domínio de objetos e suas operações, mas não define a implementação de objetos.
- » «**utility**»: é aplicado a uma classe que não possui instâncias, mas cujos atributos e operações possuem escopo de classe.

Artefato

- » «**document**»: é aplicado a um artefato que representa um documento.
- » «**executable**»: é aplicado a um artefato que pode ser executado em um nó.
- » «**file**»: é aplicado a um artefato que contém código-fonte ou dados.
- » «**library**»: é aplicado a um artefato que é um arquivo de biblioteca estático ou dinâmico.
- » «**script**»: é aplicado a um arquivo que pode ser interpretado por um sistema de computador.
- » «**source**»: é aplicado a um arquivo de origem de um arquivo executável.

Operação

- » «**create**»: é aplicado a uma operação que cria uma instância do classificador; por exemplo, se a operação for um construtor.
- » «**destroy**»: é aplicado a uma operação que destrói uma instância do classificador.

Componente

- » «**buildComponent**»: é aplicado a um componente que especifica um conjunto de componentes para o desenvolvimento organizacional ou no nível do sistema.
- » «**entity**»: é aplicado a um componente que representa um conceito de negócio.
- » «**implement**»: é aplicado a um componente que não possui uma especificação e é uma implementação de uma especificação na qual ele possui uma dependência.
- » «**process**»: é aplicado a um componente que é baseado em transação.
- » «**service**»: é aplicado a um componente que calcula um valor. Esse componente não tem estado.
- » «**subsystem**»: é aplicado a um componente que faz parte de um sistema maior.

Pacote

- » «**framework**»: é aplicado a um pacote que contém elementos reutilizáveis, como classes, padrões e modelos.
- » «**modelLibrary**»: é aplicado a um pacote que contém elementos de modelo para reutilização.
- » «**perspective**»: é aplicado a um pacote que contém apenas diagramas ou subpacotes. Os extratores ignoram pacotes que possuem esse estereótipo aplicado.

Modelo

- » «**metamodel**»: é aplicado a um pacote que contém um modelo que é uma abstração de outro modelo.
- » «**systemModel**»: é aplicado a um modelo ou pacote que contém os modelos que descrevem diferentes perspectivas de um sistema.

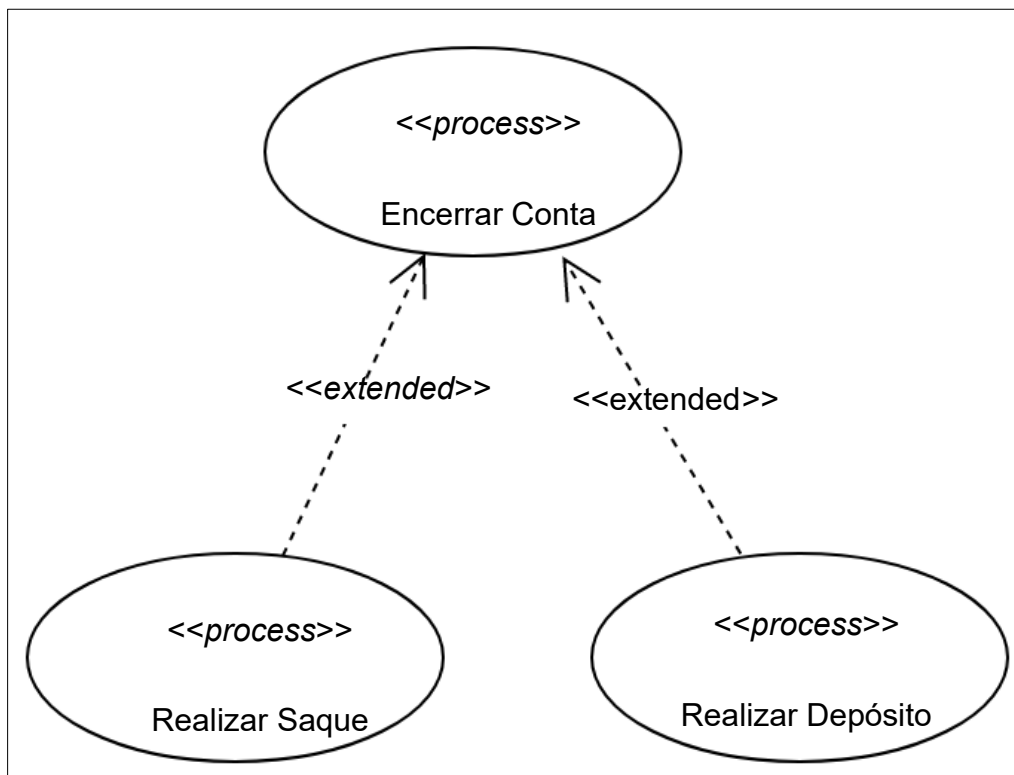
Classificador

- » «**realization**»: é aplicado a um classificador que especifica o domínio dos objetos e sua implementação.
- » «**specification**»: é aplicado a um classificador que especifica o domínio de objetos, não sua implementação.

Uso

- » «**responsibility**»: é aplicado a uma nota que descreve a obrigação de um elemento de modelo para outros elementos do modelo.

Figura 13. Estereótipos em classes.

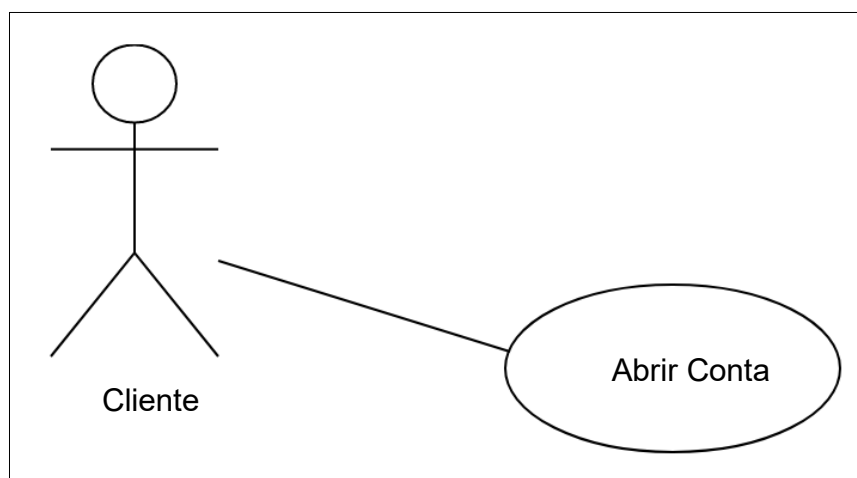


Fonte: Elaborada pelo autor.

Associações

Associações entre atores e casos de uso são indicadas nos diagramas de caso de uso por linhas sólidas. Existe uma associação sempre que um ator está envolvido com uma interação descrita por um caso de uso e são modeladas como linhas que conectam casos de uso e atores entre si, com uma ponta de seta opcional em uma extremidade da linha, em que a ponta de seta costuma ser usada para indicar a direção da chamada inicial do relacionamento ou para indicar o ator primário no caso de uso. As associações podem ser de inclusão extensão e generalização.

Figura 14. Ação entre Ator e UseCase.



Fonte: Guedes (2009, p. 58).

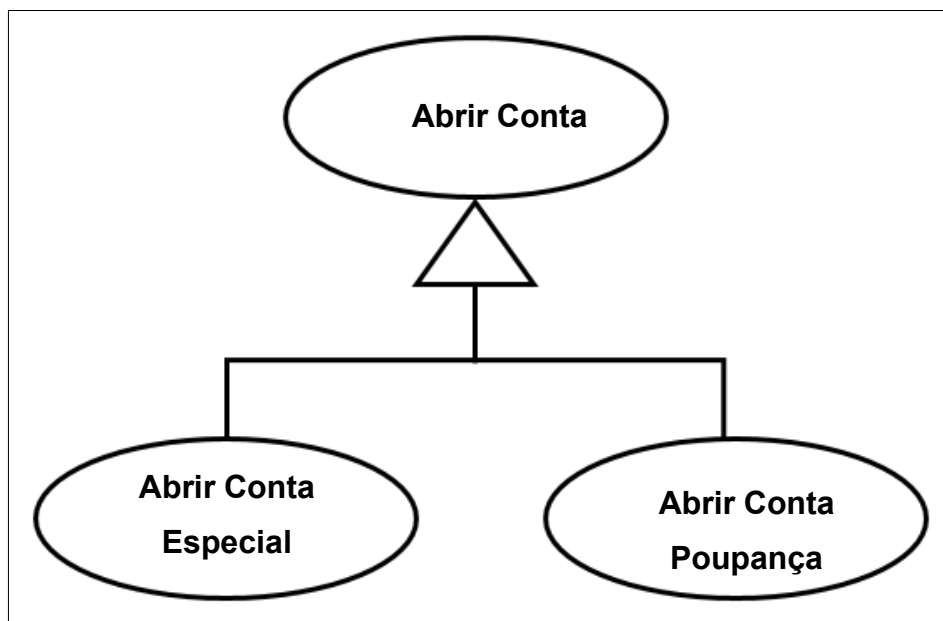
Diagrama de classes

O Diagrama de Classes é um dos diagramas mais importantes da modelagem de Software. Ele utiliza a linguagem UML para ilustrar as classes, seus atributos, métodos e seus relacionamentos.

Generalização

O relacionamento de generalização é uma forma de associação entre casos de uso na qual existem dois ou mais casos de uso com características semelhantes, apresentando pequenas diferenças entre si (GUEDES, 2009, p. 59). Ainda, segundo Guedes (2009, p. 59), os casos de uso especializados herdam também quaisquer possíveis associações de inclusão e extensão que o caso de uso venha a ter. A associação de generalização/especialização é graficamente representada com uma seta, que aponta para o uso de caso geral, partindo do uso de casos especializados.

Figura 15. Generalização.



Fonte: Elaborada pelo autor.

Inclusão

Um relacionamento de inclusão indica obrigatoriedade, ou seja, quando um determinado caso de uso tem relacionamento de inclusão com outro, a execução do primeiro obriga também a execução do segundo (GUEDES, 2009, p. 61). É representada por uma linha tracejada, com uma seta apontando para o uso de caso de inclusão. Algumas ferramentas ainda inserem o texto “*include*”.

Extensão

O relacionamento de extensão se assemelha com a inclusão nas linhas tracejadas, porém a diferença é que representam cenários opcionais, que ocorrerão em situações específicas, quando alguma premissa for satisfeita. A seta também aponta para o caso de uso que utiliza o caso de uso estendido. Algumas ferramentas utilizam a palavra *extended*, no meio da linha tracejada.

Existem muitos outros diagramas que serão utilizados em capítulos mais específicos, como *design* de modelos e de arquitetura. São eles:

- » Diagrama de Objetos;
- » Diagramas de Pacotes;
- » Diagrama de Sequências;

- » Diagrama de Comunicação;
- » Diagrama de Máquinas de Estado;
- » Diagrama de Atividade;
- » Diagrama de Visão Geral de Interação;
- » Diagrama de Componentes;
- » Diagrama de Implantação;
- » Diagrama de Estrutura Composta;
- » Diagrama de Tempo ou Temporização.

CAPÍTULO 4

Cascata, RUP, XP

Metodologias de desenvolvimento

Como todo projeto, o de software não poderia ficar de fora de certas características que definem a necessidade específica de cada situação. Um projeto é criado para resolver problemas diversos, e cada um tem suas prioridades, prazos, orçamentos, pessoas, escopo, ou seja, cada projeto é único e deve ser tratado dessa forma. Por isso, serão apresentadas algumas metodologias de projetos. Projetos que já foram muito utilizados têm sua viabilidade em declínio, e outros estão em constante evolução.

Cascata

O modelo em cascata enfatiza que uma progressão lógica de etapas deve ser feita ao longo do ciclo de vida de desenvolvimento de software. Suas fases devem ser devidamente conhecidas antes do início do projeto, sendo que a fase seguinte será iniciada somente depois que a anterior tiver sido concluída. Embora sua popularidade tenha diminuído nos últimos anos devido à flexibilidade das metodologias ágeis, a natureza lógica do processo sequencial usado no método cascata não pode ser negada e continua sendo um processo de *design* comum na indústria. Segundo Engholm (2013, p. 36), há tempo a metodologia em cascata é adequada somente para projetos pequenos, em que todos os requisitos são conhecidos no início e possui baixa probabilidade de sofrer mudanças.

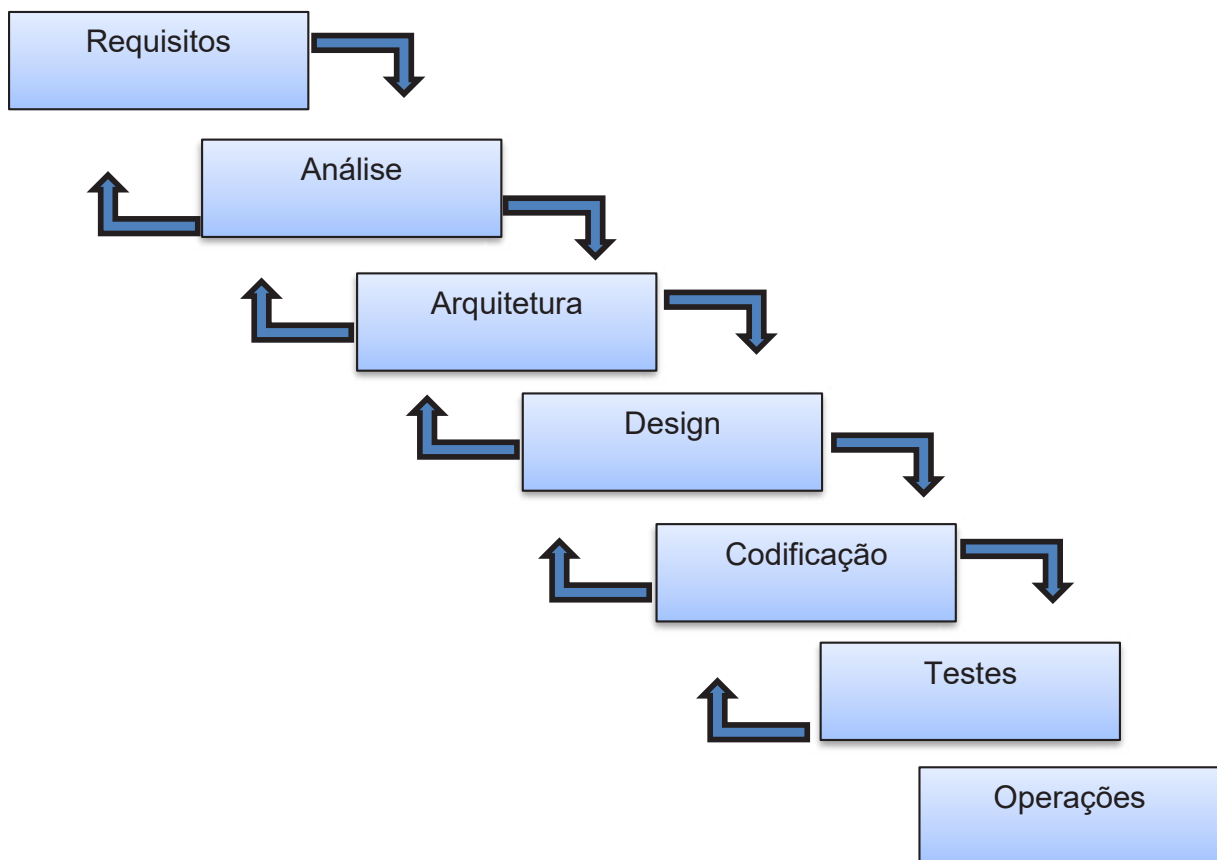
Segundo Sommerville (2011, p. 34), os principais estágios do modelo em cascata refletem diretamente as atividades fundamentais do desenvolvimento:

- 1. Análise e definição de requisitos:** os serviços, restrições e metas do sistema são estabelecidos por meio de consulta aos usuários. Em seguida, são definidos em detalhes e funcionam como uma especificação do sistema.
- 2. Projeto de sistema e software:** o processo de projeto de sistemas aloca os requisitos tanto para sistemas de hardware como para sistemas de software, por meio da definição de uma arquitetura geral do sistema. O projeto de software envolve identificação e

descrição das abstrações fundamentais do sistema de software e seus relacionamentos.

3. **Implementação e teste unitário:** durante esse estágio, o projeto do software é desenvolvido como um conjunto de programas ou unidades de programa. O teste unitário envolve a verificação de que cada unidade atenda a sua especificação.

Figura 16. Modelo em cascata.



Fonte: Elaborada pelo autor.

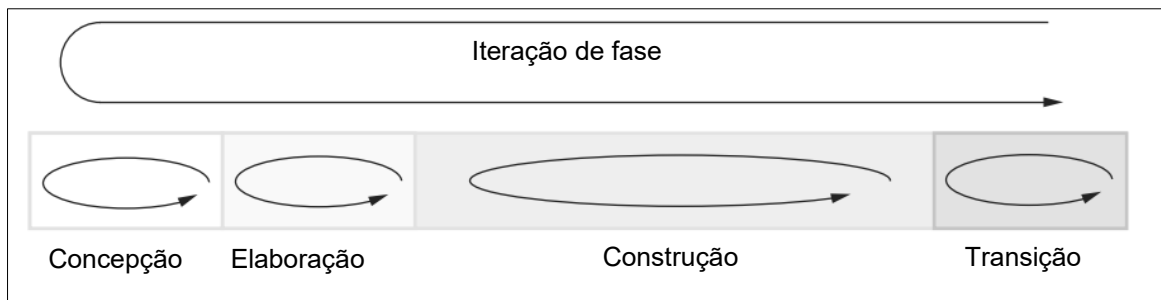
RUP – Processo unificado

O UP (*Unified Process*) é uma opção aberta de mercado, e esse desenvolvimento é realizado com fases incrementais (iterações), sendo que cada iteração possui um escopo definido e produz uma entrega modular. O ciclo de vida incremental tende a reduzir os custos do projeto, a facilitar à equipe que o planejamento dos cronogramas do projeto seja atingido, permitindo melhor acomodação de mudanças no projeto (ENGHOLM, p. 38). O “R” foi uma aquisição feita pela IBM da empresa *Rational Software Corporation*. O RUP pode ser um guia para melhor utilização da linguagem UML.

Segundo Sommerville (2011, p. 34), o RUP é um modelo constituído de fases que identifica quatro fases distintas no processo de software. No entanto, ao contrário do modelo em cascata, no qual as fases são equalizadas com as atividades do processo, as fases do RUP são estreitamente relacionadas ao negócio, e não a assuntos técnicos.

O *Rational Unified Process* divide um ciclo de desenvolvimento em quatro fases estáticas consecutivas:

Figura 17. Fases no RUP.



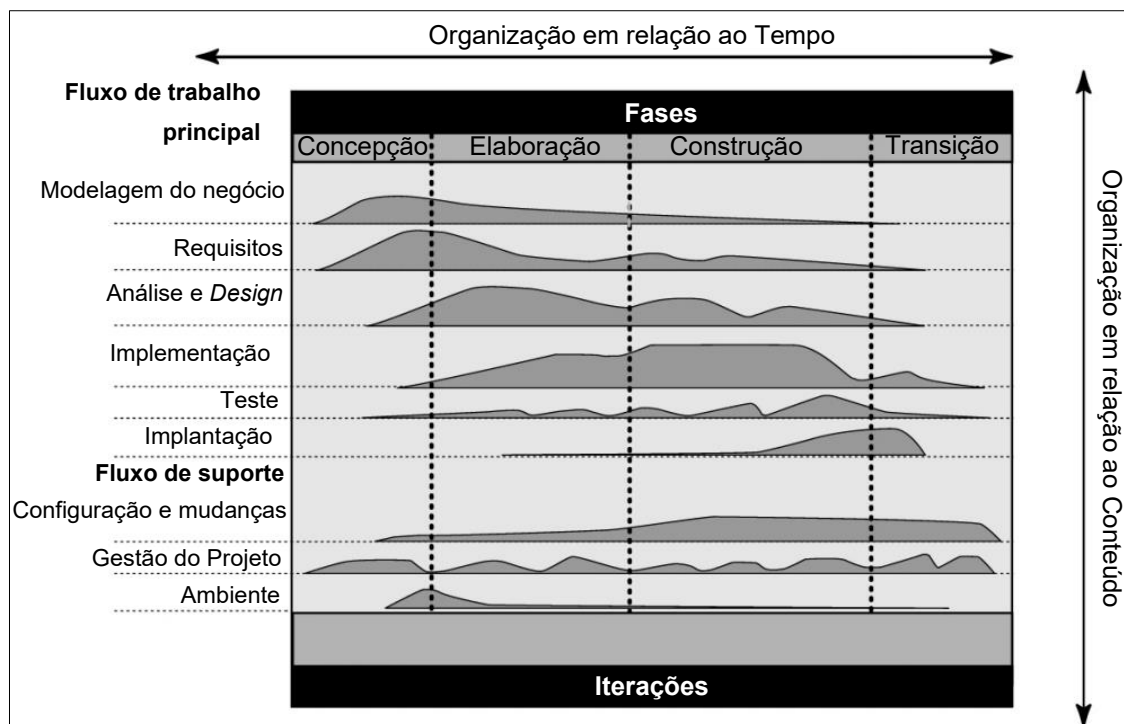
Fonte: Sommerville (2011, p. 35).

- » **Concepção:** define-se um *business case* para o sistema e delimita-se o escopo do projeto. Nessa fase, identificam-se os atores e as entidades externas com as quais o sistema irá interagir.
- » **Elaboração:** as metas da fase de elaboração são desenvolver uma compreensão do problema dominante, estabelecer um *framework* da arquitetura para o sistema, desenvolver o plano do projeto e identificar os maiores riscos do projeto. No fim dessa fase, você deve ter um modelo de requisitos para o sistema, que pode ser um conjunto de casos de uso da UML, uma descrição da arquitetura ou um plano de desenvolvimento do software (SOMMERVILLE, 2011, p. 36).
- » **Construção:** nessa fase, todos os componentes de programação e aplicações são desenvolvidos, integrados e testados. Na fase de construção, todos os recursos são direcionados na gestão de recursos e controle de cronograma.
- » **Transição:** a fase de transição é inserida quando uma parte do sistema já tenha funcionalidade que poderá ser utilizada pelo usuário final. Nessa fase, é comum surgirem novas versões, pois problemas serão corrigidos pelo fato de o usuário estar testando. A liberação para o usuário final deverá ter um bom nível de qualidade aceitável.

