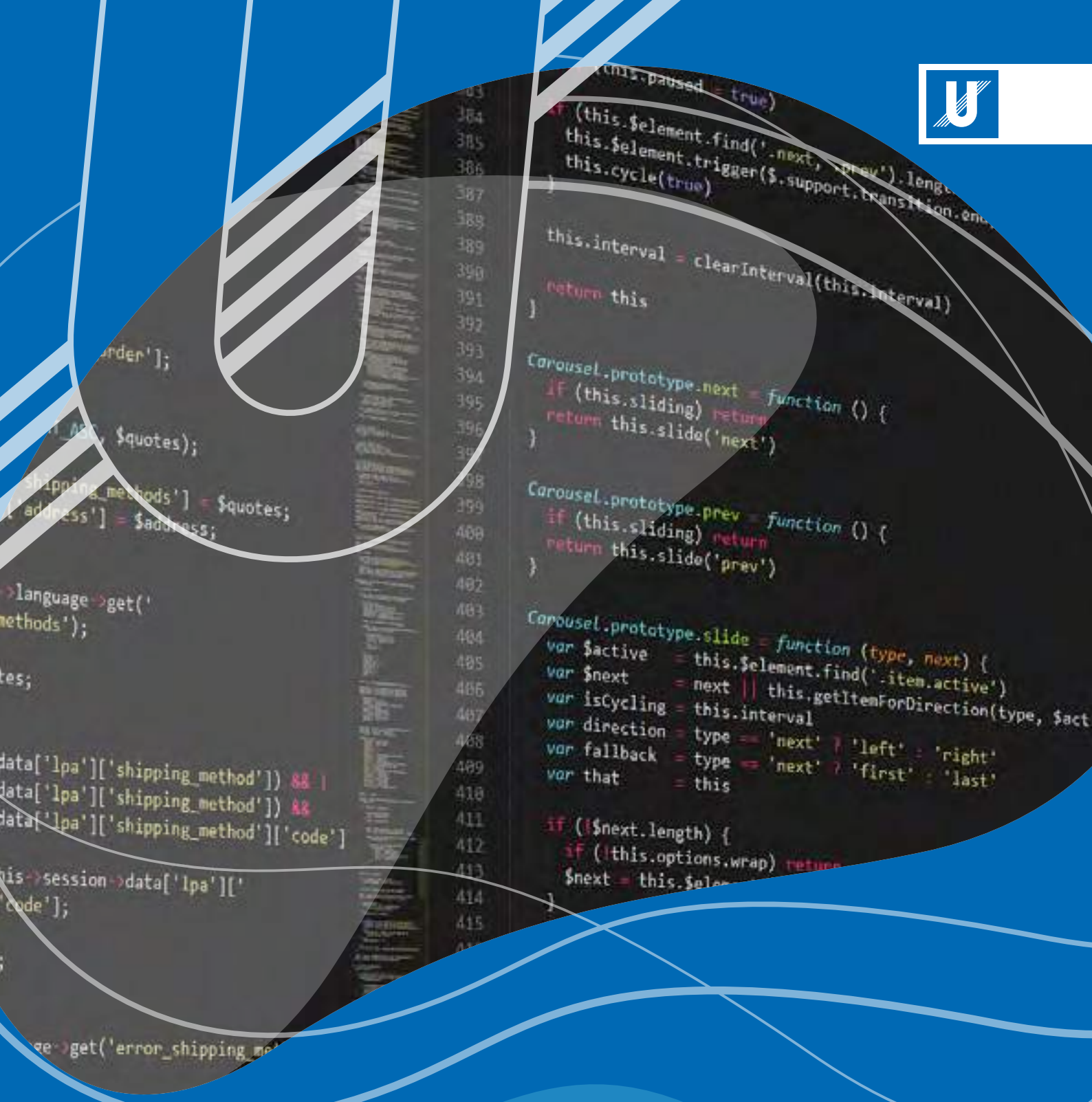


LINGUAGENS DE PROGRAMAÇÃO

Prof. Esp. RONNIE CESAR TOKUMO

Universidade de Marília
Avenida Hygino Muzzy Filho, 1001
CEP 17.525–902- Marília-SP

*Todos os gráficos, tabelas e esquemas são creditados à autoria, salvo quando indicada a referência. Informamos que é de inteira responsabilidade da autoria a emissão de conceitos.

Imagens, ícones e capa: ©envato, ©pexels, ©pixabay, ©Twenty20 e ©wikimedia

Conteúdo PROGRAMÁTICO

Introdução

A programação é uma das áreas mais importantes da TI (tecnologia da informação) e permite o desenvolvimento de soluções computacionais para problemas reais que possam ser atendidos através de uma combinação de hardware e software.

Esta premissa se baseia na ideia de que um hardware sem software, para controlar suas funcionalidades, acaba se tornando um dispositivo eletromecânico apenas, em que não há praticamente automatização alguma e assim, não existe a necessidade de programação como ocorre em máquinas de operação manual.

A implementação de software pode ser feita de mais de uma maneira, normalmente, alterando-se a lógica utilizada para o atendimento dos requisitos e funcionalidades, escolha de uma linguagem de programação adequada para cada finalidade e aspectos como qualificação da mão de obra disponível na linguagem desejada, reaproveitamento de código já feito anteriormente, etc.

De qualquer forma, não existe uma melhor linguagem de programação, na verdade, pois os problemas que podem ser solucionados são tão variados que não há uma única linguagem ideal para todos eles, e assim, com o passar do tempo, algumas linguagens se tornam mais populares, outras perdem força ou nem chegam a ter expressão no mercado.

A intenção desta disciplina é mostrar alguns aspectos que diferenciam estas linguagens e permitem a enorme variedade de soluções computacionais disponíveis no mercado, sendo que a medida que o tempo passa e a TI encontra cada vez mais nichos para ser utilizada, mais variado se torna o uso das linguagens de programações existentes, permitindo que as pessoas possam escolher entre várias linguagens a aprender e se aperfeiçoar, sabendo que existe mercado de trabalho para absorvê-las se demonstrar habilidade de programação suficiente.

Durante as aulas deste material, é possível conhecer alguns aspectos importantes de algumas linguagens de programação distintas, em sua forma de uso e aplicabilidade, assim como conhecer alguns dos diferentes paradigmas utilizados por estas linguagens, para o desenvolvimento de softwares.

Bons estudos!

Paradigmas de Programação

O desenvolvimento de soluções computacionais depende de uma série de fatores, pois representa o completo desenvolvimento e execução de um projeto que parte de problemas reais que possam ter soluções desenvolvidas com base no uso de softwares.

Além do software, é necessário pensar em detalhes relacionados a hardware, e plataformas onde será utilizada a solução desenvolvida, perfil de usuários, limites de tempo e investimento para o desenvolvimento da solução, dentre outros aspectos relevantes que também variam de um projeto para outro.

Para os estudos neste material, aspectos como estes são todos pertinentes, mas não fazem parte do foco dos estudos, servindo eventualmente como elementos de apoio para os estudos das técnicas e ferramentas de desenvolvimento de software.

Temos muitas possibilidades no desenvolvimento de softwares, pois o leque de possibilidades vem aumentando gradativamente com o passar do tempo e com a evolução da tecnologia da informação, fazendo com que projetos inteiros desenvolvidos em determinado tempo sejam inteiramente abandonados e novas versões destes projetos sejam criados com o propósito de atender às necessidades dos novos tempos, como depois do surgimento das interfaces gráficas, redes de computadores ou a popularização da Internet.

Os chamados paradigmas não se referem especificamente a uma ou outra linguagem de programação, mas às diferentes formas de implementar soluções computacionais que servem de base para que as linguagens de programação sejam definidas em termos de funcionalidades semântica e sintáticas, que serão vistas ao longo do material, sendo conceitos bastante relevantes para os estudos desta disciplina.

Um exemplo é que nem todas as linguagens de programação podem desenvolver soluções que possam ser funcionais na web, sendo estas mais específicas para determinadas finalidades ou muito antigas e não atualizadas para esta tecnologia, assim como existem linguagens que foram criadas já com a Internet popularizada e sendo o foco da maior parte do que é produzido em termos de software.

Estudar os diferentes paradigmas de programação é importante para todos que trabalham com desenvolvimento de software, pois o mercado muda de tempos em tempos devido a evoluções tecnológicas como já citado e precisa que os desenvolvedores, e principalmente líderes de projetos, possam avaliar os melhores caminhos e ferramentas a serem utilizados de forma que um projeto possa ser bem sucedido.

Esse sucesso no desenvolvimento de software não depende simplesmente de se ter uma mão de obra bem capacitada em determinada linguagem utilizada pela empresa no momento, ou foco apenas na linguagem dominante do mercado, mas de ser capaz de compreender e absorver novas ideias que surgem e que possam realmente agregar melhorias no processo de desenvolvimento de software.

Partindo então da ideia de que paradigmas de programação se referem a como programar e não as linguagens de programação propriamente ditas, imagina-se que não é necessário conhecer linguagem nenhuma para esta disciplina.

É importante que se tenha uma noção da lógica de programação e interpretação de algoritmos ao menos, sendo desejável que se tenha também a capacidade de desenvolvimento de algoritmos para melhor compreender exemplos, independente de aspectos e particularidades de uma linguagem de programação.

Ter conhecimento em alguma linguagem de programação, seja qual for, sendo este conhecimento básico, intermediário ou avançado, também ajuda em diferentes graus de relevância, assim como conhecer mais de uma linguagem também possui algumas vantagens, mas não são pré-requisitos.

Existem paradigmas bastante tradicionais e utilizados como base para várias linguagens de programação como os paradigmas conhecidos como programação estruturada e programação orientada a objetos que são os mais populares, paradigmas mais específicos como a programação funcional ou a programação lógica, além de outros como a programação orientada a aspectos, programação declarativa, etc.

Ao longo das aulas, os principais paradigmas e conceitos e exemplos sobre eles serão tratados de forma a oferecer ao leitor, condições de compreender um pouco das diferenças entre os diferentes paradigmas, suas aplicações e algo sobre linguagens que se baseiem nestes paradigmas.

Uma forma de conhecer os diferentes paradigmas é avaliando diferentes exemplos de códigos e scripts gerados em diferentes linguagens que utilizem estes paradigmas de forma a comparar a maneira como são desenvolvidas soluções distintas para problemas similares. A partir do momento em que se compreendem os diferentes paradigmas e suas características, é possível compreender códigos em diferentes linguagens e aprendê-las mais rapidamente.

Breve História das Linguagens

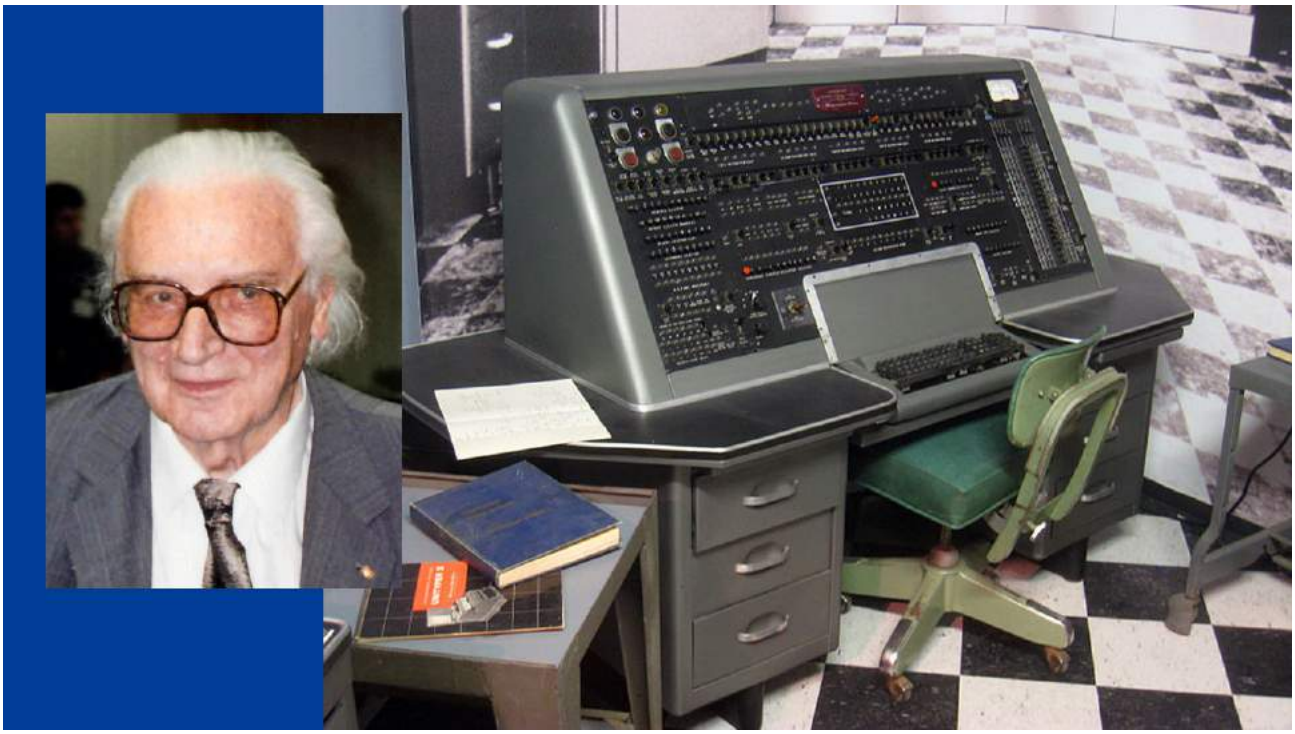
Desde meados do século XX já eram desenvolvidas linguagens de programação, mas bem diferentes em suas finalidades e forma de programação, pois nesta época o hardware era muito diferente do que se utiliza atualmente em termos de dimensões dos componentes, desempenho e quantidade de recursos disponíveis.

Em função destas diferenças e da pouca ou quase nenhuma variedade de hardware existente, o software não era genérico e adaptável a todo hardware disponível como ocorre hoje com linguagens que permitem que um mesmo software possa ser executado em diferentes plataformas de hardware e sistemas operacionais.

Na década de 1940, o engenheiro alemão Konrad Zuse (1910 - 1995) pode ter sido o criador da primeira linguagem de programação chamada Plankalkül, desenvolvida por ele para hardware criado por ele mesmo, mostrando assim o grau de especificidade de hardware e software da época.

Ainda nesta década, o UNIVAC I foi considerado o primeiro computador vendido nos Estados Unidos de forma comercial e para o desenvolvimento de software para ele, foi implementada a linguagem Short Code, própria para a arquitetura de seu hardware.

Konrad Zuse e o UNIVAC I, hoje peça de museu em Boston, Massachusetts



Fonte: Wikipedia | Wikimedia.

Desde a década de 1950, as linguagens de programação vêm sendo criadas, umas se popularizando e outras nem chegando a serem conhecidas pelos desenvolvedores em geral. Também é possível acompanhar neste vídeo a evolução de algumas linguagens nos últimos anos e declínio de outras no mesmo período, mas é importante citar como algumas duram por décadas como relevantes para o mercado de desenvolvimento de software.

Acesse o link: **Disponível aqui**

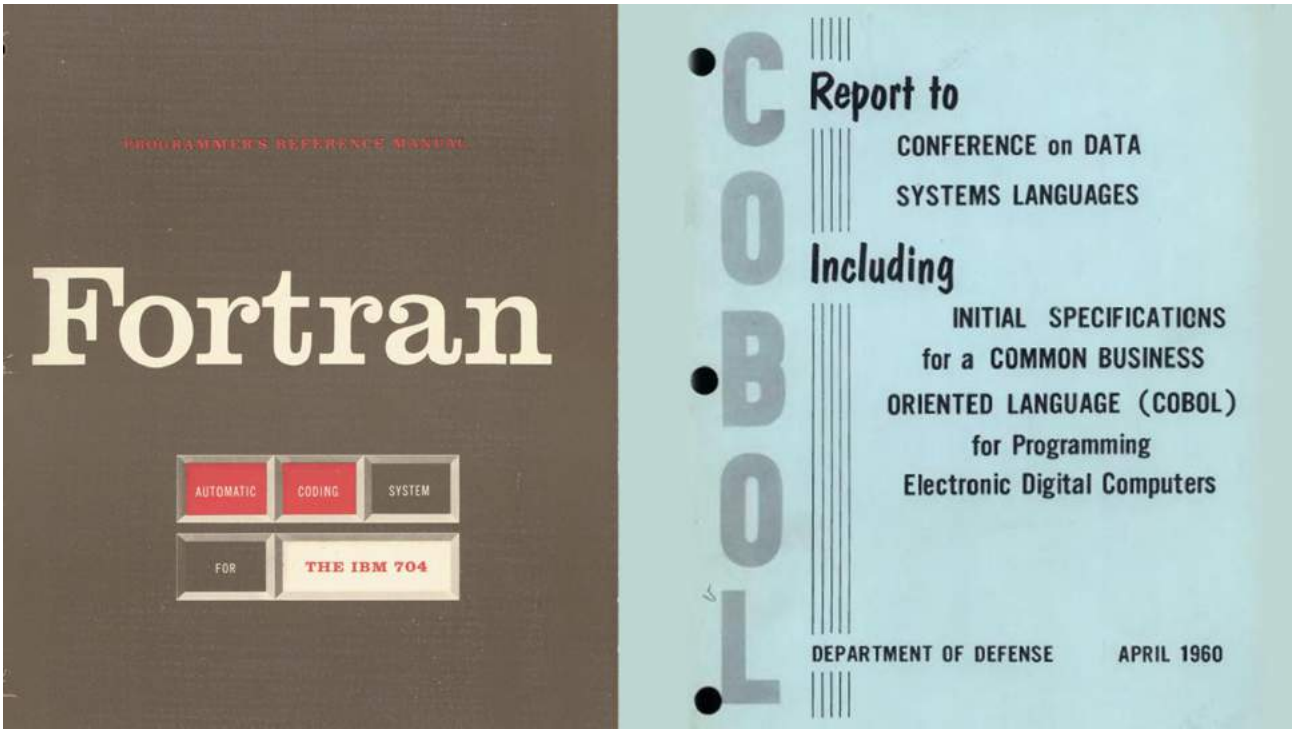
Na década seguinte de 1960, a linguagem FORTRAN (IBM Mathematical FORmula TRANslation System) foi criada dentro de um paradigma mais rígido, sendo depois atualizada para incorporar outro paradigma é um exemplo de evolução de uma linguagem para se adaptar à evolução tecnológica e da própria informática.

Também surgiu a linguagem Lisp (LISt Processor) que tinha por base outro paradigma e proporcionou uma ferramenta própria para os estudos iniciais da chamada inteligência artificial.

ALGOL (Algorithmic Language) foi outra linguem implementada na época que acabou servindo de base para outras linguagens conhecidas a partir da forma como foi estruturada sua semântica e sintaxe.

E mais ao final dessa década já adentrando na década de 1960, surgiu uma das mais influentes linguagens dessa época e algumas décadas mais chamada de COBOL (COmmon Business Oriented Language) que foi rapidamente popularizada devido a sua capacidade de manipulação de bancos de dados.

Capas da primeira apostila de Fortran e de Cobol



Fonte: Wikipedia | Wikimedia.

Ainda existem sistemas ditos legados implementados há décadas que sofrem apenas manutenções e atualizações mantendo a mesma linguagem COBOL como base em computadores do tipo Mainframe, ainda bastante confiáveis para estes softwares e bases de dados.

Já na década de 1970, o surgimento da linguagem BASIC (Beginner's All-purpose Symbolic Instruction Code) foi uma das responsáveis pela futura onda de popularização dos computadores pessoais que viriam a ser comercializados a partir da década de 1980.

Esta linguagem mais fácil de interpretar códigos e de implementá-los pelo seu alto nível, permitiu que muitas pessoas iniciassem estudos na área de programação e uma explosão de softwares novos surgisse gradativamente.

Nem todas as linguagens de programação foram bem sucedidas e se popularizaram, ou não passaram de alguns anos de uso como ocorreu com linguagens como APL, SIMULA, ALGOL, B e CPL, mas mesmo não sendo muito utilizadas, puderam contribuir em alguns aspectos com a evolução das linguagens que surgiram posteriormente.

Uma das linguagens que teve como base outra que deixaria de ser utilizada foi a popularíssima Pascal que tinha como base a linguagem ALGOL, sendo a linguagem implementada na década de 1970 suficientemente completa e de boa interpretação de códigos, tornando-se uma linguagem de base para o meio acadêmico.

Dela, surgiu uma variante muito popular e já adaptada ao desenvolvimento de interfaces gráficas chamada de Delphi nos anos 2000, tendo até hoje muitos softwares desenvolvidos comercialmente em uso.

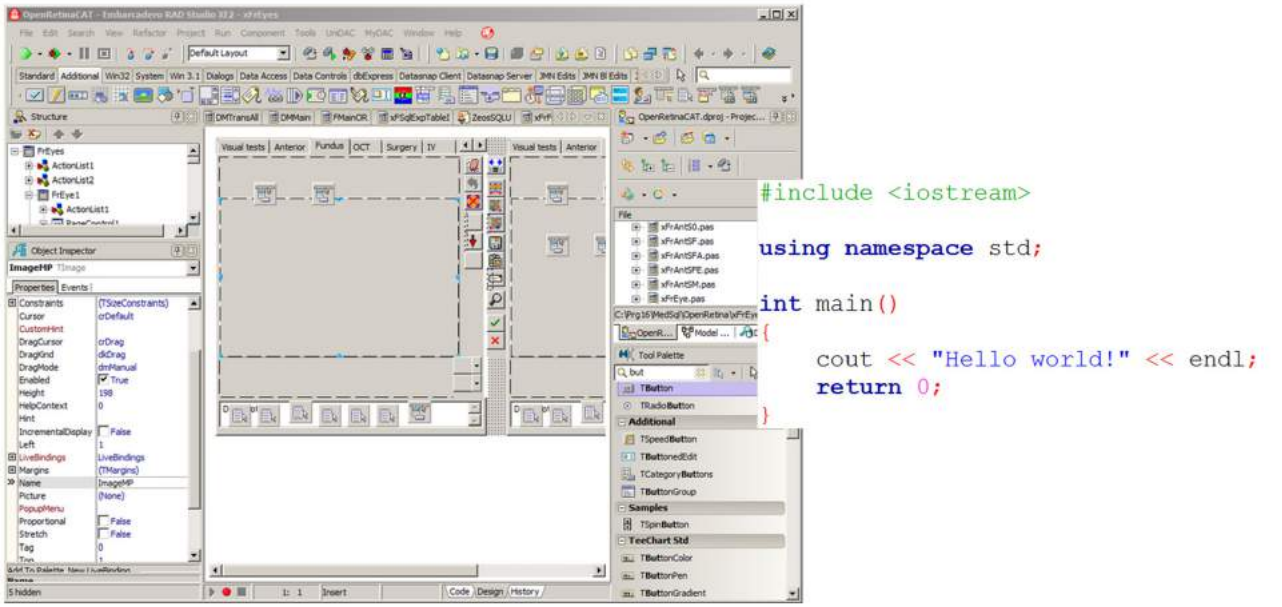
Outra linguagem que aproveitou a estrutura de linguagens anteriores e surgiu na década de 1970 foi a linguagem C, baseada nos conceitos de linguagens como B e CPL dos anos 1960.

A linguagem C, de propósito geral, foi tão bem estruturada e capaz de ser aprimorada que se popularizou muito ao ponto de desbancar outras populares e se tornar a linguagem mais utilizada do mundo durante determinado período.

A linguagem C possuía uma grande capacidade de solucionar problemas em função de permitir o desenvolvimento de software que converse diretamente com o hardware e gerenciar o uso destes recursos por usuários, permitindo o desenvolvimento desde sistemas operacionais para diferentes tipos de hardware até softwares de aplicação e jogos.

Essa linguagem serviu de base para diversas outras linguagens que foram surgindo seguindo a mesma nomenclatura como C++ na década de 1980, C# e Objective-C décadas depois, para atender às demandas da época como o uso de interfaces gráficas, novos paradigmas e o surgimento dos dispositivos móveis.

Interface Delphi e o programa Hello World, escrito na linguagem de programação C++



Fonte: Wikimedia.

Ainda na década de 1970, surgiram linguagens como Scheme e ML que tinham como base a linguagem Lisp, tendo também foco na programação mais específica e não para propósitos gerais.

Já na década de 1980, além da linguagem C++, surgiram também as linguagens Smalltalk, que utiliza como base a chamada programação OO (Orientação a Objetos ou Object Oriented), Perl, Quick Basic e Haskell.

Na década de 1990, muitas das linguagens mais relevantes para o mercado atual surgiram como as linguagens Visual Basic, Python, PHP, Ruby, Java e JavaScript que juntas representam uma grande porcentagem em termos de uso para desenvolvimento de software nos dias atuais, observando que a linguagem Python recentemente despontou para o mercado como ideal para aplicações baseadas nas tecnologias mais atuais e vem então ultrapassando todas as demais em termos de uso por desenvolvedores.

Nos anos 2000, muitas linguagens sofreram atualizações e assim foram implementadas versões mais modernas de linguagens já citadas, e assim, foram implementadas as versões 5.6, 6.0 e 7.0 de Java, surgiu a linguagem C# que logo evoluiu para as versões 2.0, 3.0 e 4.0 na mesma década.

Foram também desenvolvidas as versões Ruby 1.9 e 1.0, Fortran 2003 e 2008, Python 2.0 e 3.0. Surgiu também nesta década a linguagem Clojure que possui fins específicos também.

Depois, mais recentemente, surgiram as linguagens Hack desenvolvida pela empresa Facebook para desenvolvimento de software para seu portifólio de produtos, Go desenvolvida pela Google de propósito geral e Swift desenvolvida pela Apple para ter uma plataforma moderna e própria para desenvolvimento de aplicativos dará seus produtos.

Existem outras linguagens que foram desenvolvidas neste período todo da história da tecnologia da informação como Lua, Dart, R, TCL, Action Script, Assembly (ou Assembler) e outras que possuem também relevância histórica maior ou menor, tendo destaque para a linguagem R em ascensão devido ao uso crescente da estatística aplicada à TI.

Também existem linguagens que entram nesta categoria pela sua escrita de códigos ou scripts, mas que existem divergências quanto à devida classificação como ocorre com SQL, HTML e CSS que recebem classificações ditas não exatamente de programação, mas tendo função próxima das linguagens citadas anteriormente.

