

Programação Orientada a Objetos

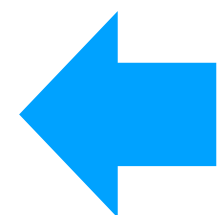
Victor Machado da Silva, MSc
victor.silva@professores.ibmec.edu.br

Índice

- [Apresentação do curso](#)
- [Configurando o ambiente](#)
- [Sobre paradigmas de programação](#)
- [Git](#)
- [Fundamentos da OO](#)
- [Introdução a Java](#)
- [Conceitos estruturais](#)
- [Introdução à UML](#)
- [Conceitos relacionais](#)

Índice

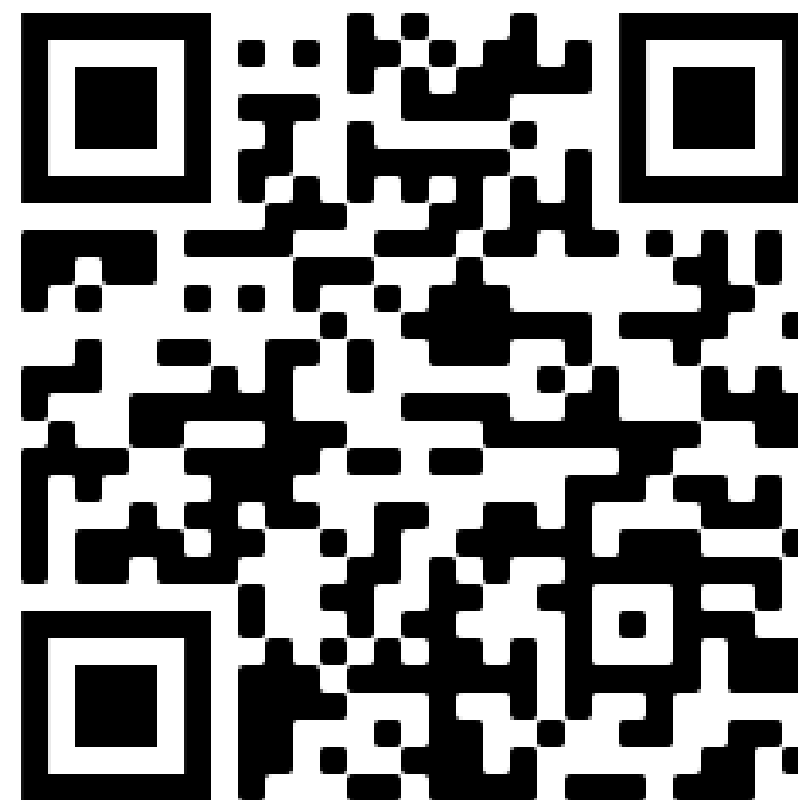
- [Conceitos organizacionais](#)
- [Estudo de caso](#)
- [UML: Diagramas de Casos de Uso](#)
- [Exceções e Tratamento de Erros](#)
- [Boas Práticas no Uso da OO](#)
- [Princípios SOLID](#)



Apresentação do curso

Apresentação do curso

- Contato: victor.silva@professores.ibmec.edu.br
- Aulas às segundas e quartas-feiras, de 7:30 às 9:20
- Grupo no Whatsapp: <https://chat.whatsapp.com/BCQDk3VSc4f0BjN2vbtpsa>
- Material no GitHub: <https://github.com/victor0machado/2021.1-progoo>



Apresentação do curso

Aula	Dia	Dia sem.	Tópico
01	22/02/2021	seg	Introdução à disciplina
02	24/02/2021	qua	Sobre paradigmas de programação / Introdução ao Git
03	01/03/2021	seg	Fundamentos de OO: abstração, reuso, encapsulamento
04	03/03/2021	qua	Conceitos iniciais de Java: parte 1
05	08/03/2021	seg	Conceitos iniciais de Java: parte 2
06	10/03/2021	qua	Conceitos iniciais de Java: parte 3
07	15/03/2021	seg	Conceitos estruturais: classe, atributo, método
08	17/03/2021	qua	Conceitos estruturais: objeto
09	22/03/2021	seg	Noções de UML: introdução
10	24/03/2021	qua	Noções de UML: diagramas de classes
11	29/03/2021	seg	Conceitos relacionais: herança e polimorfismo
12	31/03/2021	qua	Conceitos relacionais: associação
13	05/04/2021	seg	Conceitos relacionais: interface
14	07/04/2021	qua	SEM AULA (SEMANA AP1)
15	12/04/2021	seg	SEM AULA (SEMANA AP1)
16	14/04/2021	qua	SEM AULA (SEMANA AP1)
17	19/04/2021	seg	Conceitos organizacionais: pacotes
18	21/04/2021	qua	SEM AULA (TIRADENTES)
19	26/04/2021	seg	Conceitos organizacionais: visibilidades
20	28/04/2021	qua	Noções de UML: diagramas de caso de uso

Apresentação do curso

Aula	Dia	Dia sem.	Tópico
21	03/05/2021	seg	Noções de UML: diagramas de atividades
22	05/05/2021	qua	Noções de UML: diagramas de sequência
23	10/05/2021	seg	Boas práticas da OOP: parte 1
24	12/05/2021	qua	Boas práticas da OOP: parte 2
25	17/05/2021	seg	Boas práticas da OOP: parte 3
26	19/05/2021	qua	Princípios SOLID: parte 1
27	24/05/2021	seg	Princípios SOLID: parte 2
28	26/05/2021	qua	Princípios SOLID: parte 3
29	31/05/2021	seg	Princípios SOLID: parte 4
30	02/06/2021	qua	Princípios SOLID: parte 5
31	07/06/2021	seg	Design smells
32	09/06/2021	qua	Métricas de código
33	14/06/2021	seg	Frameworks Java
34	16/06/2021	qua	Frameworks Java
35	21/06/2021	seg	Frameworks Java
36	23/06/2021	qua	SEM AULA (SEMANA AP2)
37	28/06/2021	seg	SEM AULA (SEMANA AP2)
38	30/06/2021	qua	SEM AULA (SEMANA AP2)
39	05/07/2021	seg	SEM AULA (SEMANA AS)
40	07/07/2021	qua	SEM AULA (SEMANA AS)

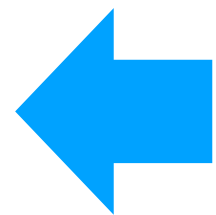
Apresentação do curso

Avaliação

- Proporção:
 - Exercícios periódicos (AC): 20%
 - Projeto (AP1): 40%
 - Projeto (AP2): 40%
- Detalhes das entregas:
 - Exercícios da AC devem ser individuais
 - Projetos de AP1 e AP2 em grupos de no mínimo 2 e no máximo 3 pessoas
- AS será uma prova com consulta, que substituirá a menor nota entre AP1 e AP2.
- Os trabalhos serão *commitados* no GitHub em um *branch* separado do *master*, em um repositório privado criado pelos alunos, e um *pull request* deverá ser enviado para avaliação.