A interação poderá ser feita em cada fase, que poderá originar um subproduto do produto final em desenvolvimento, que crescerá de forma incremental de uma interação pra outra, até se tornar o produto final.

O processo RUP poderá ser dividido em duas dimensões, onde o eixo horizontal representa o prazo e é expresso em forma de ciclo, ou fases, ou iterações. O eixo vertical representa as fases estáticas:

Figura 18. Fase do RUP.



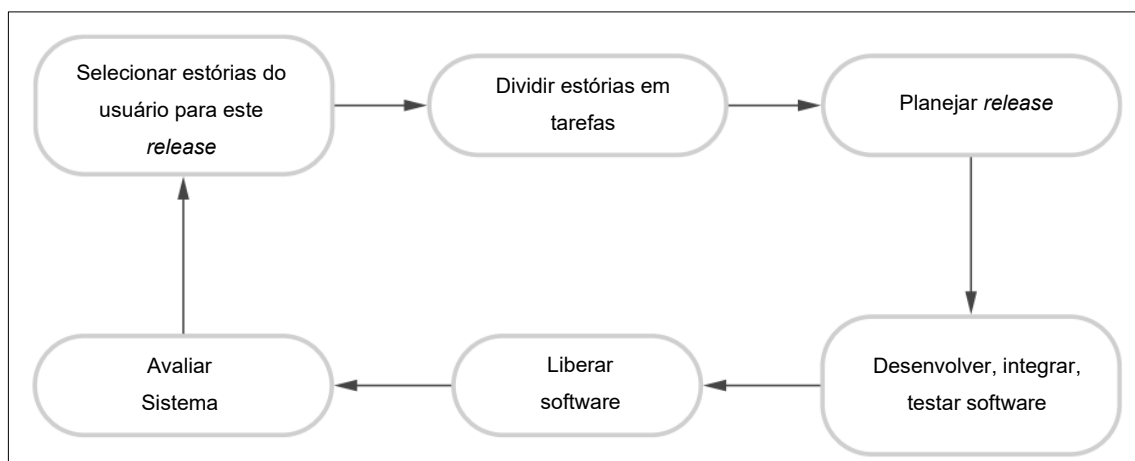
Fonte: https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TP026B.pdf.

Os nove processos principais do RUP representam todas as atividades e estão divididos em seis fluxos de processos principais, Modelagem do negócio, Requisitos, Análise e Design, Teste, Implantação, e em três fluxos de suporte, Configuração e mudanças, Gestão de Projeto e Ambiente.

XP – Extreme Programing

Com o XP, iniciaram-se os primeiros passos para o desenvolvimento ágil de software. O grande ponto forte é o desenvolvimento interativo, levado a níveis extremos, em que, segundo Sommerville (2011, p. 44), várias novas versões de um sistema podem ser desenvolvidas, integradas e testadas em um único dia por programadores diferentes, e seus princípios seguem as seguintes práticas para um ciclo de *release*:

Figura 19. Ciclo de release do XP.



Fonte: Sommerville (2011, p. 44).

CAPÍTULO 1

Classes, objetos, atributos e métodos

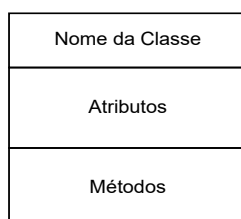
Classes

No mundo real, quando os objetos possuem características semelhantes, normalmente são agrupados e classificados. Um exemplo muito utilizado é a segmentação de mercado, realizada pelas empresas de marketing. Supondo que a empresa queira lançar um produto e esse produto só atenderá pessoas com as mesmas características, comportamentos, gostos musicais, idade etc., desse agrupamento de **indivíduos** surgirá um grupo que será classificado, como, por exemplo, **pessoas de interesse**. Analisando esse exemplo, podemos facilmente abstrair, para um projeto de modelagem, a classe **pessoa** e alguns de seus atributos, como nome, *e-mail*, idade, localização e comportamentos, como andar, comprar, vender, construir, dormir etc. Portanto, Classe será o modelo que foi devidamente classificado com todas as características, comportamento, estado e semântico semelhante, para ser criado um objeto.

Segundo Dall’oglio (2009, p. 90), classe é uma estrutura estática utilizada para descrever objetos mediante atributos (propriedades) e método (funcionalidades). A classe é um modelo ou *template* para criação de objetos. Ela é orientada ao assunto, ou seja, cada classe é responsável por um assunto diferente e possui responsabilidades sobre ele.

Graficamente, uma classe é ilustrada como um retângulo separado por três partes:

Figura 20. Definição gráfica de uma Classe.



Fonte: Elaborada pelo autor.

A parte superior define o nome único da classe no projeto. O espaçamento do meio define todos os atributos e a divisão final, os métodos da classe. O nome da classe deve ser único e de fácil entendimento. Os atributos descrevem as propriedades das classes. Ele é identificado com um nome e um tipo de dado associado. Os métodos definem as funcionalidades das classes e são compostos por nome, tipos de entradas, tipos de saídas e uma lista de argumentos.

Outro exemplo para ilustrar uma Classe é a famosa “planta de uma casa”. Quando o Engenheiro Civil projeta uma casa, ele utiliza toda a técnica, cálculos, ferramentas, para gerar um “modelo”, que é a planta da casa. Ou seja, a planta será o modelo para se criar uma casa. A casa será um “objeto” criado a partir de um modelo.

Ao final, as definições das classes, seus relacionamentos, ilustrados em diagramas, serão o resultado da etapa do projeto de software. Após serem definidas quais classes irão compor o sistema, será possível implementá-las em uma linguagem de programação.

Objetos

Objeto é uma estrutura dinâmica originada com base em uma classe. Após a utilização de uma classe para se criar diversas estruturas iguais a ela, que interagem no sistema e possuem dados nela armazenados, dizemos que estamos criando objetos ou instanciando objetos de uma classe. Diz que objeto é uma instância de uma classe, porque o objeto existe durante um instante dado de tempo, da sua criação até sua destruição (DALL’OGLIO, 2009, p. 93).

Encontrar e descrever um objeto é o principal objetivo da modelagem a objetos. Certamente, um objeto pode ser uma abstração de algo concreto, como uma pessoa, um carro, uma bicicleta, ou pode ser um conceito ou uma ideia, que possua atributos ou propriedades, que descrevem o estado de um objeto do mundo real, ações ou métodos e um identificador ou nome que o defina como sendo único na representação. Seu conceito é compreender o mundo real e oferecer um entendimento preciso para a implementação em sistemas computacionais. Descrever um objeto dependerá do problema encontrado, e não existe uma representação genérica. Por exemplo, o objeto *aluno*, modelado para o domínio de uma escola de natação será diferente quando o domínio for uma escola EaD.¹

¹ Ensino a Distância.

Para identificar objetos, Pressman (1995, p. 321) sugere que os objetos se revelam em uma das seguintes maneiras:

- » Entidades externas: outros sistemas que produzem ou consomem informações a serem usadas por um sistema baseado em computador.
- » Coisas: que fazem parte do domínio do problema. Cartazes, *displays*, relatórios.
- » Eventos²: que ocorrem dentro do contexto de operação do sistema: movimento de robes, transferência de propriedades.
- » Papéis: desempenhados por pessoas que interagem com o sistema: gerente, engenheiro.
- » Unidade organizacional: que são pertinentes a uma organização. Grupos, equipes, departamentos.
- » Lugares: que estabelecem o contexto do problema e a função global do sistema. Piso de fábrica, local de descarga.
- » Estruturas: que definem uma classe de objetos ou, ao extremo, classes relacionadas de objetos: sensores, computadores.



Muitas vezes, o projetista modela um método como sendo um objeto. Utilizando o exemplo de Pressman (1995, p. 322), imagine se o projetista modelasse como o objeto o nome ***“inversão de imagem”***. Pelo contexto do domínio, ***“inversão de imagem”*** seria um método do objeto ***imagem***.

Para nomear um objeto unicamente dentro do projeto, as boas práticas de modelagem recomendam que se inicie com letras minúsculas e que seja um resumo da classe. Também é recomendado que não se utilize acentos, espaços, caracteres especiais. Relacionado a palavras compostas, a mais recomendada é a *camel case (objetoA)*. Nos capítulos que tratam de arquitetura, será demonstrada a especificidade de cada uma.

Quando for implementado por uma linguagem de programação, será no objeto que o processamento ocorrerá. Em termos simples, a classe dará vida ao sistema, por meio da criação de objetos, e estes terão seu tempo de vida definido pelo desenvolvedor ou pelo sistema operacional.

² Esses eventos não estão relacionados com os eventos, dos componentes.

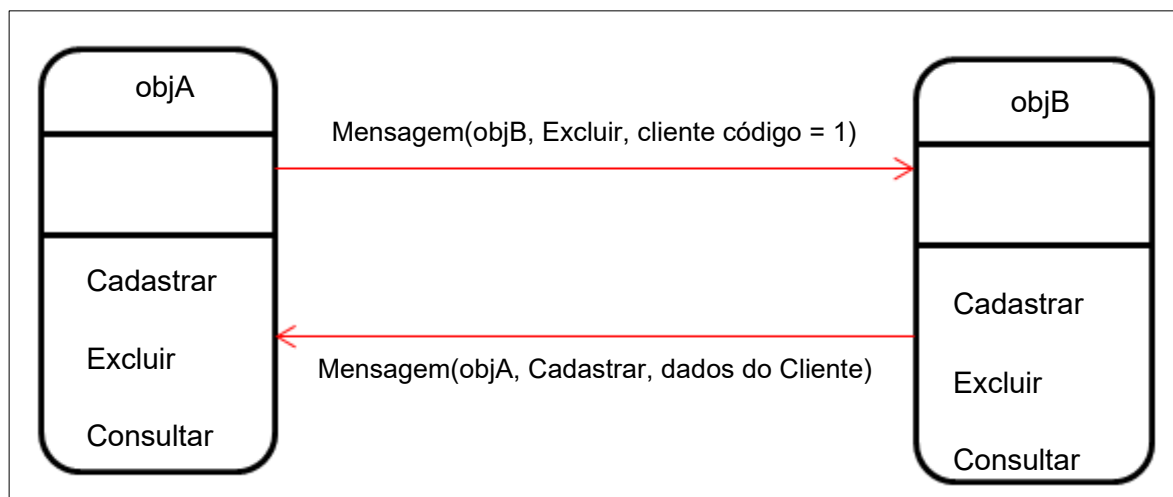
Mensagem entre objetos

Um mecanismo muito importante ao projeto deverá ser definido antes mesmo que ele entre em implementação: a mensagem. Uma mensagem define como os objetos se comunicarão, e ela terá a seguinte estrutura:

» mensagem (destino, operação, parâmetros).

O destino define qual objeto receberá a mensagem. Com as novas tecnologias, os objetos poderão estar no mesmo conjunto de objetos ou em softwares externos. A operação é o método interno ao objeto receptor que será executado assim que a mensagem chegar. Os parâmetros são informações contidas na mensagem usada para que a operação seja bem-sucedida.

Figura 21. Comunicação entre objetos.



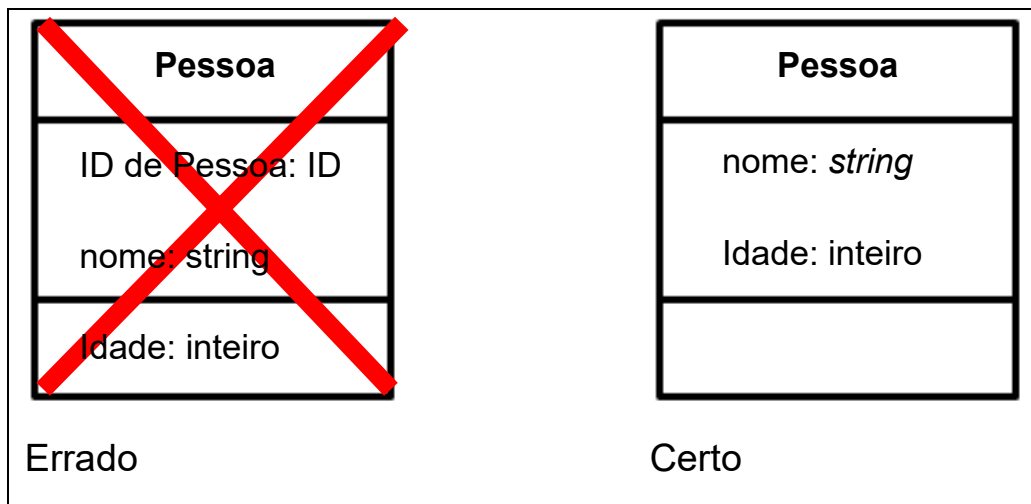
Fonte: Elaborada pelo autor.

Atributos

Os atributos são características coletadas no momento de abstrair a classe do mundo real para o mundo computacional. São propriedades como nome, *e-mail*, telefone, CPF, altura etc. Normalmente, a maioria das classes são criadas com a finalidade de armazenar dados em banco de dados e, por isso, além dos nomes, os atributos devem possuir os tipos de dados, como texto, número, data etc. Outra necessidade para os atributos possuírem um tipo é que, quando forem implementados por uma linguagem de programação, ele se torna uma variável. Em relação aos identificadores únicos, segundo Rumbaugh (1994, p. 35), os identificadores explícitos de objetos não são obrigatórios em um modelo de objetos. Cada objeto tem sua própria e única identidade, as linguagens de programação geram automaticamente identificadores

implícitos para referenciar objetos, portanto não é necessário, nem se devem explicitar identificadores.

Figura 22. identificadores internos x atributos do mundo real.



Fonte: Rumbaugh (1994, p. 35).

Os identificadores internos são simplesmente uma facilidade de implementação e não tem significado no domínio do problema. Por exemplo, número do seguro social, número da chapa do carro e número do telefone não são identificadores internos, pois não têm significado no mundo real. Esses atributos são atributos legítimos. (RUMBAUGH, 1994, p. 36).

O atributo também possui um modificador de acesso que poderá ser público ou privado. Esse modificador de acesso é utilizado para definir como um objeto pode ter acesso aos atributos e métodos de outro objeto. Os modificadores de acesso serão tratados com mais detalhes em Encapsulamento.

Métodos

São procedimentos residentes nos objetos que determinam como eles irão atuar ao serem acionados. Esse acionamento é feito por meio das mensagens, ou seja, pela comunicação entre os objetos. É o método que executa cálculos, manipula dados, retorna mensagens, cria e destrói o objeto, interage com outros objetos etc.

Os métodos são listados no terço inferior da classe, também possuem nomes, tipos de entrada, tipos de saída, tipo de acesso e devem possuir um nome único dentro da classe. Os nomes também devem seguir as boas práticas, porém, por se tratar de uma ação, o método deverá sempre ser criado como verbo: cadastrar, incluir, calcular, excluir.

Cada nome de método deve ser seguido pelos detalhes opcionais, como a lista de argumentos e o tipo de resultado. A lista de argumentos é escrita entre parênteses após o nome, e os argumentos são separados por vírgulas. Deve-se nomear cada tipo de argumento. O tipo de resultado é precedido por dois pontos e não deve ser omitido (RUMBAUGH, 1994, p. 37). Caso o método não receba argumentos, devem-se colocar parênteses vazios.

Existem alguns métodos especiais, como os Construtores, que são responsáveis por criar o objeto quando implementado em alguma linguagem de programação.

Podemos, então, criar uma classe mais completa com todos os conceitos que foram vistos até agora. Digamos que será preciso mapear uma classe, para cadastrar alunos, professores e funcionários de uma escola. Todos eles têm nome, *e-mail*, telefone como atributos e poderão se cadastrar no sistema e alterar seus dados. Para esse modelo, seria criada a seguinte classe:

Figura 23. Classe com atributos e métodos.

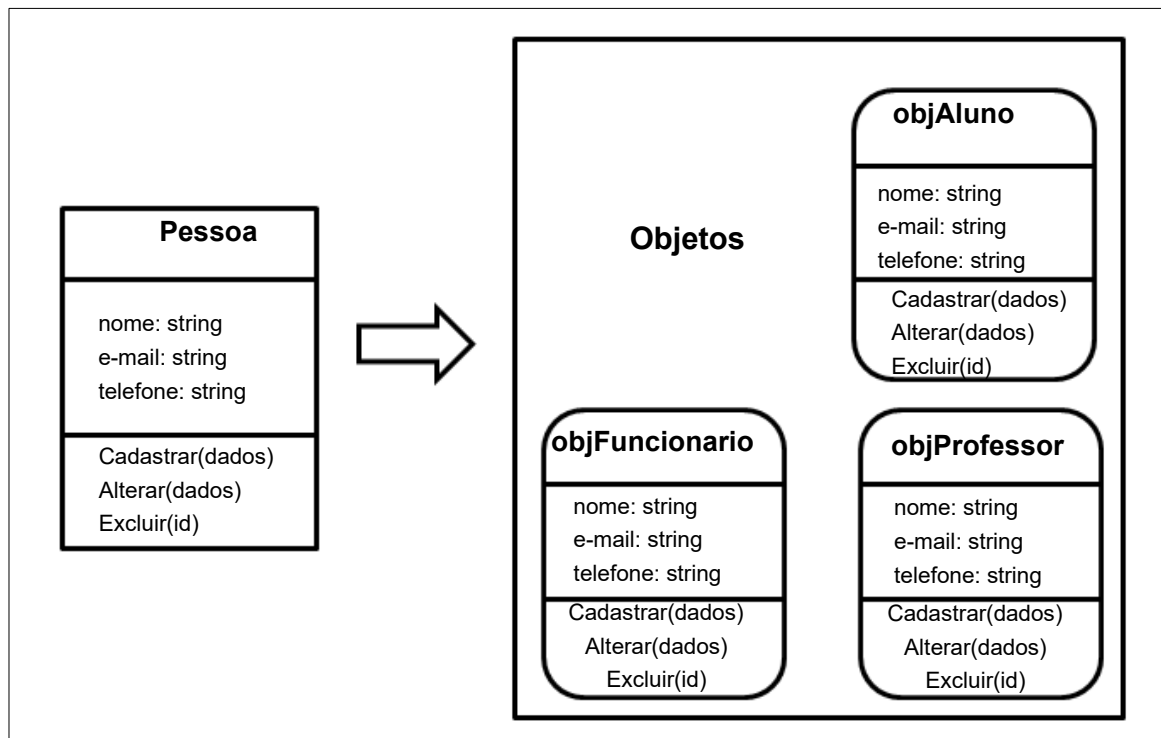
Pessoa
nome: string
e-mail: string
telefone: string
Cadastrar(dados)
Alterar(dados)
Excluir(id)

Fonte: Elaborada pelo autor.

Instanciação

Para definir instanciação, podemos dizer que a classe é modelo estático que tem o propósito de abstrair “coisas” do mundo real e transformá-la em objetos computacionais. Ou seja, quando o projetista escreve a classe e passa para o desenvolvedor implementá-la para gerar os objetos, o sistema irá instanciar os objetos baseados nas classes. Basicamente, é quando o Engenheiro passa para o construtor a planta e esse constrói a casa, baseada nas informações contidas na planta. Baseando-se na classe pessoa, poderíamos utilizar a seguinte representação gráfica para ilustrar uma instanciação:

Figura 24. Instâncias de Classes: objetos.



Fonte: Elaborada pelo autor.

CAPÍTULO 2

Herança, encapsulamento, polimorfismo

Herança e generalização

É um mecanismo que permite o compartilhamento de métodos e dados entre classes, subclasses e objetos. A hereditariedade permite a criação de novas classes programando somente as diferenças entre a nova classe e a classe-pai. Se objetos com propriedades comuns são classificados em uma classe, classes com propriedades comuns são reunidas em uma superclasse.

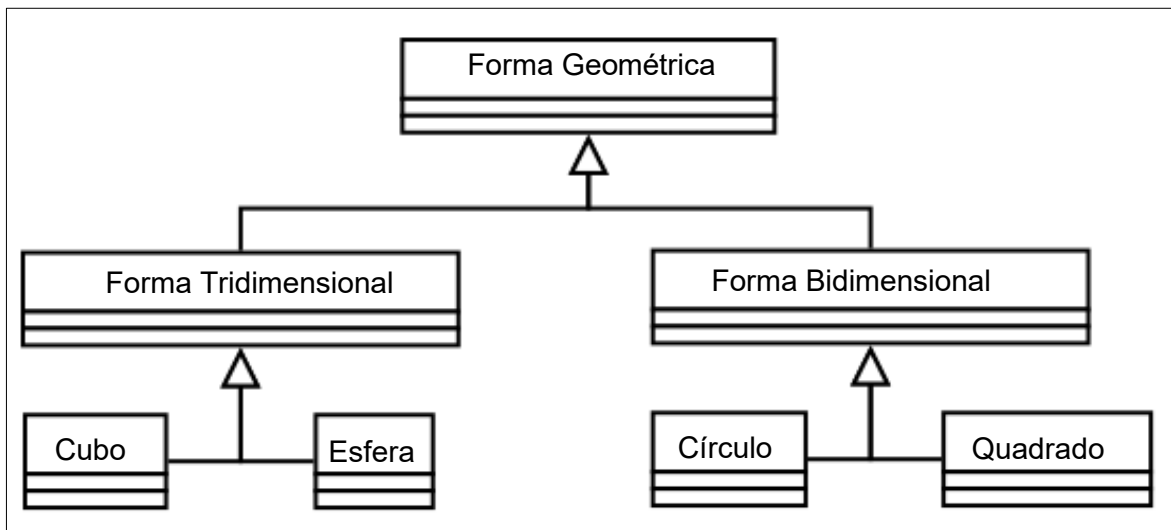
Assim, uma superclasse representa a generalização de suas subclasses, enquanto uma subclasse de uma dada classe representa uma especialização dessa classe. A subclasse Define uma nova classe originada de uma classe maior, a superclasse ou classe-pai.

