PEF – 5743 – Computação Gráfica Aplicada à Engenharia de Estruturas

Prof. Dr. Rodrigo Provasi

e-mail: provasi@usp.br

Sala 09 – LEM – Prédio de Engenharia Civil

Projeto de *Software – Parte II*

- Métodos de descrição de *software – Unified Modeling Language (UML)*
- *Design Patterns*
- Modelos em Camadas

Introdução

- Viu-se na parte anterior maneiras de organizar a criação de um código quando está em um grupo de pessoas.
- Isso mostra-se fundamental para que o trabalho progrida e que o esforço de cada um não colida, respeitando o tipo de cada projeto.
- Nessa seção serão mostrados maneiras para descrever o projeto para que ele seja executado da melhor forma por essas equipes.

Métodos de Descrição de *Software*

- Descreve o funcionamento global, mantendo uma linguagem abstrata.
- Metodologias de projeto influem nas escolhas dos métodos de descrição.
- Possíveis abordagens:
  - Baseada em cenários
  - Análises orientadas a objetos
  - Análise orientada ao fluxo
  - Baseada em classes
  - Modelo comportamental

Unified Modeling Language

- Os métodos citados fazem parte da UML.
- A UML é um conjunto de diretrizes que definem diversas formas de descrever o funcionamento de um *software* através de textos e diagramas de alto nível.
- O que se pretende é ter uma descrição tal que qualquer pessoa possa compreender o funcionamento, que ajude quando da codificação e que, nessa etapa, permita modificações sem grandes esforços.

Casos de Uso (*Use Cases*)

- Modelagem baseada em cenários.
- Os casos de uso são definidos em BOOCH, RUMBAUGH e JACOBSON (2005) como "uma descrição para um conjunto de ações, incluindo variações, que o sistema executa para obter um resultado de um valor observável pelo ator. Graficamente, um caso de uso é uma elipse".
- Os seus elementos serão mostrados a seguir.

Casos de Uso (*Use Cases*)

- **Nomes:** cada caso de uso deve possuir um nome que o distingue dos outros casos de uso. Um nome é uma ou poucas palavras que o caracterizem. É possível qualificar um caso de uso, especificando o pacote.

Casos de Uso (*Use Cases*)

- **Atores:** um ator representa um conjunto de papéis que o usuário pode ter quando interage com esses casos de uso. Em geral representa um ser humano, mas também pode representar um equipamento ou algum outro sistema de atua sobre o sistema.

Casos de Uso (*Use Cases*)

- Existem ainda possíveis relações entre os elementos anteriormente citados, como especializações (por exemplo, um tipo de ator como Empregado, que derive de um ator Pessoa, especializando esse último), dependências e relações (como quando um caso de uso se utiliza ou depende de outro).

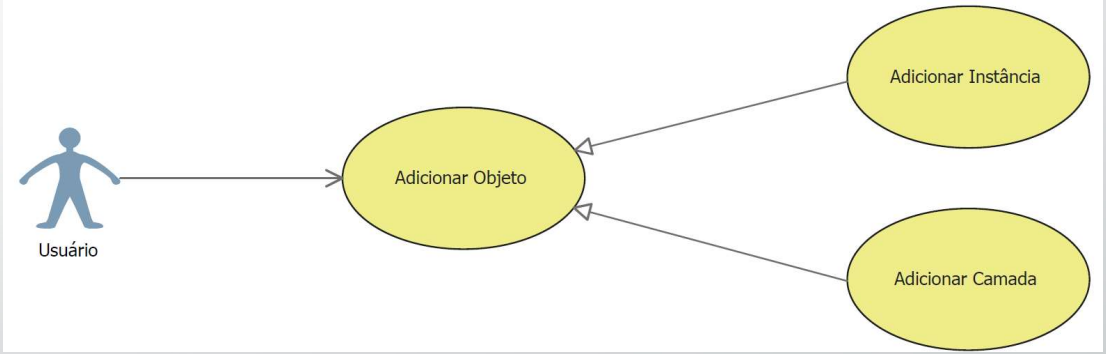


Diagrama de Atividades

- Modelagem orientada ao fluxo.
- O diagrama de atividades é definido em BOOCH, RUMBAUGH e JACOBSON (2005) como sendo "essencialmente um diagrama de fluxo, mostrando o fluxo de controle de atividade para atividade. Diferentemente de um diagrama de fluxo convencional, um diagrama de atividade mostra tanto concorrência quanto ramos de controle".