Sugestões de materiais para estudo

- Thiago Leite e Carvalho - Orientação a Objetos: Aprenda seus Conceitos e suas Aplicabilidades de Forma Efetiva (Casa do Código, 2016)
- Apostila Java e Orientação a Objetos (Caelum - <https://www.caelum.com.br/apostilas>)
- Curso Java Completo (DevDojo - https://www.youtube.com/watch?v=kkOSweUhGZM&list=PL62G310vn6nHrMr1tFLNOYP_c73m6nAzL)
- Joshua Bloch - Java Efetivo: As Melhores Práticas para a Plataforma Java (Alta Books, 2019)
- Maurício Aniche - Orientação a Objetos e SOLID para Ninjas: Projetando Classes Flexíveis (Casa do Código, 2015)

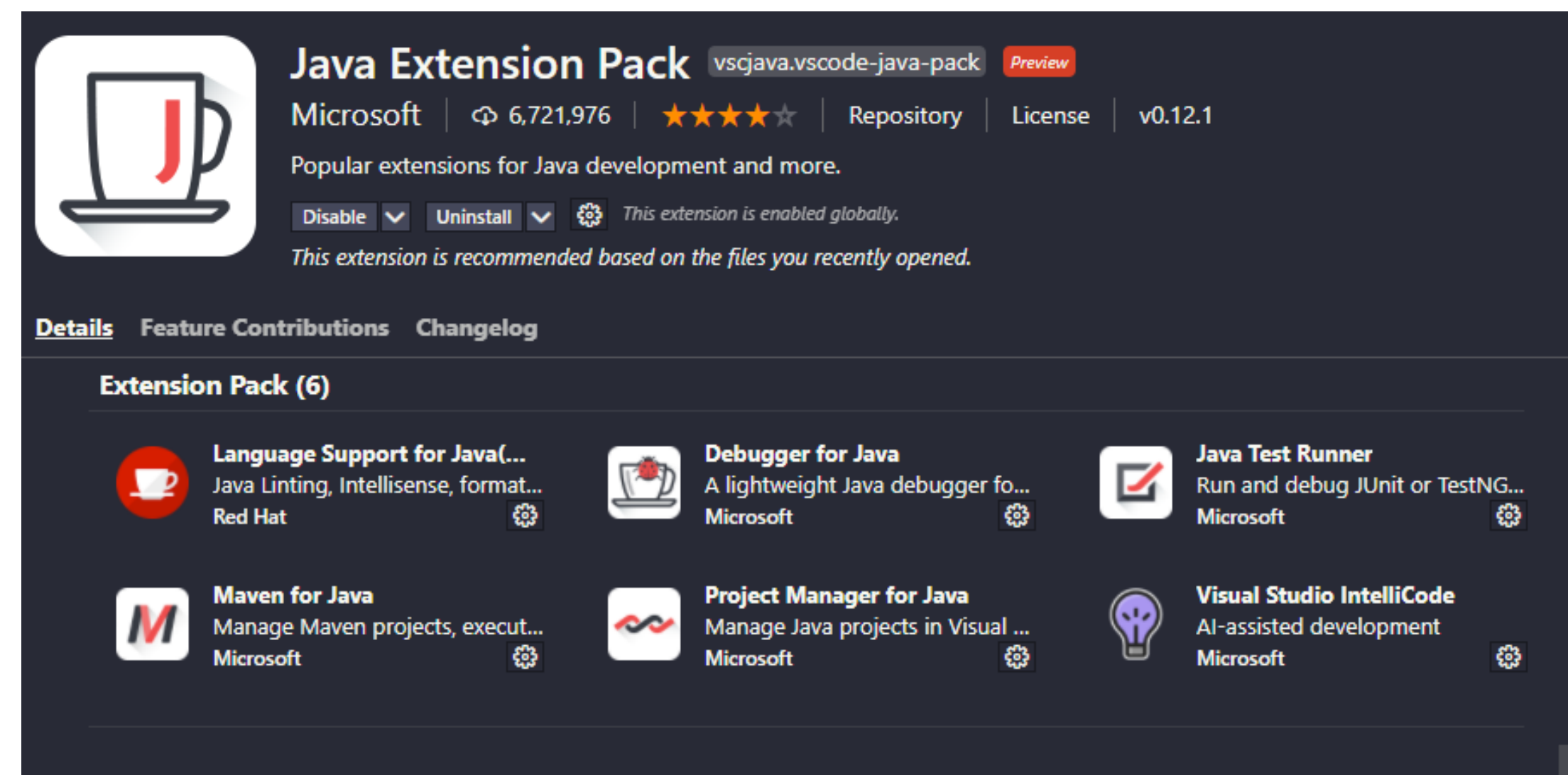


Configurando o ambiente

Configurando o ambiente

IDE:

- Neste curso usaremos o **VSCode** como IDE principal para o desenvolvimento de código. Caso você ainda não tenha, sugiro dar uma olhada nos slides da disciplina de Programação (link [aqui](#)) e fazer a instalação e configuração inicial do software.
- Dentro do VSCode, será necessário instalar a *Java Extension Pack*, um pacote organizado pela própria Microsoft, com diversas extensões específicas para Java.