Os termos ancestral e descendente referem-se à generalização de classes por meio de múltiplos níveis. Uma instância de uma subclasse é simultaneamente uma instância de todas as classes ancestrais. A notação de generalização é um triângulo interligando uma superclasse e suas subclasses. A superclasse é ligada por uma linha ao topo do triângulo. As subclasses são interligadas por linhas a uma barra horizontal ligada à base do triângulo (RUMBAUGH, 1994, p. 54).

O que devemos levar em consideração sobre herança em orientação a objetos é o compartilhamento de atributos e o comportamento entre classes de uma mesma hierarquia. As classes inferiores da hierarquia automaticamente herdam todas as propriedades e métodos das classes superiores, chamadas de superclasses (DALL’OGLIO, 2009, p. 99). Ainda segundo Dall’oglio (2009, p. 99), utilizando a herança em vez de criar uma estrutura de classes totalmente nova, é possível reaproveitar uma estrutura já existente que forneça uma base abstrata para desenvolvimento, provendo recursos básicos e comuns.

Na figura a seguir, temos uma hierarquia de classes de forma geométrica. Ainda não foram utilizados os atributos e métodos, pois precisamos de mais detalhes sobre o domínio.

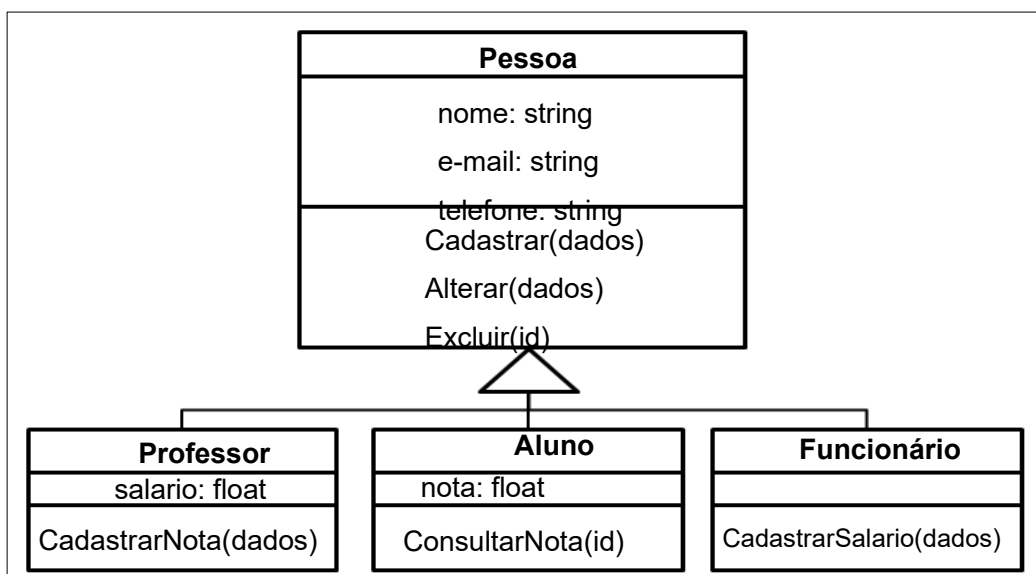
Figura 25.



Fonte: Elaborada pelo autor.

Vamos utilizar o exemplo da escola. Foram mapeadas as classes para **professores**, **alunos** e **funcionários**. Em princípio, todos puderam ser modelados em uma única classe, Pessoa. Isso foi possível porque todos tinham os mesmos atributos e métodos. Porém, para avançarmos mais no exemplo, vamos adicionar aos professores a possibilidade de cadastrar a nota do aluno e, para o funcionário, a possibilidade de cadastrar o salário do professor. Já podemos perceber que, se criarmos esses atributos e métodos na superclasse, será herdada a funcionalidade de cadastrar a nota e o salário do professor. Regras básicas da segurança da informação estariam sendo violadas: a confidencialidade, integridade e disponibilidade. Veja na figura a seguir como ficaria o diagrama de classes depois dessa atualização:

Figura 26.



Fonte: Elaborada pelo autor.

Podemos observar que, nas classes Professor, Aluno e Funcionário, não estão explícitos os atributos e métodos da Superclasse Pessoa. Porém, quando um desenvolvedor olha para esse diagrama, saberá que deverá ser implementada uma herança, por meio da linguagem de programação que esteja utilizando.

Encapsulamento

O Encapsulamento está vinculado à forma como o objeto vai expor suas propriedades e métodos. Muitos autores associam o encapsulamento a uma caixa preta. Um bom exemplo de encapsulamento é o carro. O carro foi projetado para que os motoristas tenham acesso às ações de acelerar, frear, parar, por meio de instrumentos dos quais as pessoas que estão manipulando não precisam saber como é o mecanismo de aceleração ou os componentes que fazem o carro parar. Eles sabem apenas manipular um “método” que está disponível que faça essa função ao ser acionado.

Segundo Dall’oglio (2009, p. 107), o encapsulamento é um mecanismo que provê proteção de acesso aos membros internos de um objeto. Dessa forma, existem certas propriedades de uma classe que devem ser tratadas exclusivamente por métodos dela mesma. As propriedades não devem ser acessadas diretamente de fora do escopo de uma classe, pois, dessa forma, a classe não oferece mais garantias sobre os atributos que contém. Com o encapsulamento, segundo Rumbaugh (1994, p. 10), há implementação de um objeto sem que isso afete as aplicações que o utilizam. Para melhorar o desempenho, pode-se modificar a implementação de um objeto, eliminar um erro ou consolidar um código.

Voltando ao exemplo do carro, o engenheiro que o projetou definiu que a única maneira, seja por diversos fatores, de se acelerar o carro seria pelo pedal do acelerador. Entretanto, deixo essa disponibilidade de acelerar exposta para que os demais passageiros, por meio do botão de abrir o vidro, por exemplo, pudessem acelerar o carro. Qual garantia de segurança seriam oferecidas nesse contexto?

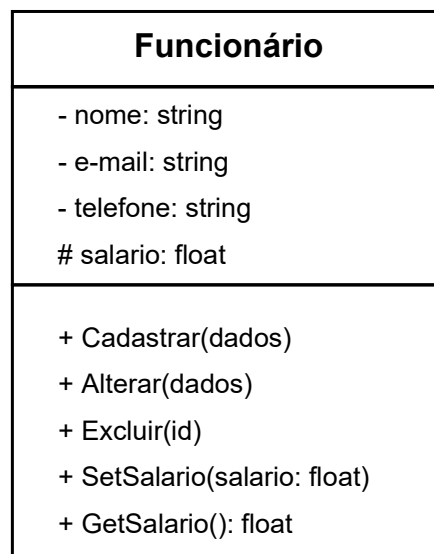
A visibilidade é uma forma de garantir como os atributos e métodos podem ser acessados e, para isso, na modelagem de objetos, existem três formas: *private*, *public* e *protected*.

- » ***private***: membros declarados como *private* podem ser acessados somente dentro da própria classe. Não podem ser acessadas pelos seus descendentes nem a partir do programa que faz uso dessa classe. Na UML, é simbolizada com um (-) em frente ao membro (DALL’OGLIO, 2009, p. 108).

- » ***protected***: membros declarados como *protected* podem ser acessados dentro da própria classe que foram acessados, mas não podem ser acessados pelo programa que faz uso dessa classe. Na UML, é simbolizada com um (#) em frente ao membro (DALL’OGLIO, 2009, p. 108).
- » ***public***: membros declarados como *public* poderão ser acessados livremente a partir da própria classe em que foram declarados, a partir de classes descendentes e a partir de programas que fazem uso dessa classe. Na UML, é simbolizada com um (+) em frente ao membro (DALL’OGLIO, 2009, p. 108).

A visibilidade é uma característica obrigatória nas linguagens de programação orientadas a objetos. Entretanto, algumas podem ter suas particularidades, que serão analisadas na Unidade IV. A seguir, veja uma classe modelada utilizando a visibilidade:

Figura 27. Classe com visualização de membros.



Fonte: Dall’oglio (2009, p. 111).

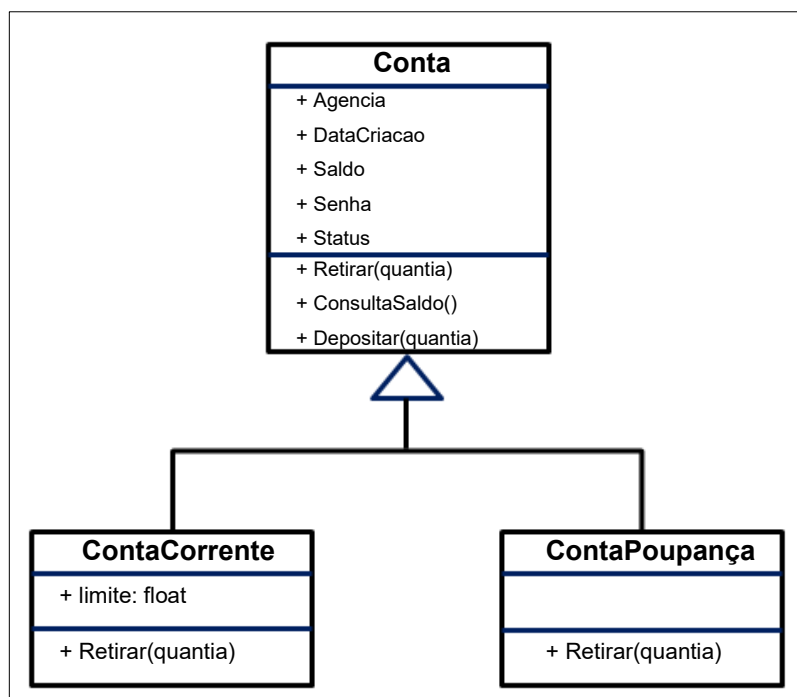
Polimorfismo

Segundo Dall’oglio (2009, p. 101), o significado da palavra polimorfismo nos remete a “muitas formas”. Já em Orientação a Objetos, polimorfismo é o princípio que permite que classes derivadas de uma mesma superclasse tenham métodos iguais, ou seja, com a mesma nomenclatura e parâmetros, mas com comportamentos diferentes, redefinidos em cada uma das classes-filhas.

Guedes (2009, p. 50) acrescenta que polimorfismo está associado à herança e polimorfismo trabalha com a redeclaração de métodos previamente herdados por uma classe. Esses métodos, embora com a mesma assinatura, ou seja, o mesmo nome, mesmos parâmetros de entrada, diferem de alguma forma da implementação utilizada na superclasse, sendo necessário reimplementá-lo na subclasse.

O polimorfismo é um assunto extenso e em cada arquitetura poderá existir uma forma de implementá-lo. A grande maioria obriga o desenvolvedor a escrever exatamente igual sua nomenclatura, parâmetros de entradas e parâmetros de saída, ficando diferente, apenas, o código que define seu comportamento ou processo. Veja no exemplo, criado por Dall’oglio (2009, p. 99) e adaptado pelo autor, o método **Retirar** existente nas três classes: na superclasse **Conta** e nas subclasses **ContaCorrente** e **ContaPoupança**.

Figura 28. Herança para explicar Polimorfismo.



Fonte: Dall’oglio (2009, p. 99).

O método **Retirar** da Super Classe **Conta** apenas efetua a retirada, deixando as demais lógicas para as Subclasses:

- » Retirada (quantia):
 - › Consulta a quantia atual.
 - › Subtrai da quantia atual a quantia passada no parâmetro.
- » Fim de Retirada.

O método **Retirar** da Subclasse **ContaPoupança** verifica se existe saldo e, caso exista, efetua a retirada:

- » Retirada(quantia):
 - › Consulta Saldo.
 - › Se a quantia for menor que o Saldo:
 - Executa o método **Retirada** da Super Classe Conta, passando a quantia.
 - › Senão:
 - Emite um alerta ao e não efetua a retirada.
- » Fim de Retirada.

Já no método **Retirar** da classe **ContaCorrente**, existe uma lógica que verifica o limite, ou seja, a retirada não poderá exceder um limite especificado, e calcula o imposto:

- » Retirada(quantia):
 - › Consulta Saldo.
 - › Consulta Limite.
 - › Calcula imposto.
 - › Se a quantia for menor que o Saldo + Limite:
 - Executa o método **Retirada** da Super Classe Conta, passando a quantia + o imposto.
 - › Senão:
 - Emite um alerta ao e não efetua a retirada.
- » Fim de Retirada.

Podemos observar que o método Retirar possui as mesmas características (nome, parâmetro) em ambas Subclasses, porém comportamento diferente em ambas.

CAPÍTULO 3

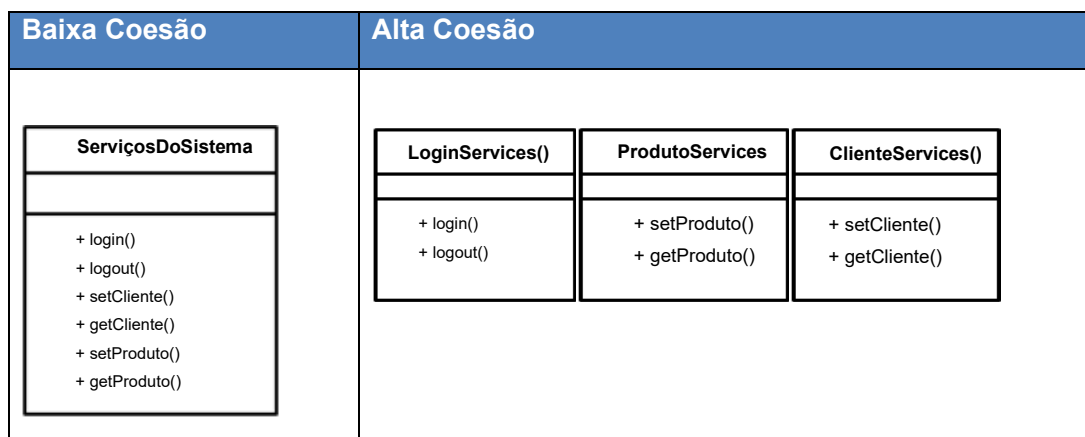
Coesão, acoplamento, abstração, interface

Coesão

Coesão está relacionada ao nível em que os métodos e atributos de uma classe estão relacionados com apenas um propósito no sistema. O ideal é ter classes com alta coesão (ENGHOLM, 2013, p. 112).

Quanto mais coesas as classes, mais relação forte representam, onde seus membros estão ligados e possuem um objetivo em comum. Se os membros não forem extremamente necessários àquela classe, não devem estar presentes no código, assim como as funcionalidades supérfluas e suas responsabilidades, o que torna o código fácil de dar manutenções sem afetar outras funcionalidades. Quando isso acontecer, significa que a coesão é baixa, e a responsabilidade deve ser transferida para a elaboração de uma nova classe.

Figura 29. Exemplo de alta e baixa coesão.

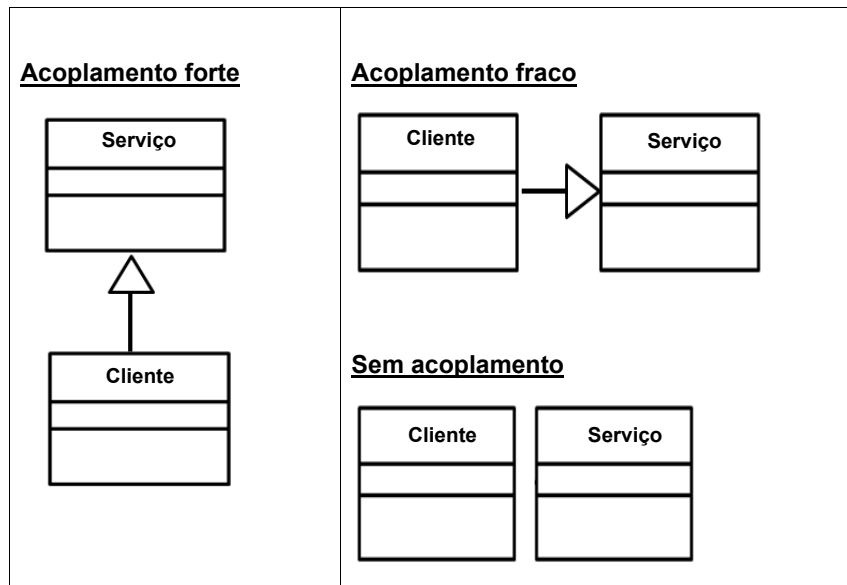


Fonte: Engholm (2013, p. 112).

Acoplamento

Representa o quanto uma classe é dependente da outra. A engenharia de software sugere como boa prática que se tenha classes com acompanhamentos fracos entre elas (ENGHOLM 2013, p. 112). Quando existe baixo acoplamento, o sistema fica flexível, reutilizável e organizado. O alto acoplamento gera problema de baixa coesão. Quando se altera o código de um bloco, corre-se o risco “alto” de impacto em outros módulos.

Figura 30.

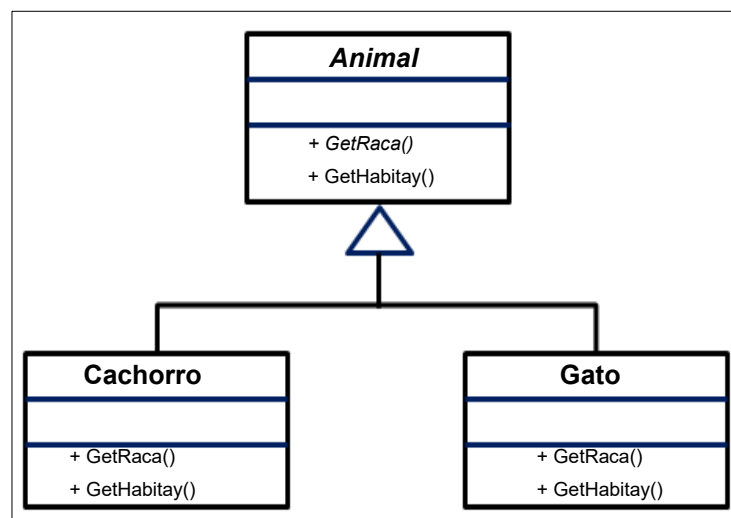


Fonte: Engholm (2011, p. 116).

Abstração

Classes abstratas são classes utilizadas apenas como um rascunho de como as classes que herdarem dela devem se comportar. Além disso, essas classes não poderão ser instanciadas. Seu propósito é fornecer uma definição comum de uma Super Classe que pode ser compartilhadas por várias Sub Classes. A Abstração também está ligada à Herança.

Figura 31. Abstração.



Fonte: Elaborada pelo autor.

Na figura 31, temos uma classe, **Animal**, e duas subclasses, **Gato** e **Cachorro**. Percebam que a notação para a abstração é a fonte em itálico, ou seja, a classe **Animal** é uma classe abstrata, e o método **GetRaca()** é um método abstrato.

Quando o desenvolvedor for instanciar uma classe, deverá utilizar sempre as subclasses. Todas as arquiteturas de linguagem de programação irão emitir mensagens de erro toda vez que for utilizada uma classe Abstrata como instância. Sempre também que instanciar as subclasses, deverá instanciar, obrigatoriamente, o método **GetRaca()**, pois, pelo fato de ser um método abstrato, torna-se obrigatório seu uso, mesmo que não seja implementado nenhum comportamento ou código na subclasse. A superclasse não deverá possuir implementação alguma, somente sua assinatura, ou seja, sua declaração.

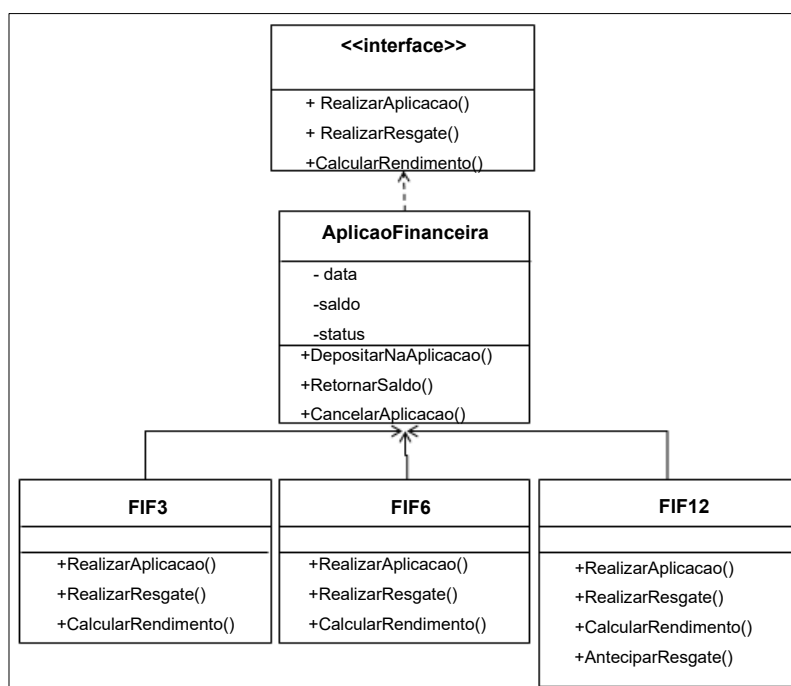
Interface

A interface é um recurso muito utilizado em OOP, para “obrigar” um determinado grupo de classes a ter métodos ou propriedades em comum para existir em um determinado contexto, entretanto os métodos podem ser implementados em cada classe de uma maneira diferente. Uma interface é um contrato que, quando assumido por uma classe, deve ser implementada. Esse recurso é importante em linguagens que não suportam herança múltipla de classes. Além disso, você deve usar uma interface se deseja simular a herança para estruturas, porque realmente não podem herdar de outra estrutura ou classe.

Cuidado para não confundir herança com Interfaces, pois:

- » a Herança limita uma classe a herdar somente uma classe pai por vez;
- » em Interfaces, é possível que uma classe implemente várias interfaces ao mesmo tempo.

Figura 32. Utilização de Interface.