Não é necessário estudar como programar em tantas linguagens de programação, pois seria uma tarefa extremamente exaustiva e sua utilização seria limitada a umas poucas, pois o mercado não necessita de um desenvolvedor de tantas linguagens simultaneamente, pois não se desenvolve aplicações em muitas destas linguagens mais. É interessante saber que existem e ler sobre elas e conhecer seus pontos fortes e limitações, lembrando que antes de estudar uma ou mais linguagens de programação, é preciso conhecer a lógica utilizada na resolução de problemas e possuir certo domínio no desenvolvimento de algoritmos.

Classificação de Paradigmas

Os paradigmas de programação representam formas variadas de se desenvolver soluções computacionais e as linguagens de programação são ferramentas para tal atividade baseadas nestes paradigmas, sendo que algumas linguagens podem utilizar conceitos de mais de um paradigma integrando estes conceitos com o propósito de oferecer soluções mais completas e atender uma maior gama de problemas.

Geralmente, as linguagens de programação são criadas com objetivos previamente definidos como para o desenvolvimento de softwares científicos como no caso da linguagem Fortran, ALGOL e Assembly que possuem capacidade de tratar cálculos complexos de ponto flutuante, ou seja, cálculos com casas decimais.

Também existem linguagens como COBOL criadas para o desenvolvimento de softwares comerciais, linguagens como Lisp, Prolog e Scheme para desenvolvimento na área de IA (Inteligência Artificial), linguagens como C para a programação de SO (Sistemas Operacionais) por serem altamente capazes de manipular hardware mesmo sendo de alto nível, e linguagens como PHP e JavaScript para desenvolvimento de conteúdo para web.

Segundo Sebesta (2011), as linguagens de programação podem ser avaliadas segundo alguns critérios como legibilidade de seus códigos, facilidade na escrita destes e a confiabilidade dos softwares desenvolvidos a partir de cada linguagem. Observe a figura 3 para conhecer estes critérios e suas características relevantes.

Imagem 3: Critérios e características de linguagens de programação.

	CRITÉRIOS		
Caracteristica	**LEGIBILIDADE**	**FACILIDADE DE ESCRITA**	**CONFIABILIDADE**
Simplicidade	●	●	●
Ortogonalidade	●	●	●
Tipos de dados	●	●	●
Projeto de sintaxe	●	●	●
Suporte para abstração		●	●
Expressividade		●	●
Verificação de tipos			●
Tratamento de exceções			●
Apelidos restritos			●

Fonte: Sebesta, 2011.

Algumas características citadas são realmente fundamentais na avaliação de uma linguagem como a simplicidade que é importante nos três critérios de avaliação, pois sendo simplificada, torna seus códigos mais fáceis de compreender e implementar, tendo pouca variedade de estruturas básicas de programação.

Outra característica importante é a ortogonalidade que toma por base a quantidade de combinações entre estruturas de dados e de controle, reduzindo a possibilidade de ambiguidades e exceções, tornando uma linguagem mais compacta e com menor possibilidade de ocorrência de erros devido a exceções. Linguagens como C, por exemplo, permitem que sejam utilizados ponteiros que são operadores de tipo que trabalham com a memória em hardware de diversas formas, reduz a ortogonalidade da linguagem devido à grande possibilidade de combinações possíveis destes operadores e tipos da linguagem.

Outra característica importante se refere aos tipos de dados e estruturas possíveis em uma linguagem que podem facilitar ou dificultar os critérios base de avaliação de uma linguagem, como no caso da linguagem C que não possui tipo lógico e utiliza valores numéricos de forma adaptada para esta funcionalidade, não sendo um defeito da linguagem, mas apenas uma característica que dificulta a legibilidade de códigos e sua interpretação clara muitas vezes.

Por fim, outra característica presente em todos os critérios se baseia no projeto da sintaxe de cada linguagem, e segundo Sebesta (2011), o formato definido como padrão de definição de identificadores, palavras reservadas da linguagem e seu uso na construção de estruturas de controle de fluxo no código, e a forma como se implementa uma instrução na linguagem possui influência direta em sua legibilidade também.

Dentro do critério de facilidade de escrita e confiabilidade, existe ainda a preocupação com o suporte à abstração que se refere a capacidade de definir em processos e estruturas de dados as características relevantes de um problema real.

Também existe a chamada expressividade que está presente na análise de facilidade de escrita e confiabilidade que se refere a aspectos como a simplificação de expressões e existência de comandos que facilitem a codificação como a existência de mais de um comando para implementação de laços de repetição, por exemplo.

Mais especificamente voltadas à confiabilidade, existem características como a verificação de tipos durante o processo de compilação de um código que é melhor que em tempos de execução em relação a custo de processamento, tratamento de exceções para melhoria da garantia de funcionamento de softwares, uso de apelidos ou alias que servem para acessar uma mesma posição de memória como no uso de ponteiros em linguagens como C, e por fim, a própria legibilidade e facilidade de escrita influenciam a confiabilidade de um código.

ABORDAGEM PRÁTICA

Compreender a estrutura de um algoritmo ou código de uma linguagem de programação é o primeiro passo essencial para se aprender a programar, mas antes ainda, é preciso aprender a interpretar problemas e deles obter informações capazes de auxiliar no início do esboço da lógica de uma provável solução computacional.

Categorias de Linguagens de Programação

Linguagens mais antigas eram criadas especificamente para determinados hardwares e o conceito de paradigmas de programação foi sendo definido ao longo dos anos para diferenciar as diferentes abordagens da implementação de software.

As três principais categorias de linguagens de programação (Sebesta, 2011) são imperativas, funcionais e lógicas, mas existe uma derivação vinda da programação imperativa chamada de orientação a objetos.

Pode-se subcategorizar as linguagens de programação em ramificações das categorias principais como no caso das linguagens que gerar scripts como JavaScript, Ruby e Perl, mas estas são basicamente imperativas.

Linguagens voltadas a interfaces gráficas foram desenvolvidas depois de muitas outras, mas são geralmente baseadas em alguma linguagem mais antiga e originalmente de modo textual como C, Basic ou Pascal que geraram versões para interfaces gráficas como Visual-C, Visual Basic e Delphi, além das ditas linguagens visuais como Visual BASIC.NET (VB.NET) (Sebesta, 2011).

A programação lógica, baseada em regras define seus programas de forma distinta da imperativa, pois as regras que servem da base são definidas sem uma ordem lógica e podem ser acessadas livremente de forma que sirvam como uma base complexa de informações que podem ser utilizadas na geração de novas informações.

Numa linguagem imperativa, existe uma ordem lógica para a execução de uma grande variedade de estruturas que podem ser organizadas como independentes ou dependentes umas das outras, mas tendo uma ordem planejada de execução.

As linguagens funcionais trazem uma perspectiva mais matemática para a programação com dados constantes e funções de comportamento fixo que facilita a reutilização e ampliação de aplicações de forma escalar em função da não mutabilidade destas funções.

A programação orientada a objetos possui foco na abstração de dados estruturando-os como atributos de classes que são evoluções das funções em programação imperativa e permitem uma maior independência entre estes componentes que

acabam sendo mais facilmente reutilizados em novos softwares.

Além da divisão dos paradigmas nestas quatro diferentes linhas de programação, algumas outras variantes surgem como propostas como propostas de variações da orientação a objetos, por exemplo, como a programação orientada a eventos ou orientada a aspectos, ou formas alternativas de programação como em linguagens que podem ser consideradas declarativas como HTML e SQL, ou linguagens de programação baseadas em blocos como Scratch e Blocky.

Existem também métodos complementares como a programação dinâmica que busca quebrar problemas mais complexos em outros de menor complexidade que trabalham com grafos, por exemplo.

Nesta ideia da programação dinâmica, este processo de resolução de subproblemas precisa ser bem pensado para que depois, estes subproblemas possam ser integrados e uma solução computacional funcional e completa seja obtida.

A ideia de problemas relacionados a custos de rotas e a obtenção dos melhores resultados é um típico problema que pode ser solucionado com este método que trabalha com entradas de dados que influenciam dinamicamente o processamento e assim, este método é capaz de gerar mudanças na execução em função destas entradas de dados por eventos que podem influenciar processos.

Este método trabalha com um conceito importante que será mais bem explorado mais adiante no material chamado de recursividade, onde trechos de código são chamados repetidamente por eles mesmos, realimentando o trecho com novos dados até que determinada condição encerre este processo.

Existem algoritmos que tratam de uma lógica mais complexa e evoluída do uso de sub-rotinas capazes de chamarem a si mesmas de forma iterativa ou repetitiva, também chamada de recursiva, onde existe um controle da quantidade de iterações baseado em uma condição que muitas vezes envolve valores numéricos que ao serem atingidos, encerram as sucessivas chamadas à sub-rotina e as seguidas repetições.

PARA GABARITAR

Cada paradigma, assim como cada linguagem de programação possui recursos que auxiliam no desenvolvimento de determinados tipos de software, como os tipos de estruturas de dados que utilizam e a forma como as manipulam, sua capacidade de lidar diretamente com hardware ou não, serem compiladas ou interpretadas, etc. Estas especificidades de cada linguagem apoiadas nos paradigmas que elas usam como base permitem que os desenvolvedores tenham opções mais adequadas para cada tipo de problema a ser solucionado. Algumas linguagens se propõem a serem de propósito geral como C++ e Java, e outras mais específicas como Haskell e Prolog, mas todas podem ser aplicadas e o mercado possui espaço para que saiba codificar em muitas destas linguagens, lembrando que o bom programador não é aquele que conhece várias linguagens, mas aquele que sabe pensar em bons algoritmos eficientes e conhece uma boa linguagem para aplicar a estes algoritmos.

Outro paradigma chamado de funcional se baseia em sub-rotinas que executam processos como se fossem funções matemáticas em que se evitam ao máximo mudanças de estado ou dados dinâmicos que são características comuns nos paradigmas imperativo e estruturado que utilizam muitas estruturas de dados variáveis e estados que mudam o tempo todo na execução.

Programação Estruturada – Conceitos

A programação possui formas alternativas de ser feita, e os paradigmas oferecem estas possibilidades, lembrando que a programação em si existe a décadas e de tempos em tempos formas alternativas foram surgindo e oferecendo novas perspectivas para o desenvolvimento de software.

Programação Imperativa

Dentre as várias formas de programação utilizadas no desenvolvimento de software, a programação imperativa foi a inicialmente utilizada pelas linguagens de programação da época como as linguagens Algol, Fortran e COBOL, e posteriormente, as linguagens C, Basic e Pascal.

Este paradigma se baseia na mudança de estados sofrida pelos dados devido ao processamento realizado por comandos contidos no código que são executadas ao longo da execução do software.

Representa um paradigma com simples interpretação de códigos, pois os códigos são implementados dentro da premissa de indicar diretamente as ações a serem realizadas pelo hardware.

A ideia base para este paradigma foi a famosa Máquina de Turing, nomeada em homenagem ao seu elaborador, o matemático britânico Alan Turing, considerado o “pai da computação”. A máquina basicamente trabalhava com base no conjunto de hardware formado por unidade lógica e aritmética, unidade de controle, unidade de entrada e saída, e por fim, memória primária, que eram responsáveis por receber dados, processá-los e retornar resultados, criando a popular base da computação formada por entrada, processamento e saída.

A Máquina de Turing e seu desenvolvedor, Alan Turing

Fonte: Disponível aqui

Algumas características clássicas deste paradigma eram a construção do código como um bloco único de comandos com apenas realizando desvios dentro do código utilizando um artifício que desvia a execução com base em marcadores numéricos ou textuais inseridos no código usando um comando comumente chamado de GOTO.

CONECTE-SE

Conheça a história de Alan Turing no filme “O Jogo da Imitação”. Durante a Segunda Guerra Mundial, o governo britânico monta uma equipe que tem por objetivo quebrar o Enigma, o famoso código que os alemães usavam para enviar mensagens aos submarinos aliados. Um de seus integrantes é o brilhante matemático Alan Turing (Benedict Cumberbatch).

Veja o trailer em: Disponível aqui

Programação Estruturada

Uma alternativa ao paradigma imperativo desenvolvido foi a programação estruturada, que agrega os conceitos fundamentais do paradigma imperativo e acrescenta novos conceitos, definindo assim o paradigma mais popular da história do desenvolvimento de código, presente até os dias atuais em grande parte das linguagens de programação e softwares desenvolvidos.

Estruturas mais complexas são definidas, como o uso das chamadas sub-rotinas e, dentro destas, estruturas de decisão e repetição, blocos de comandos dentro de instruções que permitem a estruturação de softwares com maior complexidade e entrando no universo de linguagens de programação de alto nível.

O conceito do paradigma estruturado surgiu na década de 1950 e foi utilizado inicialmente em linguagens como Algol e COBOL, mas logo outras linguagens tiveram versões implementadas neste paradigma ou foram criadas já nativas nesta forma de programação.

Linguagens como PHP, Pearl e mais recentemente a linguagem GO desenvolvida pela Google são exemplos de linguagens que inicialmente foram desenvolvidas para desenvolvimento estruturado.

Mesmo tendo sido inicialmente pensadas como linguagens para programação estruturada, puderam ajustar-se para absorver outras formas de programação como o uso dos desvios de código do tipo GOTO em alguns casos, ou paradigmas adicionais como o orientado a objetos.

Dentro das diferentes formas de programação, é sempre bom ter alguma informação sobre os diferentes paradigmas ou formas de programação, fazendo com que se adquira maior bagagem para a definição das melhores formas e ferramentas para se desenvolver soluções computacionais. Nesta página são discutidas diferenças entre os paradigmas imperativo e declarativo que é utilizado para definir linguagens com menos estruturas lógicas como HTML e SQL onde os comandos são menos dependentes entre si em scripts criados.

Fonte: **Disponível aqui**

Conceitos Essenciais

Dentro da programação imperativa, o desenvolvimento de código é mais compacto e específico, em que um software projetado para um determinada finalidade, sem tanta preocupação com portabilidade ou reúso de código que representam conceitos mais atuais em que se propõe implementar código que possa ser executado em mais de uma plataforma ou sistema operacional com pouca ou nenhuma alteração de código, e também não era uma preocupação antigamente, a ideia de que partes de um software desenvolvido poderiam ser reaproveitadas no desenvolvimento de outros softwares devido a uma menor demanda e o desenvolvimento bem específico de soluções computacionais.

As estruturas de código são menos complexas e de lógica mais simples, e uma linguagem como BASIC, por exemplo, em suas primeiras versões, possuíam poucas variações de comandos. Originalmente, foi implementada com 15 tipos de comandos,

suficientes para se produzir programas bastante completos em relação às necessidades computacionais da época, sendo sua facilidade de implementação de código e interpretação, motivos pelos quais se popularizou rapidamente.

Os comandos são geralmente construídos com base em uma palavra reservada da linguagem que possua uma finalidade especifica e pré-determinada da linguagem, e esta possua regras de aceite de dados adicionais chamados de parâmetros para ser construído um comando com a forma correta.

Essa forma correta de construção de cada comando se chama sintaxe e é uma das bases de qualquer linguagem de programação, sendo a linguagem BASIC uma opção com sintaxe bastante simples e intuitiva, tendo sido utilizada como primeira linguagem de programação estudada por um tempo.

A **sintaxe** é a base da construção de instruções em todas as linguagens de programação e é um mecanismo essencial para que interpretadores e compiladores possam avaliar a se é possível executar ou gerar softwares executáveis a partir de códigos escritos.

Além da sintaxe, a chamada semântica complementa o conceito de sintaxe de forma a ter um olhar mais voltado à estrutura de códigos com base nos paradigmas e implementação das linguagens para verificar se códigos estão escritos de forma adequada em suas construções lógicas e correta colocação de instruções dentro de uma estrutura compreensível por compiladores e interpretadores.

Assim, pode-se assumir de forma simplificada que a semântica se refere à **construção do código em si**, observando a correta colocação das instruções em seus devidos lugares, estejam elas corretas ou não, pois essa outra preocupação é função da sintaxe.

Dentro dos conceitos fundamentais, ainda é importante deixar mais claro a diferença entre os processos realizados na análise de código para posterior execução, e entram então os conceitos de compilação e interpretação.

A **compilação** é um processo realizado em partes em que um código fonte escrito é analisado e primeiramente uma das principais etapas reúne os caracteres para a construção de lexemas, ou conjuntos de caracteres reconhecidos como palavras reservadas, identificadores (nomes em geral), valores numéricos, etc., a fim de facilitar etapas posteriores do processo, e ai se compreende a necessidade de uma correta digitação de códigos fonte.

Segundo Sebesta (2011), em outra etapa, outro componente chamado de **analisador sintático** utiliza as unidades léxicas criadas a partir do código fonte para estruturar as chamadas árvores léxicas que servem para que se possa analisar se comandos foram escritos de forma adequada com a correta colocação de palavras reservadas e parâmetros obrigatórios ou opcionais possíveis.

Outra etapa, um processo de análise semântica, preocupa-se com a estrutura toda do código avaliando instruções completas e sua correta colocação dentro da lógica de programação da linguagem utilizada.

Este conjunto de etapas e outros processos realizados são a base para a transformação de um código fonte escrito e legível para um novo conteúdo em arquivo chamado objeto, intermediário no processo todo que elimina elementos desnecessários em um código fonte como comentários, e já tendo sua estrutura avaliada e preparada para os próximos passos.

Numa etapa posterior, o chamado “linkador” tem a função de agregar ao arquivo objeto intermediário, bibliotecas e demais elementos externos chamados pelo código fonte para que juntos se tornem um único bloco que representa o software em si executável, já todo verificado e funcional.

Erros de codificação são verificados, mas nem todo tipo de erro é possível de ser validado pelo processo, pois há casos onde erros na lógica ou de uso incorreto de comandos para determinadas ações não pode ser identificado na etapa de compilação, mas somente na execução e teste, pois são mais perceptíveis nestes momentos.

Já no processo de **interpretação**, o código fonte é analisado, avaliado e executado diretamente por uma ferramenta como um navegador ou interpretador que não gera software executável, mas apenas executa o código fonte sequencialmente, e este processo pode agir de duas formas.

Erros contidos no código que possam ser ignorados são descartados durante a execução, como ocorre comumente em páginas web exibidas em navegadores, em que alguns comandos com erros deixam de ser executados e parte do conteúdo a ser exibido em uma página simplesmente não aparece, sem impedir a exibição dos demais elementos se possível.

Outro tipo de interpretação realiza todo o processo de análise prévia do código antes da interpretação de forma a executar o código apenas se este estiver estruturado de forma correta sintática e semanticamente, como no processo de compilação, tanto que arquivos temporários intermediários podem ser gerados para que a interpretação seja realizada a partir deles.

Num terceiro tipo de processo, a compilação ocorre, mas um software executável independente de outras ferramentas não é gerado, parando então a execução na geração do arquivo objeto intermediário, como ocorre na linguagem Java em que se cria um código fonte e este é então convertido em arquivo de um tipo chamado de Bytecode que é depois, interpretado por um software conhecido como máquina virtual.

Esse chamado arquivo **bytecode** (objeto) é portável e pode ser geralmente executado em qualquer plataforma, desde que ela possua um interpretador próprio para este tipo de bytecode instalado, permitindo então que um mesmo código possa ser executado nestas diferentes plataformas (sistemas operacionais e tipos de hardware) sem alteração se possível, permitindo alta portabilidade de código na linguagem Java.

Essa portabilidade é reduzida em linguagens compiladas, pois além das especificidades de cada plataforma, os compiladores geralmente não são os mesmos e um mesmo código fonte deve ser compilado em cada plataforma para a geração de softwares executáveis que não podem ser executados em outras plataformas sem o uso de emuladores ou simuladores de ambientes de plataformas.

Componentes de um Código

Na estruturação de código fonte em linguagens para programação estruturada, são utilizadas estruturas variadas para compor soluções computacionais para que o processamento de dados possa ser realizado e problemas de complexidades variadas possam ser resolvidos.

Geralmente, as linguagens de programação estruturadas permitem a construção de instruções contendo blocos de comandos, e estas instruções representam a base da codificação englobando estruturas para realização de desvios condicionais e laços de repetição que serão trabalhados com exemplos na aula 04.

Também serão demonstradas a declaração e o uso de estruturas de dados variadas como variáveis que representam unidades simples de armazenamento de dados, e estruturas mais complexas como matrizes de dados, por exemplo.

Outro conceito relevante é o de subdivisão de códigos em sub-rotinas que permitem maior possibilidade de reúso de código, independência entre partes de um código, e melhor legibilidade e facilidade de leitura e compreensão de código.

Geralmente, ainda existe a possibilidade de inclusão de funcionalidades prontas armazenadas em arquivos contendo código chamadas de bibliotecas de forma geral, que auxiliam no desenvolvimento de software em menor tempo utilizando funcionalidades já desenvolvidas e testadas como para ações de entrada e saída de dados padrão.

Assim, a ideia geral do estudo desta disciplina é que sejam apresentados conceitos ligados ao desenvolvimento de soluções computacionais para solução de problemas a partir da abstração destes problemas e avaliação de algoritmos a serem posteriormente implementados em linguagens que possam oferecer melhor custo e benefício a partir de vários aspectos analisáveis.

PARA GABARITAR

Veja exemplo de programa em linguagem BASIC na imagem a seguir, em que as linhas numeradas são fundamentais como referência para os desvios do fluxo de execução realizados pelo comando GOTO que se referência a elas para este processo.

```
10 REM EXEMPLO DE PROGRAMA EM BASIC - DIVISÃO

20 READ A

30 READ B

40 IF B=0 THEN GOTO 200

50 LET C=A/B

60 PRINT "RESULTADO DA DIVISÃO: ",C

70 PRINT “DESEJA REALIZAR OUTRO CÁLCULO (S/N)?”

80 LET D

90 IF D=”S” THEN GOTO 20

100 IF D=”N” THEN GOTO 300

110 GOTO 70

200 PRINT “O DIVISOR NÃO PODE SER ZERO!”

210 GOTO 30

300 END
```

Também é interessante analisar o fluxo de execução em si em que são solicitados valores para as variáveis A e B, e caso B seja zero, é desviado o fluxo para um tratamento desta exceção na linha 200 informando o usuário

e retornando a linha 30 para a inserção de novo valor.

Depois, é realizada a operação e exibido o resultado ao usuário que tem, em seguida, a opção por realizar ou não outra divisão, resultando nas possibilidades de desvio para a linha 20 para novo cálculo, desvio para a linha 300 para encerramento e desvio para a linha 70 caso seja digitada uma opção inválida.

Programação Estruturada – Linguagem C

Conhecer a base de uma linguagem de programação é essencial para que se possa implementar software para a solução de problemas diversos, mas apenas conhecer a linguagem não é suficiente para que se possa considerar o desenvolvedor um bom programador, pois saber avaliar problemas e sua complexidade, elaborar algoritmos eficientes, ser capaz de detectar falhas ou melhorias possíveis, testar a solução pensada de forma a garantir o máximo de funcionalidade possível a ser entregue e ser capaz de implementar a solução em uma linguagem conhecida, ou escolher a mais adequada dentre as conhecidas são atribuições de um profissional dito programador ou desenvolvedor.

O conhecimento dos diferentes paradigmas ajuda inclusive a escolher as linguagens mais interessantes para se conhecer e se necessário, aprender para que se tenha conhecimento do que se pode desenvolver com determinada linguagem, mas um ponto a ser levado em consideração é que mão de obra qualificada para apenas codificar é mais barata que a necessária para se avaliar problemas e propor algoritmos que resolvam estes problemas de forma eficiente e confiável.

A linguagem C talvez seja o exemplo mais apropriado de linguagem para estudo da programação estruturada por vários motivos, sendo um deles a sua extensa lista de funcionalidades implementadas, sendo capaz de produzir código voltado para soluções de baixo a alto níveis e aumentando o leque de desenvolvimento de software a partir das suas variações C++, C# e Objective-C, por exemplo que elevaram a linguagem para desenvolvimento de software para outras áreas como da computação gráfica e desenvolvimento web.

Linguagem C

A base da linguagem C são as sub-rotinas que representam blocos de códigos inseridos em estruturas chamadas de **funções,** que funcionam independentes entre si, necessitando que dados sejam passados entre si para que possam trabalhar os mesmos dados durante a execução do software gerado a partir delas.

Todo código em linguagem C não precisa ter várias funções, mas também não pode ser implementado sem o uso de nenhuma função, pois a execução de um software desenvolvido em linguagem C parte sempre de uma função chamada main() que inclusive não pode ter este nome utilizado em nenhuma outra função no mesmo software.

Outra característica da linguagem C é que ela é case sensitive, ou seja, trata **maiúsculas e minúsculas de formas distintas,** e como padrão, todas as palavras reservadas da linguagem devem ser digitadas em letras minúsculas.

Alguns elementos que não necessariamente estão ligados à lógica, mas são necessários à sintaxe na codificação, foram adotados por muitas linguagens de programação que se baseiam em C, como a linguagem Java, por exemplo.

Os caracteres ponto e vírgula, chaves, parênteses e colchetes são delimitadores de comandos ou outros elementos que serão citados logo na sequência e que são fundamentais para que cada instrução possa ser bem definida e completa.

As chaves são utilizadas para delimitar blocos de comandos dentro de instruções mais complexas como desvios condicionais ou laços de repetição que permitem que comandos diversos (inclusive outras estruturas semelhantes) sejam inseridos dentro de seus blocos a serem executados.

ABORDAGEM PRÁTICA

O aninhamento ou encadeamento de estruturas de controle de mesmo tipo é bastante utilizado na elaboração de algoritmos e permite que algoritmos com necessidades mais complexas possam ser elaborados. Estruturas de dados maiores que variáveis, por exemplo, necessitam de recursos mais complexos para sua manipulação, assim como condições que são avaliadas em sequência podem ser encadeadas de forma a se complementarem e, assim, permitires situações em que uma avaliação é realizada apenas dependendo do resultado obtido em uma condição anterior realizada necessariamente primeiro. Estruturas de repetição encadeadas permitem que dois contadores ou mais estejam trabalhando em conjunto como nos números de um hodômetro ou ponteiros de um relógio, e assim, cada vez mais funcionalidades são obtidas pela combinação de estruturas de controle.