Configurando o ambiente

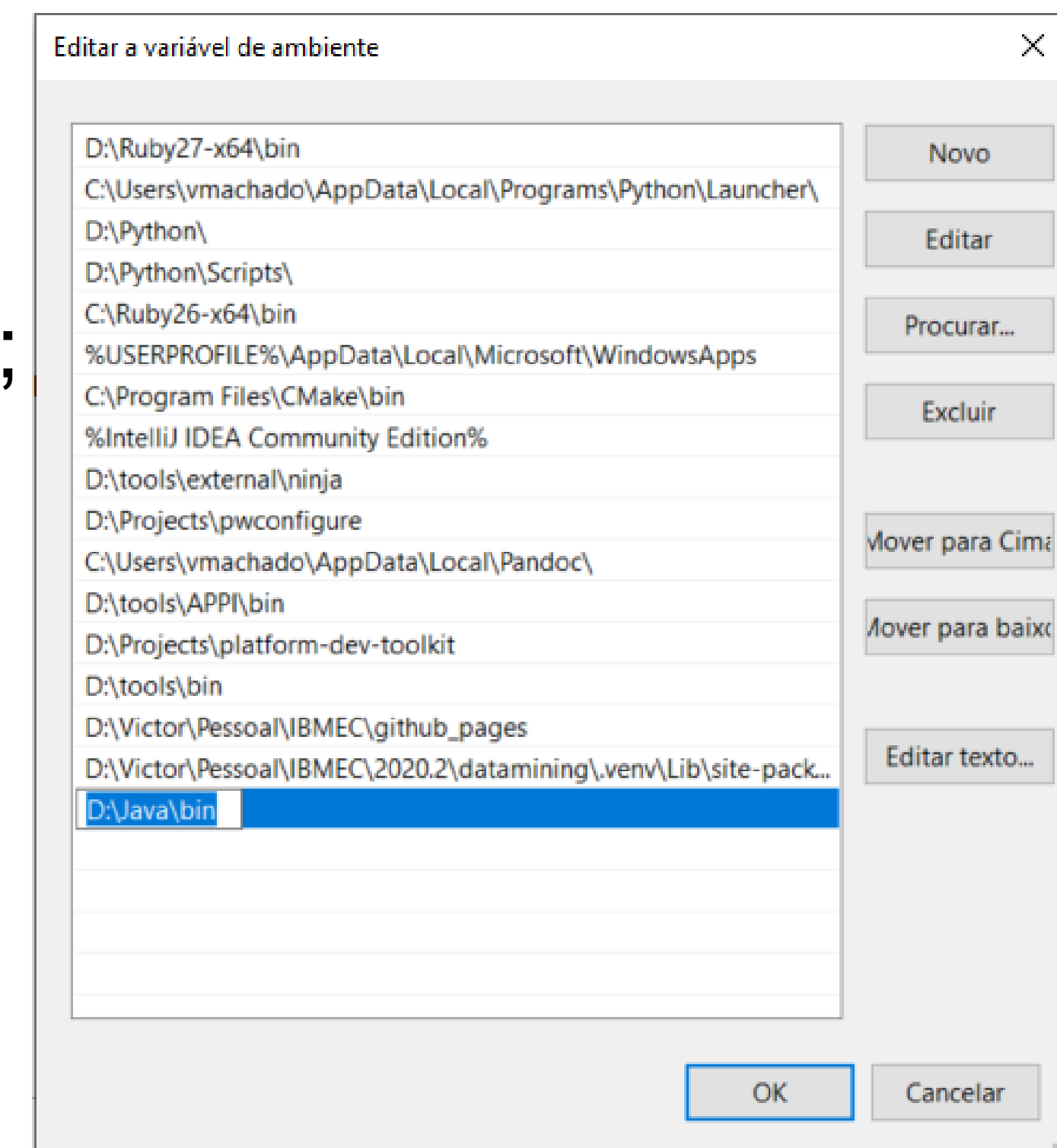
Java:

- Normalmente, todo mundo possui o Java instalado no computador. No entanto, caso você não tenha, clique [aqui](#) para acessar o site do Java, baixar o JRE (*Java Runtime Environment*) e instalá-lo;
- Além do JRE, será necessário instalar o JDK (*Java Development Kit*), plataforma de desenvolvimento para Java. Acesse [esse link](#), baixe a versão mais recente do Java SE (atualmente, é a versão 15) e instale o software;
- Como sugestão, recomendo instalar em um caminho curto, como por exemplo *D:\Java*, para facilitar o acesso. Após a instalação, coloque a pasta “bin” do diretório Java no seu PATH, para poder acessar o compilador pela linha de comando (ver mais informações no próximo slide).

Configurando o ambiente

Como adicionar um caminho ao PATH:

- Aperte a tecla do Windows e pesquise por “Editar as variáveis de ambiente para sua conta”;
- Na janela que abrir, procure pela linha com “Path”, selecione-a e clique em “Editar...”;
- Clique em “Novo” e insira o caminho desejado;
- Aperte “Ok” e “Ok” novamente para fechar tudo;
- Para confirmar se está tudo certo, abra um prompt de comando novo e chame um arquivo ou executável que está contido na pasta (no caso da pasta “bin”, chame “javac”)



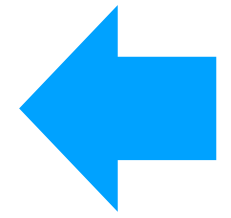
Configurando o ambiente

Git:

- Para o desenvolvimento dos trabalhos e acompanhamento da disciplina, vamos utilizar o Git para versionar os materiais;
- Para instalar o Git na máquina, acesse [esse site](#) e baixe o instalador. O processo de instalação é direto, basta clicar em “Next” para concluir a instalação;
- Também será necessário criar uma conta no GitHub (www.github.com). Caso queira, já pode me procurar por lá (victor0machado).