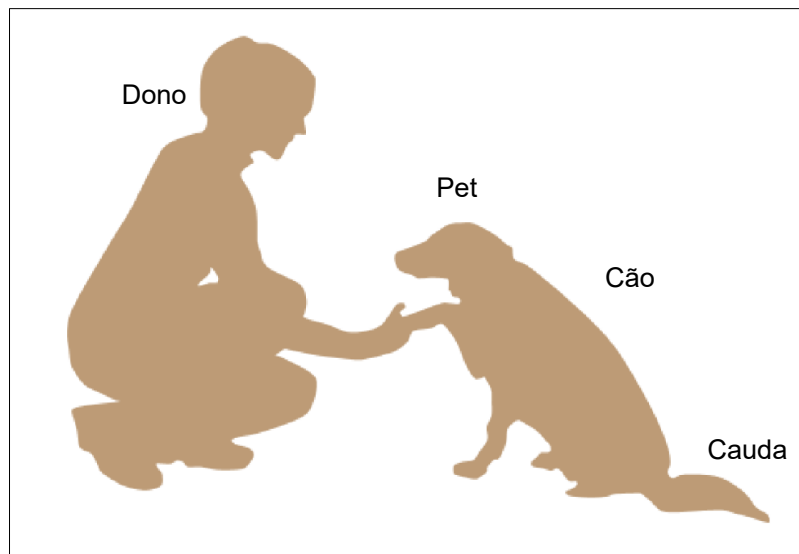


Fonte: Engholm (2013, p. 161).

CAPÍTULO 4

Relacionamento entre objetos: associação, agregação, composição

Figura 33. Ilustrando relacionamentos.



Fonte: <https://cdn.visual-paradigm.com/guide/uml/uml-aggregation-vs-composition/uml-association-aggregation-composition.png>. Acesso em: 27/9/2019.

Para iniciar esse capítulo, segue um exemplo bem didático adaptado de <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition>:

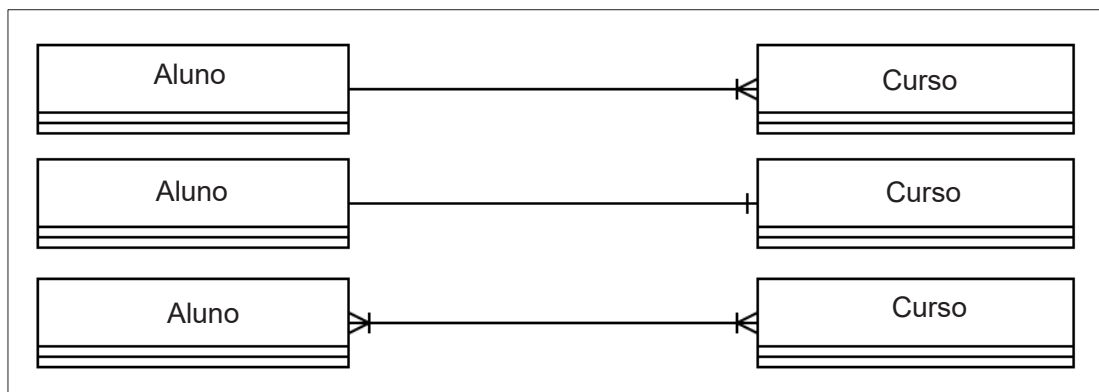
- » **Associação** – donos alimentam seus *pets*; *pets* são alimentados pelos seus donos.
- » **Agregação/Composição** – uma cauda é uma parte do *pet*.
- » **Herança/Generalização** – um cão é um *pet*.

Associação

Associação é a forma como duas classes ou objetos se relacionam, ou seja, define como eles se interagem ou colaboram. Ou seja, se duas classes precisam se comunicar, deverá haver um *link* entre elas, chamado de conector. Essa associação é utilizada na comunicação entre os objetos para que ambos troquem funcionalidades, características etc., para poderem executar operações em aplicações.

A associação pode ser representada por uma linha entre essas classes com uma seta indicando a direção da navegação. Se a seta estiver nos dois lados, a associação é bidirecional. Podemos indicar a multiplicidade de uma associação adicionando *símbolos de multiplicidade* à linha que denota a associação. A multiplicidade se dá conforme a cardinalidade dessa associação, podendo ser um-par-um, um-para-muitos, muitos-para-um, muitos-para-muitos.

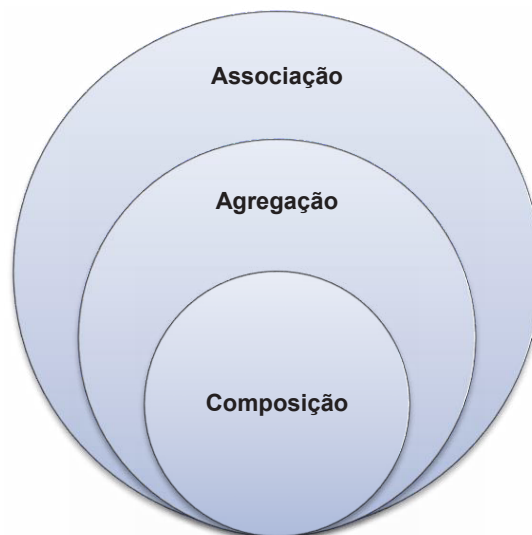
Figura 34. Multiplicidade em relacionamentos.



Fonte: Elaborada pelo autor.

A Composição e a Agregação são formas de como esses objetos se relacionam.

Figura 35. Organização entre Associação, Agregação e Composição.



Fonte: Elaborada pelo autor.

Agregação

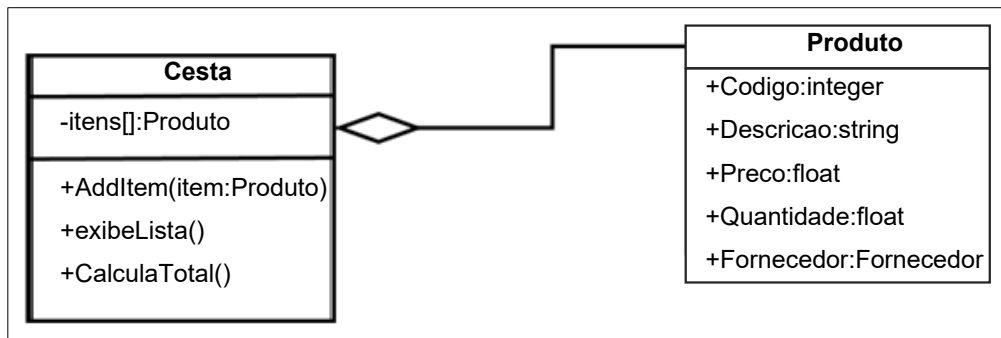
A agregação é um relacionamento “parte-todo” ou “uma-parte-de” no qual os objetos que representam os componentes de alguma coisa são associados a um objeto que

representa a estrutura inteira. A Agregação é desenhada como uma associação, exceto por um losango que indica a extremidade estrutural do relacionamento (RUMBAUGH, 1994, p. 53).

Segundo Dall’oglio (2009, p. 118), na agregação um objeto agrega outro objeto, ou seja, torna um objeto externo parte de si mesmo pela utilização de um de seus métodos. Assim, um objeto-pai poderá utilizar funcionalidades do objeto agregado.

Vamos utilizar o exemplo de Dall’oglio (2009, p. 119), em que ele utiliza uma classe chamada Cesta que representa o todo e a classe Produto representa cada uma das partes. Uma cesta é composta de inúmeros produtos (instâncias da classe Produto).

Figura 36. Agregação.

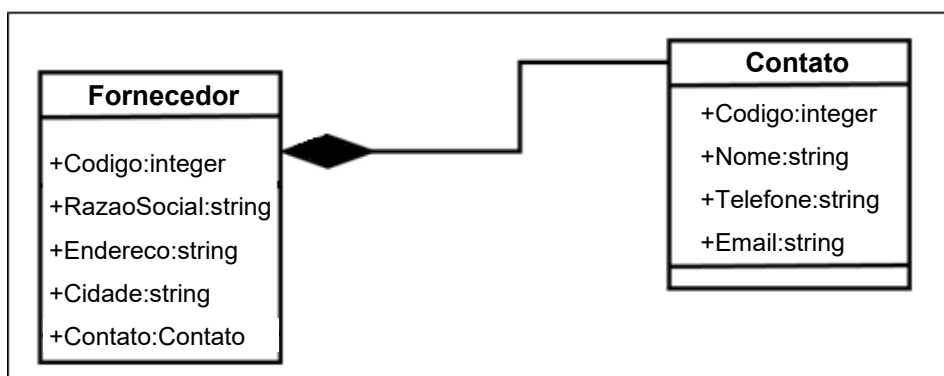


Fonte: DALL’OGLIO (2009, p. 119).

Composição

A composição é um tipo especial de relação de agregação em que as partes componentes não existem, exceto como parte da composição. Em uma Composição, duas entidades são altamente dependentes umas das outras, elas representam parte do relacionamento e, quando há composição entre duas entidades, o objeto não pode existir sem a outra entidade.

Figura 37.



Fonte: DALL’OGLIO (2009, p. 122).

Agregação *versus* composição

Segundo Dall’oglio (2009, p. 118), a diferença em relação à agregação é que, na composição, o objeto-pai ou o “todo” é responsável pela criação e destruição de suas partes. O objeto-pai “possui” as instâncias de suas partes. Na agregação, as instâncias do “todo” e das “partes” são independentes.

CAPÍTULO 1

Padrões de design e arquiteturas

Padrões de projeto são soluções para problemas de design de software orientados a objetos. Os padrões de projeto geralmente visam resolver os problemas de geração e interação de objetos, e não problemas de escala maior na arquitetura do software. Eles oferecem soluções generalizadas na forma de modelos que podem ser aplicados a problemas do mundo real.

Segundo Gamma (2000, p. 17), projetar software orientado a objetos é difícil, mas projetar software reutilizável orientado a objetos é ainda mais complicado. Você deve identificar objetos pertinentes, fatorá-los em classe no nível certo de granularidade, definir as interfaces das classes, as hierarquias, as heranças e restabelecer as ligações entre eles. O seu projeto deve ser específico para o problema a resolver, mas também genérico o suficiente para atender a problemas e requisitos futuros.

O design de sistemas pode ser visto como um processo de definir a arquitetura, os componentes, os módulos, as interfaces e os dados para determinado sistema com a finalidade de satisfazer aos seus requisitos (ENGHOLM, 2011, p. 229).

Para se obter um bom design, é preciso que o detalhamento do diagrama de classes esteja em seu nível máximo, pois problemas de negócios deverão ser resolvidos nessa etapa. Segundo Engholm (2011, p. 230), os princípios da Orientação a Objetos devem ser explorados, como Coesão, Encapsulamento, Herança, Polimorfismo, Interfaces, Acoplamento, Composição.

Segundo Leite (2000), a arquitetura de software é uma especificação abstrata do funcionamento deste em termos de componentes que estão interconectados entre si. Ela permite especificar, visualizar e documentar a estrutura e o funcionamento do software independentemente da linguagem de programação na qual ele será implementado.

Também, por meio da arquitetura, *frameworks* de desenvolvimento, de acordo com os padrões escolhidos, podem ser definidos. Outra responsabilidade da arquitetura está em dividir o software em componentes abstratos, que juntos formarão um sistema completo que irá satisfazer os requisitos, ou seja, sua funcionalidade, além de interagir e cooperar entre si.

Um grande problema quando se está iniciando o estudo de padrões de projetos é a dúvida de quando utilizá-lo. Uma dica importante de Engholm (2013, p. 233) é que padrões de projetos não representam a solução para todos os problemas encontrados no design e no desenvolvimento de software, por isso deve-se procurar o equilíbrio entre a flexibilidade e a simplicidade.

O arquiteto de softwares conhecer vários *frameworks* e vários catálogos de padrão orientado a objetos, pois deverá estar pronto para utilizá-los de acordo com cada projeto. Isso trará benefícios na reutilização e atenderá três objetivos principais do sistema: escalabilidade, performance e modularização distribuída.

O particionamento do software em componentes, segundo Leite (2000), oferece vários benefícios.

- » permite ao programador compreender melhor o software;
- » possibilita que essas partes possam ser reutilizadas mais de uma vez dentro do mesmo programa ou por outro programa;
- » podem ser gerenciadas mais facilmente quanto estiverem sendo executadas.

Além dos benefícios, o processo de selecionar corretamente uma arquitetura, segundo Engholm (2013, p. 209), deve considerar:

- » onde a empresa irá hospedar o sistema;
- » plataformas utilizadas;
- » tipo de plataforma que a empresa deseja utilizar;
- » sistemas legados aos quais o novo sistema irá se integrar;
- » tipo de interface se que deseja utilizar;
- » se necessário, qual o tipo de distribuição do sistema deverá se utilizar.

Um sistema bem desenhado, bem desenvolvido ou bem projetado, deverá possuir as seguintes características:

Escalabilidade

É a capacidade de determinado sistema continuar funcionando em virtude do aumento de usuário, trabalhando de forma concorrente. A escalabilidade deve ser transparente aos usuários caso haja necessidade de manusear sua capacidade. O hardware deve fazer parte dessa escalabilidade, pois o cálculo dessa demanda deve ser para ambos.

Disponibilidade

É o tempo que o sistema ficará disponível ao usuário, funcionando e em execução. Normalmente sua mensuração é feita em percentual de tempo de atividade.

Confiabilidade

A confiabilidade está relacionada à disponibilidade, pois sua mensuração é feita entre as falhas de disponibilidade.

CAPÍTULO 2

Modelo de design

A fase de design sempre acontece logo após o levantamento de requisitos e define a arquitetura, os componentes, as interfaces e os dados do sistema. Segundo Engholm (2013, p. 229), seu propósito é criar uma solução técnica que satisfaça aos requisitos do sistema e traduza a especificação funcional escrita basicamente na terminologia dos negócios. Seu objetivo é:

- » determinar as classes que compõem o sistema;
- » detalhar arquitetura, banco de dados, linguagem de programação, padrões de design etc.
- » diagramar sequências de classes, de estudo, de componentes, de distribuição.

No decorrer do tempo, alguns padrões de desenho de arquitetura foram aparecendo. Um padrão pode se definir como soluções para problemas específicos e recorrentes que foram utilizados no desenvolvimento de software. Quem propôs esse princípio foi Chirstopher Alexander, um engenheiro da área da construção civil. Para ele, um padrão é composto por quatro características, citadas por Gamma (2000, p. 19):

- » **O Nome do Padrão:** uma referência usada para descrever um problema de projeto, suas soluções e consequências em uma ou duas palavras.
- » **O Problema:** descreve em que situação aplicar o padrão, explica o problema e o contexto e pode descrever problemas de projetos específicos.
- » **A Solução:** descreve os elementos que compõem o padrão de projeto, seus relacionamentos, suas responsabilidades e colaborações.
- » **As Consequências:** são os resultados e análises das vantagens e desvantagens da aplicação do padrão.

Para descrever um padrão de projeto, devem-se evitar as notações gráficas e optar por um formato mais consistente. Isso facilitará a reutilização do projeto, registrando custos, análise de experiências e decisões. Gamma (2000, p. 22) propõe o seguinte gabarito, que será utilizado em seu catálogo de Padrão de Projetos:

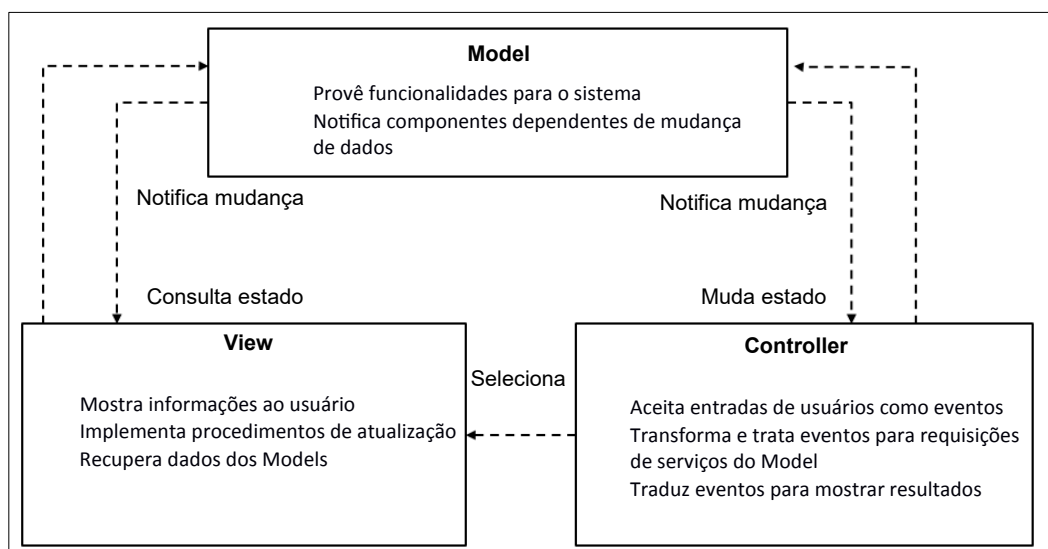
- » Nome e classificação do Projeto;

- » Intenção e objetivo;
- » Motivação;
- » Aplicabilidade;
- » Estrutura;
- » Participantes;
- » Colaboração;
- » Consequências;
- » Implementação;
- » Exemplo de Código;
- » Usos conhecidos;
- » Padrões Relacionados.

Um dos padrões de arquitetura que poderá ajudar a melhor o conceito de padrão é o MVC (*Model, View, Controller*). Ele é composto por três objetos, como cita Gamma (2000, p. 20) e Engholm (2013, p. 235):

- » **Model** é o objeto de aplicação. Ele contém os dados e as regras de negócio e determina qual View será apresentada como *interface*
- » **View** é a apresentação na tela. É responsável por manter a interação gráfica com o usuário.
- » **Controller** é o que define a maneira como a interface do usuário reage às entradas deste. Recebe os eventos e as requisições dos usuários.

Figura 38. Padrão MVC.



Fonte: Engholm (2000, p. 225).

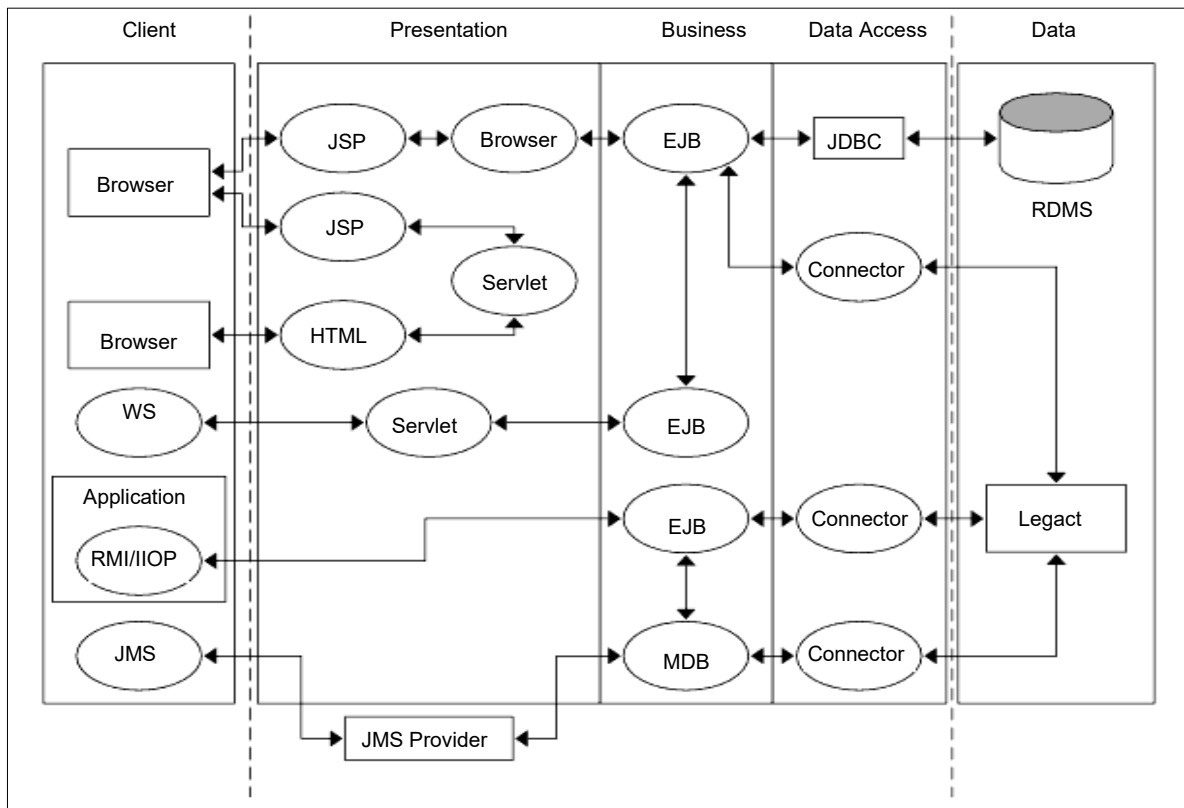
Layers

Atualmente, juntamente com MVC, o padrão de arquitetura em camadas vem se tornando mais popular. Esse modelo facilita o design do sistema, a reutilização dos componentes e a. Ela é bem flexível, e as grandes plataformas, como Java e .NET, já disponibilizaram manutenção *frameworks* para facilitar o trabalho da modelagem e programação. A abordagem multicamada é particularmente boa para o desenvolvimento de aplicativos hospedados em escala de web, de nível de produção e em nuvem com muita rapidez e relativamente sem riscos.

Esse padrão é composto de três elementos:

- » **Camada de apresentação:** é a primeira e mais alta camada presente no sistema. Fornece serviços de apresentação, ou seja, apresentação de conteúdo para o usuário final por meio da GUI (*Guide User Interface*). Pode ser acessada por meio de qualquer tipo de dispositivo cliente, como *desktop*, *laptop*, *tablet*, celular etc. Na plataforma Java, a *camada de apresentação* é onde a interface do usuário é gerada dinamicamente. Um aplicativo pode requerer componentes J2EE na camada de apresentação, como os *Servlets*, *Jsp*s, Conteúdo estático.
- » **Camada de aplicativo:** é a camada intermediária dessa arquitetura, onde a lógica de negócios do aplicativo é executada. Lógica de negócios é o conjunto de regras necessárias para executar o aplicativo de acordo com as diretrizes estabelecidas pela organização. Na plataforma Java, a *camada de lógica de negócios* geralmente contém componentes EJB implementados que encapsulam regras de negócios e outras funções de negócios em *Session Beans*, *Entity Beans*, *Message-Drive Beans*. Na plataforma .NET, a camada de negócio poderá ficar em uma DLL.
- » **Camada de dados:** é a camada mais baixa dessa arquitetura e está principalmente relacionada ao armazenamento e à recuperação de dados de aplicativos. Na plataforma .NET, a camada de dados, assim como a de negócios, também poderá ficar disponível em DLL. Na plataforma Java, na *camada de acesso a dados*, o JDBC (conectividade de banco de dados Java) é usado para conectar-se a bancos de dados, fazer consultas e retornar resultados de consultas e conectores personalizados.