Veja a imagem a seguir:

```
int avaliação(int idade, char experiencia) {
if (idade >= 18) {
            if (experiencia = “S”) {
                        printf (“Candidato aceito para
entrevista\n”);
                        return (1);
             } else {
                        printf (“Candidato não atendeu os
requisitos\n”);
                        return (0);
             }
         } else {
                         printf (“Candidato não atendeu os
requisitos\n”);
                         return (0);
   }
}
```

Fonte: autor.

Observando o exemplo acima, temos uma função chamada **avaliação** (apenas letras sem acento são permitidas em identificadores) contendo dois parâmetros que funcionam como requisitos para a funcionalidade da função.

Assim, temos que os **parênteses**, além de seu uso em expressões matemáticas, serve para **delimitar parâmetros** que representam dados passados de uma função a outra ou para atender à sintaxe de comandos – como **if** – utilizados no exemplo.

Os comandos if contêm expressões lógicas baseadas em operadores lógicos, e realizam desvios da execução do software de acordo com os valores resultantes da condição definida, assim como outra palavra reservada importante no exemplo é else que representa a contrapartida de cada comando if de desvio condicional utilizado.

O espaçamento em níveis chamado de **endentação** (ou indentação também), que não é obrigatório nesta linguagem, auxilia na legibilidade do código e para compreender as estruturas de decisão if e else, e como estão estruturados dentro da função avaliacao().

Para se conhecer uma linguagem de programação como C, é preciso se familiarizar com sua semântica, que representa a forma como um código deve ser estruturado e a variedade de diferentes estruturas sintáticas que são as regras de construção de instruções a partir de comandos e funções da linguagem. Também é muito importante ler códigos prontos e compreendê-los, mas é preciso também executá-los, modifica-los e depois tentar criar suas próprias versões de códigos e poder sentir segurança para atuar profissionalmente com uma linguagem qualquer.

O trecho de código da imagem anterior não pode ser compilado da forma como está, pois existe um complemento necessário semântico composto de outros comandos e elementos para que o compilador possa gerar um software executável, e o exemplo de estrutura básica em linguagem C da imagem a seguir pode ser observado.

```
#include <stdio.h>
int main () {
        int idade;
        char experiencia;
int vaga = 0;
        while (vaga <= 10) {
                printf (“Digite sua idade: );
                scanf (“%d”, &idade);
```

```
                    printf (“Digite sua experiência: );
                    scanf (“%d”, &experiencia);
                    vaga = vaga + avaliacao (idade, experiencia);
}
return (0);
}
```

Fonte: autor.

Neste exemplo, o restante do código necessário para que se possa gerar um software é exposto, sendo que ambos os trechos das imagens anteriores devem estar posicionados em um mesmo arquivo de texto salvo com o nome desejado, seguido da extensão **.C** indicativa de códigos desta linguagem.

Uma estrutura de repetição representada pelo comando **while** que se baseia em uma condição indicada dentro dos parênteses como parâmetro para controlar a quantidade de repetições a serem realizadas tem como base uma suposta ideia de que existe um processo de chamada de até dez pessoas para a realização de entrevista para possível contratação, por exemplo.

Dentro da função **main()** são obtidos os dados necessários para processamento na função avaliacao() através das funções printf() e scanf() que estão disponíveis numa biblioteca adicionada pela primeira instrução do código que inclui a stdio.h, e assim disponibiliza estas duas funções e muitas outras criadas para esta finalidade.

A codificação em linguagem C pode ser bastante complexa devido ao grande detalhamento dos processos que podem ser implementados, e linguagens mais modernas simplificam esse desenvolvimento, que reduz o trabalho de codificação, mas reduz também a possibilidade de manipulação de como deve ser realizado o processo.

Existem estruturas geradas em linguagem C que não existem em várias outras linguagens tais como Java, como os chamados ponteiros que servem como indicativos de outra estrutura de dados em memória e não armazenam dados em si, sendo útil em estruturas de dados mais complexas que variáveis principalmente.

```
void trocaValores (int a, int b) {
                int aux;
                aux = a;
                a = b;
                b = aux;
}

void trocaValores (int *a, int *b) {
                int aux;
                aux = *a;
                *a = *b;
                *b = aux;
}
```

Fonte: autor.

Nestes exemplos, temos duas variações de uma mesma função **trocaValores(),** que recebe dois parâmetros de formas distintas em cada exemplo. No primeiro, recebe uma cópia dos valores originais recebidos e alterações sofridas pelos valores contidos nas variáveis a e b não afetam as variáveis de origem dos valores, pois dentro de uma função, o chamado **escopo** de uma variável faz com que variáveis declaradas dentro de uma função, sejam ativas apenas durante a execução da função, tendo sido criada no momento da execução da função, e tendo o espaço ocupado pela mesma na memória liberada assim que a função é encerrada.

No segundo exemplo da imagem acima, os parâmetros são, na verdade, ponteiros que são direcionados nas posições de memória ocupadas pelas variáveis de origem indicadas como parâmetros, e quaisquer alterações sofridas pelos ponteiros da função afetam e alteram diretamente os valores das variáveis fora da função ligadas aos parâmetros da função, sendo então a maneira correta de conseguir alterar mais de um valor de uma variável em uma função, pois usando apenas parâmetros e retornos de função, a linguagem C, assim como várias outras, tem por padrão o retorno de apenas um valor simples.

Outro tipo importante de estrutura de dados da linguagem C são os **vetores e matrizes**, estruturas de dados homogêneas para conjuntos de dados, sendo vetor para listas com uma dimensão apenas, e matrizes para duas ou mais dimensões, lembrando que existem autores que utilizam apenas o temo matriz para qualquer quantidade de dimensões.

A imagem a seguir mostra exemplos de vetores e matrizes com as indicações de índices para identificação de cada célula das estruturas, observando que a numeração inicia em zero por convenção da linguagem e as referências são indicadas entre colchetes.

0	1	2	3	4	5	6	7	8	9

int lista[10];

0	1	2	3	4	5	6	7	8	9
1									
2									
3									

char dados[3][9];

Fonte: autor.

Na imagem, é apresentada um ilustração de um vetor declarado com o nome lista para 10 elementos do tipo inteiro, e depois, é ilustrada uma matriz declarada com o nome dados com 3 linhas e 9 colunas, totalizando uma possibilidade de armazenamento de 27 caracteres pelo tipo de dado char utilizado na declaração.

A programação estruturada trabalha com o desenvolvimento de softwares utilizando linguagens de propósito geral que normalmente e facilmente conseguem combinar estruturas de dados variadas com os demais componentes citados, sendo que as estruturas de repetição são essenciais para se trabalhar com vetores e matrizes devido a forma como funcionam ser adequada para se referenciar posições em vetores ou coordenadas de linhas e colunas em matrizes.

É comum ao observar códigos, encontrar estruturas de dados sendo referenciadas por seus nomes com símbolos como * ou **&** juntos, e caso não conheça a capacidade de gerenciamento de memória da linguagem C, é interessante iniciar pelo conceito de ponteiros que são apontadores de endereços de memória para variáveis e outras estruturas de dados.

Fonte: **Disponível aqui**

Programação Estruturada – Outras Linguagens

Além da linguagem C, tema da aula 04, outras linguagens também são implementadas com base na programação estruturada e no paradigma imperativo como o caso da linguagem BASIC e da linguagem Pascal, além de linguagens mais populares no mercado como PHP e GO.

Linguagem Pascal

O problema desta linguagem foi que suas evoluções acabaram sendo comercialmente fortes, como Delphi, que possui muito sistemas ativos até hoje e ainda muito confiáveis, mas mesmo Delphi não acompanhou a evolução de outras linguagens e acabou perdendo muito espaço no mercado, sendo amplamente dominado por Java, C e Python principalmente.

```
program Teste;
var
            a, b, potencia, i : integer;
begin
            writeln ('Digite um valor inteiro para a base: );
           readln (a);
           potencia := a;
writeln ('Digite um valor inteiro para o expoente:);
           readln (b);
            for i := 1 to b-1 do
                        potencia := potencia * a;
            writeln (‘Resultado da exponenciação: ‘,
potencia);
end.
```

Fonte: autor.

No exemplo da imagem acima, um código em linguagem Pascal é apresentado e sua leitura não difere muito da sintaxe da linguagem C, tendo alguns pontos de atenção a serem considerados.

Um ponto perceptível logo no início do código é que em Pascal, utiliza-se a primeira linha para nomear o código, ao invés de se incluir uma biblioteca como em C, por exemplo. Outro ponto relevante associado ao que foi citado é que os comandos de entrada e saída de dados *readln()* e *writeln()* são considerados palavras reservadas da linguagem, sendo comandos da própria implementação da linguagem.

O símbolo utilizado para atribuição de valores é := ao invés do = utilizado em C, assim como os apóstrofos utilizados como delimitadores de texto ao invés das aspas em C, sendo igualmente importantes para a correta construção de árvores sintáticas para análise sintática de instruções.

Outro ponto importante na codificação nesta linguagem é o uso das palavras reservadas *begin* e *end* como delimitadores de blocos de comandos, tornando a legibilidade mais fácil e intuitiva.

Por fim, vale citar o uso de uma estrutura de repetição baseada na palavra reservada *for* para a criação de um laço de repetição contado onde são definidas *b-1* iterações, pois como a variável *potencia* que conterá o resultado da exponenciação é inicializada com o valor da base, uma multiplicação a menos deve ser realizada.

PARA GABARITAR

As variáveis funcionam como depósitos de dados que podem ser manipulados durante a execução e algoritmos e devem ser bem compreendidas para que seu uso constante não represente um problema. As constantes complementam as variáveis como recurso para armazenamento de dados não variáveis durante a execução de algoritmos. O uso de estruturas de dados como variáveis é fundamental para o desenvolvimento de algoritmos e saber manipular seus dados de forma adequada é importante. Uma prática que auxilia de se desenvolver algoritmos seguros é a do uso da atribuição para realizar inicializações de variáveis antes de seu primeiro uso, pois assim, valores indesejados contidos na área de memória alocada para variáveis não correm o risco de serem utilizados.

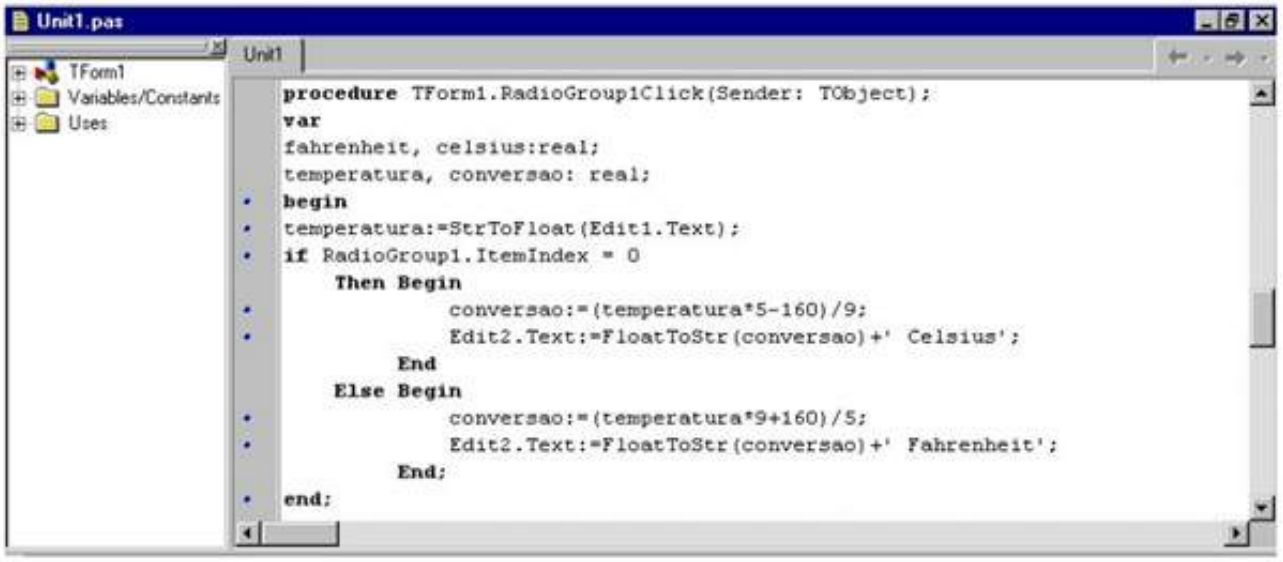
Observa-se que a linguagem é aparentemente mais simples em sua leitura de código, mas em relação à sintaxe e semântica, se assemelha a C, pois a construção das instruções é semelhante sofrendo poucas alterações perceptíveis, além de palavras reservadas e na simbologia utilizada.

Em Pascal também é possível incluir bibliotecas utilizando a palavra reservada *uses* que, como em outras linguagens, é utilizada semanticamente sempre no início do código, mas no caso de Pascal, logo depois da instrução que nomeia o código.

Delphi

Em Delphi, os softwares criados se baseiam em interfaces gráficas, e a base da linguagem Pascal se mantém, mas são adicionados conceitos responsáveis por permitir que estas interfaces sejam construídas levando em consideração que o desenvolvimento de software que deva ser executado em janelas depende da mecânica de composição destas e de seus componentes disponíveis a partir das definições do sistema operacional.

Para exemplificar este tipo de aplicação implementada sobre o mesmo paradigma, mas que pode ter elementos um tanto distintos do que foi visto até o momento, observe o exemplo ilustrativo da imagem a seguir.



Fonte: Bezerra (2018).

Neste exemplo, é perceptível a estrutura de uma janela típica de um sistema operacional com interface gráfica com botões, campos para entrada de dados e campos seletores de opções, e estes componentes precisam ser nomeados e funções relacionadas a eles podem ser aplicadas de forma a realizar processamento necessário.

A palavra reservada *procedure* neste exemplo age como gerenciador da janela em que os componentes necessários serão implementados e ações associadas a eles, mesmo que no processo de criação, inicialmente, os componentes sejam criados como se estivessem sendo desenhados em um software gráfico.

Cada componente criado se torna um objeto que possui propriedades associadas a ele como tamanho, core, nome, tipo, dado armazenado, etc., mas que podem ter processos codificados como se fosse em modo texto como atribuição ou modificação de valores ou cálculos matemáticos, por exemplo.

RadioGroup1, por exemplo, se refere ao componente que permite a seleção entre Fahrenheit ou Celcius na janela que funciona como um formulário de dados, e atributos como *ItemIndex* podem ser utilizados como for adequado ao programador, que neste caso está servindo para verificação de qual das duas opções está selecionada para cálculo da conversão de temperatura.

Dentro do bloco de comandos do primeiro comando *if* existem duas atribuições, sendo a primeira de uma expressão de cálculo da temperatura a ser atribuída à variável *conversao*. Na segunda atribuição, o valor de *conversao* é convertido em texto

e adicionado como dado a ser exibido no componente *Edit2* pelo atributo *Text*.

Detalhes como a identificação de componentes e seus atributos e o uso das sub-rotinas associadas a estes são pontos divergentes em relação à linguagem Pascal original, mas a forma como se implementa o código da estrutura lógica dos processos praticamente se manteve e assim, aqueles que aprendem a linguagem pura, possuem maior facilidade de adaptação para novas versões implementadas de uma linguagem conhecida.

Linguagem PHP

Outra opção de linguagem originalmente pensada como estruturada foi a linguagem PHP que possui um mercado muito grande para o desenvolvimento de software para sistemas web utilizando o navegador como software de interpretação de códigos.

Um ponto importante é que a linguagem PHP é associada à linguagem declarativa HTML, e juntas, realizam o processo de construção de páginas web interativas e capazes de funcionarem como sistemas criados em Delphi, por exemplo.

O HTML realiza o processo de estruturação da página em si com a inserção de elementos na página como texto, imagens, efeitos e formulários semelhantes aos criados em Delphi, sendo esta parte conhecida como Front End, ou a parte mais de interação com usuários em sistemas web.

O PHP complementa o Front End com funcionalidades como acesso a bancos de dados, processamento de dados, manipulação do que ocorre com os elementos criados em HTML e usuários, gerando funcionalidades como em sistemas desktop, processo esse associado ao termo Back End, que trabalha nos bastidores de forma mais transparente aos usuários.

```
<!DOCTYPE html>
<html>
<body>
            <h1>Exemplo HTML + PHP</h1>
<?php
```

```
                        $item = "Produto A";
                        $estoque = 100;
                        echo $item;
                         echo "</br>";
                        echo $estoque;
    ?>
    </body>
    </html>
```

Fonte: autor.

Neste exemplo de código, temos as chamadas *tags* HTML responsáveis pela construção de uma página diretamente no navegador. Elas são delimitadas por sinais de < e > para caracterização das palavras reservadas e instruções completas, sendo que as instruções são estruturadas utilizando-se um comando delimitado por < e >, e depois, ao final da instrução, delimitada por < e />, indicativos de início e fim da instrução.

Internamente ao script HTML, são inseridos scripts em PHP delimitados pelos marcadores <?php e ?> que separam o conteúdo PHP do restante do código HTML para impedir que o navegador interprete incorretamente o script, misturando as duas linguagens.

Detalhes sobre a linguagem PHP são que ela também toma por base a linguagem C, sendo visíveis algumas construções de comandos exatamente como em C, mas algumas regras foram modificadas em virtude das características de uso da linguagem como a inclusão do símbolo $ antes dos nomes de todas as variáveis, facilitando sua identificação, e agregação de *tags* HTML em meio ao código PHP como no comando *echo "</br>";* que é responsável por exibir na tela o conteúdo entre aspas, assim como a função printf*()* em C, mas o conteúdo representa quebra de linha, tendo então a simples função de pular linha e não exibir texto.

CONECTE-SE

Compreender o processo de elaboração de conteúdo web envolve o aprendizado de mais de uma linguagem e de como integrar os códigos e aplicações geradas por estas linguagens. É preciso estudar diferentes conceitos e ferramentas para que se possa ter sucesso no desenvolvimento de páginas, sites institucionais ou para comércio eletrônico, aplicativos para plataformas diversas e sistemas baseados na web.

Fonte: **Disponível aqui**

Com isto, é possível observar que várias linguagens de programação compartilham aspectos semelhantes e facilitam seu aprendizado, mas todas elas necessitam de uma correta compreensão dos problemas a serem solucionados e uma precisa elaboração dos algoritmos capazes de proporcionar funcionalidades que atendam ao que se espera como solução.

Linguagem GO

Mais recentemente, a Google, que pode ser considerada ainda uma empresa mais recente no mercado, desenvolveu a linguagem GO, de propósito geral e também baseada em C, para que pudesse servir como ferramenta de desenvolvimento de software e aplicativos otimizada para seu sistema operacional Android.

Essa linguagem rapidamente adquiriu vários adeptos em função de sua simplicidade e características próprias, fazendo com que tivesse uma participação relevante no mercado e tornando-se uma opção boa de aprendizado e busca de vagas de trabalho no mercado.

Possui como principal característica a possibilidade de desenvolvimento de software para a execução paralela de tarefas que podem ser definidas como outro paradigma chamado de programação concorrente.

Nessa forma de desenvolvimento de software, programas separados ou tarefas (*threads*) que sejam executadas paralelamente criadas por um programa, podendo ser executadas "ao mesmo tempo" pela divisão de tempo de processamento de um único processador ou pela execução das *threads* realmente em paralelo por vários processadores em um mesmo hardware, sendo que a pouco tempo foi criada a arquitetura de núcleos em um único processador que também podem ser programados para trabalharem paralelamente atendendo diferentes *threads* simultaneamente.

Mas esse recurso é opcional, e a linguagem pode agir tranquilamente como uma típica linguagem estruturada com sintaxe semelhante à linguagem C de uma forma otimizada e muito eficientes, tendo como uma importante característica, a possibilidade de uma função retornar mais de um valor ao final de sua execução, recurso não disponível normalmente nas demais linguagens populares do mercado. Veja:

```
package main
import "fmt"
func main() {
var lista = [5]int{10,20,30,40,50}
    fmt.Println ("Lista inicial:", lista)
```

```
        fmt.Scan (&lista [2])
        fmt.Println ("Lista ajustada:", lista)
        for i := 0; i < 5; i++ {
              lista [i] = lista [i] / 2
            }
      fmt.Println("Lista final: ", lista)
    }
```

Fonte: autor.

Neste exemplo de código em linguagem GO, temos primeiramente a indicação de um pacote chamado **main**, conceito que permite o agrupamento de mais de um arquivo de código fonte em linguagem GO para compor um software mais complexo e modular.

É utilizada a palavra reservada **import** para incluir bibliotecas como a chamada **fmt** para funções de entrada e saída de dados como em C, onde funções como Println() e Scan() ou alternativas Print e Scanln() tendo como diferencial, que funções Println() e Scanln() mudam de linha na tela ao final de sua execução.

A declaração de um tipo de dado do tipo vetor para 5 números inteiros é declarada de uma forma diferente de C, e utilizando uma palavra reservada *var* antes da indicação do nome de uma variável para receber o vetor, seguindo da atribuição do vetor com valores de inicialização inseridos entre chaves.

Uma estrutura de repetição com base na palavra reservada **for** é gerada já com a declaração embutida de uma variável **i** de controle, inicializada com zero e sendo incrementada em uma unidade a cada iteração, podendo ser repetido enquanto o valor de **i** seja menor que 5, para ajustar uma a um, os elementos do vetor, dividindo os valores armazenados por 2.

A precedência de operadores é fundamental para que a lógica de algoritmos seja correta, mas para isso é importante conhecer a finalidade de cada tipo e operador específico e a partir da construção de expressões diversas mudando operadores e adicionando ou removendo parênteses, testar resultados a partir do ajuste da precedência.

Com a análise dos exemplos de diferentes linguagens de programação, temos diferentes ferramentas para codificação que podem ser utilizadas geralmente para o desenvolvimento de softwares para diferentes finalidades, mas que podem em alguns casos servirem de opções para o desenvolvimento de uma mesma solução, principalmente com linguagens de propósito geral.

Pode utilizar um software desenvolvido em Delphi para armazenar dados localmente para uma finalidade comercial, por exemplo, mas pode-se utilizar PHP com HTML para que essa mesma solução funcione como página web, e os dados podem ser armazenados na nuvem ou usar a linguagem GO para criar uma versão para dispositivos Android, todas utilizando como base os fundamentos da linguagem C.

Programação Orientada a Objetos - Conceitos de Objetos, Classes, Métodos e Atributos

Mudando de paradigma de programação, iniciam-se os estudos do paradigma favorito do mercado atualmente, em que as soluções tendem a ser mais reutilizáveis, as abstrações de problemas a serem resolvidos organizadas em estruturas chamadas de **classes**, contendo características similares às da programação estruturada, mas trazendo novos conceitos que permitem algumas vantagens em relação ao paradigma estruturado.

Abstração

A base para o paradigma orientado a objetos é a abstração de problemas em modelos instanciáveis chamados de **classes,** que agrupam propriedades e ações necessárias ao desenvolvimento de uma solução computacional para um problema.

Para entender um pouco da ideia da abstração, é interessante o uso de um exemplo prático: imagine consultório veterinário em que animais representam o foco do negócio, e ao redor dos animais orbitam produtos e serviços.

A abstração pode ser utilizada para iniciar uma proposta de solução para um software que resolva algumas ou várias demandas desse negócio, e é o ponto de partida para o desenvolvimento de software com base no paradigma orientado a objetos.

Em um negócio do tipo citado, é importante avaliar o que é necessário ser trazido de um problema controlado manualmente para um problema que seja controlado via software, e então é preciso compreender o chamado escopo do problema.

Digamos que nesse caso é solicitado que um software seja capaz de cadastrar animais com dados de identificação dele e de seus donos, além de serviços realizáveis e histórico de realização mais o controle de produtos para uso ou comercialização, por exemplo.

Assim, num primeiro momento, são identificadas as principais partes do problema para que delas sejam abstraídas estruturas de dados representativas de propriedades relacionadas ao problema, e depois sejam pensados os meios para se processar os dados.

Ao abstrair as propriedades necessárias para se implementar uma solução computacional, muitos aspectos devem ser levados em consideração, e a base da manipulação de dados é muito parecida com as utilizadas na programação estruturada.

Essas propriedades, chamadas de **atributos** na programação orientada a objetos, são equivalentes a campos em estruturas do tipo registro na programação estruturada, tanto é que este conceito não é utilizado na programação orientada a objetos, pois a classe é uma evolução dos registros.

Os registros da programação estruturada são estruturas de dados mais evoluídas que vetores, ou matrizes, porque permitirem dados de tipos diferentes agrupados, funcionando como tipos especiais vinculáveis a variáveis.

No caso do veterinário, na programação estruturada, seria criado um tipo registro (*struct* em linguagem C, por exemplo) que poderia ser nomeado “animais”, contendo definições de campos para armazenar as características dos animais que sejam relevantes para o negócio como nome, raça, data de nascimento, cor, peso, etc., além dos dados do dono, mas que poderiam estar armazenados em outra estrutura de registro para dados pessoais.

Um detalhe dos registros é que como sua declaração funciona como a criação de um tipo definido pelo usuário, esses devem servir como tipos para a declaração de estruturas de dados como variáveis, por exemplo.

Mas um registro armazena dados referentes a uma ocorrência apenas (animal, cliente) quando usado para declaração de uma variável, mas se usado como tipo para declaração de vetores, pode armazenar um registro de dados por unidade do vetor, funcionando sob o mesmo princípio dos bancos de dados.

Assim, existe a possibilidade de se criar estruturas como vetores definidas a partir de estruturas de registros para que se possa trabalhar com maiores volumes de dados em memória, mas ainda é possível utilizar as estruturas de registros para organizar dados que possam ser armazenados em arquivos do tipo texto, onde cada linha do arquivo pode armazenar um registro completo, por exemplo.

Conceitos de Classe, Atributo, Método e Objeto

A organização da abstração do real para uma solução computacional é organizada em quatro elementos principais no paradigma orientado a objetos: classes, atributos, métodos e objetos.

No caso do paradigma orientado a objetos, os registros são substituídos por classes que, assim como os registros podem estar associados a dados heterogêneos armazenáveis em atributos ao invés de campos.

As classes são responsáveis por estruturar os softwares no paradigma orientado a objetos e dentro de uma classe, pode ser estruturada toda uma solução computacional, ou esta solução ser dividida em mais de uma classe dependendo de como é elaborada a abstração do problema real e sua complexidade.