Diagrama de Atividades

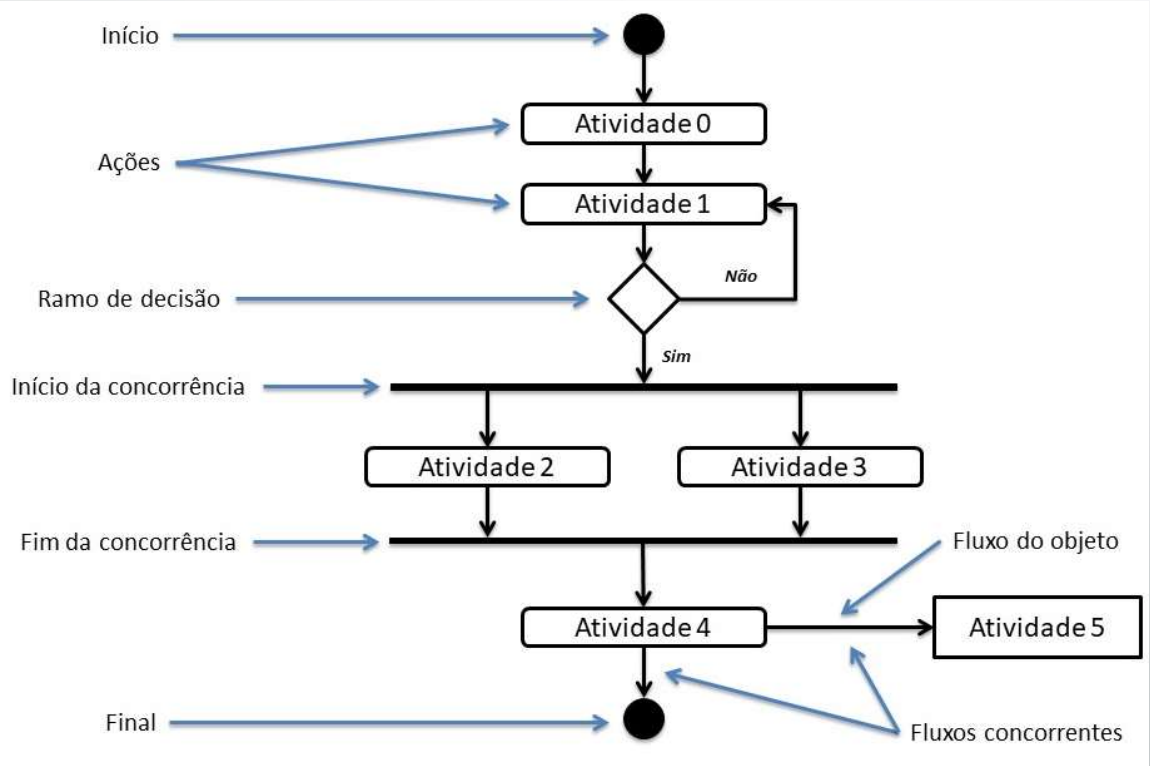
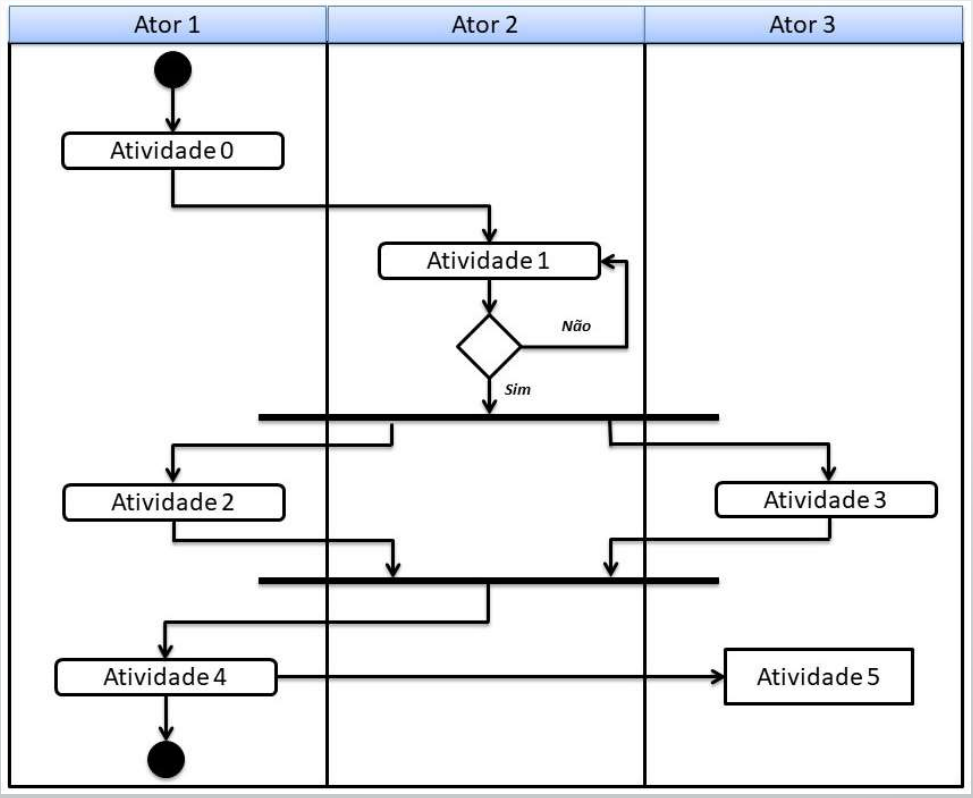


Diagrama de Raias (*Swimlane*)

- Uma variação do diagrama de atividades é o diagrama de raias (Swinlane), na qual cada atividade está separada por grupos. A figura exibe um diagrama de raias esquemático, no qual existem três atores, cada qual com determinadas atividades. Esse exemplo utilizou-se de atores, pois esses são os executores da tarefa. Nos casos práticos, essas indicam a organização responsável por aquela tarefa (seja ela um ator ou não).



Diagramas de Fluxo de Dados – *DFD*

- Diagramas de fluxo de dados (DFD ou *Data Flow Diagrams* em inglês) focam nas modificações sobre as entradas do sistema e quais operações devem ser executadas para que ocorram essas alterações. Esses diagramas podem exibir o software como o todo ou podem mostrar parte do sistema, permitindo uma maior compreensão sobre o mecanismo de funcionamento desses em projeto.
- Elementos:

Diagrama de Fluxo de Dados -DFD

- São divididos em níveis, sendo o zero o mais alto do sistema.
- Em cada nível menor, uma ação é 'explodida' em ações que ocorrem dentro da ação.