Figura 39.



Fonte: <https://docs.oracle.com/cd/E19644-01/817-5448/dgdesign.html#wp18227>. Acesso em: 1º jun. 2019.

CAPÍTULO 3

Documentação da arquitetura

O documento de arquitetura de software oferece uma visão geral de arquitetura detalhada do sistema. Ele serve como um meio de interação entre o arquiteto de software e outros membros de equipe de projeto, relacionadas às decisões de arquitetura sobre o projeto.

A documentação da arquitetura poderá ser demonstrada de maneiras diversas, pois deverá deixar claro o motivo de sua escolha. Textos e diagramas UML são as formas mais fáceis de se produzir e de apresentar. Ela poderá ser informal, semiformal ou formal. A informal, apesar de ser rápida, é fácil para se confeccionar. Sua desvantagem está na descrição detalhada do diagrama, pois não existe uma padronização. A semiformal é composta de um esquema gráfico que possui regras um pouco mais rígidas que a informal, entretanto não fornece uma descrição detalhada dos elementos. Com isso, requer conhecimento mais técnico dos diagramas mais específicos. Um exemplo é o diagrama UML. A formal está diretamente ligada à arquitetura escolhida, muitas vezes integradas na mesma ferramenta de codificação, e sua grande utilidade é gerar códigos por meio dos diagramas.

Engholm (2013, p. 213) demonstra um modelo parcialmente preenchido em seu estudo de caso, em que a abertura do projeto será composta por:

Introdução

Esse documento deve apresentar a arquitetura do sistema utilizando-se de diagramas UML, como Caso de Uso, Processos, Implantação.

Propósito

Esse documento fornece uma visualização da arquitetura do sistema, utilizando diferentes visualizações da arquitetura relacionada.

Escopo

Contém uma breve descrição da maneira de se utilizar o Documento de Arquitetura de Software.

Definições, acrônimos e abreviações

Essa subseção fornece as definições de todos os termos, acrônimos e abreviações necessárias para a interpretação apropriada do Documento de Arquitetura de Software.

Referências

Essa subseção fornece uma lista completa de todos os documentos mencionados em outra parte do Documento de Arquitetura de Software.

Haverá subseções que explicam como o Documento de Software está organizado:

- » Representação Arquitetural;
- » Restrições e Metas Arquiteturais;
- » Visão de Casos de Uso;
- » Visão Lógica.

Decomposição do modelo de design em termos de camadas e de hierarquia de pacotes:

- » Pacotes de Design Significativos do Ponto de Vista da Arquitetura;
- » Realizações de Casos de Uso;
- » Visão de Processos;
- » Visualização da Implementação;
- » Visão da Implementação.

E, ainda, o Documento de Arquitetura de Software poderá ter a definição de camadas, caso o projeto opte por esse modelo:

- » Camp:
 - › para cada camada, inclua uma subseção com o respectivo nome, uma lista dos subsistemas localizados na camada e um diagrama de componentes.
- » Visão de Dados (opcional):
 - › uma descrição da perspectiva de armazenamento de dados persistentes do sistema.

- » Tamanho e desempenho:
 - › uma descrição das principais características de dimensionamento do software que têm um impacto na arquitetura, bem como as restrições do desempenho desejado.
- » Qualidade.

CAPÍTULO 4

Padrão GoF

Um padrão de projeto nomeia, abstrai e identifica os aspectos-chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável. O padrão de projeto identifica as classes e instâncias participantes, seus papéis, colaborações e a distribuição de responsabilidades. Cada padrão de projeto focaliza um problema ou tópico particular de projeto orientado a objetos. Ele descreve em que situação pode ser aplicado, se ele pode ser aplicado em função de outras restrições de projeto e as consequências, os custos e os benefícios de sua utilização (GAMMA, 2000, p. 20).

O padrão de projeto GoF é dividido em grupos:

- » **Padrões de comportamento:** descrevem como os objetos devem se comportar em relação à interação e distribuição de responsabilidade. Os padrões comportamentais não descrevem apenas padrões de objetos ou classes, mas também os padrões de comunicação entre eles (GAMMA, 2000, p. 212).
- » **Padrão de Criação:** os padrões de criação abstraem o processo de instanciação. Eles ajudam a tornar um sistema independentemente de como seus objetos são criados, compostos e representados. Um padrão de criação de classe usa a herança para variar a classe que é instanciada, enquanto um padrão de criação de objeto delegará a instanciação para outro Objeto (GAMMA, 2000, p. 91).
- » **Padrão estrutural:** os padrões estruturais se preocupam com a forma como classes e objetos são compostos para formar estruturas maiores. Os padrões estruturais de classes utilizam a herança para compor interfaces ou implementações. Esse padrão é particularmente útil para fazer bibliotecas de classes desenvolvidas independentemente de trabalharem juntas (GAMMA, 2000, p. 140).

Cada grupo está dividido em vários padrões:

Figura 40. O espaço dos padrões de projeto.

Escopo	Classe	Propósito		
		Criação	Estrutural	Comportamental
		<i>Factory Method</i>	<i>Adapter(class)</i>	<i>Interpreter</i>
	Objeto	<i>Abstract Factory</i> <i>Builder</i> <i>Prototype</i> <i>Singleton</i>	<i>Adapter(object)</i> <i>Bridge</i> <i>Composite</i> <i>Decorator</i> <i>Facade</i> <i>Flyweight</i> <i>Proxy</i>	<i>Chain of Responsibility</i> <i>Command</i> <i>Iterator</i> <i>Mediator</i> <i>Memento</i> <i>Observer</i> <i>State</i> <i>Strategy</i> <i>Visitor</i>

Fonte: Gamma (2000 p. 26).

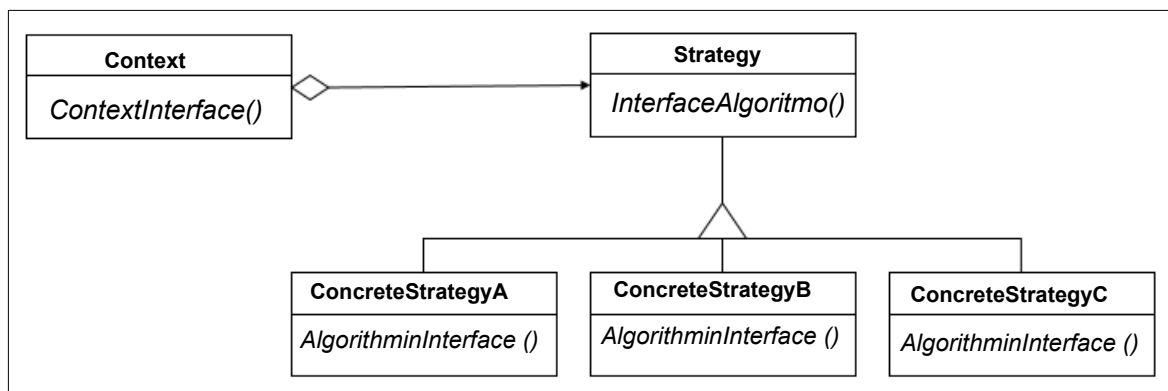
Em seguida, veremos os mais comumente usados nas linguagens de programação mais populares de mercado.

Padrão de comportamento

Strategy

Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam (GAMMA, 2000, p. 292). Sua principal utilização é quando se necessita de variantes de um determinado algoritmo.

Figura 41. Especificação do Padrão Strategy.

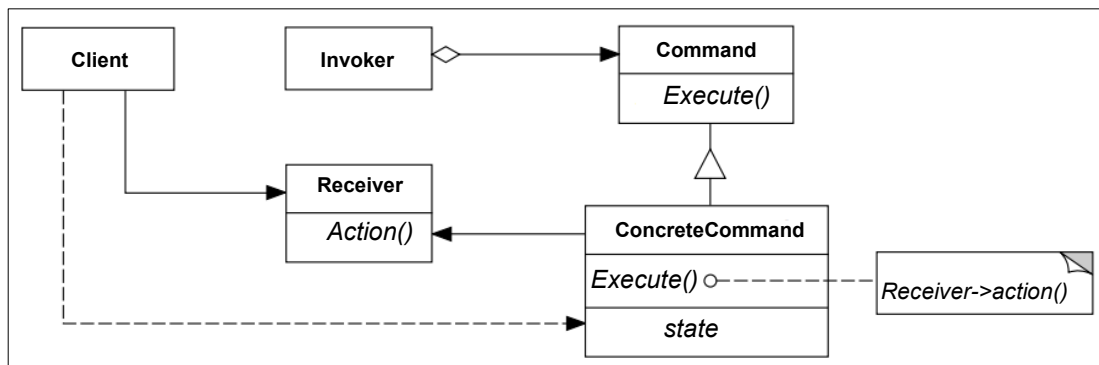


Fonte: Gamma (2000 p. 294).

Command

A intenção do Padrão Command é encapsular uma solicitação como um objeto permitindo, dessa forma, parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (log) de solicitações se suportar operações que podem ser desfeitas. Sua principal aplicação é parametrizar objetos por uma ação a ser executada, especificar, enfileirar e executar solicitações em tempos diferentes (GAMMA, 2000, p. 225).

Figura 42. Estrutura do Padrão Command.

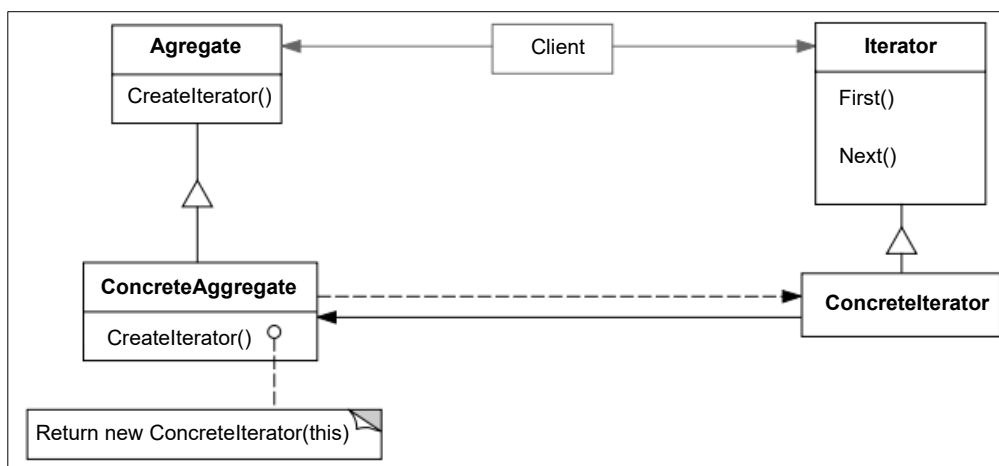


Fonte: Gamma (2000 p. 294).

Iterator

Também conhecido como cursor, fornece um meio de acessar, sequencialmente, os elementos de um objeto agregado sem expor a sua representação subjacente. Sua motivação é um objeto agregado, tal como uma lista, que deveria fornecer um meio de acessar seus elementos sem expor a sua estrutura interna.

Figura 43. Estrutura do Padrão Iterator.



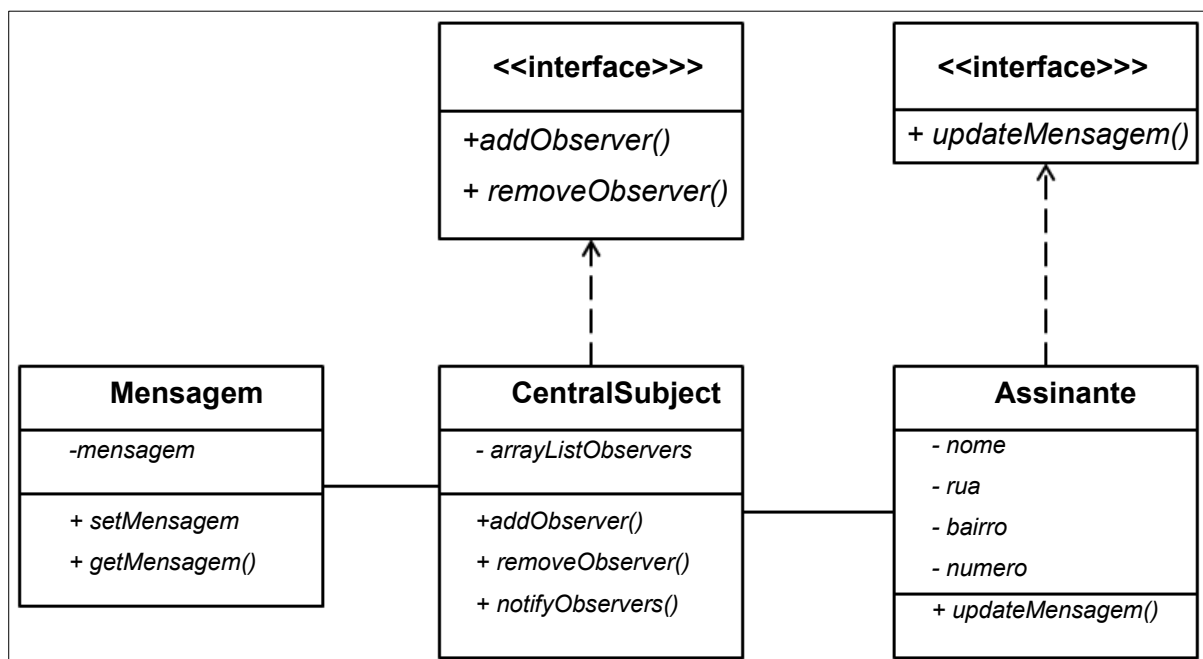
Fonte: Gamma (2000 p. 245).

Observer

No contexto do *Observer*, temos objetos observadores e observados. Ele é responsável por notificar os demais objetos sobre a mudança de estado de outros objetos.

Definir uma dependência um-para-muitos entre objetos, de maneira que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente (GAMMA, 2000, p. 274).

Figura 44. Diagrama de classe do padrão *Observer*.



Fonte: Engholm (p. 252).

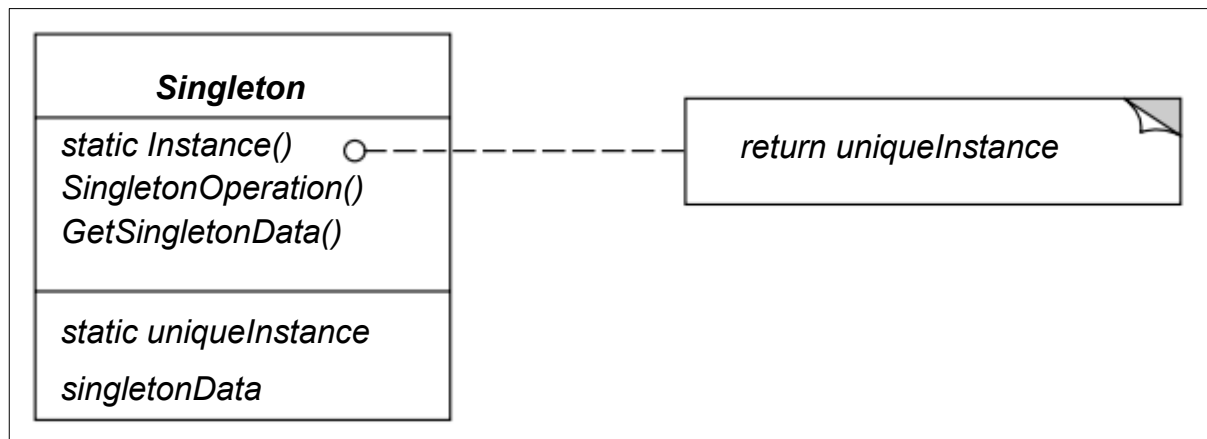
Padrão de criação

Singleton

Singleton é um padrão que permite que apenas um objeto da classe seja instanciado.

Uma solução melhor seria tornar a própria classe responsável por manter o controle da sua única instância. A classe pode garantir que nenhuma outra instância seja criada (pela interceptação das solicitações para criação de novos objetos), bem como pode fornecer um meio para acessar sua única instância (GAMMA, 2000, p. 130).

Figura 45. Estrutura Padrão Singleton.



Fonte: GAMMA (2000, p. 130).

Factory

Criar objetos dentro de uma classe com um método *factory* é sempre mais flexível do que criar um objeto diretamente. *Factory Method* dá às subclasses um gancho para fornecer uma versão estendida de um objeto e permite adiar sua instanciación (GAMMA, 2000, p. 115).

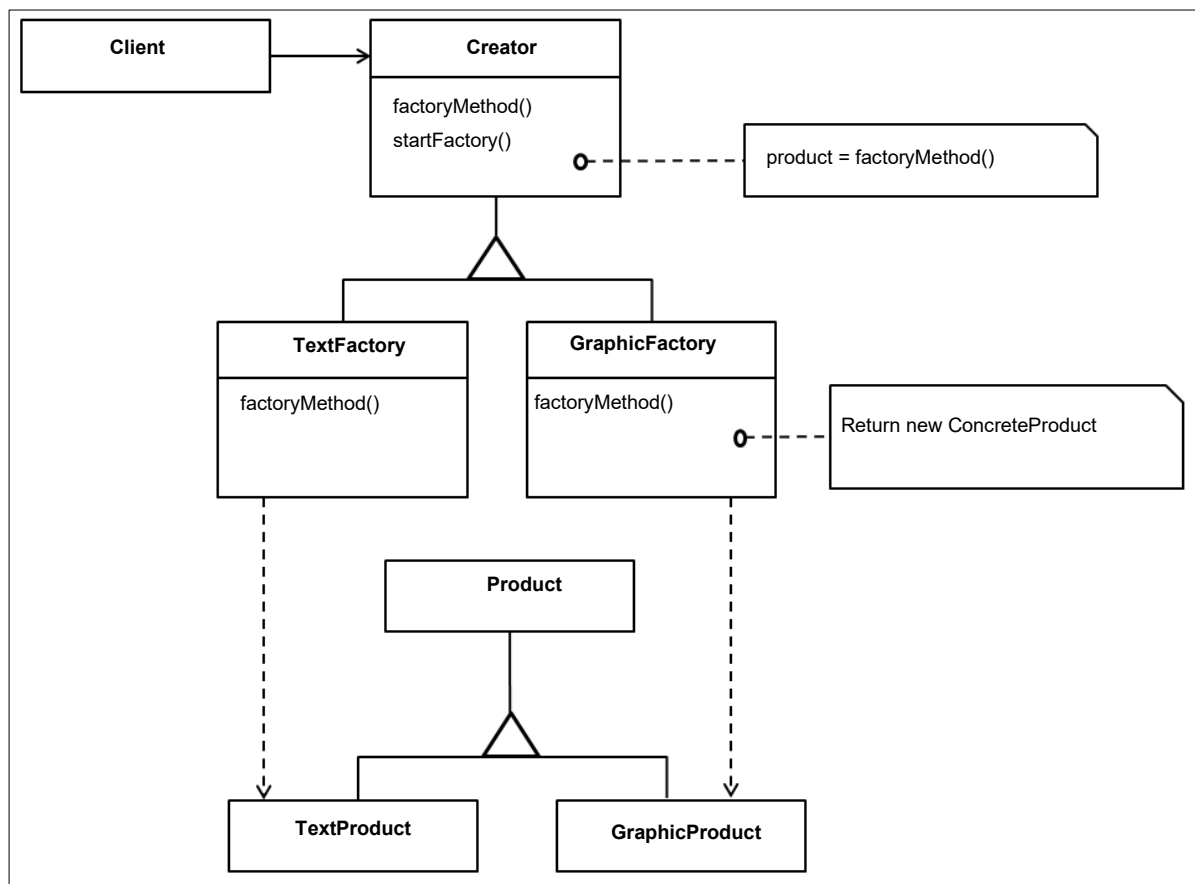
Segundo Engholm (2013, p. 258), uma vantagem da utilização desse padrão é que podemos adicionar novas subclasses ao sistema sem nenhum tipo de alteração no *framework* da aplicação. Por exemplo, em uma aplicação para hotéis, podemos adicionar novas classes concretas relacionadas ao tipo de quartos sem alterar o código que utiliza essas classes.

Segundo Gamma (2000, p. 113), devemos usar o padrão *Factory Method* quando:

- » uma classe não pode antecipar a classe de objetos que deve criar;
- » uma classe quer que suas subclasses especifiquem os objetos que criam;
- » as classes delegam responsabilidade para uma dentre várias subclasses auxiliares, e você quer localizar o conhecimento de qual subclasse auxiliar que é a delegada.

Por ser da categoria de criação, esse padrão tem relação com a criação de algo. No caso do *Factory*, o algo a ser criado é um produto que não está amarrado à classe que o cria. Para manter baixo acoplamento, o cliente faz uma solicitação por meio de uma fábrica, que cria o produto solicitado.

Figura 46. Estrutura padrão Factory.



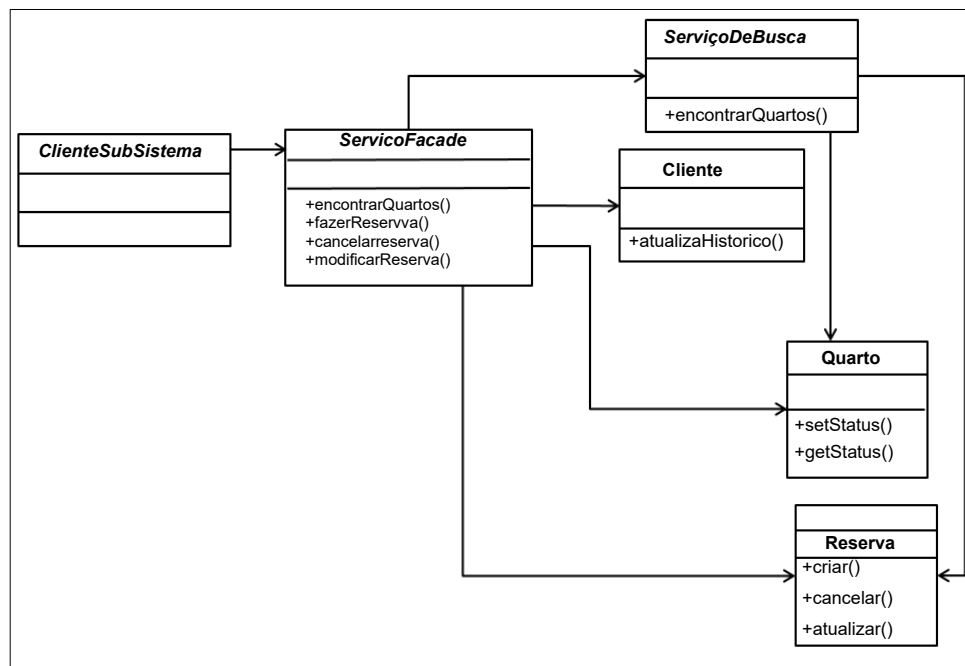
Fonte: Sanders (2013, p. 101).