Os chamados tipos de dados são fundamentais na maioria das linguagens de programação. Geralmente, uma linguagem de programação possui ao menos um tipo numérico de ponto flutuante para trabalhar com valores com a possibilidade de

terem dígitos decimais, assim como podem ter tipos destinados apenas a valores inteiros.

Tipos de dados para caracteres e texto também são importantes, e muitas linguagens possuem um tipo próprio para valores lógicos ou booleanos verdadeiro e falso, mas também é possível utilizar números inteiros 1 e 0 para representar estes valores booleanos.

Esses tipos de dados são utilizados para a criação de variáveis em praticamente todas as linguagens de programação, e no paradigma orientado a objetos, geralmente são também utilizados para definir tipos para atributos de classes, indicando os tipos de dados utilizados para representar as propriedades destas classes.

Os atributos de uma classe se parecem em alguns aspectos como variáveis, mas sua finalidade e formas de uso são distintas, mesmo podendo geralmente representar unidades de dados com um tipo definido em sua implementação, pois variáveis são de propósito geral em termos de armazenamento de dados como valores recebidos pelo software, variáveis contadoras para laços de repetição, etc.

Os atributos se referem a propriedades em classes escolhidas para representar as características de elementos reais que foram abstraídas para a modelagem de classes que simulam estes objetos reais.

Os atributos são utilizados para armazenar dados referentes a um elemento específico obtido a partir de uma instanciação de classe para que os dados de atributos que caracterizam esta instância possam ser inseridos e processados como devido pelo software.

Essa instanciação de uma classe para que um elemento possa ser devidamente criado e utilizado como estrutura de dados e comportamentos relacionados a estes dados, recebendo o nome de objeto.

Objetos são então instâncias particulares de classes que representam elementos únicos com suas características próprias, sendo que uma classe pode servir como base para a geração de inúmeros objetos.

As classes não armazenam dados em si, servindo apenas como modelo para objetos que armazenam em cada instância, um conjunto particular de dados em atributos que podem ser então manipulados pelo software.

Esta manipulação é realizada por estruturas semelhantes às sub-rotinas chamadas de métodos que são responsáveis por realizar todo o processamento de dados de objetos de forma a serem essenciais para a implementação de software.

Os métodos representam ações que podem ser realizadas com atributos de forma a manipular seus dados e atender às funcionalidades definidas em requisitos do software e uma classe pode conter de zero a muitos métodos, sendo que estes métodos devem sempre ter relação direta com as classes e seus atributos de forma a afetar diretamente métodos instanciados.

Existe um recurso importante do paradigma orientado a objetos que está ligado à instância de objetos, tanto no momento de sua instanciação quanto de sua exclusão. O chamado **método construtor** possui o mesmo nome da classe à qual pertence, e é responsável por inicializar atributos com valores para o novo objeto, de forma a garantir que sejam criados com os dados adequados, de acordo com as necessidades de cada software. No momento em que é finalizado o uso de um objeto, um método destruidor (destrutor) com o nome da classe tendo um caractere diferenciador no nome como ~ antes do nome e servindo para a definição de elementos a serem descartados na memória, por exemplo.

Assim, uma classe Animais teria um método construtor Animais() para inicialização de dados e um destrutor com nome ~Animais() para liberar recursos de hardware.

Para ilustrar esses conceitos e sua estruturação dentro do paradigma orientado a objetos, observe o exemplo da imagem **AA**, que traz a abstração do problema base de um veterinário necessitando organizar seu sistema de controle de animais, produtos e serviços.

Como os objetos representam instâncias de classes, uma forma de se definir uma classe é observando o que a abstração pode fornecer de informações sobre o problema real, de forma que se possa pensar em como seria a estrutura de um objeto instanciado com atributos relevantes e como seriam os dados contidos nestes atributos.

Como cada instância de classe representa um conjunto particular de características agrupadas em cada objeto gerado a partir da classe, imaginar o que poderia ser estruturado em forma de objeto, pode ser uma boa opção para se iniciar a definição de classes.

Os animais, por exemplo, podem ser imaginados como classes, pois objetos instanciados a partir desta classe poderiam estar associados a cada animal atendido pelo veterinário com seus atributos específicos como raça, cor, nome, idade, peso, e identificação do dono.

Já os atributos relativos aos donos não precisam ser incluídos diretamente na mesma classe dos animais, pois os atributos dos animais e seus donos não são os mesmos e nem possuem muita relação, e assim, convém definir uma classe para instanciar animais e outra classe para instanciar pessoas.

Assim, já seria possível a obtenção de duas classes com atributos característicos, e também poderiam ser definidos métodos para que os possíveis dados para objetos instanciados pudessem ser trabalhados de forma adequada.

Alguns métodos são essenciais para a entrada de dados e a obtenção de dados chamados de **set()** e **get()**, nomenclatura padrão para essas funcionalidades, mas podem ser definidos métodos menos convencionais para especificidades do problema a ser resolvido.

Os métodos necessitam de comunicação entre si muitas vezes, pois um método pode acionar outro, e a mecânica de execução de métodos é baseada em uso da memória de forma dinâmica de forma que um método ocupa a memória ao ser acionada como todas as estruturas de dados necessárias para sua execução.

Depois, caso outro método seja executado a partir de uma chamada do método em execução, o status do método chamador é pausado e uma nova alocação de memória é feita para os recursos necessários para o novo método.

Ao final da execução do método chamado, dados podem ser passados como retorno para o método chamador e a memória utilizada pelo segundo método é liberada para uso pelo sistema operacional e os dados ali contidos são considerados descartados.

O método chamador volta a atividade, e continua seu processamento, sendo que ao fim de sua execução, seus recursos de memória também são liberados ao sistema operacional, e assim se repetirá o processo para outros métodos executados posteriormente durante a execução do software em si.

Os conceitos ligados à orientação a objetos são variados, e cada linguagem de programação implementa um determinado conjunto de conceitos e os adapta às suas necessidades, supondo o uso de cada mecanismo aplicado a determinados problemas que poderão ser resolvidos com estes recursos. Assim, linguagens que se propõem a resolver problemas mais variados podem necessitar de mais conceitos, e estes implementados com maior possibilidade de variação para uma maior gama de problemas.

Acesse o link: **Disponível aqui**

Programação Orientada a Objetos – Encapsulamento, Herança e Polimorfismo

A orientação a objetos é um paradigma mais moderno que a programação imperativa ou estruturada, e por isto pode ser implementada de forma a conter conceitos mais evoluídos computacionalmente, sendo próxima das necessidades de desenvolvimento de software dos dias atuais de certa forma.

Isso em função também da forma como se desenvolve softwares atualmente, permitindo uma maior independência entre os componentes de um software ou sistema computacional composto de módulos e que funcione na web, por exemplo.

Algumas características são essenciais no paradigma orientado a objetos como a definição da estrutura dos códigos em função de classes com seus atributos e métodos para que objetos sejam instanciados para o armazenamento e processamento de dados.

Outros aspectos não são obrigatórios, mas são o diferencial deste paradigma em relação aos outros e devem ser conhecidos e explorados de forma que possam ser aplicados na implementação de códigos e o real paradigma orientado a objetos possa ser aplicado no desenvolvimento de software.

Linguagens como C e Pascal não suportam a orientação a objetos, mas variações da linguagem C como C++, C# e Objective-C suportam o desenvolvimento orientado a objetos sendo opcional seu uso ou obrigatório dependendo da linguagem derivada de C.

Outras linguagens como Python, PHP e JavaScript permitem que seus códigos possam conter aspectos do paradigma orientado a objetos ou tenham seus códigos escritos de forma estruturada, por exemplo, ficando a escolha por conta de especificidades da solução computacional ou conhecimentos do codificador.

Linguagens como Java e Objective-C trabalham com conceitos de orientação a objetos nativamente e necessitam que seus códigos sejam implementados baseados em classes, sem a possibilidade de programação puramente estruturada, por exemplo.

Encapsulamento

Um conceito muito importante da orientação a objetos é o chamado **encapsulamento,** que contribui para que as estruturas de classes sejam transparentes para outras classes e todo o conteúdo de uma classe seja empacotado e independente de meios externos à classe.

A ideia de encapsular componentes de uma classe permite maior segurança aos dados que são mantidos pela classe e os processos aos quais são submetidos, tornando também o software mais confiável, pois os dados são processados de forma mais adequada e tendo assim comportamentos menos inesperados.

O que define a base do encapsulamento são palavras reservadas contidas nas linguagens de programação que aceitam este paradigma, servindo para definir o nível de visibilidade de classes, atributos e métodos dentro de um software.

Em geral, as linguagens de programação utilizam três níveis de encapsulamento utilizando as mesmas palavras reservadas *public, private e protected* para diferenciar estes níveis e definir o grau de transparência do código.

Geralmente o chamado modificador de acesso *private* é utilizado em elementos que não devem ser exibidos fora da estrutura na qual foi declarado, pois a ideia deste modificador é deixar qualquer atributo ou método de uma classe totalmente inacessível por outras classes e assim, controlar o que ocorre com estes componentes de forma mais segura.

Na declaração de um atributo ou método é comum que a sintaxe exija que um tipo de dado seja indicado, depois o nome do atributo ou método, e por fim, parâmetros que possam ser associados aos métodos são indicados entre parênteses.

O modificador de acesso *private* é posto logo no início desta sintaxe, antes do tipo de dado associado, aumentando e complementando a sintaxe de declaração de atributos e métodos.

Classes não são normalmente definidas como privadas, pois elas são a única forma de acesso a toda a estrutura de atributos e métodos internos a ela, e se não for permitido acesso a ela, a classe toda ficaria em um nível de encapsulamento inacessível e inutilizável.

Outro tipo de modificador de acesso é público, utilizando a palavra reservada *public* que atua de forma inversa a *private*, permitindo a visibilidade de classes, atributos ou métodos dentro das limitações definidas conceitualmente por ada linguagem.

As classes são sempre do tipo *public*, pois como dito anteriormente, são o meio de acesso a todos os atributos e métodos contidos nela, sendo então necessário que esta seja visível e possa ser chamada por quaisquer outras classes pertencentes ao mesmo software que também pode ser definido por pacote, por exemplo, dependendo da linguagem de programação.

Para que se possa ter um nível adequado de encapsulamento, o uso do modificador *public* em atributos e métodos não é ideal, pois dá certo nível de liberdade de acesso que pode ser indesejável do ponto de vista de segurança e transparência no código, sendo mais comum o uso deste nível de acesso mais em classes.

Outro modificador de acesso comum em linguagens de programação é dito protegido, utilizando a palavra reservada *protected* para que atributos e métodos de uma classe possam ser acessados por classes ligadas à classe ao qual pertencem por um mecanismo da orientação a objetos chamado de **herança**, que será discutido logo adiante.

Neste nível de visibilidade, classes que sejam ligadas entre si podem compartilhar atributos ou métodos protegidos de forma que esta permissão ocorra apenas do sentido de uma das classes para a outra, sendo a classe que contém os atributos ou métodos protegidos é chamada de classe pai (mãe) ou superclasse dependendo dos autores, e a classe ou as classes que estejam interligadas à superclasse e possam ter acesso aos elementos protegidos são ditas **classes filhas ou subclasses**.

Herança

Partindo da ideia do encapsulamento, foi visto que existem diferentes níveis de visibilidade associados aos termos *public*, *private* e *protected*, sendo que este último depende do conceito a ser tratado a partir deste ponto.

A herança é outra característica muito importante do paradigma orientado a objetos, pois permite que classes ligadas possam possuir um nível de acesso diferenciado em atributos e métodos contidos nestas.

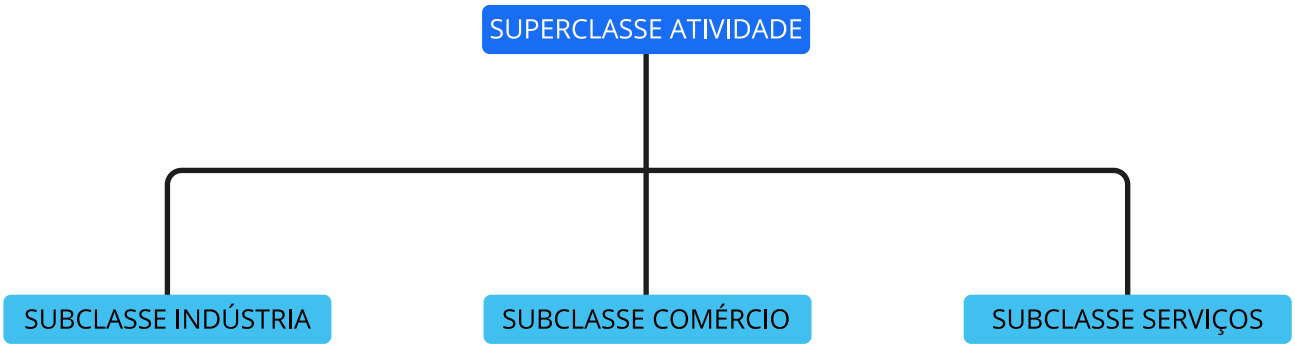
Algumas características são importantes, pois afetam diretamente a possibilidade de uso deste recurso, sendo que existem limitações que devem ser observadas no projeto de software que se baseie neste paradigma e necessite então de uma estruturação de classes de forma que não sejam confundidos os modificadores de acesso utilizados e as regras de herança entre classes respeitadas.

Uma classe pode herdar atributos e métodos de outra classe, sendo a classe que herda chamada de subclasse, podendo pela herança ter acesso a dados de atributos da superclasse, podendo modifica-los, ou usar métodos da superclasse para reduzir redundância de código.

Para ilustrar a herança, podemos imaginar classes genéricas representando veículos, animais ou produtos, sendo estas consideradas superclasses com atributos gerais como fabricante, raça ou data de fabricação respectivamente, e métodos para manipulação destes atributos como os métodos get() e set() que são os mais comuns em classes.

Classes como utilitários ou passeio podem representar outras classes mais específicas de veículos, assim como peixes e mamíferos podem representar classes mais específicas de animais, e possuem os mesmos atributos das classes veículos e animais, mas necessitam de mais atributos específicos e métodos get() e set() diferenciados.

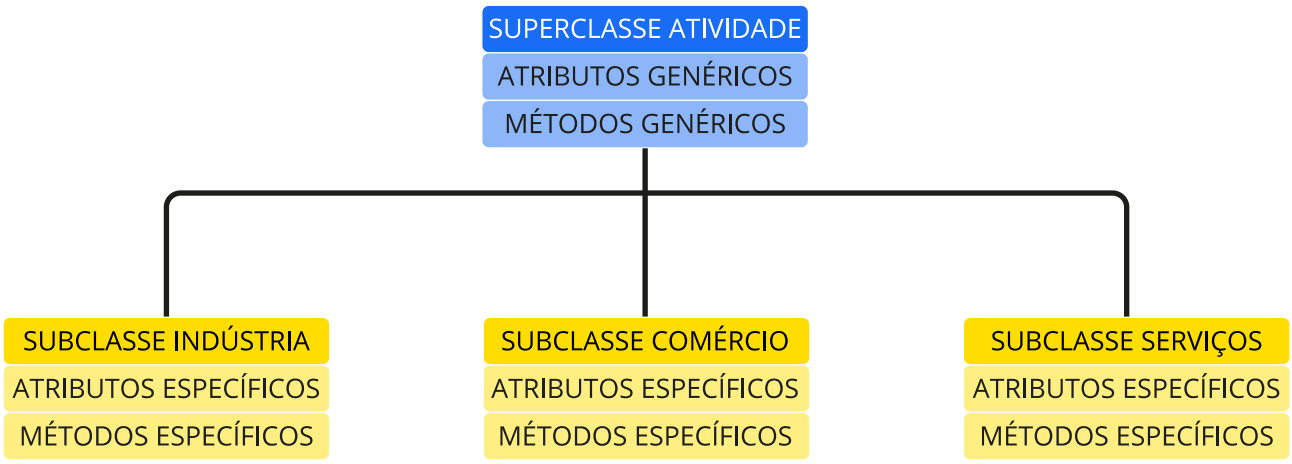
Partindo destas classes, pode-se considerar, por exemplo, veículos como sendo uma superclasse e utilitários e passeio, duas subclasses de veículos, assim como animais pode representar uma superclasse, e peixes e mamíferos, subclasses desta superclasse.



Fonte: O autor.

A imagem traz um exemplo de classes e superclasses ilustrando a classe Atividade como sendo uma superclasse mais geral e ligada a esta classe, outras três mais específicas de Indústria, Comércio e Serviços.

Estas três subclasses não são totalmente dependentes da superclasse, mas podem se beneficiar do que ela oferece para complementação e redução de código por evitar repetição desnecessária, além de estruturar de forma mais coerente a modelagem do software em si.



Fonte: O autor.

Em complemento à imagem anterior, a imagem acima traz agora o restante dos principais componentes das classes, sendo importante observar que as subclasses possuem especificidades também em seus atributos e métodos, utilizando os atributos e métodos da superclasse quando necessário através da herança.

PARA GABARITAR

A herança, assim como todos os conceitos de orientação a objetos deve ser compreendido de forma adequada, pois a programação neste paradigma pode ser simples se não forem utilizados muitos mecanismos próprios do paradigma, tornando um código similar à programação imperativa, mas se os conceitos importantes como herança e polimorfismo forem utilizados, respeitando as limitações impostas pelo encapsulamento definido em um código, o resultado final tende a se distanciar de códigos escritos em linguagens imperativas.

Polimorfismo

Outra característica importante da orientação a objetos é chamada de polimorfismo que complementa a herança, proporcionando que subclasses possam personalizar ainda mais suas funcionalidades em relação ao que existe na superclasse a qual esteja associada.

A ideia se baseia na suposição de que uma superclasse possua um método que atenda aos seus requisitos, e subclasses associadas a esta superclasse utilizem este mesmo método, mas tenham alguma variação nos requisitos para este método, e partindo dessa necessidade, o polimorfismo permite que uma subclasse reescreva métodos de uma superclasse usando o mesmo nome e assim, podendo personalizar este método para seu uso.

Tomando como exemplo a superclasse “Animais” e duas subclasses “Mamíferos” e “Peixes”, um método de locomoção poderia existir em Animais, pois todos os animais dentro do escopo do software teriam a capacidade de se mover, mas no caso dos

mamíferos, existem propriedades de locomoção diferenciadas em relação ao que ocorre na classe Peixes como o fato de mamíferos poderem andar e nadar, mas os peixes apenas nadarem, excetuando particularidades que possam ocorrer.

Dessa forma, o uso do polimorfismo se mostra como um excelente mecanismo para que classes possam estar interligadas sem prejudicar as particularidades existentes, mantendo as características de cada uma, sem a necessidade de adaptações quando ocorre a ligação por herança normalmente.

Aprender os conceitos de orientação a objetos é bastante importante para compreender códigos, e muitas vezes é preciso exemplos em alguma linguagem para que se possa compreender os conceitos. Neste site, os conceitos são avaliados sobre exemplos que podem auxiliar a compreensão destes, mas é preciso conhecer a base da linguagem para que se possa interpretar corretamente cada conceito dentro do restante do código.

Acesse o link: **Disponível aqui**

Programação Orientada a Objetos – Linguagem Java

Provavelmente, a linguagem mais popular para uso da orientação a objetos seja Java, e é a linguagem ideal para aplicar os conceitos deste paradigma com exemplos demonstrativos de aspectos ligados aos conceitos definidos na parte conceitual da orientação a objetos.

Linguagem Java

Em Java os tipos básicos de dados não variam muito da linguagem C estruturada, por exemplo, pois os dados puros em si se mantêm como números inteiros, decimais de ponto flutuante, caractere e lógico, mas as estruturas de dados sofrem muita variação em função de simplificações da linguagem em relação à C para que seja uma linguagem de mais alto nível.

Os tipos de dados em Java são pensados para que estruturas mais complexas em termos de implementação e uso sejam simplificadas em sua implementação básica, e assim, listas são mais facilmente manipuladas por tipos mais genéricos que na linguagem C, por exemplo, onde estruturas de dados como listas podem ser mais complexas em sua implementação e uso.

Existe o tipo de dado coleção, definido como *ArrayList* em Java, por exemplo, que serve para que objetos instanciados de classes sejam armazenados em forma de vetor, não podendo ser utilizado para tipos primitivos como inteiros ou caracteres, por exemplo, e para que um dado primitivo possa ser adicionado em uma *ArrayList*, é preciso que seja inserido em um objeto antes.

A medida que a linguagem evoluiu, a forma como os dados primitivos e tipos mais complexos foi sendo melhorada e a inserção de dados primitivos em um *ArrayList* foi sendo melhorada, assim como outras funcionalidades e novas implementações para tornar a linguagem Java sempre atual e com uma variedade maior de recursos a cada nova versão.

Também existem as diferenças em função da mudança de paradigma que obrigam que a estruturação de todo o código seja diferente pela inserção da ideia de classes ao invés de registros, por exemplo.

O exemplo da imagem a seguir traz a base de um código em linguagem Java para análise de sua estrutura, tipos básicos de dados e estruturação de classes com seus atributos e métodos.

```
package Exemplos;
import java.util.Scanner;
class Main {
       public static void main(String[] args) {
           int resposta;
           ValidaIdade obj;
           obj = new ValidaIdade();
           resposta = obj.setIdade();
           if (resposta == 1)
                System.out.printf("Maior de Idade");
           else
                System.out.printf("Menor de Idade");
           }
}

class ValidaIdade {
           private int idade;
           public ValidaIdade () {
                idade = 0;
           }
           public int setIdade() {
                Scanner dado = new Scanner(System.in);
                System.out.printf("Informe a idade: ");
                idade = dado.nextInt();
                dado.close();
                if (idade >= 18)
                     return 1;
```

```
                else
                    return 0;
                }
    }
```

Fonte: autor.

Observando o exemplo, temos um código simples, mas completo que ilustra alguns fundamentos da linguagem Java dentro do paradigma orientado a objetos como a definição das classes **Main** e **ValidaIdade**.

Estas duas classes servem de exemplo da estrutura semântica e sintática da linguagem, mas também apresenta a definição de um atributo idade que utiliza o modificador de acesso *private* para mantê-lo encapsulado à sua classe de origem e para que possa ser utilizado, é necessário que um objeto *obj* seja instanciado a partir da classe ValidaIdade.

O método de instanciação de objetos em Java e a compreensão da semântica e sintaxe envolvidas no processo não é algo trivial, pois os conceitos de orientação a objetos não são tão simples como os da programação estruturada em alguns aspectos, mas a simplificação do uso de estruturas de dados na linguagem Java, por exemplo, servem como compensação.

A partir do momento em que um objeto está instanciado, os métodos a serem acessados devem conter antes de seu nome, a indicação do objeto ao qual estará associado para que o processamento de dados realizado se refira aos dados contidos nos atributos deste objeto.

Para analisar o código como um todo, é importante observar a sua sequência de execução, iniciando pela classe que possui o método main() que é padrão da linguagem como ponto de partida, sendo uma exceção da própria implementação da linguagem que não precisa ser associado a um objeto instanciado, pois isto não seria possível logo na inicialização da aplicação, pois os objetos vão sendo instanciados ao longo da execução da aplicação quando necessário.

Depois de iniciada a execução do método main(), uma variável resposta é declarada, e neste momento é importante observar que não se trata de um atributo de classe, pois se encontra declarada dentro de um método, e os atributos são declarados fora dos

métodos, sendo algo comum da semântica da linguagem Java, mas que pode confundir a leitura de códigos inicialmente.

Logo em seguida, pelo fluxo de execução do software, um objeto é instanciado para a classe ValidaIdade, para que seus atributos e métodos possam ser utilizados para a realização de processos necessários, como o recebimento do dado para o atributo idade, sendo que um método set() e outro get() poderiam ser implementados apenas para gerenciamento deste atributo, mas foi optado pela criação de um método que recebe e já valida a idade para retornar um resultado já processável pelo restante do software.

Este retorno é então verificado em uma estrutura de decisão que verifica entre 0 e 1 os valores possíveis de retorno do método, sendo este retorno chamado de mensagem que é passada entre métodos durante o fluxo de execução.

Por fim, o código avalia o valor recebido na variável de apoio resposta declarada para uso pelo método main() e dependendo de cada possível resposta, mensagens são exibidas ao usuário, de maneira informativa apenas.

Alguns pontos relevantes podem ser comentados como a classe System, padrão da linguagem e que pode ser diretamente acessada sem a necessidade de importação como foi feito com a classe java.util.Scanner para que se possa receber dados pelo objeto instanciado com base nesta classe importada.

Por fim, outro ponto importante deste exemplo da imagem 21 é o uso de um construtor *public* ValidaIdade() para inicializar o atributo idade com zero, de forma a garantir que não exista um valor inadequado contido na área de memória alocada para o atributo.

Um último aspecto importante é que em Java, assim como em outras linguagens de programação, o encapsulamento pode permitir a inclusão de mais de uma classe sob um mesmo software através do uso do conceito de encapsulamento de nomeação segundo (SEBESTA, 2011), onde o uso da palavra reservada *package* permite que várias classes possam estar associadas a um mesmo pacote e façam parte de um conjunto de códigos Java que compõem a aplicação completa.

Partindo da ideia do pacote, na linguagem Java não é muito aconselhável que se tenha mais de uma classe num mesmo arquivo, mas caso sejam interligadas e uma classe seja dominante e relacionada à outra classe dependente, é possível então declarar a

classe dominante como pública e a outra ser mantida ser modificador de acesso (sendo então default) ou *private*, pois se for colocada como *public* também, cria-se uma confusão na semântica de código em relação à definição da linguagem.

```
public class Profissao {
         protected void trabalhar () {
              System.out.println( "Profissional trabalha em sua
função." );
         }
}

public class Agricultor  extends Profissao {
         private void trabalhar () {
              System.out.println( "Agricultor planta." );
         }
}

public class Pedreiro extends Profissao {
         private void trabalhar () {
              System.out.println( "Pedreiro constrói." );
         }
}
```

Fonte: autor.

No exemplo de classes da imagem, temos três classes que, pelos seus nomes, indicam que pode existir uma relação entre elas, e isto pode ser claramente observado através do uso da palavra reservada *extends* em duas das classes.

A palavra *extends* indica ao compilador que as classes “Agricultor” e “Pedreiro” são classes que funcionam como extensões da classe Profissao, e assim, as duas classes que utilizam a palavra *extends* se tornam subclasses da classe indicada após a palavra

reservada, ou neste caso, Profissao que se torna uma superclasse de onde as subclasses podem ter acesso a atributos e métodos protegidos através da herança.