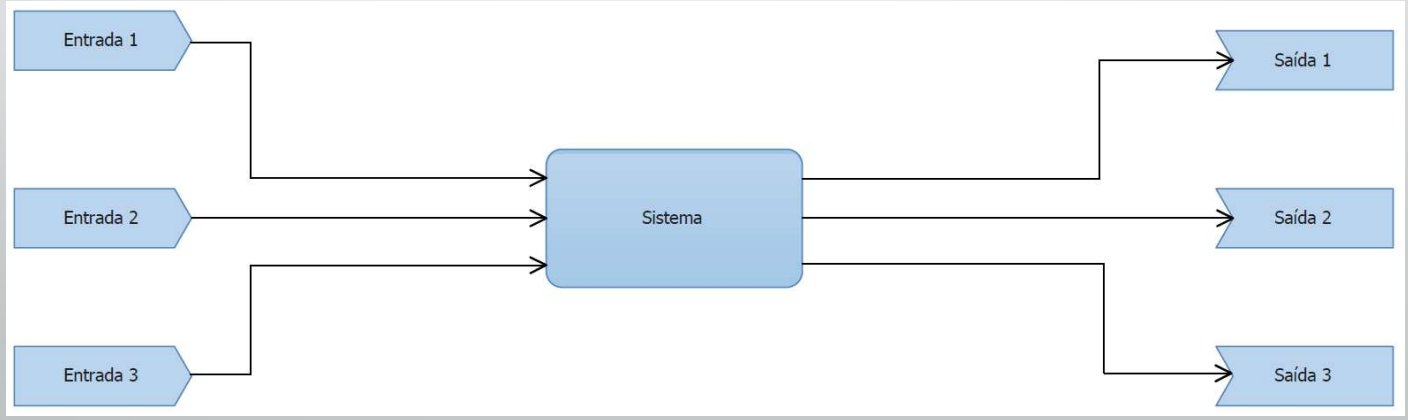


Diagrama de Fluxo de Dados -DFD

- Nível 1: Sistema decomposto em ações e banco de dados

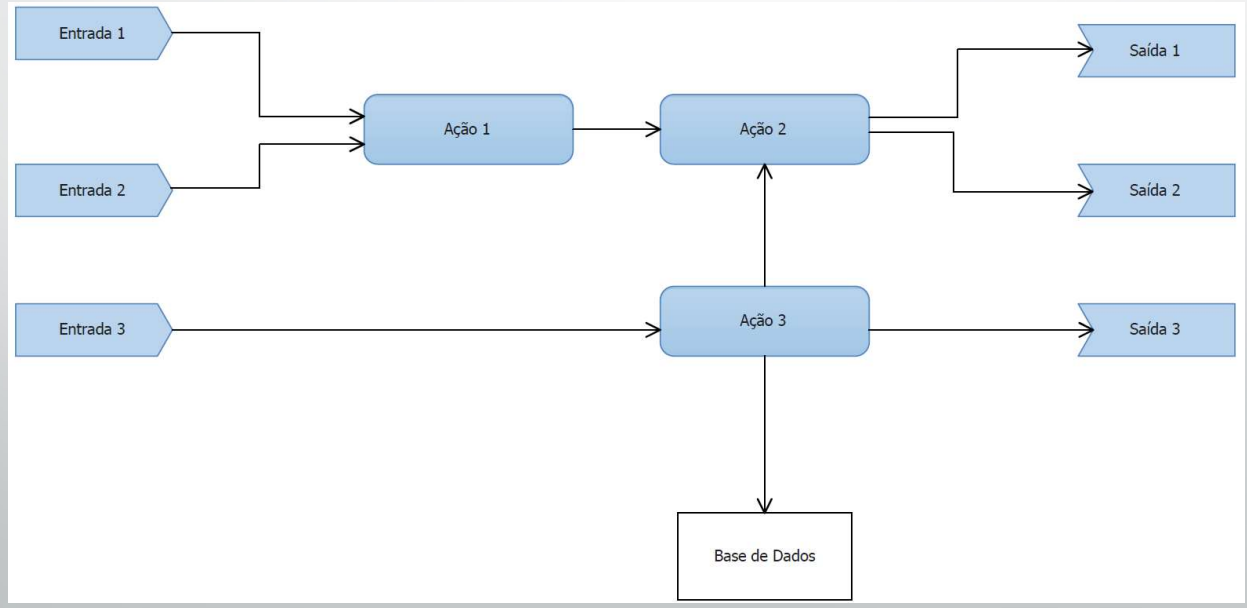


Diagrama de Fluxo de Dados – DFD

- O refinamento dos diagramas de fluxo de dados é feito até que a estrutura do programa fique compreensível para todas as partes envolvidas com o projeto do *software*, desde clientes até projetistas e desenvolvedores.
- Embora os diagramas de fluxo sejam suficientes para a maior parte dos projetos de *software*, existem alguns que não são direcionados pelo fluxo de dados, mas sim pelos eventos neles ocorridos. Para cumprir tal objetivo, utilizam-se os diagramas de estado.

Modelos baseados em classes

- O modelo baseado em classes busca identificar as classes, definir e especificar métodos, atributos e operações para a construção dos diagramas de classe.
- Esse tipo de modelagem está mais ligado à codificação, sendo em geral complementar às demais e que serve de ponto de partida para a implementação do programa.

Modelos baseados em classes

- Em geral, como observado em PRESSMAN (2005), as classes surgem de:
  - Entidades externas que produzam ou consumam informação do sistema.
  - Algo que faça parte do domínio do sistema.
  - Ocorrências ou eventos que ocorram no contexto de operação do sistema.
  - Papéis executados por pessoas que interagem com o sistema.
  - Unidades organizacionais relevantes para a aplicação.
  - Lugares que estabeleçam o contexto do problema e a função geral do sistema.
  - Estruturas que definam a classes de objetos ou classes de objetos relacionadas.

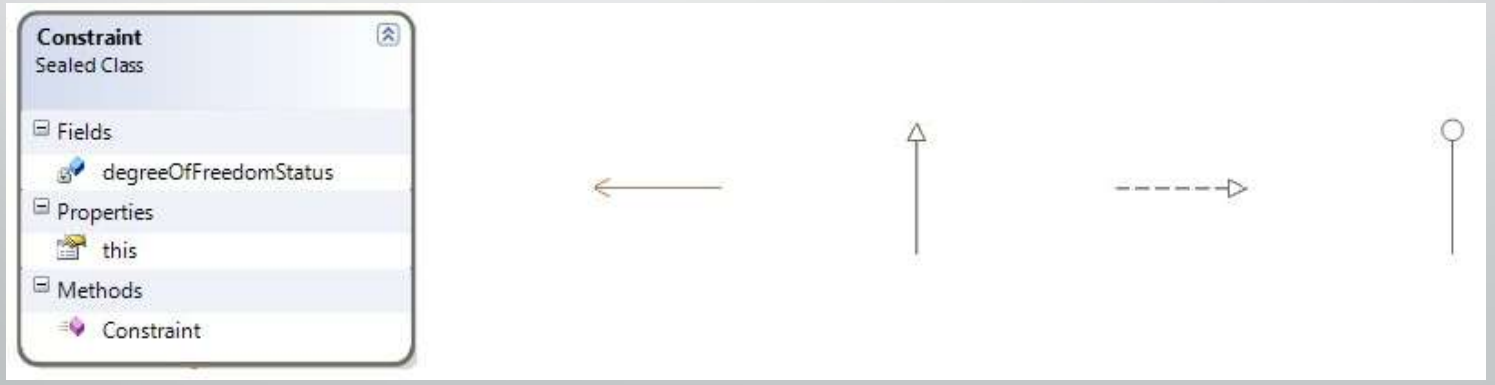
CRC – *Class-Responsability-Collaborator*

- O modelo CRC (Classe Responsabilidade Colaborador ou em inglês *Class-Responsability-Collaborator*) é descrito em PRESSMAN (2005) como, "uma coleção de cartões indexados que representam classes. Os cartões são divididos em três seções. Na parte superior do cartão escreve-se o nome da classe. No corpo do cartão listam-se as responsabilidades do lado esquerdo e as colaborações do lado direito".

Classe	
Descrição	
Responsabilidades	***Colaborador***
Nessa coluna são colocadas as	Nessa coluna são colocados os colaboradores
responsabilidades da classe.	para cada responsabilidade da coluna ao lado.