Outros programas:

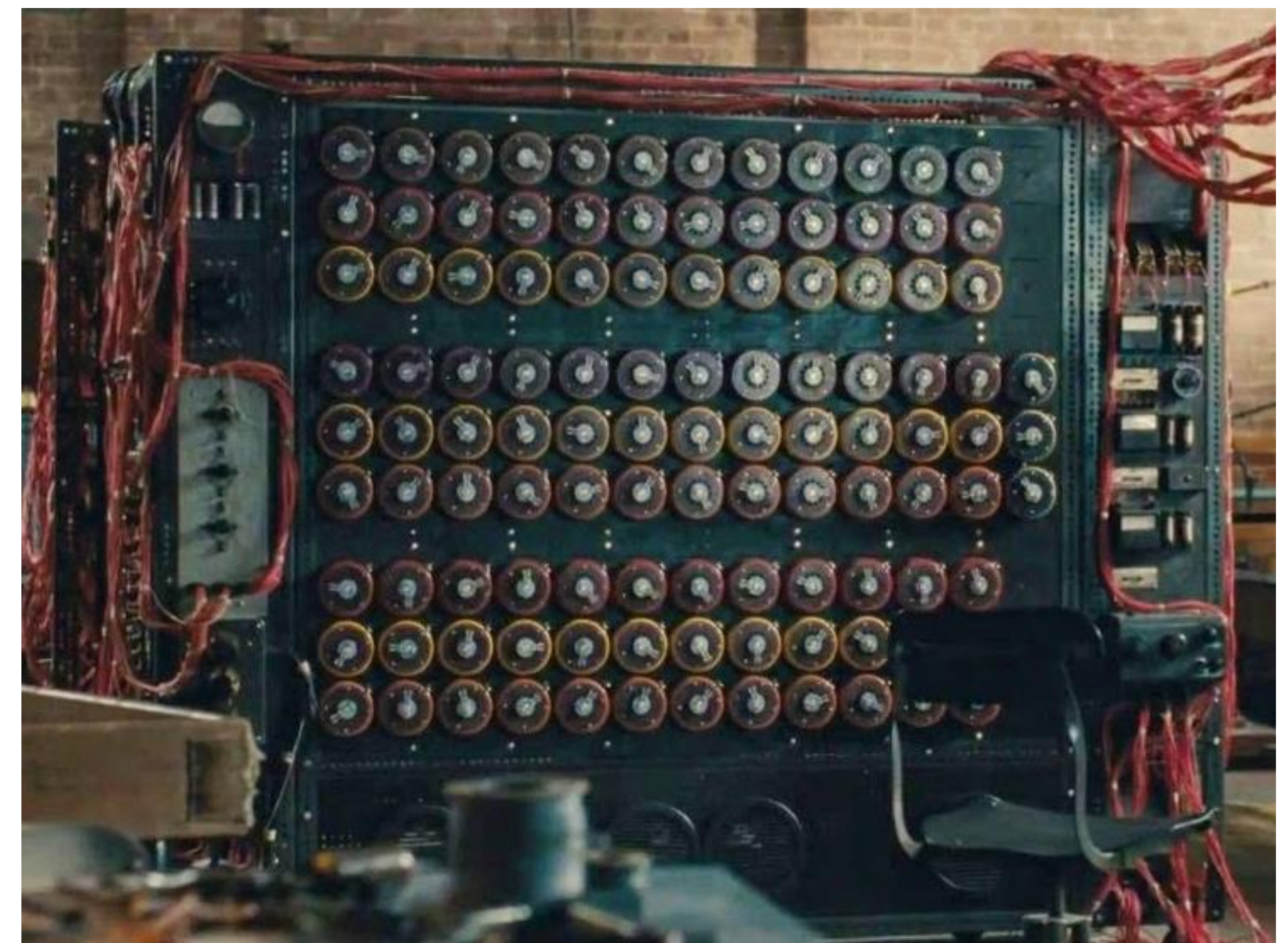
- Recomendo instalar o [notepad++](#) para um bom editor de texto simples.



Sobre paradigmas de programação

Um pouco de história...

Apesar de termos registros históricos de computadores mecânicos e eletromecânicos há bastante tempo, os primeiros computadores próximos aos que conhecemos hoje começaram a surgir na década de 1930, com Alan Turing (1912-1954) e sua definição de uma **máquina universal**, ou **máquina de Turing**, um conceito abstrato de um computador, no qual pode-se modelar qualquer computador digital.



Um pouco de história...

Durante as décadas de 1930 e 1940 surgiram diversos computadores a válvula, que ocupavam o espaço de salas inteiras e chegavam a pesar toneladas. Kits para computadores pessoais e computadores comerciais começaram a surgir na década de 1950, todos eles programáveis através de cartões que eram perfurados pelo usuário e lidos pelas máquinas, que então faziam as operações indicadas.

Até esse ponto, a memória interna de um computador era extremamente limitada. O UNIVAC, um dos primeiros computadores comerciais do mundo, tinha memória interna de 1.000 palavras de 12 caracteres. Essa memória ainda não era eletrônica, sendo utilizados tubos de mercúrio para armazenar as informações.



Um pouco de história...

A primeira linguagem de programação de alto nível implementada para um computador foi o FORTRAN, em 1957, inicialmente para o IBM 704, que já possuía memória magnética.

O primeiro compilador para FORTRAN possuía 25 mil linhas de código (de máquina). Toda máquina IBM 704 vinha com uma fita magnética contendo o compilador e um manual de 51 páginas.

Como esperado, essa linguagem era muito limitada para nossos padrões atuais. Variáveis só podiam possuir um ou dois caracteres.

O FORTRAN evoluiu e foi durante mais de 30 anos a principal linguagem de programação científica e ainda hoje é usada em muitas áreas onde se necessita de alta precisão e eficiência.



Um pouco de história...

Com o surgimento dos circuitos integrados, de computadores com cada vez mais memória e a popularização dos dispositivos para além das grandes universidades e empresas, foram surgindo outras linguagens, como:

- COBOL (1960)
- BASIC (1964)
- Pascal (1970)
- Smalltalk (1971)
- C (1972)
- PROLOG (1972)

Um pouco de história...

A linguagem Java surgiu bem depois, em 1995, depois até de Python, que surgiu pela primeira vez em 1991. Quando a linguagem surgiu, os computadores pessoais e os sistemas operacionais já estavam difundidos no mundo inteiro.

O projeto surgiu em 1991, com a perspectiva de que a próxima grande tendência em desenvolvimento de sistemas seria a junção de computadores com dispositivos digitais. Não havia ainda linguagens adequadas para esse tipo de desenvolvimento.