Isso significa que as subclasses “Agricultor” e “Pedreiro” podem acessar atributos e métodos da superclasse Profissao, mas um ponto que deve ser levado em consideração para uso da herança em Java é o encapsulamento, pois ele determina o que pode ser visto em até três níveis.

O primeiro privado utilizando a palavra reservada *private* para identificar um atributo ou método como específico da classe e invisível às demais classes de um pacote de classes que compõem uma aplicação.

O segundo é público que utiliza a palavra reservada *public* que permite que atributos ou métodos sejam sempre visíveis por outras classes de um mesmo pacote, reduzindo muito a aplicação do encapsulamento e com isto deixando toda a aplicação mais exposta.

E terceiro a visibilidade privada que utiliza a palavra reservada *protected* que permite que subclasses acessem atributos ou métodos de uma superclasse, sendo esta, a visibilidade adequada para se manter as propriedades de encapsulamento e herança adequadas a um pacote de classes de uma aplicação completa.

Outro conceito extremamente importante no exemplo da imagem acima e tda imagem a seguir, é o de polimorfismo, em que o método trabalhar() é reescrito nas subclasses para adequar seus processos às especificidades destas subclasses em relação ao que ocorre no método de mesmo nome na superclasse.

A ideia deste conceito complementar ao da herança é de que uma subclasse possui particularidades em relação a uma superclasse, e assim, pode ser necessário adaptar o processamento de dados que ocorrem na subclasse em relação ao que ocorre em um mesmo método da superclasse.

Com este exemplo então, é possível observar a ocorrência de três dos principais conceitos do paradigma de orientação a objetos:

- Encapsulamento com o uso das palavras reservadas *private*, *public* e *protected* para controlar a visualização de atributos, métodos e classes.
- Herança com o uso da palavra extends para indicar quando uma classe representa uma subclasse de outra classe que se torna superclasse e pode permitir que uma subclasse tenha acesso a atributos e métodos através de instâncias das subclasses.

- Polimorfismo para que métodos de uma subclasse equivalentes a de sua superclasse possam ser reescritos para se adaptarem aos requisitos da subclasse.

A programação orientada a objetos em linguagem Java possui muitos recursos próprios e merece atenção especial por parte dos interessados em aprender uma linguagem de programação orientada a objetos, pois uma linguagem ainda dominante no mercado e possui uma vasta coleção de bibliotecas implementadas e testadas, e frameworks desenvolvidos a partir da linguagem.

Outra possibilidade mostrada na imagem 23 é o uso de classes e métodos abstratos que podem ser utilizados em classes que apenas servem como estrutura para subclasses que irão implementar o conteúdo destas classes realmente, tendo assim, instruções dentro do métodos das subclasses, ao contrário dos métodos de classes abstratas que não contém instruções, apenas a declaração de métodos para referência estrutural das classes da aplicação.

```
public abstract class Profissao {
          public abstract void trabalhar ();
}

public class Agricultor  extends Profissao {
          private void trabalhar () {
               System.out.println( "Agricultor planta." );
          }
}

public class Pedreiro extends Profissao {
          private void trabalhar () {
               System.out.println( "Pedreiro constrói." );
          }
}
```

Fonte: autor.

Outros conceitos relacionados à orientação a objetos e presentes na linguagem Java como o uso de interfaces que servem como classe padrão para a implementação de outras classes, tendo apenas declarações de métodos sem implementá-los, pois serão implementados nas classes indicadas pela palavra reservada *implements*.

```
public interface Profissao {
          public void trabalhar ();
}

public class Agricultor  implements Profissao {
          @Override
          private void trabalhar () {
                System.out.println( "Agricultor planta." );
          }
}

public class Pedreiro implements Profissao {
          @Override
          private void trabalhar () {
                System.out.println( "Pedreiro constrói." );
          }
}
```

Fonte: autor.

Observando o exemplo de código da imagem acima é possível que se faça uma comparação com o exemplo da figura anterior, e imagine que são duas formas diferentes de se fazer exatamente a mesma coisa, mas não são métodos diferentes para atingir o mesmo objetivo.

Um ponto de atenção no exemplo é que se usa a palavra reservada @Override que possui a função de informar ao compilador que o método sendo implementado a seguir é uma modificação de outro de igual nome em uma superclasse, e caso este

nome não seja encontrado na superclasse, um erro de compilação ocorrerá, auxiliando na manutenção de código, pois nomes de classes, atributos e métodos podem ser modificados, mesmo não sendo recomendado em códigos maiores.

As interfaces são tipos de classes abstratas que permitem de forma improvisada um recurso chamado de herança múltipla que significa que uma classe pode ser baseada em mais de uma classe para ser estruturada e posteriormente instanciar objetos.

A partir da versão 8 de Java, a herança múltipla foi implementada e pode ser normalmente utilizada seguindo uma sintaxe semelhante à do exemplo da imagem a seguir em que a indicação da herança múltipla ocorre na declaração da subclasse definida a partir de duas outras superclasses separadas por vírgulas.

```
public interface Profissao {
          default void trabalhar ();
System.out.println( "Profissional trabalha em sua função." );


}
public interface Contrato {
          default void regime ();
System.out.println( "Profissional contratado CLT." );
}
public class Emprego implements Profissao, Contrato {
}
public class Cadastro {
public static void main (String[] args) {
            Emprego emprego = new Emprego();
            emprego.trabalhar();
            emprego.regime();
}
}
```

Fonte: autor.

Com estes conceitos, foi possível observar como uma linguagem de programação implementada com base no paradigma orientado a objetos é estruturada, lembrando que é apenas uma amostra da linguagem e de alguns de seus conceitos, sendo que para conhecer mesmo a linguagem, é preciso buscar mais fontes de pesquisa e praticar muito a leitura, interpretação e criação de códigos.

Existem conceitos da orientação a objetos que não são utilizados, a menos que o desenvolvedor tenha uma demanda específica em relação ao uso destes recursos ou imagine sua solução com base nestes conceitos. É o caso do uso de classes abstratas que servem de modelo para outras classes não sendo instanciadas para a geração de objetos, ou as interfaces que representam tipos especiais de classes abstratas que utilizam atributos mais específicos (*static* e *final*) tendo apenas métodos abstratos e públicos, sem terem o seu código para processamento de dados, apenas as estruturas das classes e componentes destas. Esse mecanismo de criação de interfaces permite que algumas situações sejam solucionadas como no caso hipotético onde classes A, B e C sejam herdeiras da classe abstrata Z. E queiramos atribuir apenas as classes B e C determinado comportamento. Se fosse feito pela classe abstrata Z todos herdariam, então resolvemos isso criando uma interface e a relacionamos apenas com B e C.

Programação Orientada a Objetos – Outras Linguagens

O paradigma orientado a objetos é bastante popular na atualidade, e várias linguagens de programação foram criadas já usando como base este paradigma – é o caso da linguagem Java. Mas outras foram adaptadas para permitirem o desenvolvimento de código neste paradigma, como ocorreu com a linguagem C que na implementação C++ adicionou a possibilidade de uso deste paradigma.

Linguagem Smalltalk

Uma linguagem que contempla os conceitos do paradigma orientado a objetos por ser a linguagem que nasceu juntamente com o paradigma é a linguagem Smalltalk, também responsável pelo conceito de interação com usuários através de janelas, segundo Wangenheim (2002).

Por ser uma linguagem aparentemente com pouca variedade de funcionalidades, representa uma linguagem com boa ortogonalidade, ou seja, possui baixa redundância na geração de código.

Outra característica é que as classes que servem para instanciar objetos também são objetos, pois tudo na linguagem Smalltalk são objetos segundo (Wangenheim, 2002), e para se definir classes existe o conceito de metaclasse, e assim, as classes não precisam ser instanciadas para a geração de objetos.

Assim, nessa linguagem, tanto instâncias de classes quanto objetos são criados por um mecanismo de envio de mensagens através do uso da palavra reservada new, assim como em Java.

As classes em Smalltalk possuem também atributos e métodos, além da linguagem ter implementado os mecanismos de herança simples apenas, e tudo que pertence a uma superclasse é automaticamente herdado por suas subclasses. O polimorfismo também faz parte da implementação da linguagem permitindo que duas classes distintas possuam um mesmo nome em superclasse e subclasse.

Outro ponto relevante dentro da linguagem Smalltalk que se refletiu em Java foi a compilação para geração de arquivo intermediário chamado de *bytecode* e que deve ser interpretado por uma máquina virtual instalada e compatível com o sistema operacional em uso.

Com relação à definição de identificadores, a linguagem Smalltalk, assim como C e Java, é *case sensitive*, diferenciando letras minúsculas de maiúsculas, podendo então ser definidas regras de nomenclatura como uso de letras minúsculas para iniciar nomes de atributos e método, e de maiúsculas para nomear classes.

Em relação aos tipos primitivos de dados, a forma como a linguagem trabalha em torno de objetos pode parecer estranha, mas mesmo números simples são considerados objetos instanciados de classes referentes a tipos numéricos, e 10 é considerado um Smallinteger, 23.90 é considerado *float* de ponto flutuante, e um tipo diferente é o fracionário Fraction que será instanciado em frações com divisão não inteira como em 7/3.

A sintaxe da linguagem Smalltalk é um pouco diferente do usual, pois sendo uma das linguagens antigas que serviram de base para outras, foi implementada de forma singular como se pode observar nos exemplos de instruções da imagem:

```
10 timesRepeat: [Transcript show: 'Smalltalk']

1 to: 100 by: 5 do: [: cont |
Transcript show: (cont printString)
]

100 to: 200 do: [: cont |Transcript show: (cont printString)]

"exemplo de utilização do método whileTrue"
|c|
c := 0.
[Transcript show: ('numero' , c printString ).
Transcript cr.
c := c + 1.
c < 10] whileTrue.

"exemplo de utilização od método whileFalse"
|c|
```

```
c := 0.
[Transcript show: ('numero' , c printString ).
Transcript cr.
c := c + 1.
c >= 10] whileFalse.
```

Fonte: Wangenheim (2002).

Nos exemplos de instruções na imagem, nas primeiras três instruções, são definidas estruturas de repetição de três formas semelhantes à maneira como se implementa laços contados em outras linguagens que utilizam a palavra reservada for.

Em Smalltalk, pode optar por utilizar a palavra reservada *timerRepeat* que tem em sua sintaxe diretamente uma determinada quantidade de iterações, e depois, num segundo exemplo, é usada a palavra reservada *to* para indicar delimitadores de início e fim de contagem, e a palavra reservada *by* para informar de quantas em quantas unidades a contagem deve ser realizada.

Depois, ainda no exemplo da imagem acima, as palavras reservadas *whileTrue* (enquanto verdadeiro) e *whileFalse* (enquanto falso) servem para que condições possam ser estabelecidas dentro da sintaxe da linguagem para determinar condições de parada das iterações.

No exemplo contendo *whileTrue*, é declarado um objeto pela instrução |c| que cria uma variável para receber inicialmente o valor 0 para depois ser iniciado um bloco de comandos que primeiramente exibe o valor de c, depois incrementa o valor da variável em uma unidade, executando iterações enquanto o valor de c for menor que 10.

No exemplo utilizando *whileFalse*, o mesmo conjunto de instruções é executado trabalhando sobre o valor de uma variável c, alterando a lógica de controle das iterações enquanto o valor da variável C não for maior ou igual a 10.

Alguns detalhes de sintaxe relevantes são o uso de aspas duplas para comentários e simples para textos a serem exibidos pelo método show(). Também são usados os símbolos de dois pontos para a indicação de parâmetros para comandos e do ponto final como término de comandos. Os colchetes são usados como delimitadores de blocos de comandos e o símbolo | é usado para delimitar identificadores de variáveis.

Assim, percebe-se que conhecendo a base sintática da linguagem, pode-se compreender seus códigos mais facilmente, e a semântica da linguagem não difere muito das demais linguagens, tendo sua estrutura particular de construção de códigos, mas sendo de interpretação até simples em códigos não muito elaborados.

O aprendizado de novas linguagens de programação pode ser mais difícil do que se imagina, pois uma nova linguagem sendo estudada pode ser de um diferente paradigma, fato que por si só já é um dificultador. Pode ocorrer também que a linguagem aceite o mesmo paradigma da linguagem que se domina, mas possa ser multiparadigma e misture conceitos que podem confundir a interpretação de códigos. É importante ler sobre a linguagem e conhecer seus pontos fortes e fracos, saber com base em qual paradigma foi implementada, ver pequenos códigos que sirvam de base para os principais conceitos e estar com a mente aberta para evitar vícios de programação da linguagem já dominada e que possam não ser adequados ou aceitos pela nova linguagem.

Linguagem C++

Outra linguagem de programação que pode ser utilizada para complementar os estudos neste paradigma é a linguagem C++ que possui as funcionalidades da linguagem C, mas agregando, entre outras funcionalidades e conceitos, o paradigma orientado a objetos.

Complementarmente às estruturas que são adequadas ao uso para armazenamento de dados, as classes com seus atributos e métodos foram implementados, permitindo que a base do paradigma orientado a objetos pudesse ser agregada à programação estruturada, criando assim uma linguagem multiparadigma.

Como na linguagem Java, existem três tipos de modificadores de acesso público, protegido e privado, usando as mesmas palavras reservadas *public*, *protected* e *private* para determinar o grau de encapsulamento de atributos e métodos dentro de uma classe, permitindo assim que além do encapsulamento, herança e polimorfismo tenham espaço para serem implementados como recursos na linguagem.

```
#include <iostream>
using namespace std;
class Animal
{
            int codigoAnimal;
            int raca;
            string cor;
            string nome;
            string dono;
public:
            void setCodigo (int codigo);
            void setRaca (int raca);
};
```

Fonte: autor.

No exemplo da imagem, um código é implementado, mas se for observado, não há efetivamente código presente, apenas definições de uma classe com seus atributos e métodos, servindo como conteúdo para um arquivo de cabeçalho para o código efetivamente, sendo então estruturado como uma biblioteca que pode ser nomeada “animal.h” ou ‘animal.hpp” (extensão mais específica para a linguagem C++).

O código em si pode ser implementado em outro arquivo chamado “animal.cpp” tendo a extensão padrão da linguagem C++ e a biblioteca criada sendo agregada pelo comando include como é habitual nos códigos da linguagem C e C++ para bibliotecas padrão, e também para aquelas implementadas para a aplicação.

```
#include "animal.hpp"

void Animal::setCodigo (int codigo) {
            codigoAnimal = codigo;
}

void Animal::setRaca (int racaAnimal) {
            raca = racaAnimal;
}
```

Fonte: autor.

Neste exemplo da imagem, um arquivo animal.cpp, pode conter a implementação dos métodos indicados nas definições inseridas no arquivo animal.hpp, gerando assim a possibilidade de instância da classe e uso de métodos para ajustar dados de atributos como nas demais linguagens orientadas a objeto.

Um ponto importante é o uso de #include “animal.hpp” para que as definições geradas da classe possam ser acessadas pelo compilador, e assim, sendo feita uma ligação entre o “arquivo .cpp” e o “arquivo .hpp” no momento de se gerar um software executável.

Por fim, o código contido no exemplo da imagem 30 representa o componente principal da aplicação toda, onde a função main() é utilizada, sendo a responsável por iniciar a execução do software, observando que os dois outros arquivos não continham esta função.

Nela, são inseridos os comandos necessários para que a aplicação possa funcionar como esperado, recebendo dados e inserindo-os em atributos de objetos instanciados através dos métodos disponíveis.

```
#include "animal.hpp"


void main()
{
            Animal pet;
            pet.setCodigo (1);
            pet.setRaca (3);
}
```

Fonte: autor.

O exemplo da imagem traz a instância do objeto pet utilizando como referência à classe Animal e em seguida, utiliza os métodos set() para atribuir valores aos atributos *codigo* e *raca* do objeto.

Esse terceiro arquivo poderia ter um nome já pensando no software como se chamará e não exatamente como é o nome de uma classe associada, como por exemplo, “petshop.cpp” para que após compilado, o executável final seja chamado por este mesmo nome, simplificando a documentação e compreensão do código todo da aplicação.

Os conceitos de construtores e destrutores também estão presentes, e permitem a garantia de uma correta inicialização de atributos em objetos instanciados através de métodos construtores com o mesmo nome da classe obrigatoriamente. Uma classe também pode ter vários construtores diferentes implementados e só podem ser utilizados através do uso da palavra reservada new ou na instância de um objeto.

O destrutor é chamado quando um objeto não necessita mais ser utilizado devendo ser chamado ao término de seu processamento, usando a interrogação antes do nome de um método de mesmo nome da classe, assim como o construtor.

```
class Animal
{
            int codigoAnimal;
            int raca;
            string cor;
            string nome;
            string dono;
public:
            void Animal (void) {
                                    codigoAnimal = 0;
                        raca = 0;
cor = “”;
                        nome = “”;
                        dono = “”;
            }
            void ~Animal (void) {
                        cout << “Objeto excluído\n”;
            }
};
```

Fonte: autor.

O exemplo de construtor e destrutor da imagem mostra como se pode implementar este tipo de método, e no construtor são inicializados os valores dos atributos com dados de segurança e no destrutor é exibida uma mensagem informativa da exclusão do objeto.

O encapsulamento pode ser implementado de formas variadas em uma linguagem orientada a objetos, de acordo com as diferentes funcionalidades desejadas como oferta pela linguagem. A linguagem C# é um exemplo de linguagem com modificadores de acesso um pouco diferentes das linguagens C++ e Java, tendo além dos modificadores público, protegido e privado, os modificadores interno, interno protegido, e protegido particular.

Fonte: **Disponível aqui**

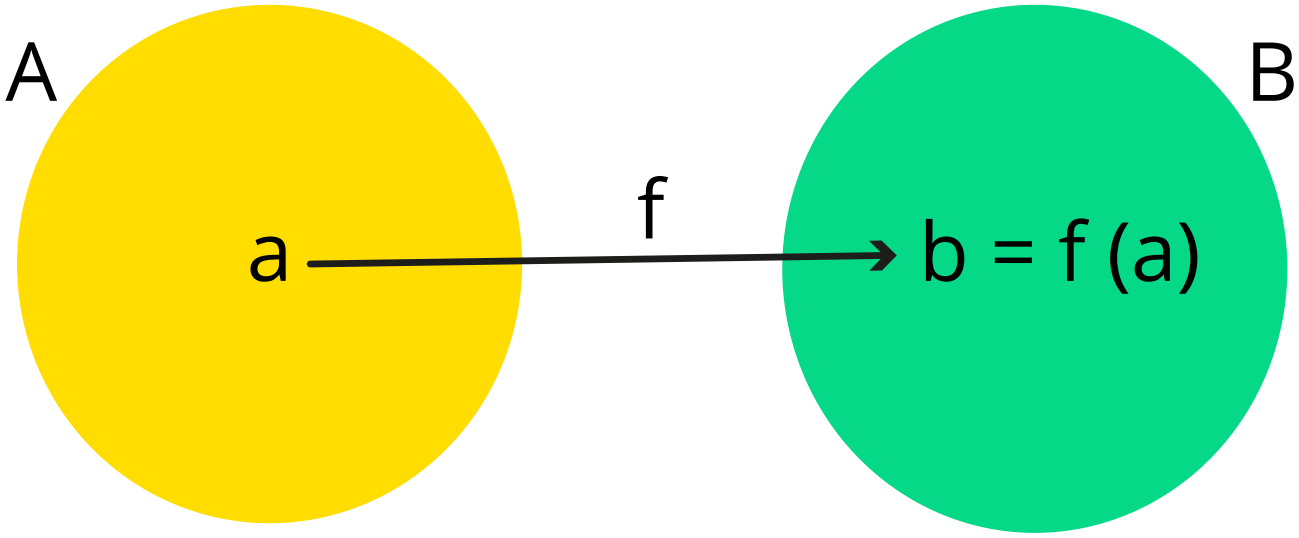
Programação Funcional - Conceitos

O chamado paradigma funcional possui um viés mais matemático e permite uma forma mais específica de desenvolvimento de código onde toda a codificação se baseia no conceito utilizado em funções matemáticas e representa um importante paradigma não imperativo de programação.

Linguagens como Lisp, ML, Scheme e Haskell se baseiam neste paradigma para a estruturação de sua semântica e sintaxe inicialmente, mas podendo adicionar conceitos de outros paradigmas para que pudessem se tornar linguagens mais versáteis.

Funções

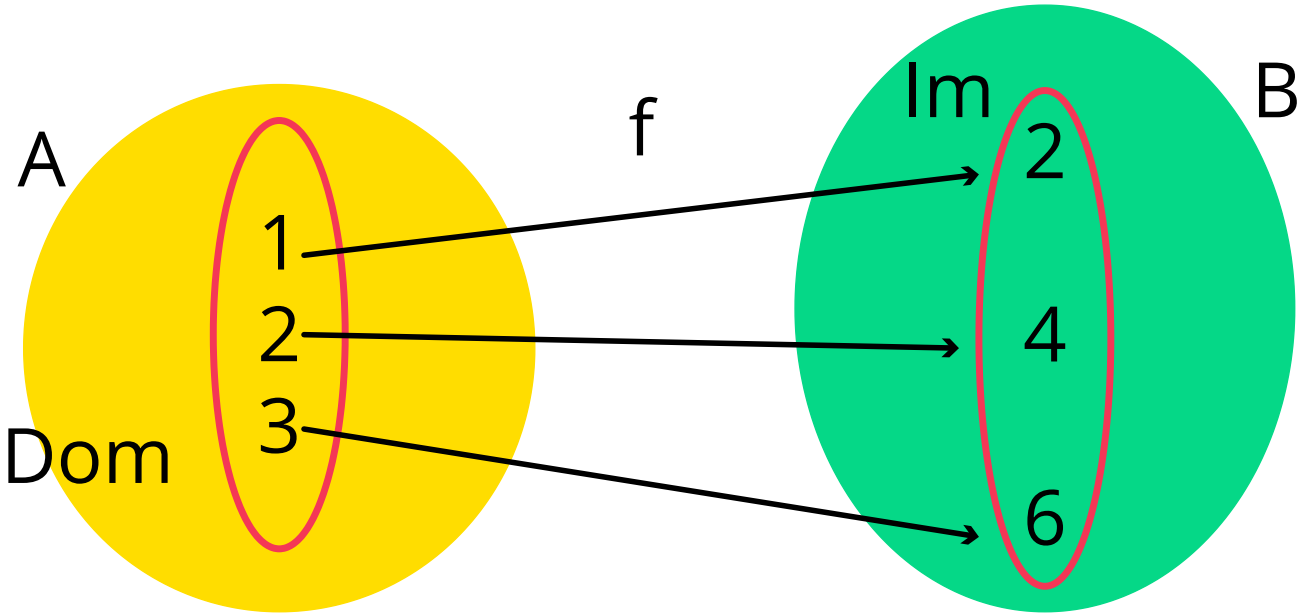
Matematicamente, uma função representa uma relação entre elementos numéricos de um conjunto chamado **domínio** e outro chamado **imagem**, partindo de alguma operação ou expressão aplicada nos elementos do domínio, que resultam sempre em elementos do conjunto imagem. Observe a imagem a seguir para compreender o conceito de forma gráfica.



Fonte: O autor.

No exemplo, é exibida uma representação gráfica genérica com base do chamado diagrama de Venn que usa círculos para representar conjuntos e setas representando as relações entre os elementos de cada conjunto, sendo que a origem da seta ocorre

em um elemento do conjunto domínio, e a seta aponta para um elemento do conjunto imagem.



RELAÇÃO MUITOS PARA MUITOS

Fonte: O autor.

Um exemplo prático de função matemática é mostrado na imagem acima, em que é possível observar que a aplicação de determinada operação sobre os elementos do conjunto domínio A, resultam nos elementos de um conjunto imagem B, e a operação neste exemplo seria a de multiplicar os elementos de A por 2.

No caso da programação, geralmente as funções recebem parâmetros de entrada e os processamentos são realizados com base neste parâmetro ou a função não recebe parâmetros, mas pode manipular dados de fora dela.

Um ponto importante é a forma como os parâmetros podem ser definidos, havendo diferenças entre as implementações das linguagens que podem aceitar apenas variáveis declaradas com tipos primitivos, podem aceitar estruturas de dados mais complexas, ou até objetos e estruturas inteiras.

De forma similar às sintaxes de programação em geral, uma função é escrita iniciando-se pelo identificador da mesma, seguido por um dado, variável ou expressão entre parênteses, funcionando como os parâmetros em programação e depois, a forma como os dados de parâmetros são processados.

A execução do código em linguagens deste tipo ocorre na ordem definida para as operações, tendo regras definidas e não executadas da forma sequencial como ocorre na programação imperativa.

PARA GABARITAR

As chamadas funções puras possuem uma característica importante que se chama imutabilidade que define que uma função, ao receber um parâmetro e utilizar seu valor no processamento interno, sempre deve retornar o mesmo resultado. Uma função que recebe um valor multiplica esse valor por ele mesmo e retorna este resultado obtido é imutável, pois sempre que um valor for passado como parâmetro, o mesmo resultado do valor ao quadrado será retornado, ao passo que numa função de receba um valor e utiliza um dado presente em uma variável externa à função para uso em cálculos, se torna impura, dependendo do valor desta variável para determinação do resultado. O cálculo da cotação de uma moeda que recebe de fora da função o valor para conversão, sem que este seja recebido como parâmetro torna a função impura e assim, menos confiável em relação a não ocorrência de exceções. Funções imutáveis são excelentes no processo de teste e implantação de softwares, pois existe uma confiança de que os resultados são realmente previsíveis e as chances de exceções reduzidas.

Em Lisp, dois tipos de dados básicos são utilizados apenas, chamados de átomos e listas, e estes átomos representam tanto dados numéricos, quando alfanuméricos (caracteres).

As listas são declaradas entre parênteses e representam conjuntos de átomos ou outras listas que também usam parênteses para definir seus próprios elementos. Um exemplo de lista contendo átomos e uma lista poderia ser representado por (1 2 3 (4 5) 6), sendo que 1, 2, 3 e 6 são átomos contidos na lista, e (4 5) representa uma lista contida dentro desta outra lista maior. Pode-se dizer então que a lista possui 4 átomos e uma lista, mas de forma geral, possui 5 elementos componentes.