Padrão estrutural

Facade

Em sistemas maiores, a comunicação entre subsistemas pode ser complexo e desorganizado. Com padrão *Facade*, é possível unificar para um conjunto de interfaces. Segundo Gamma (2013, p. 179), *Facade* define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado, além de estruturar um sistema em subsistemas para reduzir a complexidade.

O exemplo a seguir foi criado por Engholm (2013, p. 263) e representa o uso de um sistema de hotel em que o funcionário realiza a reserva de determinado quarto para um provável hóspede. Nesse caso de uso, o sistema deve apresentar os quartos disponíveis no período desejado pelo cliente. No diagrama, podemos observar que o *ServicoReservaFacade* possui uma série de métodos que são disponibilizados pelo cliente de subsistemas e que, ao chamar qualquer método disponível no Facade, o cliente não precisa de nenhum conhecimento das classes envolvidas.

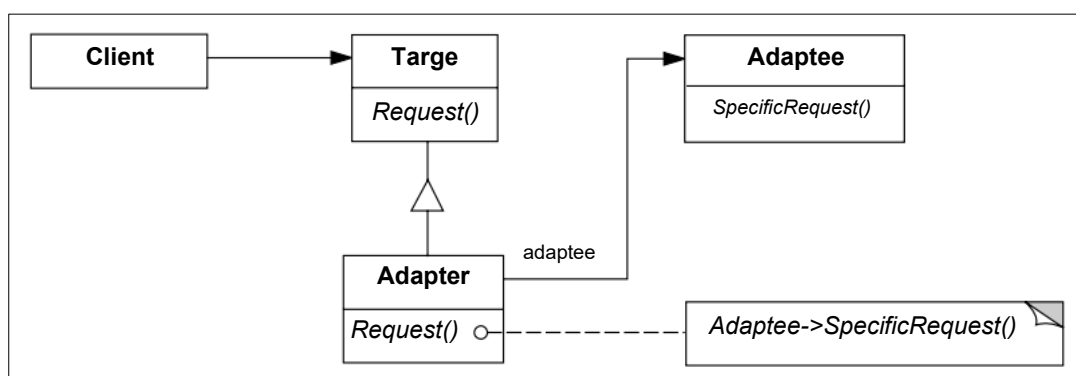
Figura 47. Diagrama de Classe do Padrão *Facade*.

Fonte: Engholm (2013, p. 266).

Adapter

A intenção do padrão *Adapter*, segundo Gamma (2000, p. 140), é converter a interface de uma classe em outra interface, esperada pelos clientes. O *Adapter* permite que classes com interfaces incompatíveis trabalhem em conjunto – o que, de outra forma, seria impossível. Utiliza-se *Adapter* quando é preciso utilizar uma classe já existente, mas sua interface não corresponde à interface que necessita utilizar algum recurso, ou reutilizar classes que coopere com classes não relacionadas. *Adapters* são úteis se você quiser usar uma classe que não tenha exatamente os métodos exatos que você precisa, e se a classe original não puder ser mudada. O *Adapter* pode pegar os métodos que você pode acessar na classe original e adaptá-los aos métodos que você precisa.

Figura 48. Especificação de Adapter com Composição.

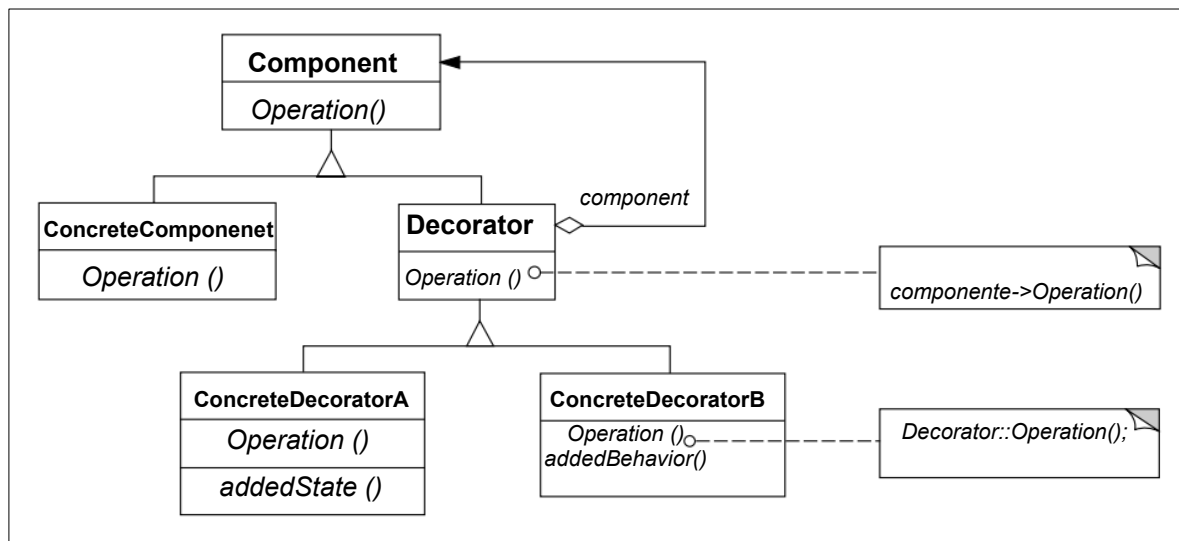


Fonte: Gamma, 2000 p. 142.

Decorator

O padrão decorador atribui responsabilidades adicionais a um objeto dinamicamente. Os decoradores fornecem uma alternativa flexível à subclasse para estender a funcionalidade. Algumas vezes, queremos acrescentar responsabilidades a objetos individuais, e não a toda uma classe. Às vezes, um grande número de extensões independentes é possível, e isso poderia produzir uma explosão de subclasses para suportar cada combinação (GAMMA, 2000, p. 172).

Figura 49. Implementação do Padrão *Decorator*.



Fonte: Gamma (2000 p. 172).

CAPÍTULO 1

Padrão Java

Arquitetura

O Java é uma plataforma composta de uma máquina virtual (JVM), inúmeras Bibliotecas e a linguagem de programação Java. Por isso, é considerado a plataforma mais completa, pois, além do desenvolvimento, já possui o ambiente de execução. A linguagem de programação é Orientada a Objetos, madura, de alto nível, robusta, multi *threaded*, portátil. Com ela, podemos criar várias aplicações como *Desktop*, *Web*, *Mobile*, Embarcados e cada tipo contém uma edição ou plataforma:

- » **Java SE (*Java Standard Edition*):** é a edição mais básica do Java e inclui as APIs mais simples de programação, como `java.lang`, `java.io`, `java.math`, `java.net`, `java.util` etc.
- » **Java EE (*Java Enterprise Edition*):** é uma edição mais robusta e possui aplicações de larga escala. É usada principalmente para desenvolver aplicativos da Web e corporativos. É construído no topo da plataforma Java SE. Inclui tópicos como Servlet, JSP, Web Services, EJB, JPA etc.
- » **Java ME (*Java Micro Edition*):** é uma plataforma bem reduzida, utilizada para aplicativos móveis e embarcados.

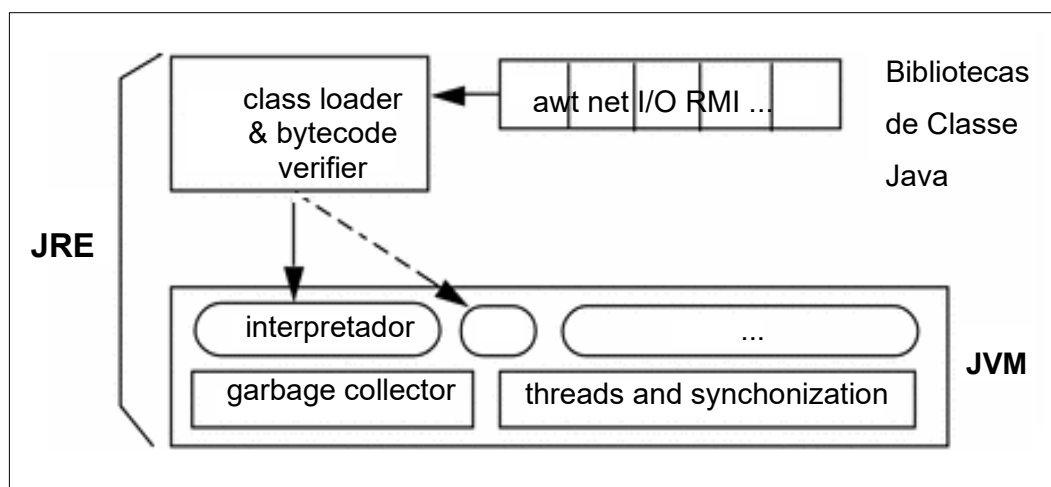
Todas essas plataformas funcionam por meio da implementação do JDK (*Java Development Kit*), que é um ambiente usado para desenvolver aplicativos Java. Nele estão contidas uma JRE e uma JVM.

A JRE (*Java Runtime Environment*) é um conjunto de ferramentas utilizadas para desenvolver aplicativos Java. É o ambiente de software no qual os programas

compilados para uma implementação típica da JVM podem ser executados. O sistema de tempo de execução inclui:

- » código necessário para executar programas Java, vincular dinamicamente métodos nativos, gerenciar memória e manipular exceções;
- » implementação da JVM.

Figura 50. Relacionamento Funcional com o JRE e Bibliotecas de Classes.



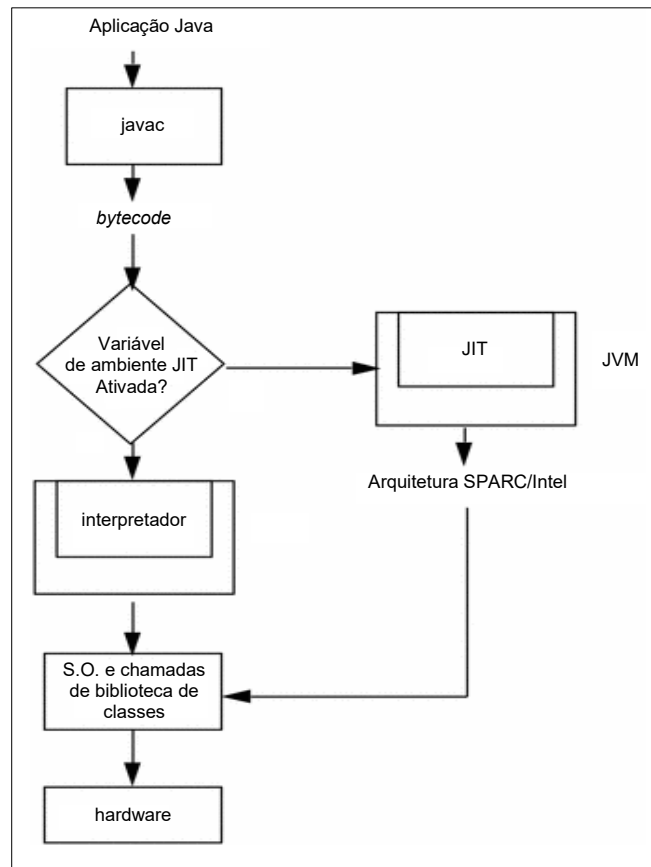
Fonte: <https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html>. Acesso em: 12 jun. 2019.

A **JVM** (*Java Virtual Machine*) é uma máquina virtual que abstrai a camada de hardware e faz a comunicação com o Sistema Operacional. É a JVM que faz com que o Java seja multiplataforma, ou seja, o mesmo software poderá funcionar em vários hardwares e Sistemas Operacionais nos quais a JVM esteja funcionando. A diferença para outras linguagens compiladas, como a linguagem C, por exemplo, é que esta abstrai as bibliotecas da arquitetura que o compilador estiver gerando o programa final. Ou seja, se for gerado um executável, em uma plataforma Intel com sistema operacional Windows, esse programa não funcionará em um sistema operacional Linux.

Para executar o código escrito em linguagem Java, é preciso transformá-lo em um padrão chamado *bytecode* e colocá-los em um arquivo *.class*, por meio de um compilador JIT (*Just-in-Time*) incluído na JDK.

Ativando a variável de ambiente do compilador JIT, que estará ativado por padrão, a JVM lê e faz a análise do arquivo *.class* e o transmite para o JIT. O JIT, por sua vez, compila os *bytecodes* em código nativo para a plataforma que está executando. Veja como demonstra na figura a seguir:

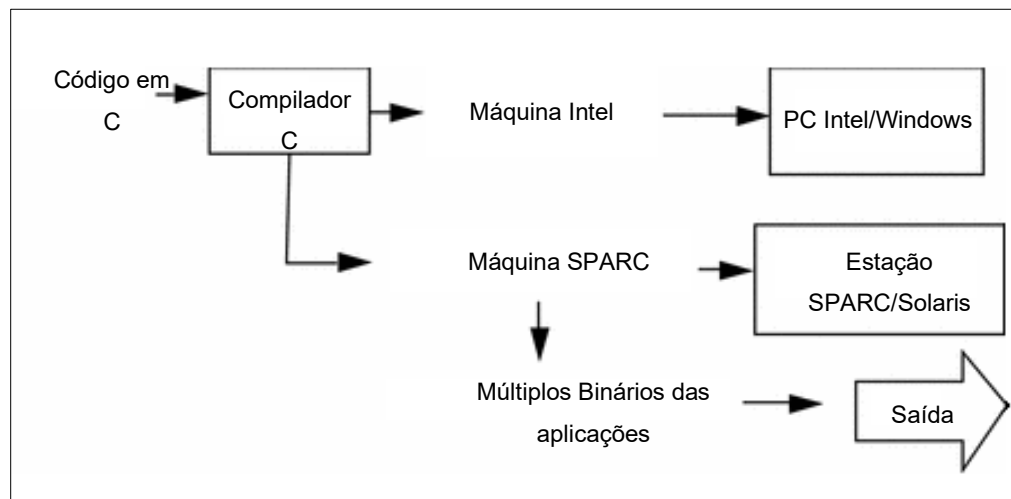
Figura 51. Processo de compilação JIT.



Fonte: <https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html>. Acesso em: 12 jun. 2019.

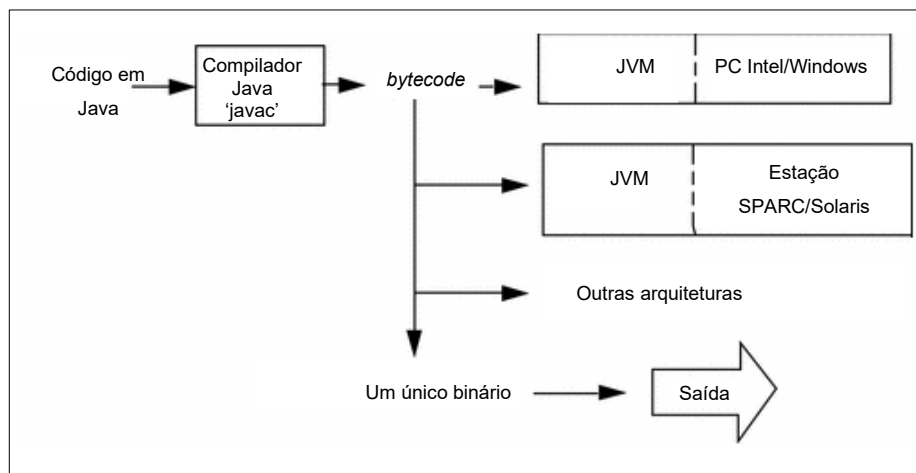
Logo depois de gerar, o *bytecode* será interpretado pela JVM que o executará. Nesse momento, o programa em execução não saberá nem o hardware nem o sistema operacional em que ele estará rodando. Veja a diferença entre um código em C em um ambiente tradicional e um código Java sendo compilados:

Figura 52. Ambiente de compilação tradicional.



Fonte: <https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html>. Acesso em: 12 jun. 2019.

Figura 53. Ambiente portátil do compilador Java.



Fonte: <https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html>. Acesso em: 12 jun. 2019.

Classe

Deve começar com letra maiúscula.

O nome da classe deve ser precedido da palavra reservada `class` e do modificador de acesso.

Nomes compostos devem constar como uma única palavra, como *PessoaFisica*.

Métodos

Deve começar com letra minúscula.

Deve ser um verbo como *main()*, *print()*, *println()*.

Se o nome contiver várias palavras, inicie-o com uma letra minúscula seguida por uma letra maiúscula, como *consultarExtrato()*.

Objetos

O objeto é uma instância de uma classe, ou seja, ele é a classe (arquivo texto) transformada em um bloco que ficará em memória. O **Estado** do objeto representa os dados (valor) de um objeto. O **Comportamento** representa o comportamento (funcionalidade) de um objeto, como depósito, retirada etc. A **Identidade** é geralmente implementada por meio de um ID exclusivo. O valor do ID não é visível para o usuário externo, sendo usado somente pela JVM.

Construtor

Contrutor é um tipo *especial* de método utilizado quando o objeto é instanciado, ou seja, ele é o primeiro método a ser executado. Em algumas arquiteturas, os

Contrutores podem ser inicializados com parâmetros ou sem e podem ser explícitos ou não. O nome do construtor deve ser o mesmo que o nome da classe, não deve ter nenhum tipo de retorno e, em certas linguagens, não pode ser abstrato, estático, final e sincronizado. Um construtor é denominado padrão quando é evocado logo na criação do objeto. Sua função é fornecer valores padrão para sua inicialização.

No código a seguir, na linha seis foi criado um construtor customizado que recebe dois parâmetros, e, na linha doze, o construtor padrão. No Java, o construtor padrão pode ser omitido caso não exista nenhum construtor customizado.

No método principal, TestaConta, foram criados dois objetos, **conta1** e **conta2**. O **conta1** utiliza o construtor padrão para iniciar o objeto, e o **conta2** utiliza o construtor parametrizado.

Figura 54. Códigos de exemplo.



```

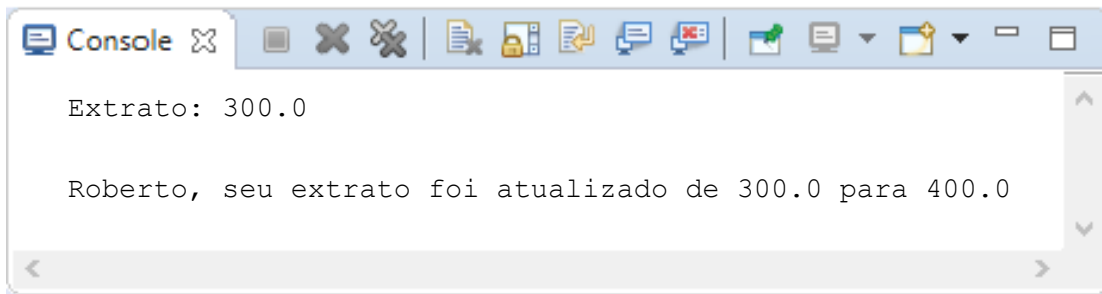
1  class Conta{
2      double _saldo = 300.00;
3      String _nome;
4
5      // Construtor parametrizado
6      Conta(double deposito, String nome){
7          this._saldo += deposito;
8          this._nome = nome;
9      }
10
11     // Construtor padrão
12     Conta() {}
13
14     //método
15     public double ConsultarExtrato() {
16         return this._saldo;
17     }
18 }
19
20 public class TestaConta {
21
22     public static void main(String[] args) {
23         Conta conta1 = new Conta();
24         System.out.println("Extrato: " +
25             conta1.ConsultarExtrato());
26
27         Conta conta2 = new Conta(100, "Roberto");
28         System.out.println(conta2._nome +
29             ", seu extrato foi atualizado de " +
30             conta1._saldo + " para " +
31             conta2._saldo);
32     }
33 }

```

Fonte: Elaborada pelo autor.

O resultado no console foi o seguinte: o objeto conta1 efetuou a Consulta de Extrato, que, naquele momento, era de 300.00. Em seguida, foi criada a conta2, inicializada com um depósito de 100.00.

Figura 55. Saída em console.