A nova linguagem foi demonstrada em um dispositivo interativo portátil projetado para a indústria de TV a cabo. Esse conceito de dispositivos móveis ainda demoraria muitos anos até se tornar economicamente viável, portanto pode-se dizer que Java foi uma linguagem criada muito à frente do seu tempo.



Um pouco de história...

Java foi, de fato, a linguagem base que permitiu o surgimento dos dispositivos móveis. Até 2017, Java era a linguagem oficial do sistema Android, e o Kotlin, atual linguagem oficial, foi desenvolvido para que tenha uma interoperabilidade com Java.

De acordo com o índice TIOBE, principal medidor de uso de linguagens de programação, inclui Java em primeiro ou segundo lugar há 20 anos, dividindo o pódio com C, C++ e, desde 2019, Python.

Paradigmas de programação

Um paradigma, por definição, é um conceito que define um exemplo típico ou modelo de algo. É a representação de um padrão a ser seguido.

Um **paradigma de programação** é uma forma de classificação de linguagens de programação, baseada nas suas funcionalidades. Um paradigma é, portanto, um tipo de estruturação ao qual a linguagem deverá respeitar.

Cada paradigma surgiu de necessidades diferentes. Dado isso, cada um apresenta maiores vantagens sobre os outros dentro do desenvolvimento de determinado sistema. Sendo assim, um paradigma pode oferecer técnicas apropriadas para uma aplicação específica.

Paradigmas de programação

Paradigma imperativo ou procedural:

- As instruções devem ser passadas ao computador na sequência em que devem ser executadas;
- Como linguagens mais conhecidas temos COBOL, FORTRAN e Pascal;
- O código programado através desse paradigma é uma espécie de passo-a-passo dos procedimentos que a máquina deverá executar;
- A solução do problema será muito dependente da experiência e criatividade de quem trabalha com a programação - foco no “como”;
- Recomendado em projetos onde não se espera que haja mudanças significativas com o tempo;
- É eficiente e permite uma modelagem tal qual o mundo real, porém o código tende a ser de difícil legibilidade.

Paradigmas de programação

Paradigma declarativo:

- Foca mais no “quê” deve ser resolvido, ao invés do “como”;
- Nível de abstração é maior;
- O foco deixa de ser como o resultado vai ser computado, e sim em como funciona a sequência lógica e qual o resultado esperado;
- Como exemplos de linguagens declarativas, temos Lisp, Prolog e Haskell.

Paradigmas de programação

Paradigma funcional:

- O uso de funções é destaque;
- O programa é dividido em blocos e, para sua resolução, são implementadas funções que definem variáveis em seu escopo e retornam algum resultado;
- São exemplos de linguagens o Lisp, o Scheme e o Haskell;
- É bastante indicado quando a solução requerida é fortemente dependente de uma base matemática;
- O paradigma funcional tem alocação de memória automática, eliminando possíveis “efeitos colaterais” nos cálculos matemáticos das funções.

Paradigmas de programação

Paradigma lógico:

- Deriva do paradigma declarativo;
- Utiliza formas de lógica simbólica como padrões de entrada e saída, para realizar inferências para produzir os resultados;
- Entre as linguagens que usam esse paradigma, podemos citar Mercury e Prolog;
- São utilizadas na solução de problemas que envolvem inteligência artificial, criação de programas especialistas e comprovação de teoremas.

Paradigmas de programação

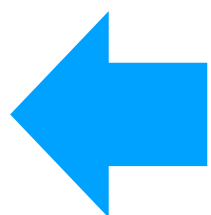
Paradigma orientado a objetos:

- Foi popularizado na década de 1990 com a linguagem Java, apesar de existir há mais tempo, em linguagens como Smalltalk e C++;
- Permite uma programação multiplataforma de uma mesma maneira;
- Apoia-se nas abstrações de classes e objetos ao tentar retratar a programação tal qual se enxerga o mundo real;
- Todos os objetos têm determinados estados e comportamentos. Esses estados são descritos pelas classes como atributos, e a forma como os objetos se comportam é definida por meio de métodos, que são equivalentes às funções do paradigma funcional;
- Três alicerces básicos: herança, polimorfismo e encapsulamento.

Paradigmas de programação

Paradigma orientado a eventos:

- Usado por toda linguagem de programação que tem uso de recursos gráficos, como jogos e formulários;
- Dessa forma, a execução do programa se dá à medida que determinados eventos são disparados pelo usuário;
- Como exemplos, temos Visual Basic, C# e Delphi.



Git

Introdução

- Talvez você reconheça esse nome por causa de sites como *github* ou *gitlab*. **Git** é um sistema *open-source* de controle de versionamento.
- Versionar projetos é uma prática essencial no mundo profissional e, em particular, na área de tecnologia, para manter um histórico de modificações deles, ou poder reverter alguma modificação que possa ter comprometido o projeto inteiro, dentre outras funcionalidades que serão aprendidas na prática.
- Os projetos são normalmente armazenados em **repositórios**.
- Vamos aqui falar de alguns conceitos principais sobre como funciona o sistema, independente da plataforma que vamos utilizar (p.ex., *github*). Em seguida vamos aplicar alguns desses conceitos na prática.

Para mais informações...

- O tempo que temos em aula não é suficiente para conseguirmos discutir todos os detalhes por trás dessa tecnologia. Portanto, seguem abaixo algumas sugestões de conteúdos extras para estudarem:
 - <http://git-scm.com/book/en/Getting-Started-About-Version-Control>
 - <http://git-scm.com/book/en/Getting-Started-Git-Basics>
 - <http://learngitbranching.js.org/>
 - <http://try.github.io/levels/1/challenges/1>

O que é Controle de Versões?

- Controle de Versões é um sistema de grava mudanças aplicadas em um arquivo ou um conjunto de arquivos ao longo do tempo, para que o usuário possa lembrar ou recuperar depois.
- Na área de tecnologia o controle de versões já é algo consolidado, uma vez que inúmeras alterações são aplicadas em um software durante a sua implementação e manutenção. No entanto, adotar um sistema de controle de versões (VCS, da sigla em inglês) é algo recomendado para qualquer área.
- Quando se quer adotar um VCS, normalmente o primeiro passo é trabalhar com um controle local, fazendo cópias dos arquivos e os renomeando com algum padrão (p.ex., incluindo a data ao final do nome do arquivo).
- O Git veio para otimizar esse controle de versões, permitindo inúmeras alterações que seriam muito complicadas se fossem feitas manualmente.