Neste paradigma, os tipos de dados são um diferencial e como não é um paradigma de propósito geral, sendo pensado em aplicações dentro de um escopo reduzido, não são necessários muitos tipos diferentes e complexos, sendo este ponto, um elemento que simplifica o aprendizado de linguagens de programação funcionais.

Geralmente, as funções são escritas de forma simples com sintaxes bem intuitivas e de fácil interpretação, onde a operação ou função propriamente dita pode ser indicada como primeiro elemento da lista, seguido dos átomos a serem utilizados na função. Esta notação pré-fixada (também conhecida como notação polonesa) é utilizada em calculadoras financeiras, por exemplo.

Na década de 1940, Alonzo Church definiu o termo **lambda** para funções não nomeadas, conceito que foi adotado pelo paradigma e um dos conceitos que é adotado em linguagens de programação que agregam este paradigma ao(s) já existentes como fez a linguagem Java em sua versão 8.

As funções definidas neste paradigma devem ser de natureza pura, ou seja, se um determinado dado for utilizado como parâmetro de entrada para ser utilizado nos processos internos de uma função, esta deve retornar sempre o mesmo resultado, evitando assim, efeitos colaterais indesejáveis causados por resultados não esperados.

Para uma função ser pura e gerar sempre um mesmo resultado, é preciso garantir que isto ocorra, e um dos motivos de modificação e resultado da uma função é o uso de dados de fora da função que não sejam adquiridos através dos parâmetros de entrada, como no uso de dados de variáveis globais em uma função, pois estes podem ser alterados a cada execução e os retornos de uma função para os mesmos parâmetros de entrada poderiam ser diferentes.

Esse problema pode ser resolvido pelo não uso de dados externos além dos recebidos por parâmetros e isto é facilmente resolvido apenas adicionando o valor da variável global como parâmetro da função, e por garantia, se possível, mudar a variável para local de forma que sempre que precise ser utilizada ou ter seu dado atualizado, isto seja feito por passagens de parâmetros e atribuições de dados vinculados a retornos de funções, como uma alternativa de solução.

Conceitos comumente utilizados em outros paradigmas podem ferir o fundamento de pureza de uma função, como na passagem de arquivos como parâmetros, pois podem ter seu conteúdo alterado, e assim, podendo gerar mudanças no retorno da

função, assim como funções que utilizam números aleatórios em seu processamento, tendo assim também, a possibilidade de geração de diferentes retornos com um mesmo dado utilizado como parâmetro de entrada.

Os efeitos colaterais estão ligados a ações que uma função pode realizar e que não sejam totalmente previsíveis ou seguras como alterações de valores em estruturas de dados fora da função que não dependam de aguardar o retorno de dados ao final da execução da mesma, ou seja, atribuições de valores em variáveis ou outras estruturas de escopo global pela função.

O uso de funções puras traz benefícios para desenvolvedores, pois são mais previsíveis em função do maior controle dos dados e processos, mais estáveis e robustos, além de simplificarem a leitura do código e sua compreensão.

Uma função sendo pura favorece seu teste, seu reuso e sua integração, pois se sabe que uma função assim é mais facilmente testável e manipulável, facilitando a compreensão do código.

O conceito de linguagem funcional pura, segundo Sebesta (2011), não utiliza variáveis e nem instruções de atribuição de dados, reduzindo preocupações com recursos de hardware, mas sem a disponibilidade de variáveis, não se pode gerar laços de repetição, pois são controlados justamente por variáveis para atualização do número de iterações e a recursividade é uma solução à ausência de estruturas de repetição.

As linguagens funcionais mais gerais agregam recursos como o uso de variáveis e acabam gerando mudanças de estado e outros recursos que aumentam a versatilidade destas linguagens, mas consomem mais recursos de hardware e possuem menor desempenho em relação a uma linguagem funcional pura. Um aspecto que pesa no desempenho é a recursão que é mais lenta que a estrutura de repetição equivalente, mas existem compiladores que convertem recursões em estruturas iterativas para reduzir este mecanismo mais lento sem que o código necessite ser escrito de outra forma ou outra linguagem.

Linguagens funcionais possuem funções primitivas, mecanismos para a construção de funções complexas a partir de funções primitivas, e formas de representação de dados para parâmetros e retornos destas funções.

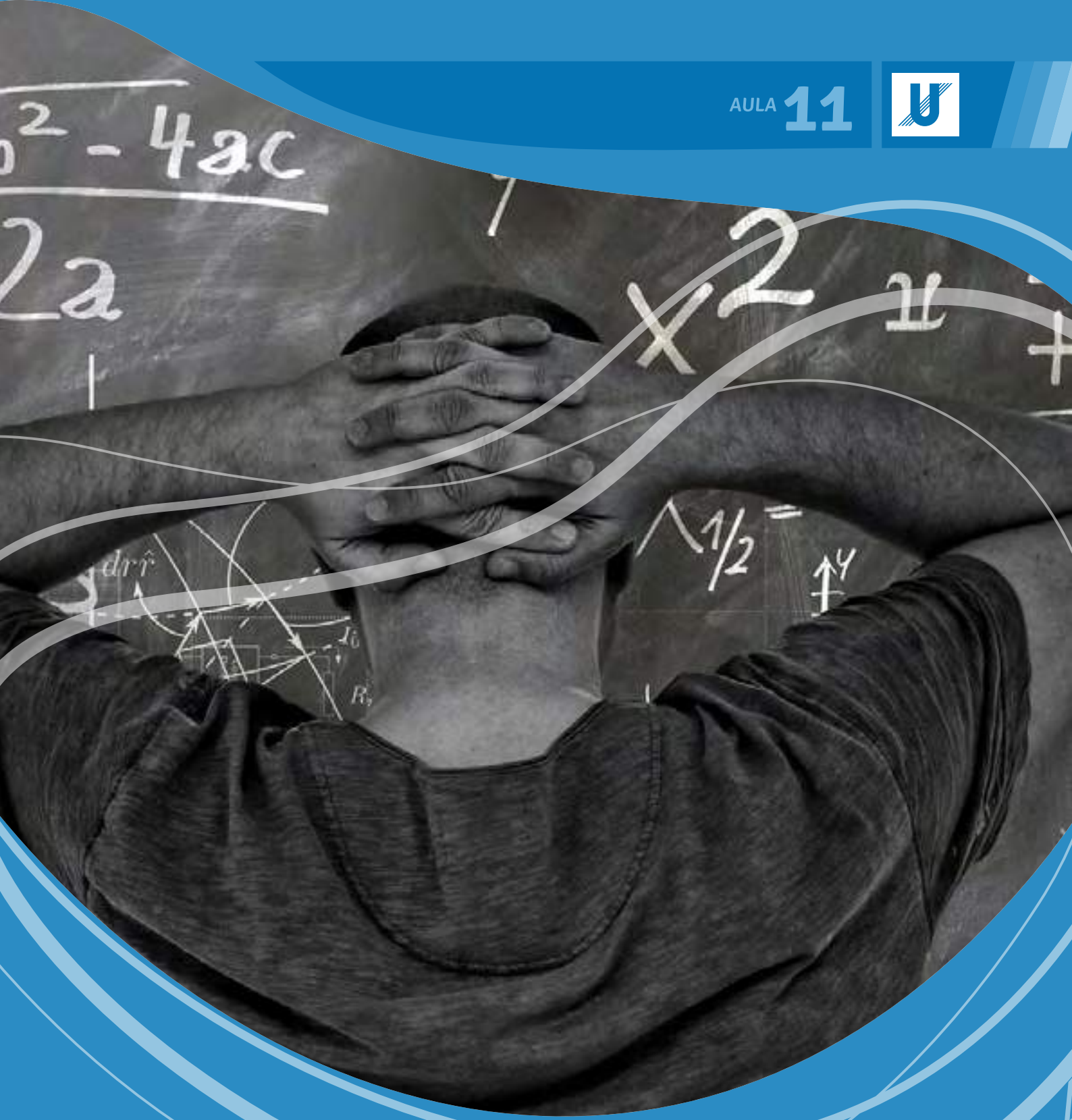
Quando se une uma linguagem funcional com uma imperativa, por exemplo, gerando uma linguagem multiparadigma, existem recursos que precisam ser analisados como o caso de uma linguagem imperativa permitir retornos de tipos mais limitados geralmente, como tipos primitivos, e isto limita o potencial do paradigma funcional.

A programação funcional possui forte referência matemática, mas pode ser aplicada no desenvolvimento de soluções para outros fins. O importante é que as características deste paradigma possam oferecer meios de se desenvolver software mais eficiente para determinados tipos de soluções computacionais.

> *Isso significa que, no paradigma funcional, você tem uma função, coloca um dado de entrada, aplica várias operações e obtém uma saída. É possível alterar as operações e, consequentemente, a saída, mas a entrada sempre permanecerá a mesma.*

Os códigos em programação funcional tendem a ter menos linhas, sendo uma alternativa de linguagem que tende a ser mais fácil de manter e evoluir códigos, testar e avaliar falhas.

Fonte: **Disponível aqui**



Programação Funcional – Linguagem Clojure

Uma opção de linguagem que pode exemplificar a programação funcional é a linguagem Clojure, fruto de uma seleção de aspectos de várias outras linguagens como Ruby, Python, Java e Haskell para oferecer soluções a problemas enfrentados por desenvolvedores destas linguagens.

Linguagem Clojure

Linguagem interpretada que funciona com base na máquina virtual Java (JVM), pode se utilizar de bibliotecas Java, que por si só já lhe confere uma extensa gama de facilidades, permitindo inclusive que aplicações Clojure e Java sejam empacotadas juntas.

Por ser baseada em Lisp, possui características boas da linguagem que permitem que tenha desempenho até superior a outras linguagens como Java e Python que possuem origem baseada em C e Algol.

Por ser uma linguagem funcional, trabalha com conjuntos de dados imutáveis (átomos) e funções primárias, redução de problemas com um controle forte de mudanças de estados, permitindo o desenvolvimento de soluções que utilizem os mecanismos de concorrência e paralelismo.

Para isso, permite que se desenvolvam aplicações utilizando tecnologias de hardwares como de múltiplas CPUs e múltiplos núcleos e computação distribuída, segundo Emerick (2012) que necessitam de linguagens de programação que possuam mecanismos para suportar estas tecnologias.

Outro mecanismo interessante da linguagem Clojure é que ela permite carregamento de código em tempo de execução, sendo importante em aplicações de missão crítica que necessitam estar no ar o tempo todos e assim, permitindo atualizações para correção ou melhorias sem interrupção dos serviços.

A forma como são elaborados os códigos no paradigma funcional é bastante diferente dos demais e inicialmente é bem complexa sua lógica e consequentemente, seu aprendizado, mas com o tempo, a simplicidade dos códigos de programação funcional vão sendo assimilados e a qualidade de ser necessária uma menor quantidade de código em relação a linguagens imperativas ou orientadas a objetos para as mesmas funcionalidades se torna uma vantagem.

No exemplo da figura a seguir, é possível observar um pequeno código que seria maior em praticamente qualquer linguagem em paradigmas estruturado ou orientado o objeto devido ao fato de lidar com lista de valores e funções.

```
(defn media
      [numeros]
      (/ (apply + numeros) (count numeros))
)
```

Fonte: autor.

Nele, uma função media é definida com o uso da função primitiva defn que recebe uma lista de números entre colchetes. A função possui a função primária apply realizando a soma dos números informados e a outra função primária contando a quantidade de elementos do conjunto fornecido.

Assim, pela utilização de uma função de soma dos valores dos elementos e outra de contagem de elementos passados como parâmetros, é possível aplicar a última função primitiva de divisão do resultado da função de soma pelo resultado da função de contagem, obtendo-se então a média aritmética dos valores recebidos como parâmetros pela função media.

Os códigos foram utilizados no ambiente online para teste de programação em diversas linguagens chamado de REPL.IT (https://repl.it) que será utilizado como padrão para teste dos exemplos desta e de outras linguagens de programação ao longo do material como já citado.

```
 (def agenda {:nome "Fulano de Tal"
       :fone "3322-5566"})


(:nome agenda)
" Fulano de Tal"
```

Fonte: autor.

Outro exemplo de código é mostrado na imagem acima, na utilização da função primitiva *def* para que uma estrutura de dados chamada agenda possa receber dados indicados por rótulos **:nome** e **:fone**, que sendo referenciados durante a execução, retornam o dado relacionado ao rótulo informado (nome neste caso).

Nessa situação da imagem acima, uma diferença importante fica por conta da delimitação dos dados rotulados com o uso de chaves, ao invés da lista de elementos delimitados por colchetes na imagem anterior.

Colocando apenas o nome de uma função que necessita de parâmetros não traz resultado algum. Inserir os parâmetros após o nome da função simplesmente como poderia ser feito em outras IDEs e com algumas outras linguagens funcionaria. Inserir os parâmetros corretamente em forma de lista para a soma está correto, mas sem o uso dos parênteses que caracterizam funções no paradigma também geram erros de sintaxe.

Como já citado, a forma como são estruturadas as expressões em programação funcional também são diferentes, pois seguem a chamada notação polonesa ou prefixa onde os operadores são colocados antes dos operandos (valores) na ordem em que devem ser utilizados, e aparentemente é uma notação mais confusa que a tradicional infixa que todos estão acostumados onde os operadores ficam intercalados com os valores.

Um exemplo simples seria a soma 3 + 4 utilizada nas linguagens de paradigmas estruturados e orientados a objeto, por exemplo, e em notação prefixa ficaria + 3 4 que é a forma padrão do paradigma funcional.

Alguns operadores também são escritos de forma diferente em Clojure em relação a C, por exemplo. O operador para negação ! é substituída pelo operador *not*, o operador ++ antes de uma variável é substituído pelo operador *inc*, etc.

Como toda a semântica da linguagem é pensada em funções e notação prefixa, a elaboração de códigos tende a ser mais complexa por não ser algo natural do ser humano esta forma de elaboração de pensamento, mas com o tempo e prática de códigos diversos para compreender todo o conceito de uma linguagem funcional, é possível que sejam produzidos códigos muito bons e altamente eficientes.

```
(defn hypot
      [x y]
      (let [x2 (* x x)
            y2 (* y y)]
            (Math/sqrt (+ x2 y2))
      )
)
```

Fonte: Sebesta (2011).

O código da imagem traz uma aplicação de cálculo da hipotenusa em um triângulo, que representa o maior lado, baseado nos valores dos catetos que estão representados pelos dois lados restantes do triângulo.

Para isto são declaradas duas variáveis x2 e y2 que recebem os valores de x e y respectivamente através da função primitiva *let* que permite que as funções contidas dentro dos colchetes sejam processadas e os valores de x sejam elevados ao quadrado, assim como o valor de y, de acordo com a fórmula trigonométrica.

PARA GABARITAR

Uma excelente maneira de se conhecer as qualidades da programação funcional é através do estudo de uma linguagem que lhe permita utilizar o que se conhece e agregar o novo paradigma.

A linguagem JavaScript é uma boa alternativa, pois é muito utilizada no mercado (fato que pode render um emprego rápido) e aceita o desenvolvimento de códigos com ou sem o uso do paradigma funcional, permitindo uma migração mais tranquila.

A sintaxe das linguagens funcionais é muito similar a matemática e por mais que não haja uma variação muito grande na sintaxe para construção de funções, a lógica utilizada nestas construções pode ser bastante complexa e unir grande quantidade de símbolos e ações não convencionais em linguagens imperativas, por exemplo.

Imagem 39

CÓDIGO	EXECUÇÃO
(defn doubler **[f]** **(fn [& args]** **(* 2 (apply f args)))** **)**	**>>> (def double-+ (doubler +))** **>>> (double-+ 1 2 3)** **>>> 12**

Fonte: Adaptado de Sebesta (2011).

No exemplo da imagem 39 é mostrado uma forma muito diferente de execução de uma aplicação onde o código traz uma forma aberta de definição de função doubler() onde & *args* permite que seja definida uma função double() inclusa dentro de doubler(), e que aplica a multiplicação por 2 sobre a expressão definida pela chamada da função double() definida em tempo de execução.

Na coluna da esquerda, o código possui apenas a função doubler() que está completa, mas deixa em aberto a definição de uma função que será realizada, sendo então definida durante a execução por (def double-+ (doubler +)).

Depois a chamada da função double() com a operação soma indicada para os três valores passados como parâmetros de entrada, realiza a soma dos três e depois o resultado é aplicado na multiplicação da função principal doubler().

Para ter uma ideia de uma aplicação um pouco maior utilizando o paradigma funcional, um exemplo que é muito comum de ser implementado no estudo da programação em si é uma calculadora, pois ajuda a conhecer operações e expressões, funções e estruturas de decisão num mesmo código, oferecendo bom nível de treino e aprendizado.

CÓDIGO	EXECUÇÃO
(defn soma [a c] **(+ a c))** **(defn sub [a c]** **(- a c))** **(defn mult [a c]** **(* a c))** **(defn div [a c]** **(/ a c))** **(defn calc** **[a b c]** **(cond** **(= b +) (soma a c)** **(= b -) (sub a c)** **(= b *) (mult a c)** **(= b /) (div a c)** **)** **)**	**>>> (calc 4 + 2)** **>>> 6** **>>> (calc 8 * 3)** **>>> 24** **>>> (calc 2 - 3)** **>>> -1** **>>> (calc 4 / 2)** **>>> 2** **>>> (calc 7 / 2)** **>>> 7/2** **>>> (calc 3 / 0)** **>>> Divide by zero !!!**

Fonte: O autor.

No código da imagem é clara a ideia da codificação em funções, pois são definidas funções para cada diferente cálculo (soma, *subt*, *mult* e *div*) e uma função calculadora para conter as instruções essenciais da função de chamada do programa e que pode acessar as demais funções declaradas.

Neste exemplo foram testados alguns cálculos, e um ponto curioso é que foram solicitadas três divisões para mostrar o que ocorre com variadas situações, mostrando a importância de serem realizados testes com aplicações.

Na primeira divisão, temos o resultado 2 para uma divisão exata, mas no segundo, a fração 7/2 é retornada como resultado por se tratar de uma operação com resultado de ponto flutuante com casas decimais, e por fim, no terceiro teste, uma convenção básica da matemática é ferida e ocorre um erro na tentativa de divisão de um número qualquer por zero que é uma operação indefinida.

Os conceitos e exemplos desta linguagem servem como ilustração de alguns dos conceitos de uma linguagem funcional, mas em caso de interesse, existe muito material adicional que pode ser explorado sobre a linguagem Clojure.

A linguagem Clojure, por conseguir comunicar-se com bibliotecas Java, sendo este um excelente argumento de uso, mas ainda existem variações da linguagem como Clojure CLR para conversão de código e adaptação para plataformas .NET ou ClojureScript que converte códigos da linguagem para JavaScript.

Cerca de 90% dos micro serviços utilizados pela Nubank estão escritos em Clojure, e diversas bibliotecas foram desenvolvidas pela própria equipe de desenvolvimento da empresa para agilizar o desenvolvimento de aplicações onde se pudesse reaproveitar código.

Acesse o link: **Disponível aqui**

Programação Funcional – Outras Linguagens

A linguagem Lisp é a linguagem mais antiga que utiliza o paradigma funcional e evoluiu com o tempo, mas mesmo assim não representa a linguagem mais popular e também mais completa.

Linguagem Lisp

Como citado anteriormente, esta linguagem utiliza dois tipos de estruturas para dados em sua versão original, sendo elas átomos que representam elementos puros como letras ou números, por exemplo, e listas que representam conjuntos formados por átomos e listas que também podem ser utilizadas como elementos de outra lista.

As funções são escritas de forma simples, com uma sintaxe básica onde é indicada uma função matemática como primeiro átomo dentro dos parênteses e depois, os demais átomos a serem utilizados na função. A função (+ 5 5) é um exemplo que retornaria o valor 10 como resultado da soma dos átomos 5 e 5.

Com o tempo, novas formas de construção de código em Lisp foram implementadas para que funções pudessem ser vinculadas a outras funções como no exemplo (* 2 (- 10 5)), com resultado 10. Comandos foram agregados com o passar do tempo para manter a linguagem popular e aumentar sua eficiência, mas mesmo assim, como já dito, a mesma não se conquistou uma grande popularidade no mercado devido às limitações em termos de aplicações práticas desejadas pelo mercado provavelmente.

Uma linguagem implementada recentemente pela JetBrains é chamada kotlin e propõe uma programação eficiente no desenvolvimento para web, mobile e desktop. Esta linguagem possui como base a orientação a objetos, mas por permitir que funções sejam escritas fora de classes, pode implementar códigos puramente funcionais como em linguagens baseadas no paradigma funcional.

Fonte: Disponível aqui

Linguagem Scheme

A linguagem Scheme, criada na década de 1970, foi desenvolvida com base na linguagem Lisp e trata funções de maneira um pouco diferente, podendo ser valores de expressões ou elementos de listas, e assim, podendo ser utilizadas como dados para variáveis ou parâmetros, recurso que não era oferecido pela versão original de Lisp segundo Sebesta (2011).

Sendo uma linguagem simples, de fácil compreensão de sua sintaxe e semântica, funciona como boa opção para uso educacional, e seu interpretador que interage com o usuário permite que diferentes solicitações sejam feitas e avaliadas em tempo de execução.

Como em Lisp, expressões são escritas entre parênteses que funcionam como delimitadores e como elementos de controle da precedência das operações realizadas nas funções.

Expressão	Resultado
10	**10**
(+)	0
(*)	1
(+ 2 2)	4
(- 5 3)	2
(* 2 (+ 1 3))	8
(/ 3 (+ 4 5))	3

Fonte: O autor.

Nos exemplos da imagem, o primeiro informa ao interpretador apenas um elemento primitivo e o mesmo é retornado como resultado, mas nos segundo e terceiro exemplos, como as expressões são escritas tendo apenas o operador sem elementos de parâmetros, os resultados são 0 e 1, respectivamente os elementos neutros de cada operação.

Na sequência, os demais exemplos trazem expressões completas utilizando apenas átomos ou expressões compostas de átomos e outras expressões com operações primitivas, as regras de precedência são aplicadas e os cálculos realizados, retornando os valores expostos ao lado das expressões.

Existem diversas linguagens de programação utilizando o paradigma funcional para poderem implementar a função lambda e outros recursos, e linguagens como Java a partir de sua oitava versão e Python, possuem além dos paradigmas orientado a objeto, o paradigma funcional agregado.

Python é uma linguagem que utiliza muito mais naturalmente o multiparadigma orientação a objetos e funcional em uma sintaxe limpa e que facilita interpretação e manutenção de código, mesmo misturando paradigmas.

```
def par (x):
  return (x % 2) == 0
```

Fonte: Acesse o link **Disponível aqui**

Neste código, é fácil imaginar uma linguagem tipicamente funcional pelas características sintáticas presentes, mas é um código completo e funcional em linguagem Python pelo paradigma funcional existente.

A semântica básica da linguagem Scheme se baseia em definições de funções, e escrevê-las corretamente é o que deve ser primeiramente compreendido para que se possam gerar aplicações para esta linguagem.

Um detalhe curioso é que o código pode ser escrito de forma que o usuário interaja com o mesmo a partir do prompt do interpretador, ou pode ser incluído no código, expressões a serem executadas logo que o código tenha sido executado e como próximo passo, estaria aguardando interação.

```
(define dobro
   (lambda (x)
       (* 2 x)))
(dobro 5)
```

Fonte: autor.

No exemplo da figura, temos a definição de uma função chamada dobro() usando a função primitiva **define** que associa uma função lambda() que aplica num valor recebido para x, o cálculo que realiza a multiplicação por 2. Após a definição da função dobro(), é inserida uma linha de comando que já atribui o valor 5 como parâmetro para a função dobro() permitindo assim que a aplicação já possa executar diretamente o cálculo para este valor.

A sintaxe da linguagem Scheme para ações comuns como entrada e saída de dados é bastante simples e é baseada também na construção de expressões como funções delimitadas por parênteses.

CÓDIGO	EXECUÇÃO
(display "Digite um valor: ") **(define x (read))** **(display "X ao quadrado: ")** **(display (* x x))** **(newline)** **(display "X ao cubo: ")** **(display (* x x x))** **(newline)**	**Digite um valor: 3** **X ao quadrado: 9** **X ao cubo: 27**

Fonte: O autor.

Pelo exemplo da imagem, é possível observar a forma como se estrutura código contendo entrada de dados através da função (*read*) inserida em outra função que usa a palavra reservada define para atribuir o retorno da função de leitura de dados a uma variável x.

Também são utilizadas funções (*display*) para exibir mensagem a usuários, sendo padronizados os conteúdos para exibição como texto delimitado por aspas, ou expressões construídas entre parênteses.

Por fim, para que a execução tenha uma aparência mais agradável, e tanto a mensagem que traz uma explicação do valor do cálculo do quadrado do valor de x, quanto do cubo de x sejam exibidas de forma a facilitar a leitura e interpretação, após a exibição do resultado de cada função, a função (*newline*) se encarrega de mudar de linha na tela de exibição.

Por esses exemplos das imagens já apresentadas nesta aula, fica nítida a construção de códigos em forma de funções delimitadas sempre por parênteses, mostrando a simplicidade sintática do paradigma e da linguagem, além de uma manutenção mais fácil de código se necessário.

Um dos fundamentos da programação funcional que é o uso de funções serve também de base para um dos conceitos mais importantes da programação, a recursão.

Na figura a seguir, temos um exemplo clássico de cálculo de fatorial que utiliza o mecanismo de recursividade para realizar sucessivas iterações e retornar ao final destas o valor calculado desejado.

```
(define (fat n)
     (if (= n 0)
         1
         (* n (fat (- n 1)))))
```

Fonte: autor.

A recursão no paradigma funcional da linguagem Scheme não se diferencia muito do recurso em uma linguagem imperativa, por exemplo, e a premissa de que a função recursiva deve ser composta por uma estrutura de decisão onde a condição representa um critério de parada, e enquanto esta condição não for satisfeita, a função chama a si mesma de forma que a cada iteração ela caminhe na direção de atender à condição de parada é o mesmo.

PARA GABARITAR

A linguagem Haskell é mais um exemplo de linguagem funcional, mas pura, similar à linguagem ML, mas com diferenças sintáticas importantes, pois a linguagem ML possui uma construção de códigos mais semelhante às demais linguagens utilizadas nesta aula, mas Haskell possui uma sintaxe mais parecida com a escrita de sistemas de equações matemáticas.