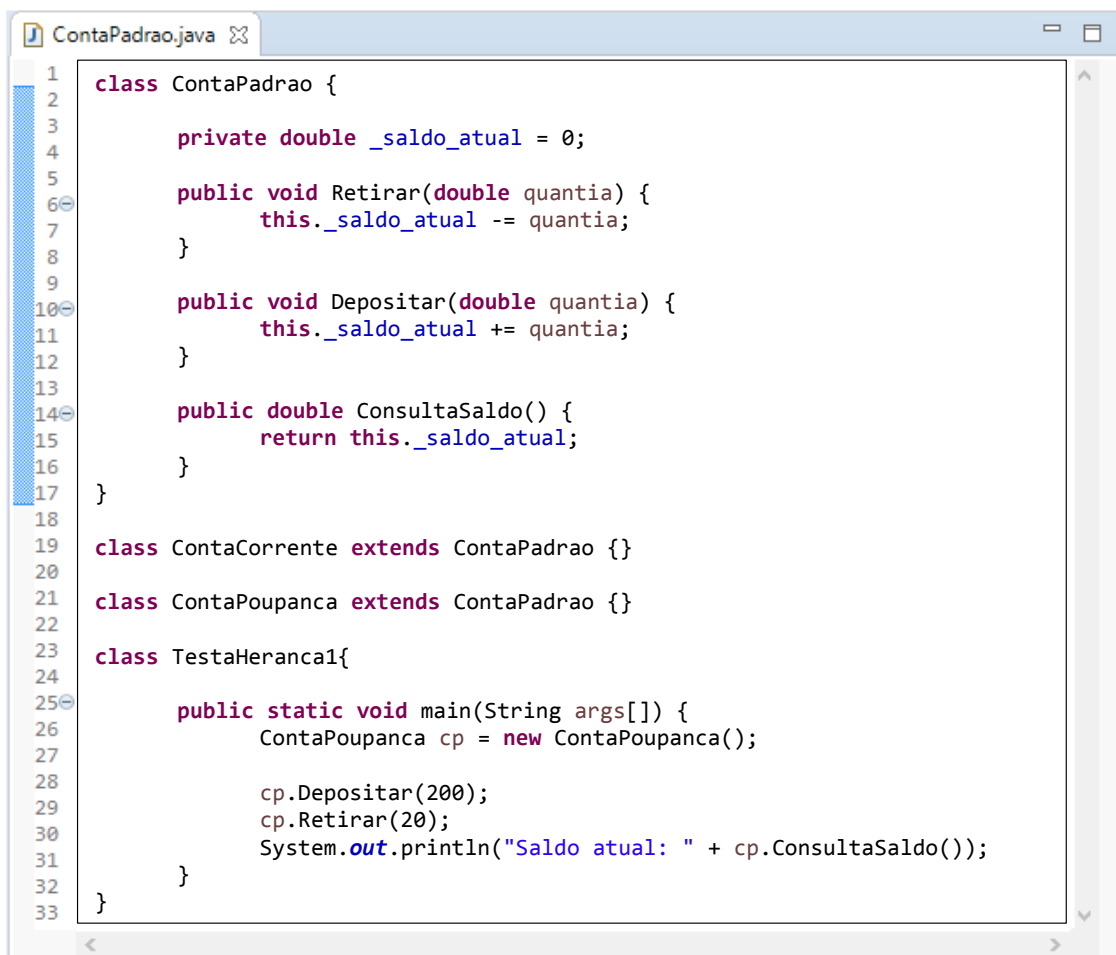


Fonte: Elaborada pelo autor.

Recursos de programação

No código a seguir, existe um exemplo de Herança em que as duas classes *ContaCorrente* e *ContaPoupanca* herdam ou estendem (*extends*) de *ContaPadrao*.

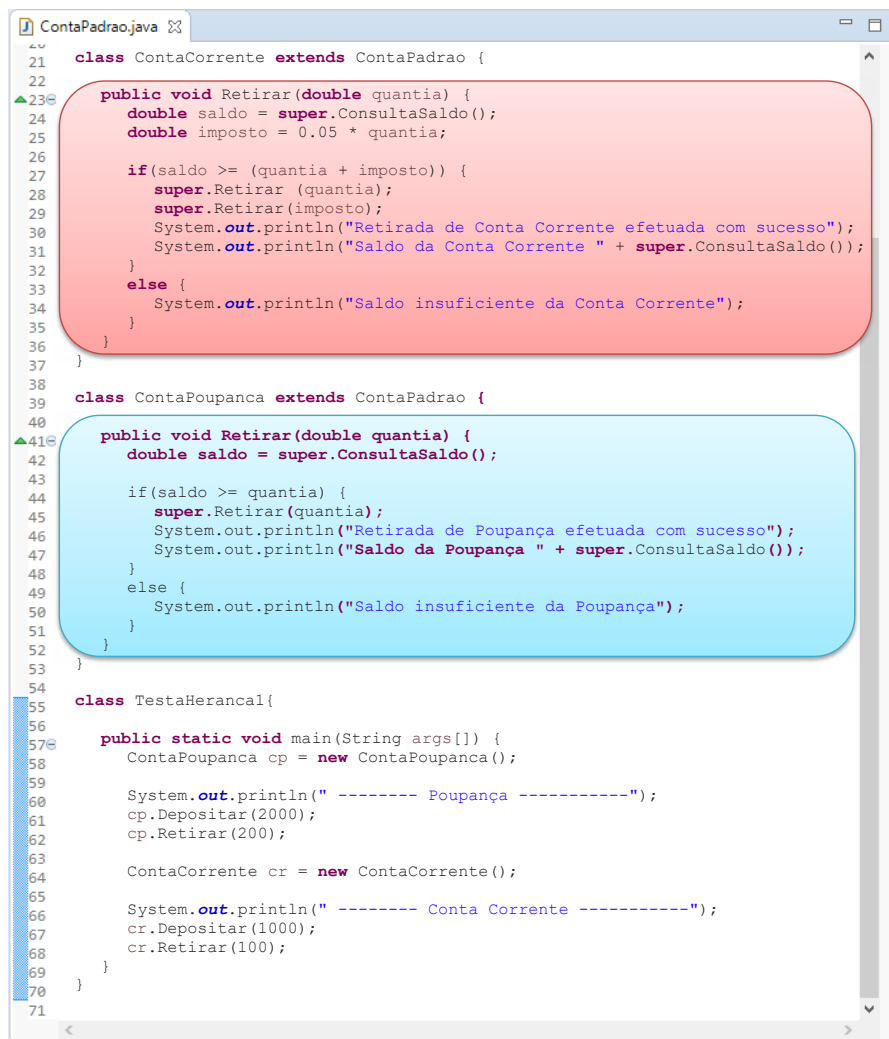
Figura 56. Herança e Encapsulamento.



Fonte: Elaborada pelo autor.

Em seguida, usando o mesmo exemplo, utiliza-se Polimorfismo, ou seja, o método *Retirar* da *ContaPoupanca* e *ContaCorrente* tem o mesmo nome e a mesma função. Porém, em cada subclasse há um comportamento diferente da superclasse.

Figura 57. Exemplo de Polimorfismo.



```

21 class ContaCorrente extends ContaPadrao {
22
23     public void Retirar(double quantia) {
24         double saldo = super.ConsultaSaldo();
25         double imposto = 0.05 * quantia;
26
27         if(saldo >= (quantia + imposto)) {
28             super.Retirar(quantia);
29             super.Retirar(imposto);
30             System.out.println("Retirada de Conta Corrente efetuada com sucesso");
31             System.out.println("Saldo da Conta Corrente " + super.ConsultaSaldo());
32         }
33         else {
34             System.out.println("Saldo insuficiente da Conta Corrente");
35         }
36     }
37 }
38
39 class ContaPoupanca extends ContaPadrao {
40
41     public void Retirar(double quantia) {
42         double saldo = super.ConsultaSaldo();
43
44         if(saldo >= quantia) {
45             super.Retirar(quantia);
46             System.out.println("Retirada de Poupança efetuada com sucesso");
47             System.out.println("Saldo da Poupança " + super.ConsultaSaldo());
48         }
49         else {
50             System.out.println("Saldo insuficiente da Poupança");
51         }
52     }
53 }
54
55 class TestaHeranca1{
56
57     public static void main(String args[]) {
58         ContaPoupanca cp = new ContaPoupanca();
59
60         System.out.println(" ----- Poupança -----");
61         cp.Depositar(2000);
62         cp.Retirar(200);
63
64         ContaCorrente cr = new ContaCorrente();
65
66         System.out.println(" ----- Conta Corrente -----");
67         cr.Depositar(1000);
68         cr.Retirar(100);
69     }
70 }
71

```

Fonte: Elaborada pelo autor.

Além da arquitetura básica para o conceito de Orientação a Objetos, o Java também proporciona a implementação do padrão GoF.

Implementando padrão GoF – Singleton

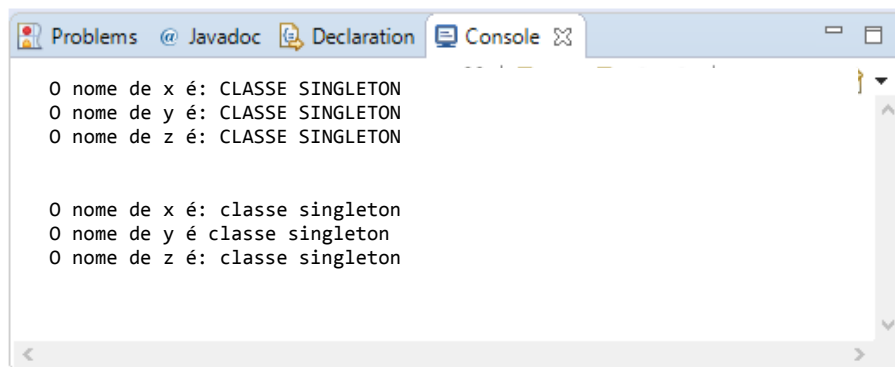
Na programação orientada a objetos, uma classe Singleton é uma classe que pode ter apenas um objeto (uma instância da classe) por vez. Assim que for instanciada, a nova variável apontará exatamente para a primeira instância criada, ou seja, qualquer modificação feita em qualquer variável dessa classe Singleton, por meio de qualquer instância, afetará diretamente a variável única criada, sendo visível o acesso em qualquer variável desse tipo de classe. Para projetar uma classe *singleton*:

- » Faça o construtor como privado.
- » Escreva um método estático que tenha o objeto de tipo de retorno dessa classe *Singleton*.

A diferença na classe normal e *Singleton* em termos de instanciação é que, para a classe normal, utiliza-se um construtor padrão, enquanto, para a classe *Singleton*, utiliza-se o método *getInstance* ().

Veja que, no exemplo, serão instanciados vários objetos da classe *Singleton*, e o resultado será sempre a mesma instância.

Figura 58. Resultado da classe *Singleton*.



```

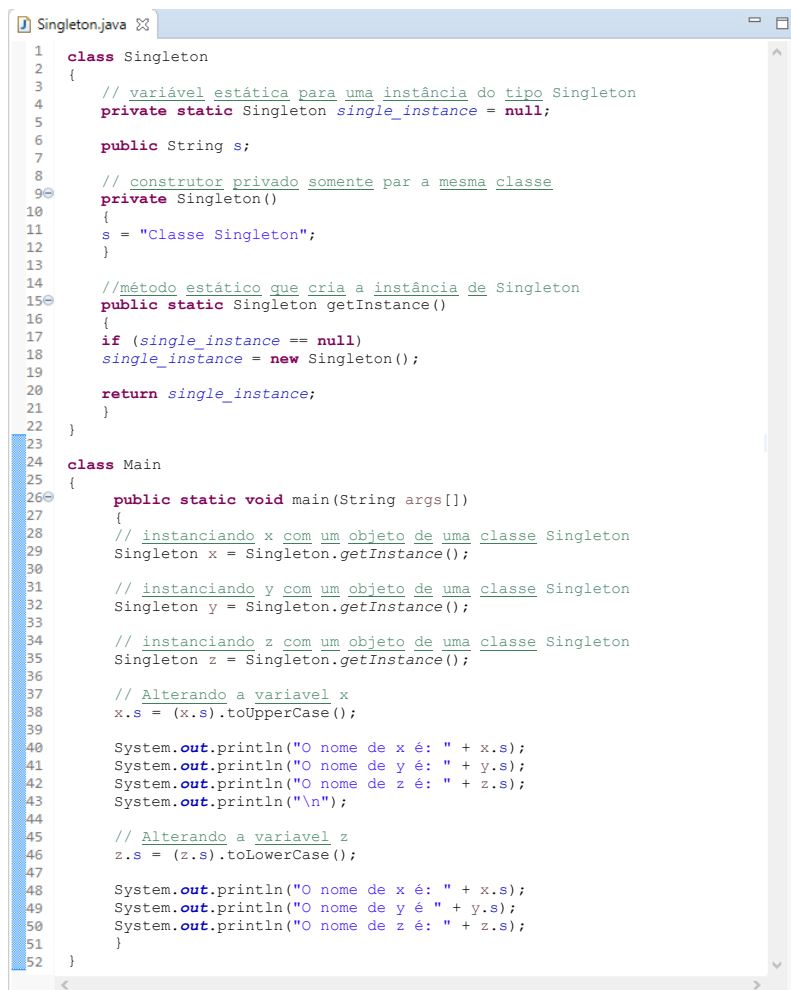
O nome de x é: CLASSE SINGLETON
O nome de y é: CLASSE SINGLETON
O nome de z é: CLASSE SINGLETON

O nome de x é: classe singleton
O nome de y é: classe singleton
O nome de z é: classe singleton

```

Fonte: Elaborada pelo autor.

Figura 59. Exemplo de Padrão Singleton.



```

1  class Singleton
2  {
3      // variável estática para uma instância do tipo Singleton
4      private static Singleton single_instance = null;
5
6      public String s;
7
8      // construtor privado somente par a mesma classe
9      private Singleton()
10     {
11         s = "Classe Singleton";
12     }
13
14     // método estático que cria a instância de Singleton
15     public static Singleton getInstance()
16     {
17         if (single_instance == null)
18             single_instance = new Singleton();
19
20         return single_instance;
21     }
22 }
23
24 class Main
25 {
26     public static void main(String args[])
27     {
28         // instanciando x com um objeto de uma classe Singleton
29         Singleton x = Singleton.getInstance();
30
31         // instanciando y com um objeto de uma classe Singleton
32         Singleton y = Singleton.getInstance();
33
34         // instanciando z com um objeto de uma classe Singleton
35         Singleton z = Singleton.getInstance();
36
37         // Alterando a variável x
38         x.s = (x.s).toUpperCase();
39
40         System.out.println("O nome de x é: " + x.s);
41         System.out.println("O nome de y é: " + y.s);
42         System.out.println("O nome de z é: " + z.s);
43         System.out.println("\n");
44
45         // Alterando a variável z
46         z.s = (z.s).toLowerCase();
47
48         System.out.println("O nome de x é: " + x.s);
49         System.out.println("O nome de y é: " + y.s);
50         System.out.println("O nome de z é: " + z.s);
51     }
52 }

```

Fonte: Adaptado de <https://www.geeksforgeeks.org/singleton-class-java>. Acesso em: 1º jun. 2019.

CAPÍTULO 2

Padrão .NET

A plataforma .NET (lê-se *dot net*) é uma plataforma *open source* de desenvolvimento de software criada pela Microsoft. Ela fornece ferramentas e tecnologias necessárias para criar aplicativos em rede, bem como Web Services, sistemas distribuídos, aplicativos da Web, e seu principal componente é o .NET Framework.

Arquitetura

.NET Framework

Diferentemente do JVM do Java, o .NET Framework não é uma máquina virtual, é um ambiente de execução gerenciado para as plataformas Windows e fornece vários serviços aos aplicativos em execução. Ele é composto de dois componentes principais: o CLR (*Common Language Runtime*) e a biblioteca de classes.

CLR

A função do *Common Language Runtime* é gerenciar memória, execução de *threads*, execução de código, verificação de segurança do código, compilação e outros serviços do sistema.

Em relação à segurança, os componentes gerenciados pelo CLR recebem níveis de confiança variados, dependendo do número de fatores que incluem sua origem (como a Internet, a rede corporativa ou o computador local). Isso significa que um componente gerenciado pode ou não ser capaz de executar operações de acesso a arquivo, operações de acesso a Registro ou outras funções confidenciais, mesmo que esteja sendo usado no mesmo aplicativo ativo.

O código implementado tem sua execução e sua tipagem forte gerenciada pela CTS (*Common Type System*) que assegura que esse código seja autodescritivo, eliminando muitos problemas comuns em softwares, como liberação de memória de componentes que não estejam mais sendo utilizados. Um recurso chamado compilação JIT (*Just-In-Time*) permite que todos os códigos gerenciados sejam executados na linguagem de computador nativa do sistema que está em execução.

Biblioteca de classes .NET Framework

A biblioteca de classes .NET Framework é uma coleção de componentes reutilizáveis que se integram plenamente ao *Common Language Runtime*. A biblioteca de classes é orientada a objeto, fornecendo tipos dos quais seu próprio código gerenciado deriva funcionalidade. Isso não apenas torna os tipos do .NET Framework fáceis de usar, como também reduz o tempo associado ao aprendizado de novos recursos do .NET Framework. Além disso, componentes de terceiros são totalmente integrados a classes do .NET Framework.

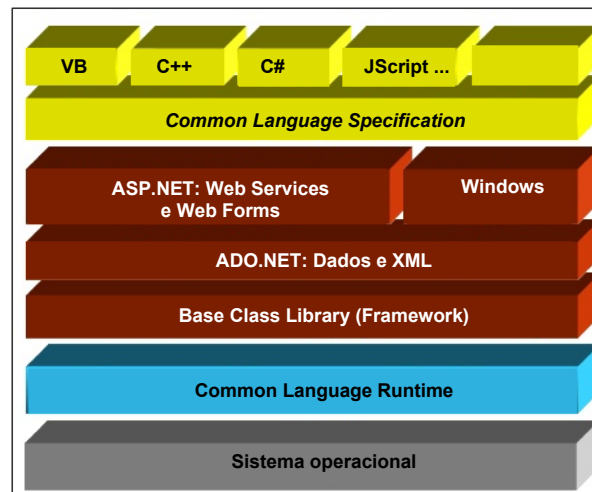
Por exemplo, as classes da coleção do .NET Framework implementam um conjunto de interfaces que você pode usar para desenvolver suas próprias coleções de classes, as quais se combinam perfeitamente às classes do .NET Framework.

Além de tarefas comuns, a biblioteca de classes inclui tipos que dão suporte a vários cenários de desenvolvimento especializados:

- » aplicativos de console;
- » aplicativos GUI do Windows (*Windows Forms*);
- » aplicativos WPF (*Windows Presentation Foundation*);
- » aplicativos ASP.NET;
- » serviços do Windows;
- » aplicativos orientados a serviço usando o WCF (*Windows Communication Foundation*);
- » aplicativos habilitados para fluxo de trabalho usando o *Windows Workflow Foundation* (WF).

As classes *Windows Forms* são um conjunto abrangente de tipos reutilizáveis que simplificam muito o desenvolvimento de GUI Windows. Ao criar um aplicativo Web Form do ASP.NET, você poderá usar as classes Web Forms.

Figura 60. Arquitetura da Plataforma Framework.



Fonte: <https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet>. Acesso em: 1º jun. 2019.

Recursos de programação

Na plataforma .NET, é possível trabalhar com várias linguagens de programação suportadas pelo Framework. Entretanto, devido à grande facilidade de implementação e à possibilidade de utilizar em várias arquiteturas (*web, mobile, desktop*), a linguagem mais utilizada nessa plataforma é a C# (C sharp).

Herança, encapsulamento, polimorfismo e construtores em C#

A sintaxe para implementar conceitos de Orientação a Objetos em C# é praticamente igual ao Java em muitos aspectos. Veja o seguinte exemplo:

Figura 61. Herança, Polimorfismo e Encapsulamento em C#.

```

Program.cs* - Program
Main(string[] args)
1  using System;
2
3  class ContaPadrao
4  {
5      private double _saldo_atual = 0;
6
7      public void Retirar(double quantia) ...
12
13      public void Depositar(double quantia)...
17
18      public double ConsultaSaldo() ...
22
23      class ContaCorrente : ContaPadrao {
24
25          public void Retirar(double quantia) ...
26
27      }
44
45      class ContaPoupanca : ContaPadrao {
46
47          public void Retirar(double quantia) ...
62
63      }
64
65      class Program
66      {
67          static void Main(string[] args) ...
81

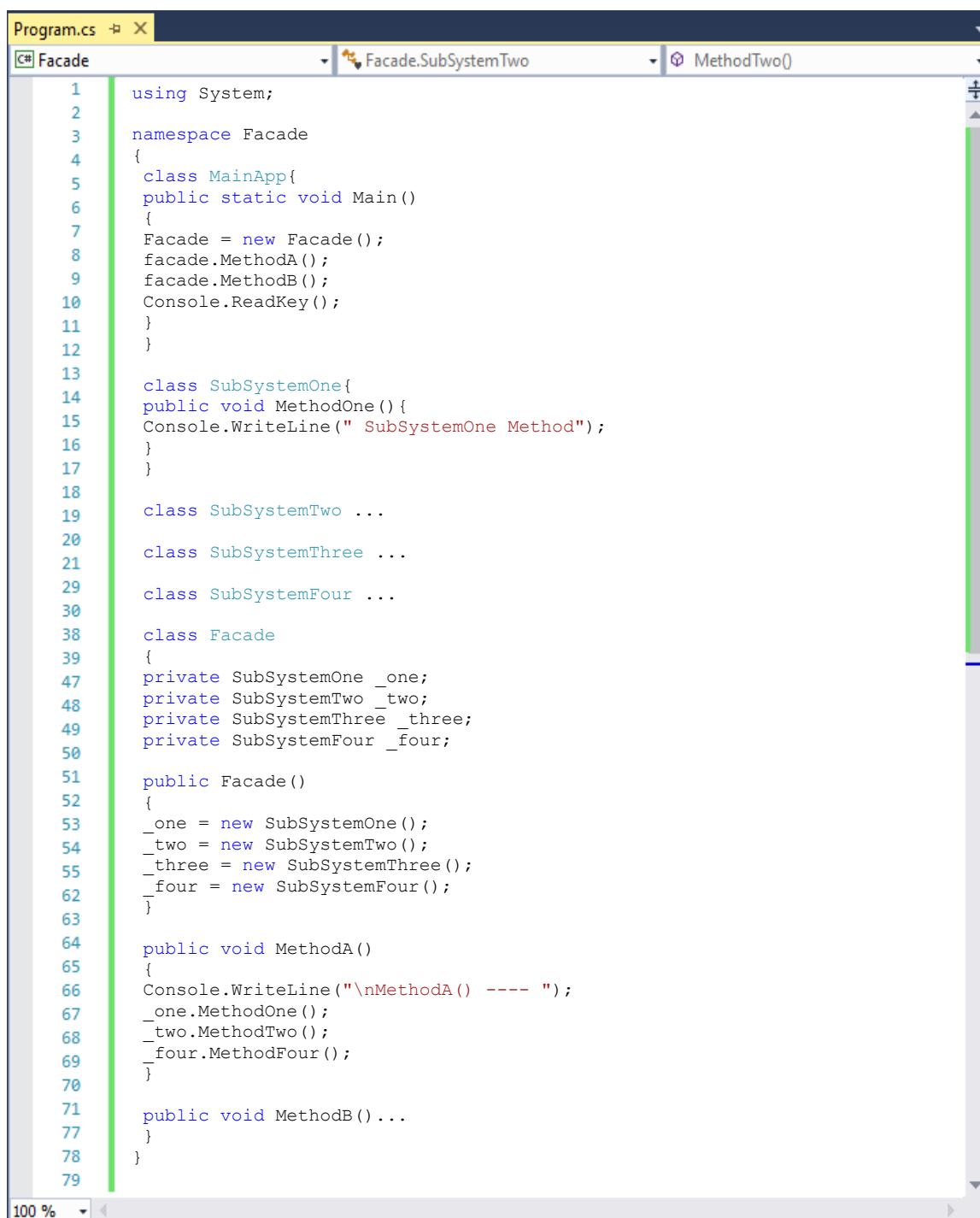
```

Fonte: Elaborada pelo autor.