Diagrama de Classe

- Os Diagramas de Classe são confeccionados uma vez que a definição das classes é concluída. O uso deste não é restrito à modelagem baseada em classes, sendo que, quando um *software* começa a ser confeccionado, criam-se diversos modelos e por último, parte-se para a confecção dos diagramas, que resultarão nas classes do *software* final.
- Elementos:

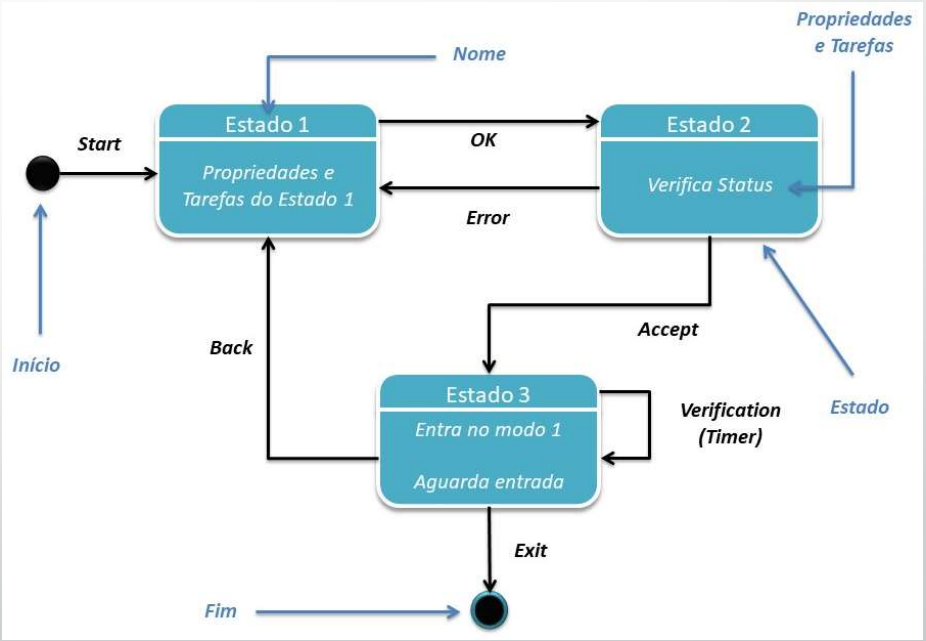


Modelagem comportamental

- A modelagem comportamental procura descrever como o software se comporta dinamicamente, uma vez que os modelos como os de classes anteriormente descritos mostram o funcionamento estático do mesmo. Busca-se identificar características que representem os eventos ocorridos no sistema. Essas podem ser observadas inicialmente nos casos de uso, sendo necessária uma busca dos possíveis eventos neles contidos.
- Uma vez encontrados os eventos, confeccionam-se diagramas de estados ou utilizam-se diagramas de sequência.

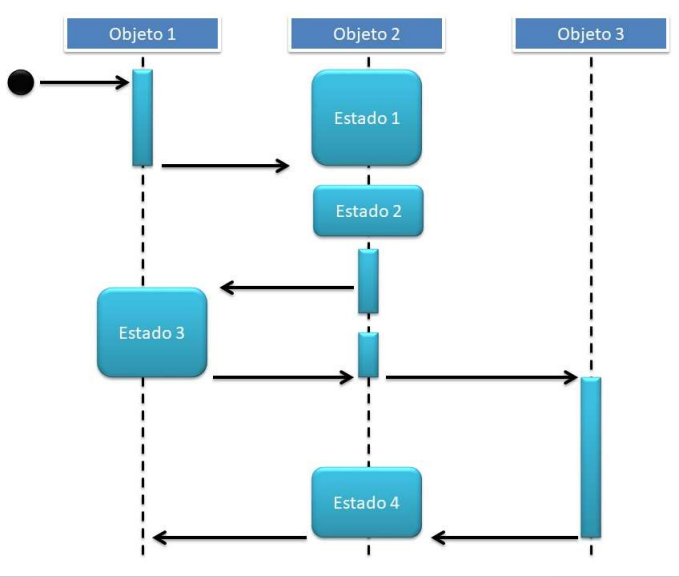
Diagrama de Estados

- Como definido em BOOCH, RUMBAUGH e JACOBSON (2005), "um diagrama de estado mostra uma máquina de estados. Enquanto um diagrama de atividades mostra o fluxo de controle de atividade em atividade, um diagrama de estado mostra o controle de fluxo de estado para estado de um único objeto".



Diagramas de Sequencia

- Os diagramas de sequência indicam como eventos geram mudanças de objeto para objeto, sendo essencialmente uma representação de como os eventos fluem de um objeto para outro em função do tempo.
- Nesse diagrama cada linha tracejada indica um objeto, enquanto os quadrados indicam o tempo de permanência processando uma atividade e os quadrados de cantos arredondados os estados.
- As setas servem para indicar a mudança do controle de informação de um objeto para outro.



Modelo de Camadas

- Os modelos apresentados cada vez mais descem nos níveis hierárquicos e fica mais próximo da codificação.
- Um *software* normalmente tem seu desenvolvimento dividido em módulos para facilitar sua manutenção e entendimento. Pode-se fazer outra de caráter mais específico: de acordo com sua função. Esse modelo de divisão é conhecido como modelo em *n*-camadas.
- As camadas são feitas para ser possível a distribuição dessas em diversos computadores de uma rede distribuída. Isso não necessariamente implica que o software será distribuído, mas esse é de onde advém o conceito.

Modelo de Camadas

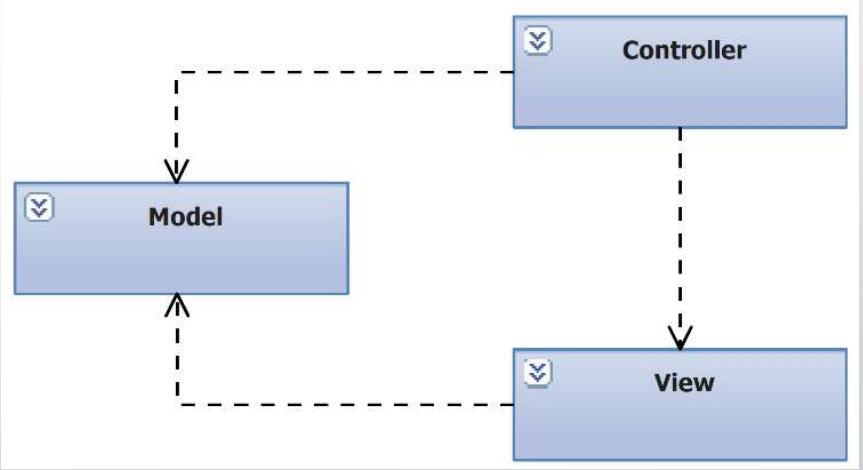
- Dos modelos de *n*-camadas, o mais utilizado é o de 3-camadas, sendo essas a interface com o usuário, a lógica de negócio e o banco de dados.
- Na primeira delas, toda a lógica de interação com o usuário, bem como as requisições são recebidas.
- Na camada de negócios os dados são obtidos da camada de banco de dados, as requisições são tratadas e seus resultados são enviados à camada de interface.
- Na camada de banco de dados fica o repositório do aplicativo e todas as manipulações inerentes ao banco.
- Normalmente os outros modelos que possuem menos camadas concatenam alguma dessas numa mesma camada, ao passo que os que apresentam mais camadas dividem essas camadas em algumas mais especializadas.