Imagem 47

ML	Haskell
fun fact(n : int): int = if n = 0 then 1 else n * fact(n - 1);	fact 0 = 1 fact n = n * fact (n – 1)

Fonte: Adaptado de Sebesta, 2011.

É intuitiva a análise dos dois códigos presentas na imagem 47, pois ambos possuem uma função chamada fact(), uma condição de parada associada ao valor 0, e chamadas recursivas à própria função com redução do valor da variável n utilizada como parâmetro de entrada.

Programação Lógica - Conceitos

Baseada na lógica formal, ramo da matemática, utilizam-se diversos conceitos dessa área para a definição do paradigma lógico, e consequentemente, as linguagens baseadas neste paradigma também possuem conceitos adaptados da lógica formal.

Lógica Formal

Segundo Sebesta (2011), um dos conceitos fundamentais deste paradigma são as proposições que são compostas por sentenças lógicas que podem resultar verdadeiras ou falsas, sendo que a base para estas sentenças é a construção de relacionamentos entre elementos.

Existe também um forte uso de símbolos para a construção de proposições de forma que neste paradigma, muitos dos símbolos matemáticos utilizados na programação mudam seu grau de relevância em relação a outros paradigmas.

Os elementos (objetos) utilizados na composição de proposições podem ser representados por termos simples, variáveis ou até constantes como nos demais paradigmas de uma forma similar, mas com características um pouco distintas.

Uma constante é um símbolo que representa um objeto. Uma variável é um símbolo capaz de representar objetos diferentes em momentos diferentes, apesar de, em certo sentido, essas variáveis serem muito mais próximas da matemática do que as variáveis em uma linguagem de programação imperativa. (Sebesta, 2011).

As proposições podem ser simples ou compostas, e uma proposição simples é mais fácil de ser construída e avaliada, pois existe apenas uma avaliação a ser feita como verdadeira ou falsa apenas, ao passo que numa proposição composta, mais de uma proposição simples precisa ser avaliada e se torna necessário um elemento adicional para que os resultados das proposições simples possam ser avaliados em conjunto.

Para realizar a avaliação das proposições simples em uma proposição composta, é preciso que sejam utilizados operadores lógicos representados por símbolos para que os resultados das proposições simples sejam avaliados como verdadeiras ou falsas e representam o resultado da proposição composta.

Na lógica os operadores podem representar negação, conjunção, disjunção, equivalência ou implicação, mas na programação, nem todos os mesmos conceitos são utilizados e alguns símbolos podem ser diferentes. Observe os principais símbolos matemáticos e seus significados na imagem a seguir:

Nome	*Símbolo*	*Exemplo*	*Significado*
negação	¬	¬ *a*	não *a*
conjunção	∩	*a* ∩ *b*	*a* e *b*
disjunção	∪	*a* ∪ *b*	*a* ou *b*
equivalência	≡	*a* ≡ *b*	*a* é equivalente a *b*
implicação	⊃	*a* ⊃ *b*	*a* implica em *b*
	⊂	*a* ⊂ *b*	*b* implica em *a*

Fonte: Sebesta (2011).

Expressões como “a ⊃ b ou “a ∩ b” são muito comuns na matemática como relações entre elementos de conjuntos, mas também podem representar as operações lógicas E e OU que são operações lógicas muito comuns na programação.

Observe a tabela verdade na imagem a seguir para compreender como são avaliados os operadores quanto aos resultados que podem ser obtidos com proposições verdadeiras ou falsas.

P	**q**	**p ^ q**
V	**V**	**V**
V	**F**	**F**
F	**V**	**F**
F	**F**	**F**

Fonte: O autor.

Na imagem acima, pode ser observado que somente quando ambas as proposições simples resultam verdadeiro, a aplicação da conjunção entre estes valores resulta também verdadeiro.

O mesmo vale para disjunções que representam o operador OU e somente resultam em falso no caso de todos os resultados de proposições simples são falsas como é mostrado na imagem a seguir.

P	q	p v q
V	V	V
V	F	V
F	V	V
F	F	F

Fonte: O autor.

PARA GABARITAR

Para melhor compreender os fundamentos do uso de operadores para a construção de proposições, é importante consultar materiais de aplicação de operadores lógicos. Geralmente associam-se tabelas verdade aos operadores lógicos para mostrar como um operador E resulta verdadeiro apenas se todos os valores avaliados pelo operador sejam verdadeiros, ou no caso do operador OU, pelo menos um valor precisa ser verdadeiro para que toda a proposição seja verdadeira.

Fonte: **Disponível aqui**

Termos compostos são representações de funções com sintaxe similar à utilizada em outros paradigmas e suas linguagens e representam as principais estruturas em linguagens de programação baseadas no paradigma lógico.

Paradigma Lógico

Este paradigma trabalha com proposições atômicas compostas por relações semelhantes à da sintaxe de programação funcional onde duas partes compõe a sintaxe, e é a relação e uma lista de elementos entre parênteses que componham esta relação.

```
homem (joao).
mulher (maria).
casados (joao,maria).
```

Fonte: autor.

Pelo exemplo de termos compostos contidos nos exemplos da imagem, temos diferentes situações, em que os dois primeiros exemplos apresentam fatos, pois indicam que determinada relação se aplica a determinado elemento indicado dentro dos parênteses.

As proposições atômicas são uma nomenclatura que pode ser utilizada nestes casos onde a combinação de símbolos de predicados com termos simples ou compostos indicados entre parênteses como mostrado na última imagem.

No terceiro exemplo, existe um tipo diferente de relação onde dois elementos são relacionados entre si a partir da função indicada, e da mesma forma que num termo com apenas um parâmetro, pode ser verdadeiro ou falso.

Linguagens que se baseiam no paradigma lógico são ditas declarativas e são de certa forma mais simples que linguagens imperativas em termos de semântica, mas com menor variedade de soluções computacionais possíveis, sendo por isto mais limitada no mercado.

Por ser um paradigma não procedural, o paradigma foca na forma dos resultados de processamentos realizados e não tanto no processamento em si, fazendo com que o programador não tenha tanto controle sobre o processo em si como no paradigma imperativo onde toda a sequência de ações é determinada sequencialmente.

Em linguagens procedurais, um algoritmo deve ser aplicado sobre dados que contém uma sequência de ações a serem realizadas até que todo o processo esteja concluído e algum resultado desejado seja obtido, ou não seja possível a obtenção de resultados desejáveis por quaisquer problemas.

Em linguagens não procedurais, apenas as características dos elementos que contêm dados devem ser descritos e as relações entre eles, de forma que se possa trabalhar estes dados e obter ou não os resultados esperados, como na ordenação de elementos de uma lista que necessitam apenas que os adjacentes sejam comparados com base em uma relação que trabalhe estes dados.

ABORDAGEM PRÁTICA

Áreas como a de banco de dados podem utilizar em suas buscas determinadas estruturas de dados que servem de índices e podem ser utilizadas em conceitos de lógica e mecanismos presentes no paradigma lógico para obter melhor desempenho. Para grandes volumes de dados, o uso de relações algébricas pode ser muito vantajoso dependendo de como for implementado o mecanismo para uso adequado dos recursos de hardware também. Também pode ser utilizado em aplicações voltadas à área de inteligência artificial e aprendizado de máquina devido à possibilidade de se obter novos conhecimentos a partir da construção de novas relações a partir das proposições já existentes.

Programação Lógica – Linguagem Prolog

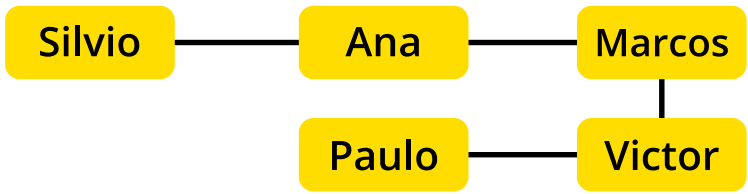
A linguagem Prolog trabalha com o conceito de declarações de fatos sobre objetos que servem para a estruturação de relacionamentos ou regras entre fatos de objetos e objetos em si. Utiliza, além de fatos e regras, a possibilidade de uso de variáveis e listas de dados, além da possibilidade de implementação de recursividade.

Linguagem Prolog

Diferentemente da programação imperativa e estruturada em que algoritmos se baseiam na manipulação de dados, no paradigma lógico, a proposta é de se pensar em o que serve de base para processamento ao invés de como devem ser processados os dados.

As chamadas proposições atômicas são formadas por relações que possuem uma sintaxe não muito diferente da programação funcional, e as relações incidem sobre termos (átomos em Lisp) inseridos em listas indicadas entre parênteses e separados por vírgulas.

amigo(Ana, Marcos)
amigo(Ana, Silvio)
amigo(Paulo, Victor)
amigo(Victor, Marcos).



Fonte: O autor.

No exemplo da imagem, existem 4 fatos determinados pelas proposições indicadas que servem de base para os fatos da relação amigo, e esta pode ser ilustrada como diagrama apenas para que fique clara a ideia de quem é amigo() de quem pela ordem dos elementos.

No código, a relação amigo() é aplicada aos termos ana, marcos, silvio, paulo e victor, sendo cada linha chamada de instância. As instâncias são aplicações diretas de relações sobre listas de termos, e o conjunto de instâncias representa a relação em si.

Esta diferenciação de ordem na relação amigo() pode não ser relevante, pois esta relação possui a propriedade comutativa, ou seja, se for escrito amigo(ana, marcos) ou amigo(marcos, ana) a relação terá o mesmo resultado e significado pela relação ser assim.

Se fosse outra relação como altura, idade ou cargo em uma hierarquia, seria diferente e, por exemplo, uma relação chefe(ana, marcos) não poderia ter o mesmo valor lógico de chefe(marcos, ana), pois uma poderia ser verdadeira, mas daí, certamente a outra não poderia ser, pois a relação chefia não é comutativa.

Em relações não comutativas como a usada no exemplo chefe(), poderiam ser utilizadas setas para indicar no diagrama, a ordem dos termos que representariam a correta aplicação da relação entre os elementos.

O problema é que na programação, a linguagem não interpreta o sentido do nome da relação, e para o interpretador, a relação se chamar amigo(), chefe() ou abc() não faz diferença, pois o que vale são as definições de instância para que a relação seja definida.

Para esta linguagem, pode ser usada a plataforma https://swish.swi-prolog.org/ para execução dos códigos em linguagem Prolog, sendo uma plataforma online para interpretação e teste de códigos.

Para verificar se a relação está corretamente definida por suas instâncias, pode-se testar a mesma utilizando comandos a serem interpretados durante a execução que podem verificar se o código foi corretamente escrito e a relação definida de maneira precisa.

Imagem 1

```
? amigo(X, Y).
X = ana,
Y = marcos
X = ana,
Y = silvio
X = paulo,
Y = victor
X = victor,
Y = marcos
```

Imagem 2

```
? amigo(marcos, Y).
false
```

Com esses dois testes colocados na imagem, temos a indicação de que algo não está correto, pois se solicitamos que todas as instâncias sejam exibidas indicando variáveis em aberto X e Y para a relação, qualquer termo das instâncias é aceito e todas as instâncias definidas aparecem na tela.

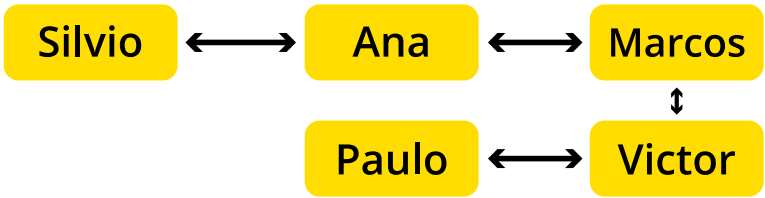
No segundo teste, o resultado acusa falso para amigos de marcos, mas isto não deveria ocorrer, pois **marcos** é amigo de **ana** e **victor**. Isto ocorreu porque nas instâncias, marcos está como segundo termo da relação na instância e no comando de teste, foi colocado como primeiro.

Como o interpretador não "imagina" que a relação possa ser comutativa, simplesmente não encontra instâncias onde marcos esteja à esquerda em conjunto com qualquer outro termo pertencente às instâncias da relação.

Para que a relação entenda a comutatividade e o lado que for referenciado para a busca por amigos se torne irrelevante, como deve ser, é possível ensinar ao interpretador que a ordem inversa dos termos também representa uma mesma relação, e para isto, é preciso adicionar as relações invertidas indicadas nas relações já definidas.

Imagem 3

amigo(Ana, Marcos)
amigo(Ana, Silvio)
amigo(Paulo, Victor)
amigo(Victor, Marcos)
amigo(Marcos, Ana)
amigo(Silvio, Ana)
amigo(Victor, Paulo)
amigo(Marcos, Victor).



Adicionando instâncias com a ordem dos termos invertidos à relação anterior da imagem 1, a nova relação mostrada na imagem 3 agora está com a comutatividade representada, e assim, representando as possibilidades de amizades corretamente.

Com isto, um novo teste com a nova relação, utilizando o mesmo comando que resultou falso no exemplo da imagem 2, agora deve retornar o resultado que aparece na imagem a seguir.

Imagem 4

```
? amigo(marcos, Y).
Y = ana
Y = victor
```

Prolog utiliza termos definidos por constantes que possuem nomes que iniciam por letras minúsculas ou variáveis definidas por nomes que iniciam por letras maiúsculas, ou outras estruturas que representem relações.

As relações podem ser definidas em sequência no código, montando uma base mais complexa de fatos e relações que podem oferecer resultados mais interessantes à medida que a lista de instâncias aumenta, variando as relações e estas, vão tendo ligações entre si.

Um dos exemplos clássicos para demonstração de Prolog se aplica ao conceito de árvore genealógica e para exemplificar como uma linguagem deste tipo pode organizar uma estrutura assim, observe o exemplo da imagem 5.

Imagem 5

```
pai(joao, pedro).
pai(joao, ana).
mae(maria, pedro).
mae(maria, ana).
irmãos(pedro, ana).
irmãos(ana, pedro).
```

Fonte: autor.

Partindo das instâncias definidas nestas relações da imagem 5, existe uma mescla entre relações comutativas e não comutativas onde irmãos() é uma relação comutativa e pai() e mae() são relações não comutativas, onde não se pode inverter os termos das instâncias para mantê-las como verdadeiras.

PARA GABARITAR

O melhor caminho para se aprender a elaborar relações é tentar imaginar problemas que necessitem que dados sejam relacionados entre si, como ocorrem em bancos de dados, e implementar instâncias que simulem as relações entre dados. Observar depois se as relações foram corretamente definidas por um conjunto adequado de instâncias informadas e testar todas as possibilidades de forma a encontrar falhas na definição das relações como foi visto em relações comutativas onde as duas possibilidades de instância com os mesmos termos não é definida.

Imagem 6

```
? pai(joao, X).
X = pedro
X = ana

? mae(maria, Y).
Y = pedro
Y = ana

? irmãos(Z, W).
W = ana,
Z = pedro
W = pedro,
Z = ana
```

Como é possível conferir pelos resultados exibidos na imagem 6, as relações estão definidas a partir das instâncias, e os irmão aparecem duas vezes como resultados em função da comutatividade.

Também é possível realizar processamento de dados em tempo de execução e expressões podem ser calculadas com dados contidos nas instâncias, por exemplo. Observe o exemplo da imagem 7.

Imagem 7

```
% pessoa(Nome, Horas trabalhadas, Salário)
pessoa(jair, 160, 2000).
pessoa(paulo, 150, 3000).
pessoa(felipe, 172, 2500).
```

No exemplo da imagem 7, é fornecida uma base de dados pessoais em três instâncias que definem a relação pessoa. Nestas três instâncias, são fornecidos termos em forma de texto e número para que se obtenha uma pequena base de dados heterogêneos.

Com base nos dados contidos nos fatos apresentados, alguns novos dados podem ser obtidos a partir da aplicação de cálculos realizados sobre os dados presentes nos termos das instâncias.

Imagem 8

```
? pessoa(X,Y, Z), Media is Z/Y.
Media = 12.5,
X = jair,
Y = 160,
Z = 2000

Media = 20,
X = paulo,
Y = 150,
Z = 3000

Media = 14.534883720930232,
X = felipe,
Y = 172,
Z = 2500
```

Neste exemplo da imagem 8, utiliza-se no momento da execução uma solicitação que utiliza os termos como base para se realizar um cálculo para descobrir a média salarial por hora de cada pessoa, associando a divisão do salário pela quantidade de horas trabalhadas, armazenando-as num novo termo chamado Media.

A linguagem Prolog, assim como as demais citadas até aqui, possui diversas outras funcionalidades que podem ser exploradas com estudos mais aprofundados. Para os estudos deste material, apenas alguns aspectos são trazidos de forma a apresentar algumas das diversas linguagens de programação existentes no mercado.

CONECTE-SE

A linguagem Prolog mostra-se bastante diferente na construção de soluções computacionais, mas como toda linguagem, pode ser utilizada na solução de determinados tipos de problemas geralmente.

Algumas aplicações que podem trabalhadas com a linguagem Prolog são nas áreas de computação simbólica como prova automática de teoremas, bancos de dados relacionais, linguagem natural, solucionadores de jogos e sistemas especialistas, dentre outros.

Fonte: **Disponível aqui**

Outros Paradigmas e Formas de Programação

Além dos tradicionais quatro paradigmas tratados ao longo do material, existem alguns diferentes paradigmas que surgiram com base em um ou mais dos paradigmas tradicionais, ou apenas definições mais específicas para determinados tipos de programação utilizados para solucionar problemas específicos e necessidades pontuais.

Programação Orientada a Eventos

Um paradigma alternativo existente é o **orientado a eventos,** que trata da interação de software com eventos esperados com base em rotinas de tratamento de gatilhos acionados por eventos do sistema como mecanismos de entrada, eventos de softwares em execução ou do sistema operacional, por exemplo.

Não é necessariamente uma extensão do paradigma orientado a objetos, mas pode utilizar conceitos deste paradigma, como ocorre com a linguagem C#, por exemplo.

Aceita o uso de recursos como *design patterns* que não representam códigos prontos para o desenvolvimento de softwares, mas modelos para auxiliar em problemas a partir de soluções já desenvolvidas, utilizadas e testadas.

Utilizado em aplicações que utilizam interfaces gráficas para interação com usuários, recebem estas interações como gatilhos para eventos e estes são tratados pelos códigos associados a estes eventos.

Eventos ocorrem em momentos que podem ser previsíveis ou não, envolvendo interação humana ou não, sendo representados por cliques, teclas apertadas, botões de controles, toques em tela sensíveis ao toque, e entradas de dados automatizadas, por exemplo.

Os eventos são monitorados e quando eventos esperados ocorrem podem tratados pelo software usando recursos do próprio sistema operacional, e quando eventos inesperados ocorrem, podem ser tratados diretamente pelo sistema operacional pelo fato de não serem tratados pelo software. Quanto mais completo o tratamento de eventos pelo software, mais completa e robusta é a aplicação.

As linguagens podem ter recursos específicos para o tratamento de eventos em forma de comandos nativos da linguagem, bibliotecas padrões da linguagem ou criadas pelos próprios desenvolvedores para atender necessidades pontuais, etc.

Aplicações para as áreas de bancos de dados, sistemas de arquivos, softwares compartilhados de rede, ferramentas para sistemas operacionais, tratamento de interrupções e aplicações comerciais onde eventos devam ser capturados e tratados.

A interface gráfica desenvolvida por muitos softwares para interação com usuários é um meio tipicamente orientado a eventos relacionados a cliques de mouse, por exemplo, sendo estes capturados primeiramente pelo sistema operacional e tratados pelo software de acordo com os objetos acionados e a forma como ocorre o clique do mouse (botão, arrasto, etc.).

Neste tipo de software, normalmente existe uma rotina que fica sendo executada por padrão e é interrompida por eventos para que haja o processamento relacionado ao tratamento destes eventos, mas depois, a rotina principal que aguarda eventos volta a funcionar a partir dos retornos das funções acionadas nos eventos.

Linguagens que geralmente criam soluções baseadas em interfaces gráficas como as versões Visual Basic, Visual C e Delphi que criam aplicações desktop tipicamente dotadas de interfaces gráficas são um bom exemplo, assim como linguagens como C#, Java e JavaScript que geralmente lidam com o tratamento de eventos em formulários web.

Esta forma de programação acaba tendo um fluxo de execução diferente dos paradigmas imperativo e orientado a objetos por não ter um fluxo contínuo determinado normalmente pelo próprio processamento de dados na aplicação, e sim pode eventos que ocorrem de forma inesperada pelo software normalmente, podendo inclusive, nunca ocorrer.

Muitas vezes, softwares embarcados em dispositivos IoT funcionam com base em laços de repetição infinitos que são interrompidos apenas por eventos ocorridos durante a execução ou interrupções de funcionamento, como ocorre em placas Arduino que recebem código em uma versão própria da linguagem C.

Imagem 1

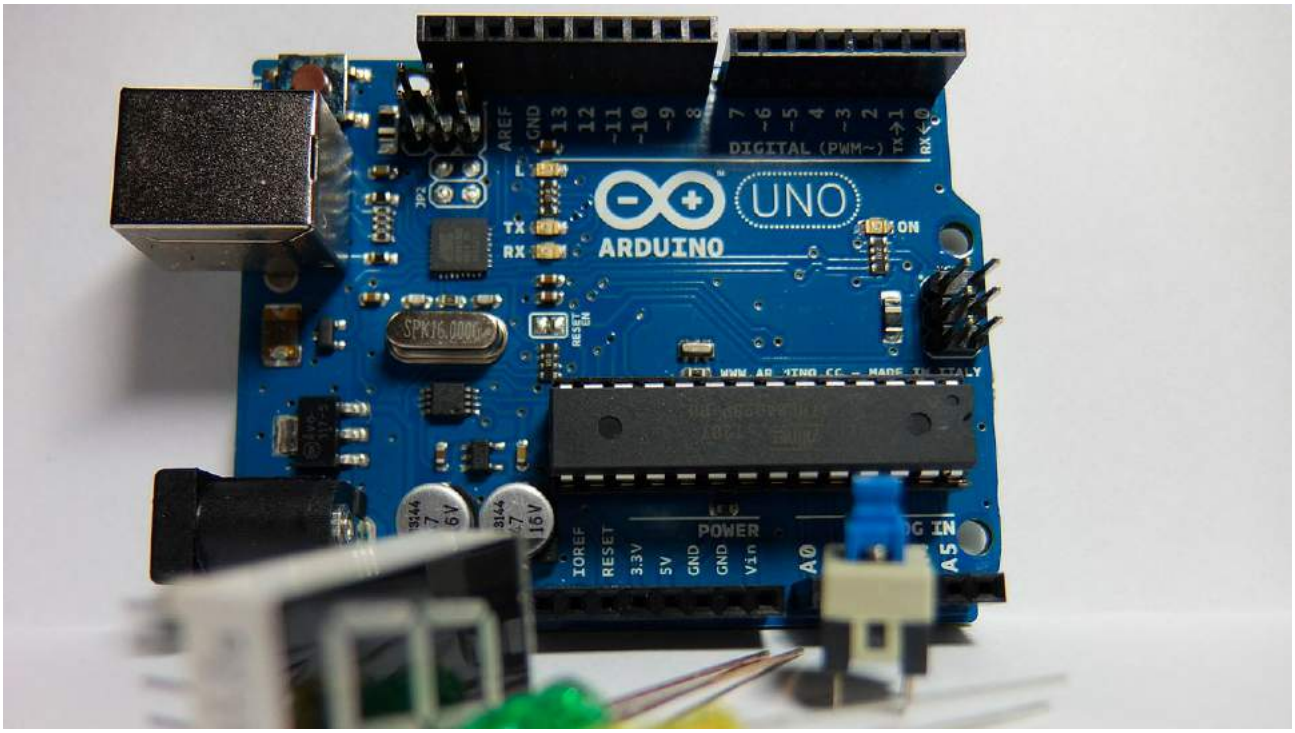
```
int azul = 13;
int vermelho = 14;

void setup() {
            pinMode (azul, OUTPUT);
                        pinMode (vermelho, OUTPUT);
}

void loop() {
            digitalWrite (azul, HIGH);
delay (100);
                        digitalWrite (vermelho, HIGH);
                        delay (100);
                        digitalWrite (azul, LOW);
                        delay (100);
                        digitalWrite (vermelho, LOW);
                        delay (100);
}
```

O código presenta na imagem 1 traz um exemplo de software que pode ser embarcado em uma placa Arduino como a mostrada na imagem 63 que permite que códigos em linguagem C sejam inseridos no chip de memória da placa para serem executados assim que a placa seja carregada eletricamente.

Esse código traz um exemplo de aplicação da placa onde dois leds de cores azul e vermelho podem ser ligados nos pinos 13 e 14 respectivamente para que sejam acesos e apagados sequencialmente a cada período de tempo indicado na função delay().

O código é dividido em duas funções, sendo a função setup() responsável pelas configurações físicas da placa em relação ao código, e a função loop() o laço de repetição infinita que aguarda os possíveis eventos de interrupção do modo de espera da placa aguardando eventos.

Imagem 2



Fonte: Pixabay.

Interrupções de hardware como botões pressionados em um projeto Arduino ou interrupções de software como cliques geram tratamentos diferentes por parte do conjunto hardware, sistema operacional e software, e até interpretadores de comandos de linguagens interpretadas são exemplos de orientação a eventos, aguardando comandos do usuário.

PARA GABARITAR

Outra forma distinta de programação é feita por blocos de forma gráfica, bastante intuitiva e de facílima compreensão utilizada em linguagens como Scratch ou Blocky que oferecem objetos em formas apropriadas para encaixe uns com os outros de forma que comandos de repetição tenham um formato que sirva para englobar outros comandos, gerando instruções completas em blocos, e daí a ideia da programação em blocos.

Extremamente didáticas, essas linguagens se tornaram perfeitos métodos de inserção de pessoas de todas as idades no aprendizado de lógica de programação, oferecendo meios de compreender ludicamente, o desenvolvimento das principais estruturas de código geralmente associadas ao paradigma imperativo. Com uma sintaxe e semântica extremamente simples, mas completa, essas linguagens ganharam, espaço na área de educação e se tornaram propostas de método de programação para o futuro, onde algoritmos formados por blocos podem ser traduzidos em outras linguagens e gerar aplicações funcionais.