Implementando padrão GoF – Facade

No capítulo sobre *Facade*, vimos que, em sistemas maiores, a comunicação entre subsistemas pode ser complexo e desorganizado. Com o padrão *Facade*, é possível unificar vários membros requeridos, para um conjunto de interfaces. O exemplo a seguir possui dois métodos e quatro subsistemas organizados em uma única interface.

Figura 62. Exemplo de Padrão *Facade*.



```

1  using System;
2
3  namespace Facade
4  {
5      class MainApp{
6      public static void Main()
7      {
8          Facade = new Facade();
9          facade.MethodA();
10         facade.MethodB();
11         Console.ReadKey();
12     }
13
14     class SubSystemOne{
15     public void MethodOne(){
16     Console.WriteLine(" SubSystemOne Method");
17     }
18
19     class SubSystemTwo ...
20
21     class SubSystemThree ...
22
23     class SubSystemFour ...
24
25     class Facade
26     {
27     private SubSystemOne _one;
28     private SubSystemTwo _two;
29     private SubSystemThree _three;
30     private SubSystemFour _four;
31
32     public Facade()
33     {
34         _one = new SubSystemOne();
35         _two = new SubSystemTwo();
36         _three = new SubSystemThree();
37         _four = new SubSystemFour();
38     }
39
40     public void MethodA()
41     {
42         Console.WriteLine("\nMethodA() ---- ");
43         _one.MethodOne();
44         _two.MethodTwo();
45         _four.MethodFour();
46     }
47
48     public void MethodB()...
49     }
50 }

```

Fonte: <https://www.dofactory.com/net/facade-design-pattern>. Acesso em: 1ª jun. 2019.

CAPÍTULO 3

Padrão PHP

PHP (um acrônimo para PHP: *Hypertext Preprocessor*) é uma linguagem de *script* geralmente usada no desenvolvimento de web, orientada a objetos, na maioria das vezes utilizada do “lado do servidor” para gerar páginas dinâmicas.

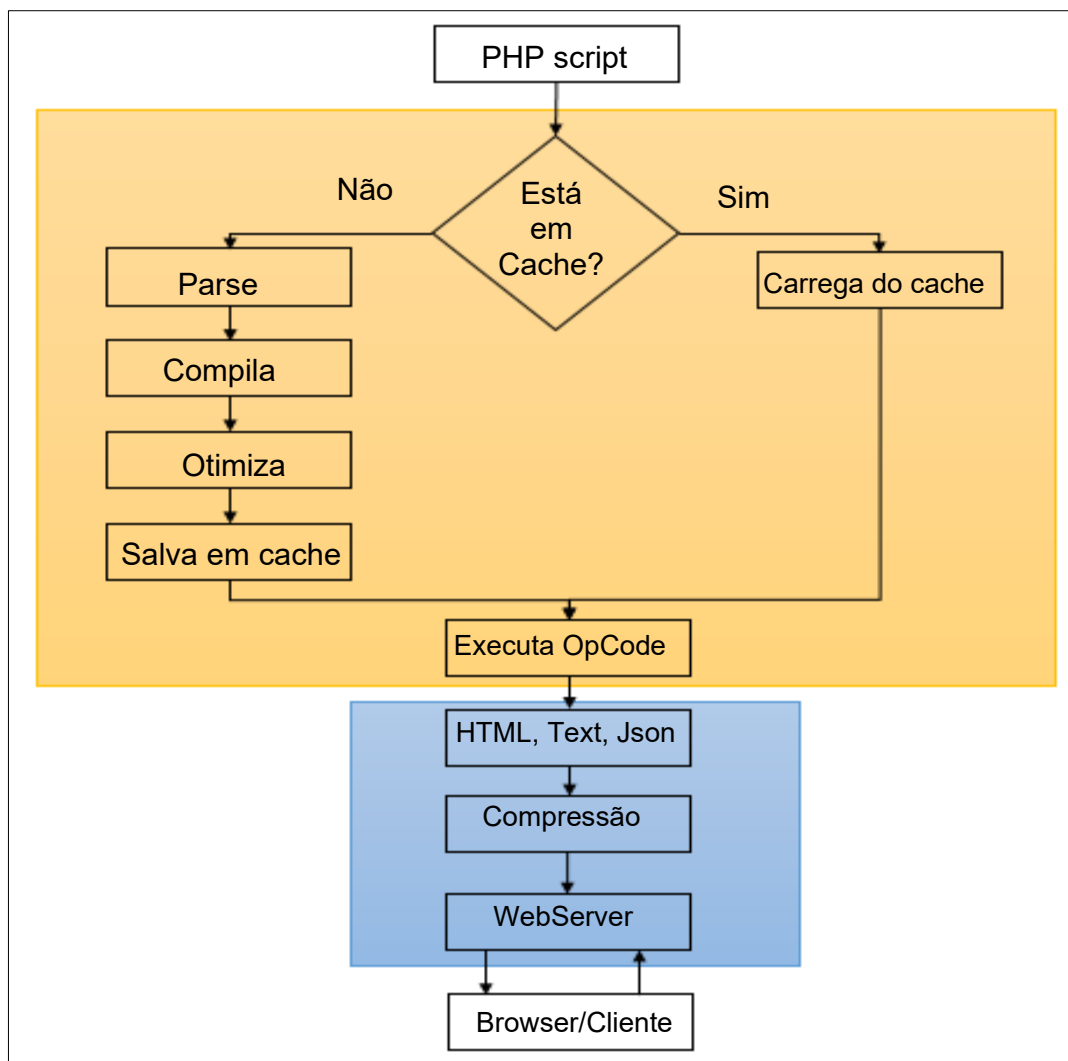
Linguagens de *scripts* são linguagens que não geram um programa executável, como o Java e o .NET. São utilizadas para fazer manutenção, como o *shell script*, no desenvolvimento do lado do cliente, como *javascript*, e também do lado do servidor, como o PHP. Para a plena utilização em páginas dinâmicas, é necessário instalar um *webserver* de terceiros, como o Apache, para que o PHP seja executado.

O PHP, como é conhecido hoje, é na verdade o sucessor para um produto chamado PHP/FI. Criado em 1994 por Rasmus Lerdof, a primeira encarnação do PHP foi um simples conjunto de binários *Common Gateway Interface* (CGI) escrito em linguagem de programação C. Originalmente usado para acompanhamento de visitas para seu currículo *on-line*, ele nomeou o conjunto de *scripts* de *Personal Home Page Tools*, mais frequentemente referenciado como *PHP Tools*. Ao longo do tempo, mais funcionalidades foram desejadas, e Rasmus reescreveu o *PHP Tools*, produzindo uma maior e rica implementação. Esse novo modelo foi capaz de interações com banco de dados, fornecendo uma estrutura na qual os usuários poderiam desenvolver simples e dinâmicas aplicações web, como um livro de visitas. Em junho de 1995, Rasmus liberou o código fonte do *PHP Tools* para o público, o que permitiu que desenvolvedores usassem da forma como desejassem. Isso permitiu – e encorajou – usuários a fornecerem correções para *bugs* no código e, em geral, aperfeiçoá-lo. https://www.php.net/manual/pt_BR/history.php.php. Acesso em: 1º jun. 2019.

Arquitetura

Seu mecanismo de execução funciona da seguinte maneira:

Figura 63. Mecanismo de execução do PHP.



Fonte: Elaborada pelo autor.

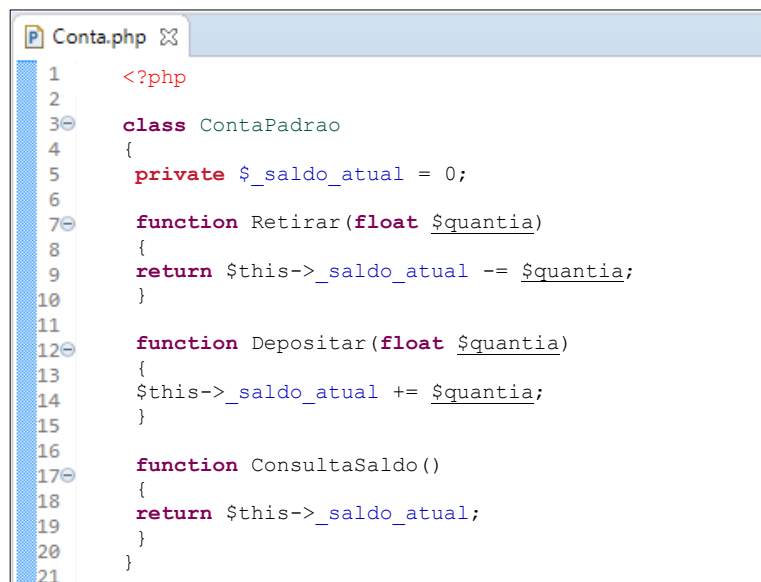
Recursos de programação

A sintaxe de programação tem muita similaridade com o JAVA e o C#, pois ambos utilizam dicionários do compilador C e C++. Uma das grandes diferenças entre Java e C# para PHP é a *tipagem* dos dados. Java e C# são linguagens de programação conhecidas como fortemente *tipadas*. Isso significa que uma variável, quando inicializada com um tipo *double*, por exemplo, vai permanecer assim até o momento de sua “morte”. Já no PHP, o compilador consegue reconhecer o tipo de dados que o programador está tentando atribuir à variável. O compilador, nesse momento, faz uma inferência de tipos, ou seja, se atribuir um número decimal a uma variável e, em seguida, atribuir um número inteiro, o compilador reconhecerá automaticamente, não emitirá nenhuma mensagem de erro e dará sequência ao fluxo de código. Seus métodos também não esperam que o tipo de retorno seja informado, e o PHP retorna o tipo de dados que estará no comando de retorno.

Herança, encapsulamento, polimorfismo e construtores em PHP

A forma de polimorfismo implementar o também veio da forma como o C++ e se assemelha muito com o C# e Java.

Figura 64. Classe ContaPadrao.



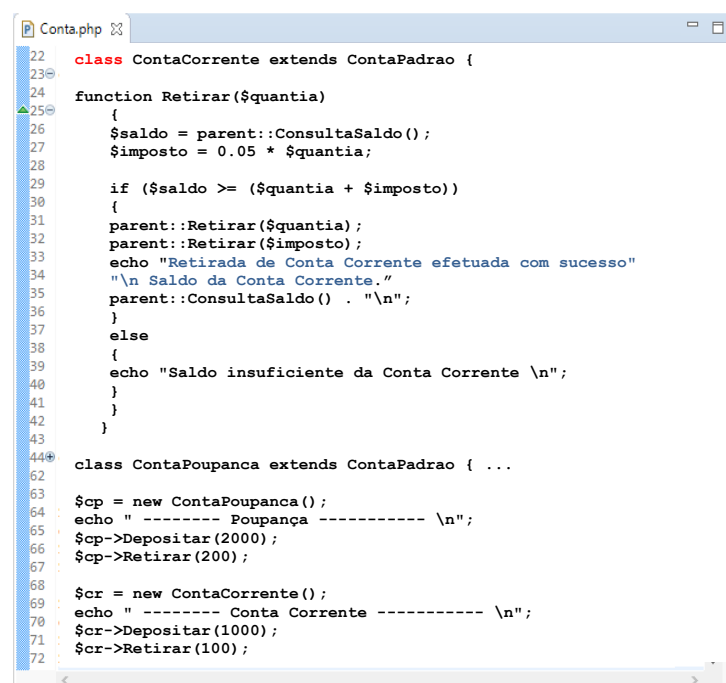
```

1  <?php
2
3  class ContaPadrao
4  {
5      private $_saldo_atual = 0;
6
7      function Retirar(float $quantia)
8      {
9          return $this->_saldo_atual -= $quantia;
10     }
11
12     function Depositar(float $quantia)
13     {
14         $this->_saldo_atual += $quantia;
15     }
16
17     function ConsultaSaldo()
18     {
19         return $this->_saldo_atual;
20     }
21 }
  
```

Fonte: Elaborada pelo autor.

Em seguida, foram criadas as classes *ContaCorrente* e *ContaPoupança*, que herdaram de *ContaPadrao*:

Figura 65. Subclasses de ContaPadrao.



```

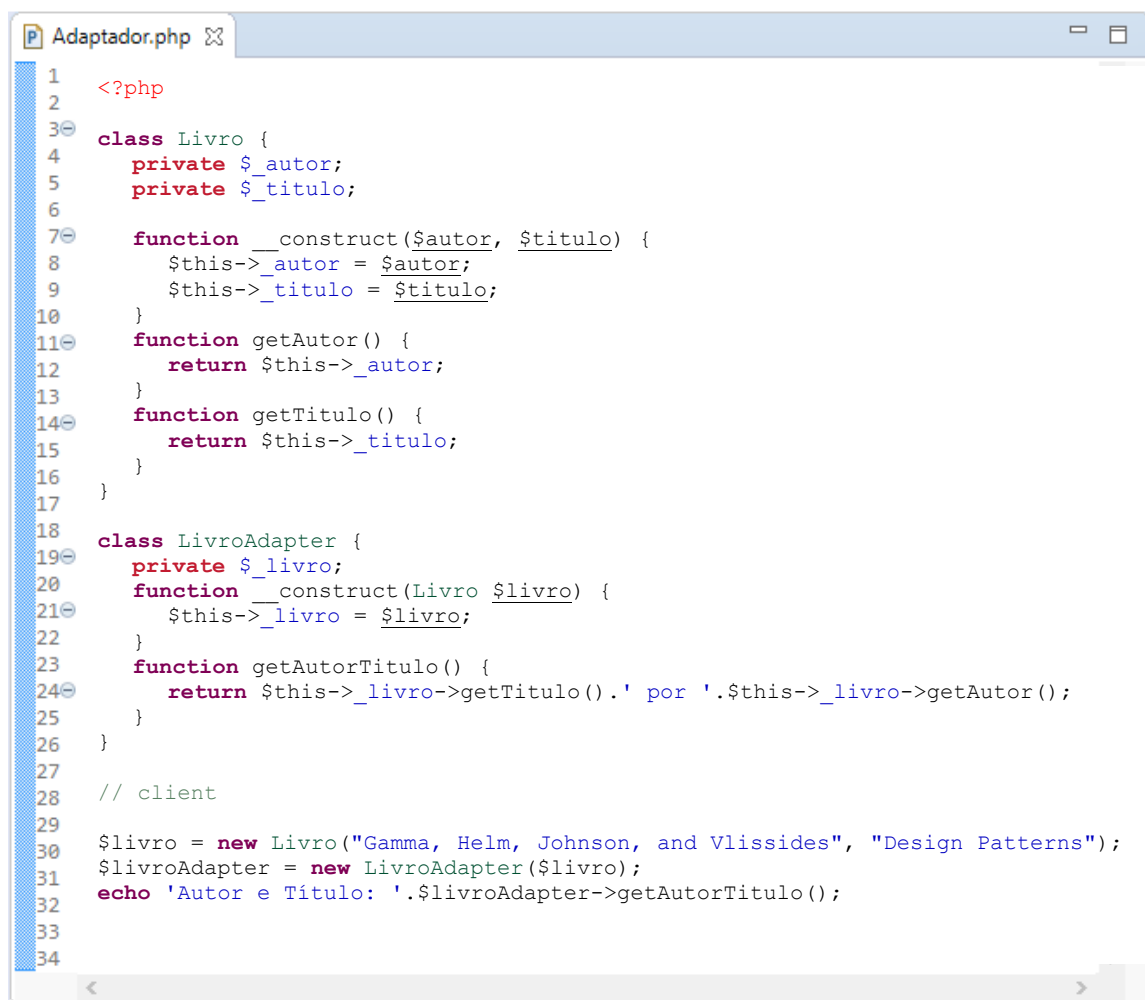
22  class ContaCorrente extends ContaPadrao {
23
24      function Retirar($quantia)
25      {
26          $saldo = parent::ConsultaSaldo();
27          $imposto = 0.05 * $quantia;
28
29          if ($saldo >= ($quantia + $imposto))
30          {
31              parent::Retirar($quantia);
32              parent::Retirar($imposto);
33              echo "Retirada de Conta Corrente efetuada com sucesso"
34              "\n Saldo da Conta Corrente."
35              parent::ConsultaSaldo() . "\n";
36          }
37          else
38          {
39              echo "Saldo insuficiente da Conta Corrente \n";
40          }
41      }
42  }
43
44  class ContaPoupanca extends ContaPadrao { ...
45
46      $cp = new ContaPoupanca();
47      echo " ----- Poupança ----- \n";
48      $cp->Depositar(2000);
49      $cp->Retirar(200);
50
51      $cr = new ContaCorrente();
52      echo " ----- Conta Corrente ----- \n";
53      $cr->Depositar(1000);
54      $cr->Retirar(100);
55  }
  
```

Fonte: Elaborada pelo autor.

Implementando padrão GoF – Adapter

Como foi estudado, *Adapters* são úteis se você quiser usar uma classe que não tenha exatamente os métodos exatos dos quais você precisa, e você não pode mudar a classe original. No exemplo a seguir, foi utilizada a classe *Livro*, que possui dois métodos *getTitulo()* e *getAutor()*. Porém o cliente espera um método *getAutorLivro()*. Para “adaptar”, será feita uma classe *LivroAdapter()*, que recebe uma instância de *Livro* e usa os métodos *getAutor()* e *getTitulo()* em seu método *getAutorLivro()*.

Figura 66. Exemplo de *Adapter* em PHP.



```
1  <?php
2
3  class Livro {
4      private $_autor;
5      private $_titulo;
6
7      function __construct($autor, $titulo) {
8          $this->_autor = $autor;
9          $this->_titulo = $titulo;
10     }
11     function getAutor() {
12         return $this->_autor;
13     }
14     function getTitulo() {
15         return $this->_titulo;
16     }
17 }
18
19 class LivroAdapter {
20     private $_livro;
21     function __construct(Livro $livro) {
22         $this->_livro = $livro;
23     }
24     function getAutorTitulo() {
25         return $this->_livro->getTitulo(). ' por ' . $this->_livro->getAutor();
26     }
27 }
28 // client
29
30 $livro = new Livro("Gamma, Helm, Johnson, and Vlissides", "Design Patterns");
31 $livroAdapter = new LivroAdapter($livro);
32 echo 'Autor e Título: ' . $livroAdapter->getAutorTitulo();
33
34
```

Fonte: Adaptada de https://sourcemaking.com/design_patterns/adapter/php.

Referências

DALL’OGLIO, Pablo. **PHP: Programando com orientação a objetos**. 2. ed. São Paulo: Novatec Editora. 2009. p. 571.

ENGHOLM, Júnior Hélio. **Análise de design Orientados a Objetos**. São Paulo: Novatec Editora, 2013.

GUEDES, Gilleanes T. A. **UML 2: uma abordagem prática**. 2. ed. São Paulo: Novatec Editora, 2011. p. 479

GAMMA, Erich. **Padrões de Projeto: soluções reutilizáveis de software orientado a objetos**. / Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Porto Alegre: Bookman, 2000.

LEITE, Jair C. **Notas de Aula de Engenharia de Software**. Acesso em: 10 jun. 2019.

OMG – Object Management Group. Unified Modeling Language, versão 2.5.1. <https://www.omg.org/spec/UML/2.5.1/PDF>. Acessado em: 9 maio 2019.

PRESSMAN, Roger S. **Engenharia de Software**. São Paulo: Pearson Makron Books, 1985.

PFLEEGER, Shari Lawrence. **Engenharia de Software: teoria e prática**. 2. ed. São Paulo: Prentice Hall, 2004, p. 532.

RUMBAUGH, James *et al.* **Modelagem e Projetos baseados em Objetos**. Rio de Janeiro: Campus, 1994.

ROYCE, Dr. Winston W. **Managing the Development of large software Systems**, Disponível em: <http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf>. Acesso em: 1º jun. 2019.

SANDERS, William. **Aprendendo padrões de projeto em PHP**. São Paulo: Novatec, 2013.

SAUVÉ, Jaques Philippe. **O que é Polimorfismo**. Disponível em: http://www.dsc.ufcg.edu.br/~jacques/cursos/p2/html/oo/o_que_e_polimorfismo.htm. Acesso em: 1º jun. 2019.

SOMMERVILLE, Ian. **Engenharia de Software**. Tradução: Ivan Bosnic e Kalinka G. de O. Gonçalves; revisão técnica Kechi Hirama. 9. ed. São Paulo: Pearson Prentice Hall, 2011. p. 551.

Sites

<https://www.dimap.ufrn.br/~jair/ES/c1.html>.

<https://brstatic.guiainfantil.com/pictures//2315-dibujos-de-una-lancha-para-colorear-dibujos-de-barcos-para-ninos.jpg>.

<https://www.artesanatopassoapassoja.com.br/wp-content/uploads/2019/02/moto.jpg>.

<https://www.artesanatopassoapassoja.com.br/wp-content/uploads/2019/02/fusca-novo.jpg>.

<https://www.omg.org/spec/UML/2.5.1/PDF>.

<https://cdn.visual-paradigm.com/guide/uml/uml-aggregation-vs-composition/uml-association-aggregation-composition.png>.

<http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf>.

https://www.ibm.com/developerworks/rational/library/content/03July/1000/1251/1251_bestpractices_TPO26B.pdf.

<https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html>.

https://sourcemaking.com/design_patterns/adapter/php.

<https://hub.packtpub.com/what-is-multi-layered-software-architecture/>.

<https://docs.oracle.com/cd/E19644-01/817-5448/dgdesign.html#wp18227>.