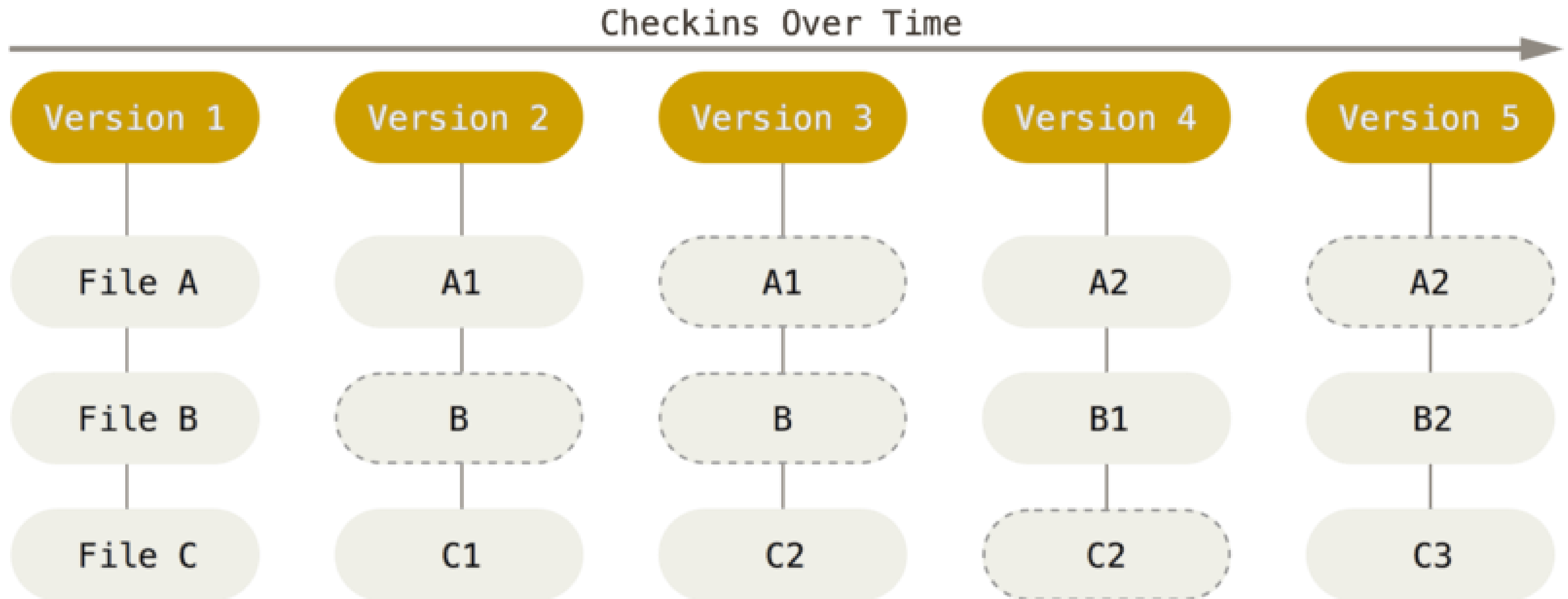
Uma breve história do Git

- A criação do Git está atrelada à criação do Linux, quando o time responsável utilizou uma solução de controle de versão que, eventualmente, tornou-se paga.
- A comunidade Linux, portanto, começou a planejar um sistema que aproveitasse algumas das características da solução anterior, porém evoluindo diversos aspectos.
- A comunidade tinha os seguintes objetivos para o novo sistema:
 - Velocidade
 - Design simples
 - Suporte forte para desenvolvimento não-linear, com milhares de atividades sendo realizadas em paralelo
 - Completamente distribuído, permitindo o acesso de qualquer lugar
 - Eficiente no suporte a grandes projetos

O que é Git?

- **Git** funciona pensando que os dados de um repositório compõem uma série de “fotografias” de um sistema de arquivos em miniatura.
- No Git, a cada vez que você aplica uma alteração (ou *commit*), ou salva o estado do seu projeto, o Git basicamente tira uma “foto” de como os arquivos do repositório estão naquele momento e então ele armazena uma referência a essa foto.
- Para ser eficiente, se os arquivos não foram alterados, o Git não altera os arquivos novamente, apenas um link para a última versão do arquivo armazenada.
- Sendo assim, o Git trabalha os dados como um **fluxo de fotografias**.

O que é Git?