Modelo de Camadas

- *Modelos de camadas:*
  - *MVC - Model-View-Controller*
  - *MVP - Model-View-Presenter*
  - *MVVM - Model-View-ViewModel.*

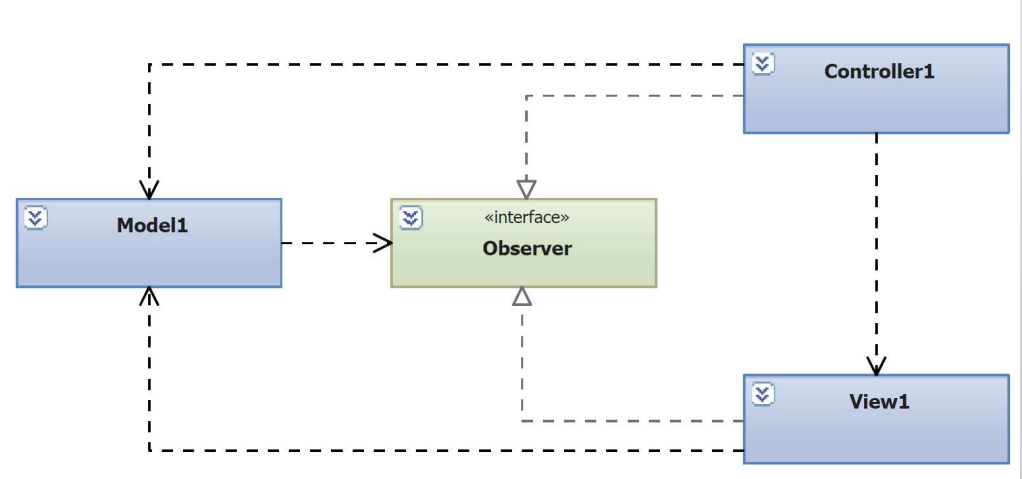
MVC - Model-View-Controller

- O modelo MVC é um dos modelos de três camadas mais utilizado.
- Esse padrão propõe a divisão da aplicação numa camada de dados (o *Model*), uma camada de interface (o *View*) e uma camada para tratar das ações de entrada (teclado, mouse, etc) e fazer a ligação entre modelo e interface (o *Controller*).



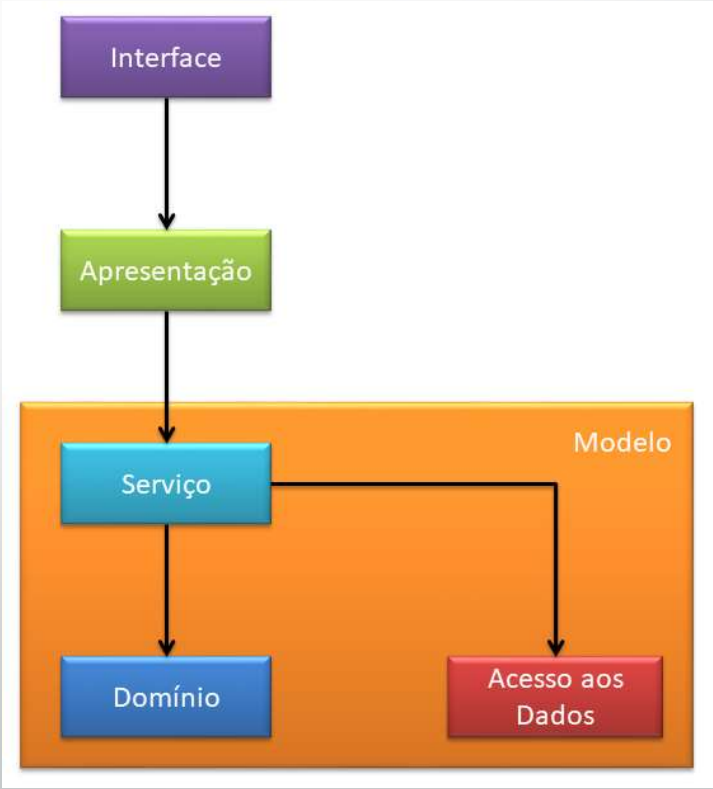
MVC - Model-View-Controller

- Como dito em Microsoft (2010), nota-se que tanto o *Controller* quanto o *View* dependem do *Model*, embora esse não dependa dos outros.
- Isso propicia uma separação que permite que o modelo seja implementado e testado separadamente, sem interferir no projeto da interface e do controlador. Apesar disso, há uma grande dependência entre *Controller* e *View*. Para minimizar esse fato, propõe-se a utilização de um padrão de projeto do tipo *Observer*.