Programação Orientada a Aspectos

Outra forma de programação relevante é a relacionada ao chamado paradigma orientado a aspectos, em que é possível organizar o código a partir de sua relevância dentro dos requisitos e propósitos em cada aplicação, complementando o paradigma orientado a objetos que se baseiam em comportamentos de objetos, agindo como código auxiliar nas aplicações.

Paradigma desenvolvido por Gregor Kiczales e a sua equipe na Xerox PARC, a divisão de pesquisa da Xerox, serviu de base para o desenvolvimento de uma linguagem de programação orientada a aspectos chamada AspectJ que oferece mecanismos para que classes públicas comuns possam ser reorganizadas como classes em aspectos.

A linguagem AspectJ não cria aplicações completas, mas complementos para aplicações Java, assim como linguagens como PHP e JavaScript podem ser ligadas a scripts HTML e CSS para funcionarem conjuntamente.

Aspectos são organizações de partes de um código de acordo com seu nível de importância dentro de toda a aplicação, permitindo que haja um encapsulamento em módulos separados de código secundário, estrutura que não é contemplada pela linguagem Java, gerando mecanismos para controle transversal de código.

Um *aspect weaver* é uma ferramenta que age como um compilador, compondo uma aplicação completa que contém toda a parte fundamental do código em importância unida a todos os aspectos que contém códigos transversais de menor relevância através de um processo chamado *weaving*.

A imagem 2 traz um exemplo de aplicação de código conjunto Java e AspectJ separados em dois arquivos que são unificados no processo de compilação para geração de arquivo *bytecode* para interpretação pela máquina virtual Java como padrão da linguagem.

Imagem 2

```
// Arquivo Java - Mensagem.java
public class Mensagem {
            public static void entrega(String mensagem) {
                        System.out.println(mensagem);
            }

            public static void entrega(String person, String
mensagem) {
                        System.out.print(person + ", " + mensagem);
            }
}
```

```
//Arquivo AspectJ - Aspecto.java
public aspect Aspecto {
            pointcut enviaMensagem()
                     : call( Mensagem.entrega("Mensagem enviada!");
);

            before() : enviaMensagem () {
                  System.out.print("Testando... ");
            }
}
```

Fonte: Laddad (2003 - Adaptado).

O código do arquivo Java na imagem 2 existe um típico código em linguagem Java que simplesmente tem a função de exibir uma frase para o usuário e na sequência, no código do arquivo Aspecto.java, a declaração de um aspecto, onde ocorre a declaração do *pointcut*, uma espécie de desvio que contém o chamado *join point* que define a ligação entre classe e aspecto.

Recursos para controle de prioridades na execução são definidos através do uso da palavra reservada *before* que indica que o conteúdo do método enviaMensagem() desta linha deve ser executado antes do conteúdo do método enviaMensagem() principal indicado pelo *pointcut call* representando um desvio incondicional durante a execução como era feito com o uso de GOTO décadas atrás.

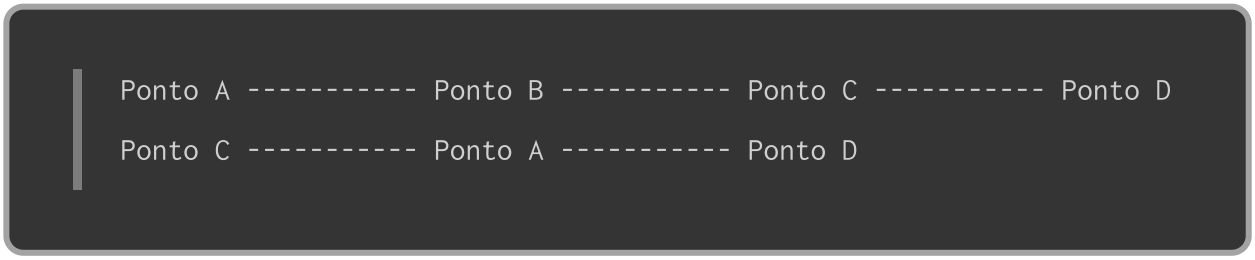
A adição deste mecanismo de aspecto faz com que a mensagem indicada por *before* no aspecto seja exibida antes da mensagem indicada pela palavra reservada *call* obtendo como resultado “Testando... Mensagem enviada!”, mostrando assim que mecanismos de tratamento de erros de abertura de arquivos ou tratamentos de erros diversos que são executados durante a execução normal da aplicação possam ser agrupados em forma de aspectos.

Programação Dinâmica

Uma última forma de programação a ser estudada se chama programação dinâmica e permite que problemas complexos possam ser reduzidos a problemas de menos complexidade, de forma a obter otimização no processo de desenvolvimento de software.

Uma aplicação para este mecanismo é no uso da teoria de grafos aplicável em muitas soluções computacionais como de rotas logísticas, onde pontos de referência das rotas podem ser representados por vértices e os caminhos possíveis serem definidos por arestas que ligam pares de vértices e podem servir como base para avaliação de menores caminhos ou que agreguem menores ou maiores custos de forma geral.

Imagem 3



Uma estrutura como a mostrada na imagem 3 que pode também de representada em forma de diagrama ou como uma estrutura de dados do tipo vetor ou matriz, é uma aplicação para a programação dinâmica.

As relações entre os pontos podem ser definidas apenas por serem adjacentes, ou seja, vizinhos como o Ponto A e o Ponto B, mas podem ser agregar valores chamados de pesos que influenciam diversas situações tais como na escolha dos algoritmos adequados para buscar as melhores rotas.

Conhecer os conceitos básicos da teoria de grafos é bastante relevante para estudos na área de TI e deve fazer parte dos conhecimentos essenciais de desenvolvedores de soluções computacionais.

Fonte: **Disponível aqui**

Uma característica importante da programação dinâmica é a possibilidade de estruturação de problemas de otimização de etapas a serem resolvidas sequencialmente, considerando os problemas fracionados como completos, desde que seja possível a integração das partes como uma solução completa para o problema original.

Problemas como de controle de estoque pode ser considerado como problema dinâmico, principalmente no comércio eletrônico onde o processo de movimentação do estoque é mais rápido e podendo ter uma complexidade maior que estoques físicos.

Uma entrada de dados pode servir como base para o processamento realizado pelo programa, sendo decisiva para como o processamento deve ser realizado, e como pode depender de vários aspectos para que os processos sejam realizados, sua previsibilidade se torna fraca e o software necessita então ser capaz de se adaptar a eventos.

Alguns problemas como no caso dos softwares de rota para GPS precisam ser dinâmicos, pois seus cálculos podem ser influenciados a todo instante por fatores inesperados como acidentes, obras ou o simples caso de o motorista se equivocar nos comandos dados pelo software.

Na otimização, os estados do processo se referem ao que é necessário para que o software possa realizar seus processos adequadamente de acordo com a dinâmica dos eventos que podem ocorrer inesperadamente ou de forma previsível, observando impactos nos processos realizados e retornos gerados.

Algumas características que auxiliam na determinação dos estados são relativas a o método com o qual os estados são obtidos não deve afetar a tomada de decisão sobre como ocorrem as transmissões de dados nos estados.

A linguagem Clojure é uma linguagem que aceita programação dinâmica, e mesmo sendo tipada, durante o processo de compilação de seus códigos, os tipos não são verificados, e como as estruturas de dados em Clojure são listas geralmente, podendo ser manipuladas em tempo de execução pode-se interferir nas funcionalidades do código em tempo real.

Linguagens e o Mercado

A escolha pela primeira ou pela próxima linguagem de programação para aprender pode ser uma tarefa mais complexa que se imagina a princípio, pois além de existirem muitas linguagens, não é apenas a popularidade da linguagem que deve ser levada em consideração.

Mesmo linguagens antigas e utilizadas em sistemas legados como COBOL possuem mercado ainda para empresas governamentais com sistemas implantados há décadas e que ainda suportam as aplicações necessárias, pagando bons salários em função da oferta cada vez menor de profissionais capacitados nestas linguagens.

Outro caso bastante conhecido é o da linguagem Delphi que ainda possui inúmeros sistemas em operação pelo planeta, sendo confiáveis e suficientes para as demandas existentes, também com bons salários em virtude da mão de obra em redução também.

Estes sistemas em algum momento podem ser migrados para outras linguagens mais modernas e a demanda por estas linguagens praticamente desaparecer como ocorreu com algumas linguagens ao longo das décadas, e é importante levar em conta esta possibilidade também na escolha de uma linguagem a aprender.

Conhecer uma das dez linguagens mais populares do mercado é um bom negócio, pois possui grande oferta de vagas de trabalho em todo o planeta, mas não apenas estas linguagens são um bom investimento intelectual, pois além destas, outras possuem boa inserção no mercado ou estão em ascensão devido a novas tecnologias, mudanças no mercado ou surgimento de novas linguagens.

A questão é que existem muitas linguagens de programação disponíveis, umas voltadas à educação, outras ao controle de hardware em sistemas embarcados ou drivers, por exemplo, linguagens web, para desenvolvimento de aplicativos, linguagens de propósito geral e baseadas em um ou mais paradigmas.

De acordo com o IEEE Computer Society, existem sete linguagens de programação para se ficar de olho atualmente por diversos motivos, mas antes cita que é essencial que um bom desenvolvedor possua conhecimentos em uma linguagem de sistema como C ou C++, uma orientada a objetos como Java ou Python e uma para geração de scripts web como JavaScript. As atuais evoluções tecnológicas e da área de TI impulsionaram linguagens que antes eram irrelevantes, assim como novas linguagens foram sendo implementadas e obtiveram boa parcela no mercado.

Fonte: **Disponível aqui**

Linguagens de Programação Promissoras

Grandes empresas do mercado desenvolvem ou adquirem linguagens para que tenham a garantia de que estas linguagens recebam a devida manutenção e tenham suporte pelo tempo que for conveniente para estas empresas.

Um caso relevante ocorreu com a Apple que utilizava como padrão, a linguagem Object-C, própria para desenvolvimento de aplicativos para o sistema IOS da marca, mas com o tempo, resolveu investir no desenvolvimento de uma linguagem mais adequada às suas necessidades específicas e próprias para a plataforma, pois não existe grande preocupação da marca em relação à portabilidade de suas aplicações e ter uma linguagem sob total controle de suas necessidades.

Outra empresa que ofereceu ao mercado sua própria linguagem foi a Google, que disponibilizou a linguagem GO de propósito geral e com características interessantes como a possibilidade de retorno de vários valores de tipos diferentes ao término da execução de uma função, sendo está uma característica bastante incomum e muito interessante.

A Facebook também disponibilizou uma linguagem adequada aos interesses de desenvolvimento de software da empresa chamada Hack, baseada em desenvolvimento web, pode ser utilizada em conjunto com HTML e PHP, mas diferente da forma como essas duas linguagens podem se mesclar em um mesmo código, códigos Hack utilizam o paradigma orientado a objetos.

A Oracle, gigante do mercado de TI a décadas também agiu no mercado e adquiriu a Sun, agregando ao seu portifólio a linguagem Java, tendo assim controle de como a linguagem evoluirá daqui para frente, mas da forma como tem se comportado nas versões que surgiram depois da aquisição, Java deve continuar ainda como uma das preferidas do mercado.

Outra linguagem que pode ter impacto no mercado é Kotlin, muito utilizada no desenvolvimento de aplicativos Android, multiparadigma, aceita interação com Java e a uns anos foi oficializada pela própria Google como linguagem oficial para desenvolvimento Android.

A Microsoft, que teve como base a linguagem C desde o princípio, acompanhou sua evolução para C++, sendo base para todos os seus softwares praticamente, desde os sistemas operacionais desktops a videogames, e depois C# para desenvolvimento web usando sua IDE Visual Studio, realiza testes com uma linguagem bastante promissora chamada Rust, desenvolvida pela Mozilla Research, multiparadigma que possui mecanismos muito eficientes de controle de memória, ponto fraco da linguagem C que permite muita manipulação direta de memória, aumentando muito a quantidade de erros de memória que possam ocorrer.

Também busca uma identidade própria de programação com o desenvolvimento de uma nova linguagem chamada Bosque que pode ser interessante devido a busca em se obter uma linguagem mais simples e que possa trabalhar com as melhores características dos paradigmas, usando valores imutáveis, funções lambda, simplificação no uso de texto, e mecanismos mais avançados de estruturas de dados e processamento.

Linguagens Alternativas

Além das linguagens ligadas a planos de grandes empresas, existem várias outras linguagens que despertam o interesse em partes da comunidade de desenvolvimento de software em todo o planeta.

Para conhecer tantas linguagens de programação e perceber afinidade com algumas e incompatibilidade pessoal com outras, a melhor forma é buscar meios de testar códigos nestas linguagens e observar semântica, sintaxe, recursos disponíveis, bibliotecas implementadas e áreas de aplicação, pois uma linguagem de programação possui suas características, mas se elas servem no momento em que são criadas, só o mercado e a comunidade de desenvolvedores pode dizer com o tempo. Buscar meios para se testar códigos como a plataforma de testes REPL.IT que utiliza o navegador como base é interessante, ainda mais pelo fato deste site aceitar diversas linguagens de programação, tendo sido um dos locais utilizados como plataforma de teste dos códigos exemplo neste material.

Dart é outra linguagem desenvolvida pela Google para desenvolvimento web que permite uso de interpretação com máquina virtual ou compilação em formato JavaScript e possui já muitos adeptos desenvolvedores em função de suas qualidades.

Imagem 1

```
int timesTwo(int x) {
     return x * 2;
}
int timesFour(int x) => timesTwo(timesTwo(x));
int runTwice(int x, Function f) {
     for (var i = 0; i < 2; i++) {
          x = f(x);
     }
     return x;
}
main() {
     print("4 times two is ${timesTwo(4)}");
     print("4 times four is ${timesFour(4)}");
     print("2 x 2 x 2 is ${runTwice(2, timesTwo)}");
}
```

Fonte: Dart.

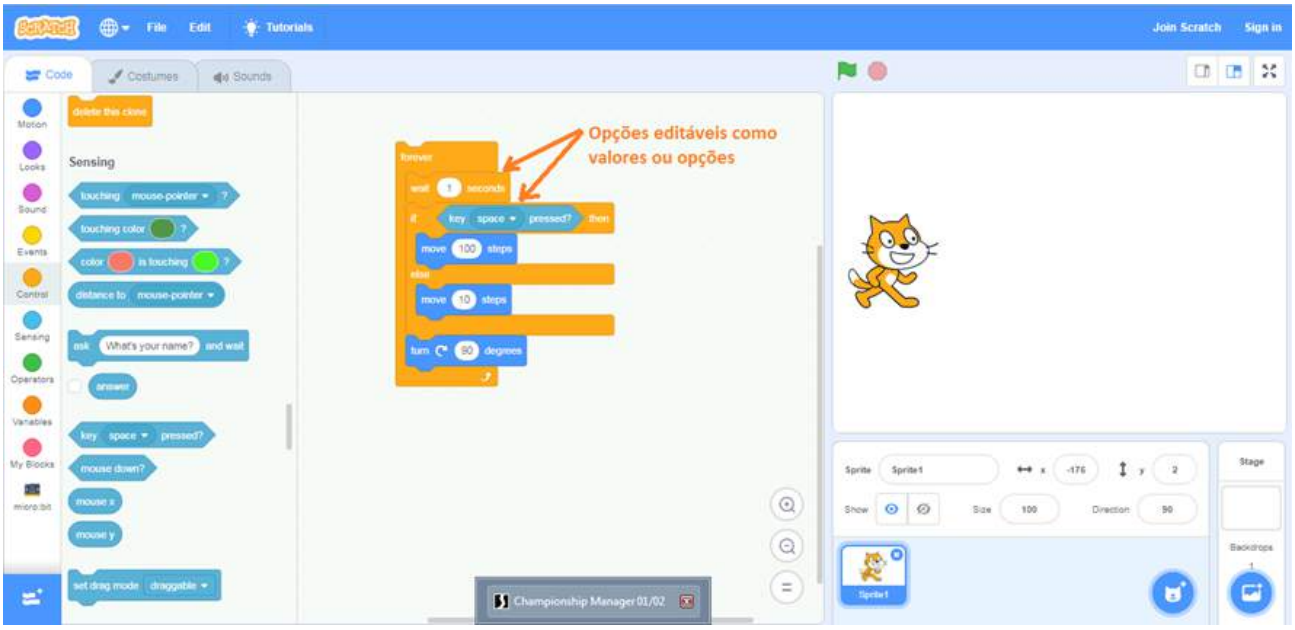
Partindo do exemplo da imagem 1, pode-se observar como as linguagens de programação que surgem possuem similaridades com outras talvez conhecidas já e oferecem uma curva rápida de aprendizado da estrutura e conceitos básicos.

Nesse exemplo, é fácil perceber semelhanças com a linguagem C pelo uso de uma função main() que inicia a aplicação e comandos conhecidos como for, print, uso de delimitadores chaves e aspas, finalização de comandos com ponto e vírgula e declaração de funções tipadas e com parâmetros também tipados.

Assim, observando exemplos de códigos prontos de várias linguagens, é possível com o tempo estabelecer relações entre diferentes linguagens e traçar ligações entre aqueles que originaram outras.

Linguagens como Scratch e APP Inventor desenvolvidas no MIT (Massachusetts Institute of Technology ou Instituto de Tecnologia de Massachusetts), Blocky desenvolvido pela Google que utiliza programação em blocos como base que pode ser convertida para códigos em linguagens como JavaScript, Python, PHP, Lua e Dart, e Swift Playgrounds desenvolvido pela Apple, são exemplos de linguagens de programação por blocos que são opções bastante intuitivas e simplificadas de alto nível, ideais para introdução da lógica de programação para iniciantes de quaisquer idades, trabalham com belas interfaces de desenvolvimento como se pode ver na IDE utilizada pela Scratch (imagem 2) e APP Inventor (imagem 3).

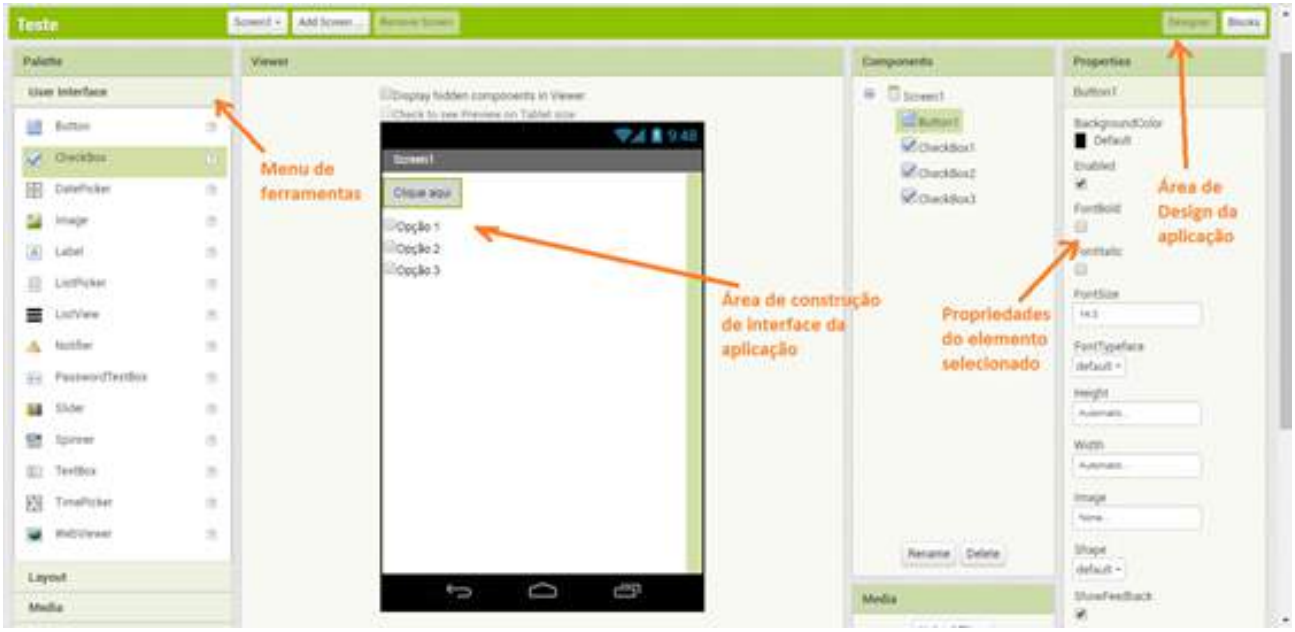
Imagem 2



Fonte: O autor.

O APP Inventor possui uma proposta bastante moderna e completa em que se pode conter uma plataforma baseada em uma IDE para código em blocos e ferramentas de construção visual de aplicativos para construção de aplicativos e formulários como opções de interação com usuários (botões, caixas de seleção), opções de layout da aplicação, elementos multimídia (sons, câmera), sensores típicos em smartphones.

Imagem 3



Existem linguagens com muitos atributos positivos como Scala que compartilha a máquina virtual Java e por isso, é compatível com as bibliotecas Java, sendo bastante eficiente, ou como R que é uma linguagem multiparadigma voltada à análise de dados, tendo forte uso estatístico e consequentemente em ciência de dados.

PARA GABARITAR

Para conhecer novas propostas de linguagens de programação é importante acompanhar as notícias relacionadas ao tema em blogs, canais e notícias relacionadas à área, pois os movimentos ocorrem de maneira imprevisível muitas vezes e apenas com um acompanhamento das notícias é possível estar por dentro das linguagens mais elogiadas, em evolução no mercado, e caindo em desuso.

Conclusão

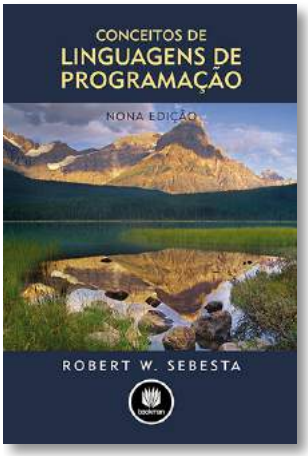
Conhecer os diferentes paradigmas e linguagens de programação existentes de forma simplificada e objetiva auxilia em tomadas de decisão com relação ao que estudar e o que utilizar para resolver problemas, mas não traz a solução de todos os problemas.

A busca por estudos adicionais é importante e a proposta do material foi e auxiliar no processo de escolha do que procurar e estudar, além de oferecer recursos para poder interpretar códigos em diferentes paradigmas e até em diferentes linguagens.

Alguns paradigmas parecem sempre mais assustadores que outros, mas a ideia é a mesma dos idiomas que são utilizados para comunicação, onde quem nasce no Brasil e tem como sua língua nativa, o português, assim como nascidos na Rússia falam russo.

A linguagem que se aprende primeiro é a porta de entrada para várias outras, mas pode ser preocupante em outras, em função de mudanças de paradigmas e estruturas de programação, pois quando se conhece alguma linguagem, usa-se para compreender a outra, mas isto pode não ser o melhor método, e talvez seja melhor ter a mente “limpa” para se iniciar os estudos de uma linguagem de programação.

Material Complementar



Livro

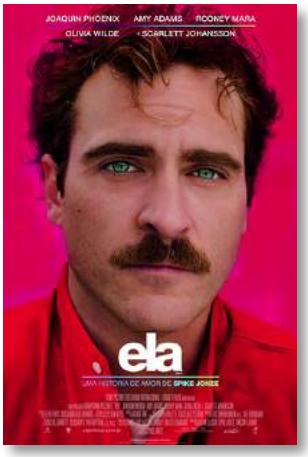
Conceitos de linguagens de programação

Autor: Robert W. Sebesta.

Editora: Bookman

Sinopse: Livro referência da área contendo conteúdo de todos os principais paradigmas e mecanismos de programação disponíveis em diversas linguagens de programação existentes, sendo uma boa fonte de referência no conhecimento das linguagens.

Comentário: Livro bastante completo e que permite uma boa compreensão dos diferentes paradigmas de programação.



Filme

Ela

Ano: 2013

Sinopse: Um homem solitário resolve adquirir um novo sistema operacional bastante complexo que acaba cativando-o tanto que ele se envolve emocionalmente com o mesmo.

Comentário: Filme reflexivo sobre a evolução do desenvolvimento de software para inteligência artificial.

Web

Todo mundo deveria aprender a programar

Vídeo bastante interessante com relatos de alguns das maiores personalidades da área dentre outros que comentam a importância da programação e como ela pode ser importante na educação desde a infância.

Acesse o link

Referências

BEZERRA, Cicero Aparecido. **Introdução à linguagem de programação Delphi.** Universidade Federal do Paraná. Curitiba-PR, 2018. Disponível em https://acervodigital.ufpr.br/. Acesso em: 15 ago. 2020.

Blockly. Disponível em: https://developers.google.com/blockly/. Acesso em: 20 de jan. 2019.

CAFFÉ, Leandro. **O que é a Programação Orientada a Eventos?.** Stackoverflow. 2015. https://pt.stackoverflow.com/. Acesso em: 20 jan. 2019.

Dart. Disponível em https://dart.dev/. Acesso em: 30 ago. 2020.

EMERICK Chas; CARPER Brian; GRAND Christophe. **Clojure Programming. Sebastopol**, CA : O'Reilly Media, 2012.

LADDAD, Ramnivas. **AspectJ in Action** - practical aspect-oriented programming. Greenwich: Manning, 2003.

MANZANO, J. A. N. G. **Java 7:** programação de computadores: guia prático de introdução, orientação e desenvolvimento. São Paulo: Érica, 2011.

MOWFORTH, Chris. **Dynamic Programming + Clojure**. Disponível em: https://speakerdeck.com/m0wfo/dynamic-programming-plus-clojure?slide=17. Acesso em: 30 jan. 2019.

Swift Playgrounds. **Apple**. Disponível em: https://www.apple.com/. Acesso em: 20 jan. 2019.

System. Windows. Forms Namespace. Microsoft. Disponível em: https://docs.microsoft.com/. Acesso em: 20 jan. 2019.

SEBESTA, Robert W. **Conceitos de linguagens de programação** [recurso eletrônico] / Robert W. Sebesta; tradução técnica: Eduardo Kessler Piveta. – 9. ed. – Dados eletrônicos. – Porto Alegre: Bookman, 2011.

WANGENHEIM, Aldo Von; ABDALA, Daniel Duarte. **Conhecendo o Smalltalk** - Todos os Detalhes da Melhor Linguagem de Programação Orientada a Objetos. Visual Books, 2002.