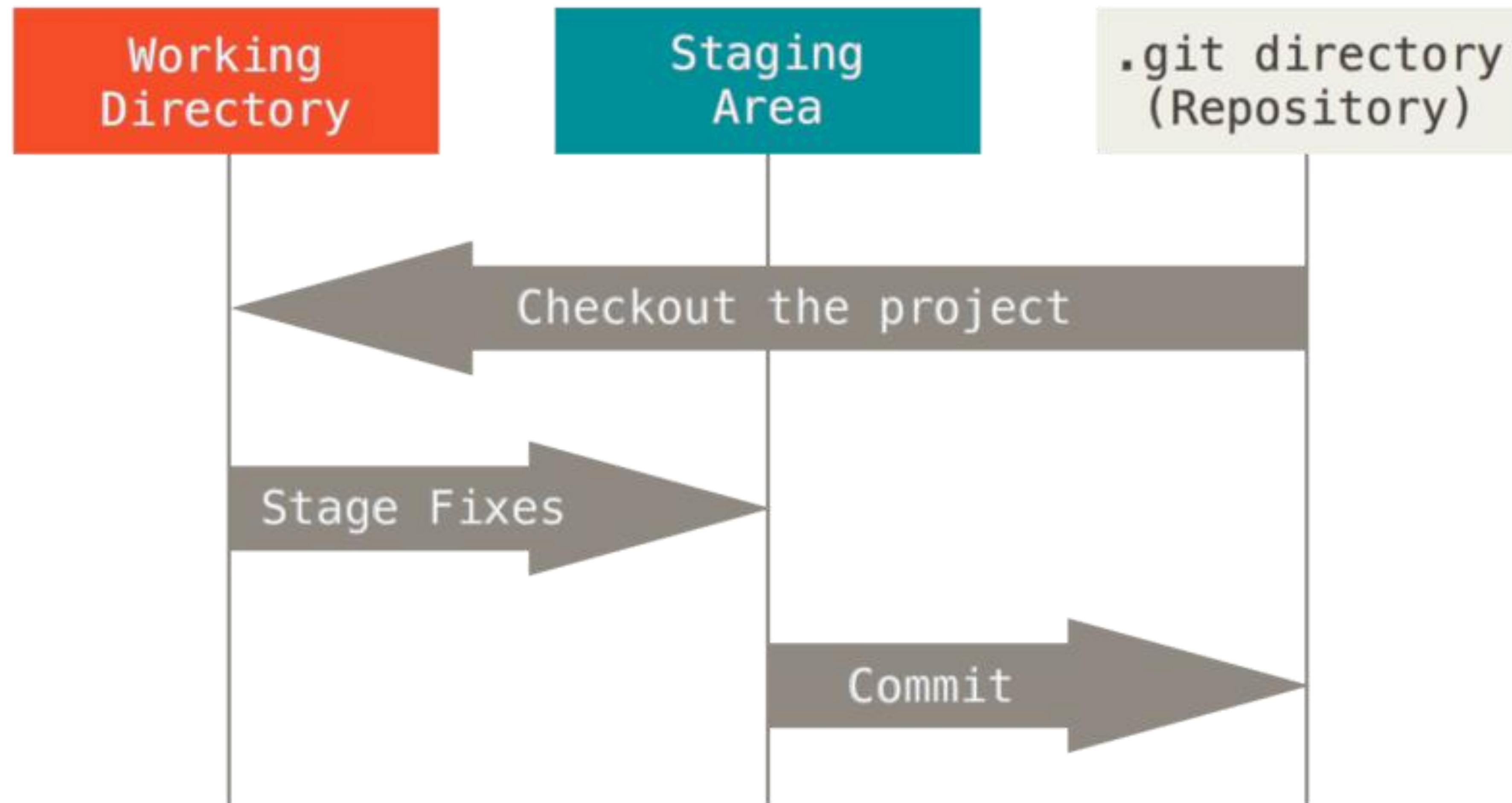
O que é Git?

- A maior parte das operações no Git precisa apenas de arquivos e recursos locais para operarem, e normalmente nenhuma informação é necessária de outro computador na rede. Isso fornece uma velocidade de operação que outros VCS não possuem. Como cada usuário possui todo o histórico do projeto no computador, a maioria das operações aparenta ser quase instantânea.
- Para todos os efeitos, na prática, o Git normalmente só acrescenta informações ao seu banco de dados, nunca removendo informações. É muito difícil, e não recomendado, gerar operações que removam informações, já que essas operações podem afetar o histórico do seu projeto.

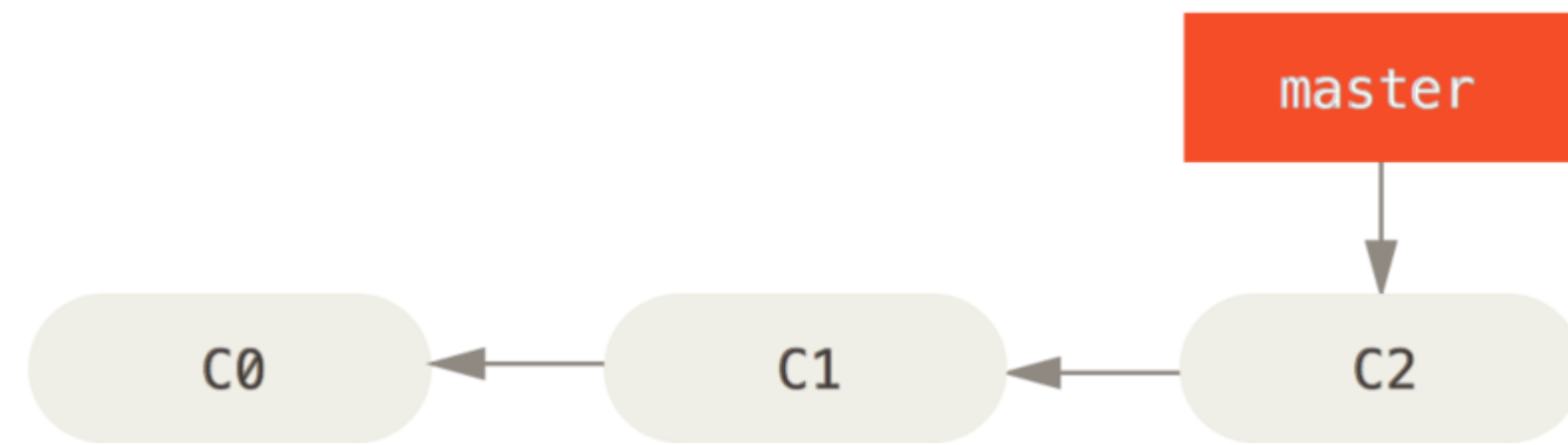
Os três estados

- O Git tem três estados principais nos quais os arquivos de um repositório podem se encontrar: **modificado**, **preparado** e “**commitado**”:
- **Commitado** significa que os dados estão armazenados de forma segura em seu banco de dados local;
- **Modificado** significa que você alterou o arquivo, mas ainda não fez o commit no banco de dados;
- **Preparado** significa que você marcou a versão atual de um arquivo modificado para fazer parte do seu próximo commit.
- Isso leva a três seções principais de um projeto Git: o diretório Git, o diretório de trabalho e área de preparo.