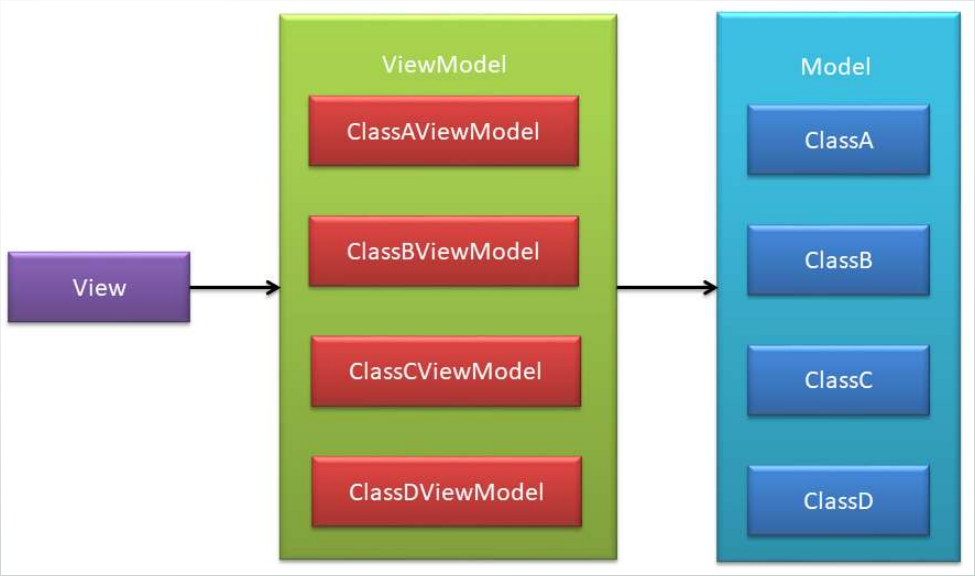
MVP – Model-View-Presenter

- Esse padrão é, de certa forma, uma modificação do MVC, porém ele prevê uma separação completa entre interface e modelo.
- Nesse padrão a interface trata os eventos de mouse e teclado, mas toda a comunicação com o modelo fica a cargo do *Presenter*. Isso desacopla os módulos de interface e modelo, que podem ser feitos separadamente. O ponto que demanda maior atenção é justamente o módulo que faz a ligação entre *Model* e *View*: o *Presenter*.



MVVM – Model-View-ViewModel

- O padrão *MVVM* é um modelo mais novo.Cria-se uma abstração que liga o modelo à interface: o *ViewModel*. Essa ligação é de alguma maneira equivalente ao *Presenter* do *MVP*. A diferença aqui é que para cada classe do modelo há um *ViewModel* correspondente.
- Um fato que é destacado na referência é que o padrão é extremamente comum quando se utiliza *Windows Presentation Foundation* para a confecção das classes por propiciar um acoplamento fraco entre *View* e *ViewModel* através de vínculos conhecidos como *Binding*.



Padrões de Projeto (*Design Patterns*)

- Padrões de projeto são soluções gerais para problemas que ocorrem no desenvolvimento de um *software*.
- Um padrão é uma solução que não é acabada, mas que pode ser transformada diretamente em código, ou seja, uma descrição de como resolver de maneira eficiente o problema encontrado.
- Eles são divididos em três categorias:
  - os de criação, que estão relacionados à instanciação de novos objetos.
  - os estruturais, que procuram criar estruturas entre classes e objetos.
  - os de comportamento, que tratam da comunicação entre os elementos.

Padrões de Projeto Criacionais

- Nessa categoria estão os padrões relacionados à criação dos objetos. Estes ainda podem ser divididos nos que são responsáveis pela criação classes (que se baseiam no conceito de herança) e nos que são responsáveis pela instanciação dos objetos (que se baseiam em delegar funções).
- São eles:
  - ***Factory Method:*** define uma interface para a criação do objeto, mas deixa a cargo das subclasses decidir qual a classe a ser instanciada. Esse padrão permite que o código seja flexível, facilitando a inclusão de novos elementos ao programa sem que haja grandes modificações no código já existente.
  - ***Abstract Factory:*** cria instâncias de famílias de classes relacionadas ou dependentes sem especificar a classe concreta. Esse padrão é similar ao *Factory*, mas leva em conta o relacionamento da família de classes a ser criada.

Padrões de Projeto Criacionais

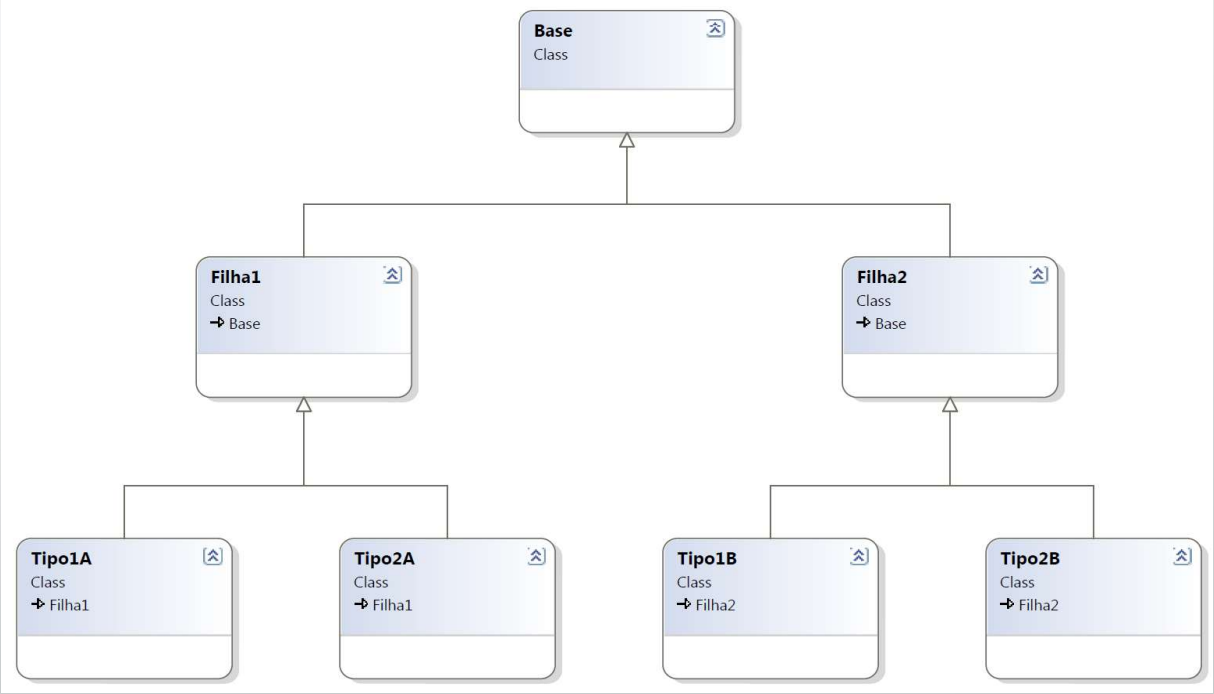
- ***Builder:*** separa a construção do objeto de sua representação. Quando a construção de um objeto é complexa e pode variar (ou seja, ter como parâmetro um objeto abstrato) esse tipo de padrão facilita o processo de instanciação.
- ***Object Pool:*** cria um pool para gerenciar a vida de um objeto. Aqui monitora-se sua vida e libera os recursos por ele utilizados quando não está mais em uso.
- ***Prototype:*** cria protótipos de classe. Para criar novos objetos, utiliza-se cópias do protótipo. Esse padrão cria um tipo básico a ser instanciado nas classes filhas utilizando parâmetros definidos no protótipo.
- ***Singleton:*** define uma classe que deve ter apenas uma instância. Esse tipo de padrão é útil quando se deseja que a durante a execução do *software* apenas uma instância da classe exista.

Padrões de Projeto Estruturais

- Os padrões de projeto estruturais utilizam-se de herança para compor interfaces e criar as estruturas. Eles definem maneiras de composição de objetos para obter-se novas funcionalidades.
- São:
  - ***Adapter:*** cria uma interface de uma classe para ser utilizada em outro cliente, tornando classes compatíveis. Esse tipo de padrão também pode ser visto como um *wrapper*, ou seja, ele expõe estruturas de um conjunto de classes para que outras (ou outro programa, pacote, etc.) possam utilizá-las.

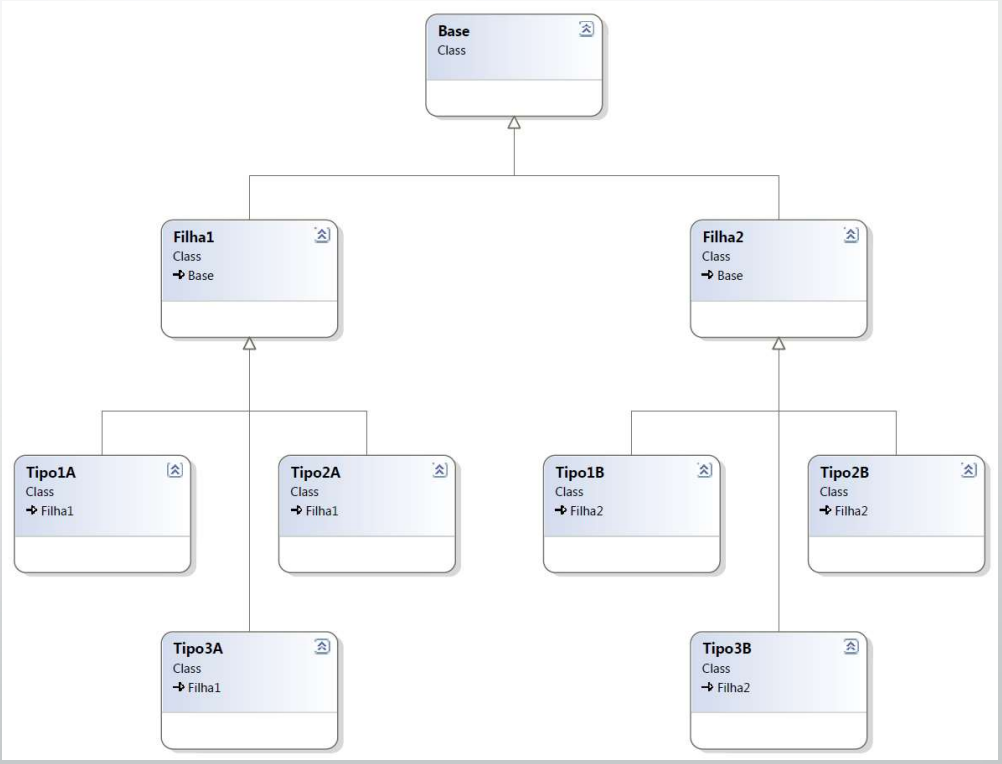
Padrões de Projeto Estruturais

- ***Brigde:*** desacopla uma abstração de sua implementação para que as duas possam ser tratadas independentemente. Nesse caso, o padrão fornece uma alternativa para classes abstratas que também possuem classes abstratas.



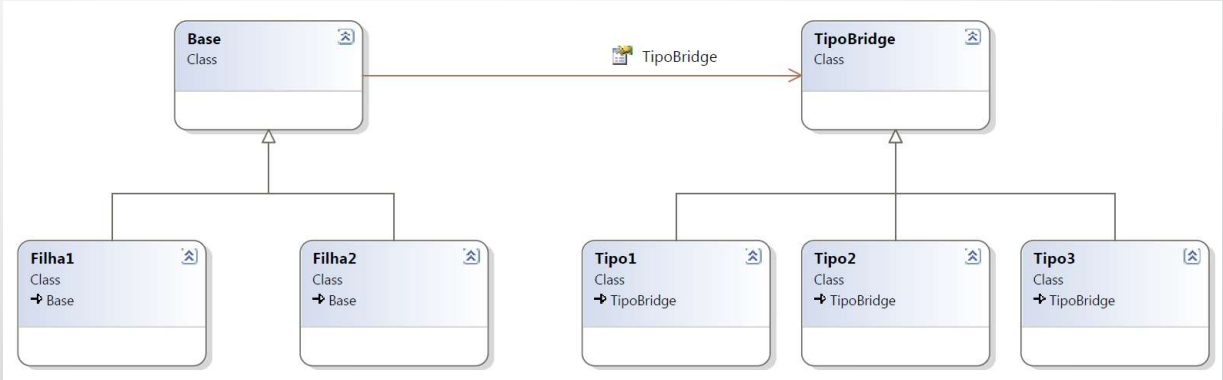
Padrões de Projeto Estruturais

- ***Brigde:*** desacopla uma abstração de sua implementação para que as duas possam ser tratadas independentemente. Nesse caso, o padrão fornece uma alternativa para classes abstratas que também possuem classes abstratas.



Padrões de Projeto Estruturais

- ***Brigde:*** desacopla uma abstração de sua implementação para que as duas possam ser tratadas independentemente. Nesse caso, o padrão fornece uma alternativa para classes abstratas que também possuem classes abstratas.



Padrões de Projeto Estruturais

- ***Composite:*** cria objetos em estruturas de árvores para representar a hierarquia completa. Esse padrão permite tratar objetos individuais e compostos uniformemente.
- ***Decorator:*** adiciona recursos à estrutura da classe para que essa permita que funcionalidades sejam adicionadas dinamicamente. O processo de obtenção desse dinamismo é dado pela inclusão de interfaces e classes abstratas de mesma raiz, formando lista de diversas camadas com as funcionalidades desejadas.
- ***Facade:*** proporciona uma interface unificada para um subsistema. O sistema por trás da classe de interface não precisa ser conhecido, desde que a interface faça a comunicação e tratamento necessários. Esse padrão funciona como uma interface de alto nível que facilita a utilização do subsistema.

Padrões de Projeto Estruturais

- ***Flyweight:*** faz uso de estruturas de compartilhamento para suportar vários objetos pequenos de forma eficiente.
- ***Private Class Data:*** controla o acesso à gravação dos atributos da classe, separando os dados dos métodos que o usa. Esse tipo de padrão faz com que os pontos em que os dados são modificados fiquem concentrados apenas nas classes.
- ***Proxy:*** cria uma classe que reserva o espaço ocupado por um objeto pesado, procurando inicializá-lo somente quando o necessário. De certa forma, impõe uma camada extra para o acesso do objeto, protegendo-o, além de permitir maior controle e melhor acesso.

Padrões de Projeto Comportamentais

- Os padrões de projeto comportamentais estão ligados à comunicação entre os objetos. São:
  - ***Chain of responsability:*** esse padrão propõe a troca de uma estrutura de comunicação do tipo *bus* onde emissor, receptor e interessados estão conectados à linha principal por uma linha do tipo *pipeline*, sendo que os interessados estão colocados sequencialmente.
  - ***Command:*** esse padrão encapsula uma ordem feita a um objeto, facilitando seu tratamento. Ele se mostra especialmente útil para geração de filas e arquivos de log, uma vez que o processamento dos pedidos é padronizado, independente do objeto.

Padrões de Projeto Comportamentais

- ***Interpreter:*** apresenta uma solução para tratar da interpretação de linguagens. Esse padrão propõe uma estrutura que armazena essa linguagem (em forma de árvore), bem como o seu interpretador. Dessa maneira, a organização e composição de mensagens complexas fica facilitado.
- ***Iterator:*** permite acessar sequencialmente itens de uma coleção. Esse padrão propõe uma estrutura para gerar esse iterador, sem que as estruturas necessárias sejam expostas. Algumas linguagens de programação ainda oferecem elementos que facilitam a implementação.
- ***Mediator:*** esse padrão cria uma forma de comunicação entre grupos de objetos, deixando as regras dessa comunicação apenas no mediador. Esse acoplamento leve aumenta a independência do código.

Padrões de Projeto Comportamentais

- ***Memento:*** adiciona uma funcionalidade que permite reverter o objeto a um estado anterior. Ele é aplicável quando há a necessidade de se desfazer mudanças que porventura o objeto tenha sofrido.
- ***Null Object:*** cria uma maneira de adicionar um objeto nulo ao projeto. Esse padrão se mostra útil quando o objeto em questão requer uma ação de uma classe colaboradora ou quando se deseja que esse objeto não execute nenhuma ação, num cenário de que a criação do objeto implica em uma ação.
- ***Observer:*** permite que mudanças no estado do objeto sejam notificadas e alguma ação seja tomada mediante essas modificações. Ele funciona definindo que uma mudança no objeto seja notificada a todos os objetos que dependem desse e faz com que sejam atualizados automaticamente.

Padrões de Projeto Comportamentais

- ***State:*** permite um objeto modificar seu comportamento quando seu estado muda. Esse padrão funciona com base em diferentes estados que uma classe pode assumir e se comportar.
- ***Strategy:*** esse padrão de projeto encapsula algoritmos permitindo que possam ser permutáveis na execução. A classe responsável pela estratégia define qual o algoritmo a ser utilizado pelo cliente.
- ***Template Method:*** implementa um conjunto de passos a ser utilizado em um algoritmo, permitindo que classes derivadas modifiquem alguns dos passos. Isso permite que, de acordo com a aplicação, o algoritmo adequado seja executado.
- ***Visitor:*** esse padrão permite que uma operação seja feita sobre elementos de uma estrutura sem que haja alteração na classe de elementos em que atuam.

Boas práticas

- Não basta utilizar os métodos de descrição de projeto e procurar incorporar *Design Patterns*. O código resultante precisa ser bem trabalhado afim de evitar problemas, facilitar a leitura e a manutenção. Algumas práticas são recomendáveis para melhorar esses problemas:

Boas práticas

- Escreva os programas de maneira bem simples e direta (sem complicar)
- Leia documentos a respeito da linguagem que estiver usando
- Faça pequenos programas para testar os comandos da linguagem
- Leia e estude cada mensagem de erro emitida pelo compilador
- Comece todo programa com um comentário descrevendo a finalidade do mesmo
- Coloque nas funções, que imprimem alguma mensagem

Boas práticas

- Recue o corpo de uma função (endentação), fazendo com que a mesma fique mais legível
- Estabeleça um critério de espaços nos recuos (Padrão C#: 4 espaços)
- Declare cada variável em uma linha para facilitar a escrita de comentários
- Coloque um espaço após cada vírgula (ou ponto e vírgula) na estrutura *for*
- Escolha nomes significativos para as variáveis
- Comece o nome dos identificadores com letra MAIÚSCULA, e, da mesma forma, cada palavra das variáveis com nomes compostos

Boas práticas

- Declare as variáveis sempre no início das funções
- Coloque sempre uma linha em branco entre a declaração de variáveis e os comandos seguintes
- Coloque espaços dos dois lados dos operadores aritméticos
- Escreva parênteses, ainda que redundantes, para facilitar o entendimento de expressões aritméticas
- Quebre um comando longo em comandos menores e mais simples
- Faça endentação no corpo de uma estrutura
- Escreva um comando por linha

Boas práticas

- Separe os comandos longos em várias linhas, quebrando-as em pontos que façam sentido
- Consulte as tabelas de precedência dos operadores aritméticos e, em caso de dúvida, escreva parênteses para efetuar as operações da forma pretendida
- Use um "pseudocódigo" para "bolar" um programa
- Endente o corpo do comando if e else
- Endente todos os níveis com o mesmo espaçamento

Boas práticas

- Escreva sempre chaves nos comandos de controle (if/else, while, for etc.), mesmo que não seja obrigatório
- Digite as chaves, parênteses e colchetes (abrindo e fechando) antes de digitar os comandos ou expressões entre eles
- Inicialize contadores, somatórios ou produtórios
- Imprima mensagens de erro em expressões que tenham restrições de valores, tais como divisão por zero, raiz quadrada de número negativo, entrada de dados inválidas, dentre outras

Boas práticas

- Solicite ao usuário os valores a serem digitados, com mensagens adequadas
- Explicite os valores dos "flags" nas entradas de dados que os contêm
- Use o valor absoluto da diferença entre dois números reais, menor que um valor pequeno, em vez de comparar números reais com sinal de igualdade ou desigualdade
- "Inicialize" as variáveis ao declará-las, sempre que possível
- Escreva os operadores unários sempre próximos das variáveis (sem espaços)

Boas práticas

- Controle as repetições com valores inteiros
- Evite muitos níveis de endentação, pois estes tornam os programas difíceis de serem entendidos
- Use <= ou >= em lugar de < ou >, nas estruturas de repetição, para evitar executar uma iteração a menos
- Evite alterar as variáveis de controle do laço for, dentro do corpo do laço
- Coloque sempre o default no comando switch para chamar a atenção em casos excepcionais
- Teste os programas exaustivamente!