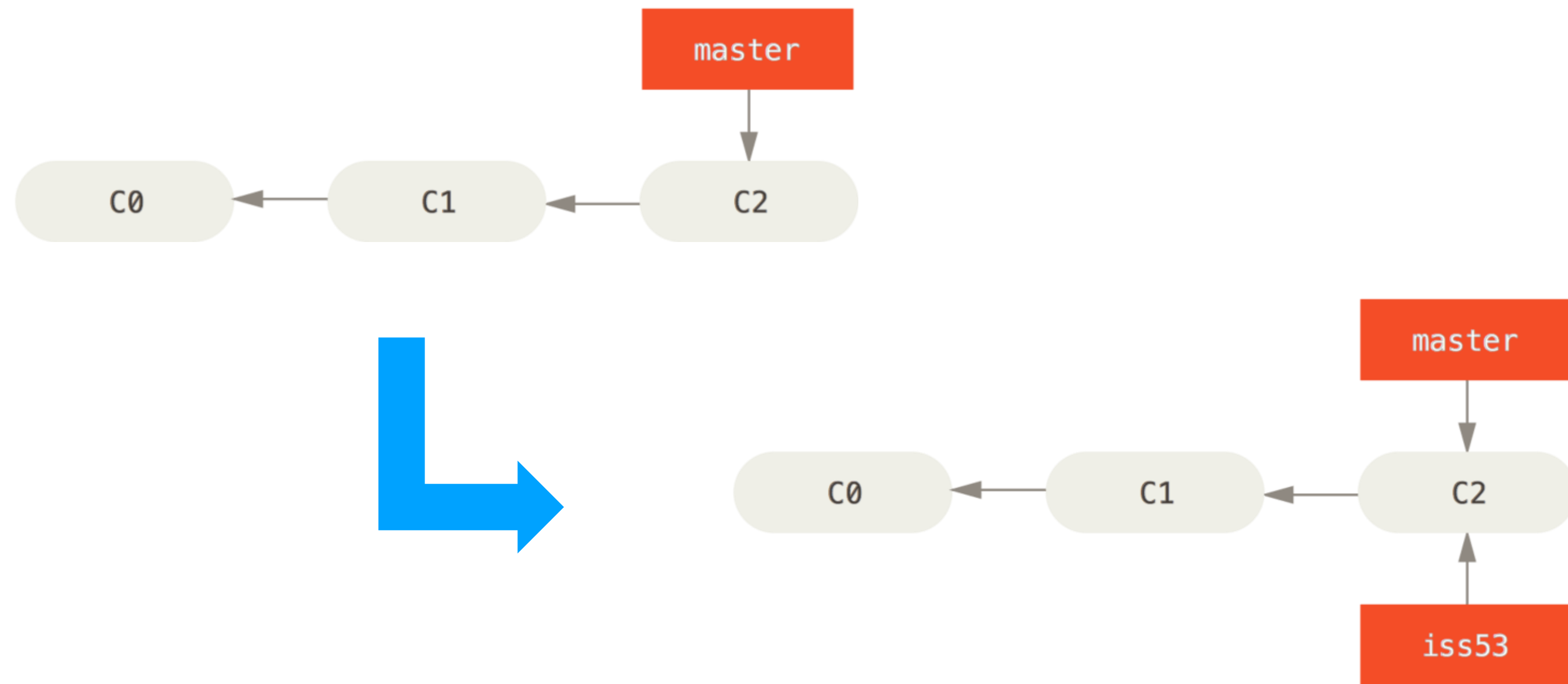
Os três estados



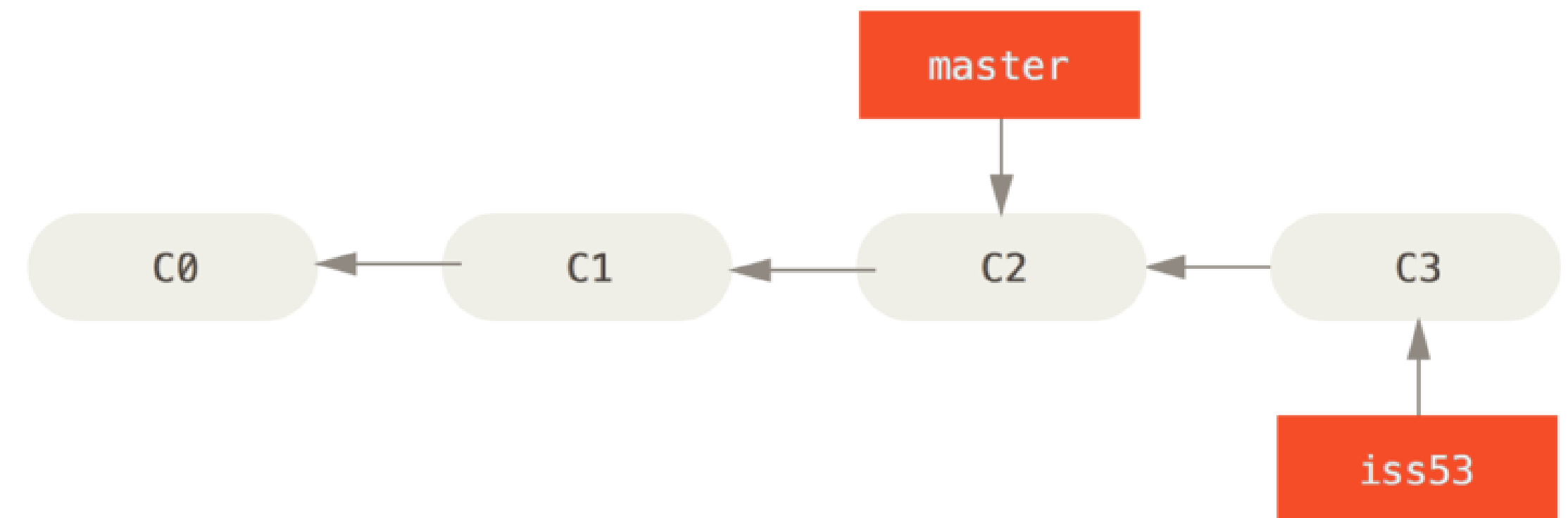
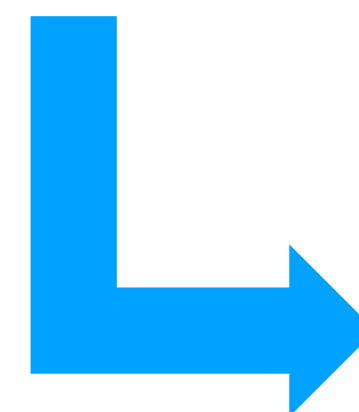
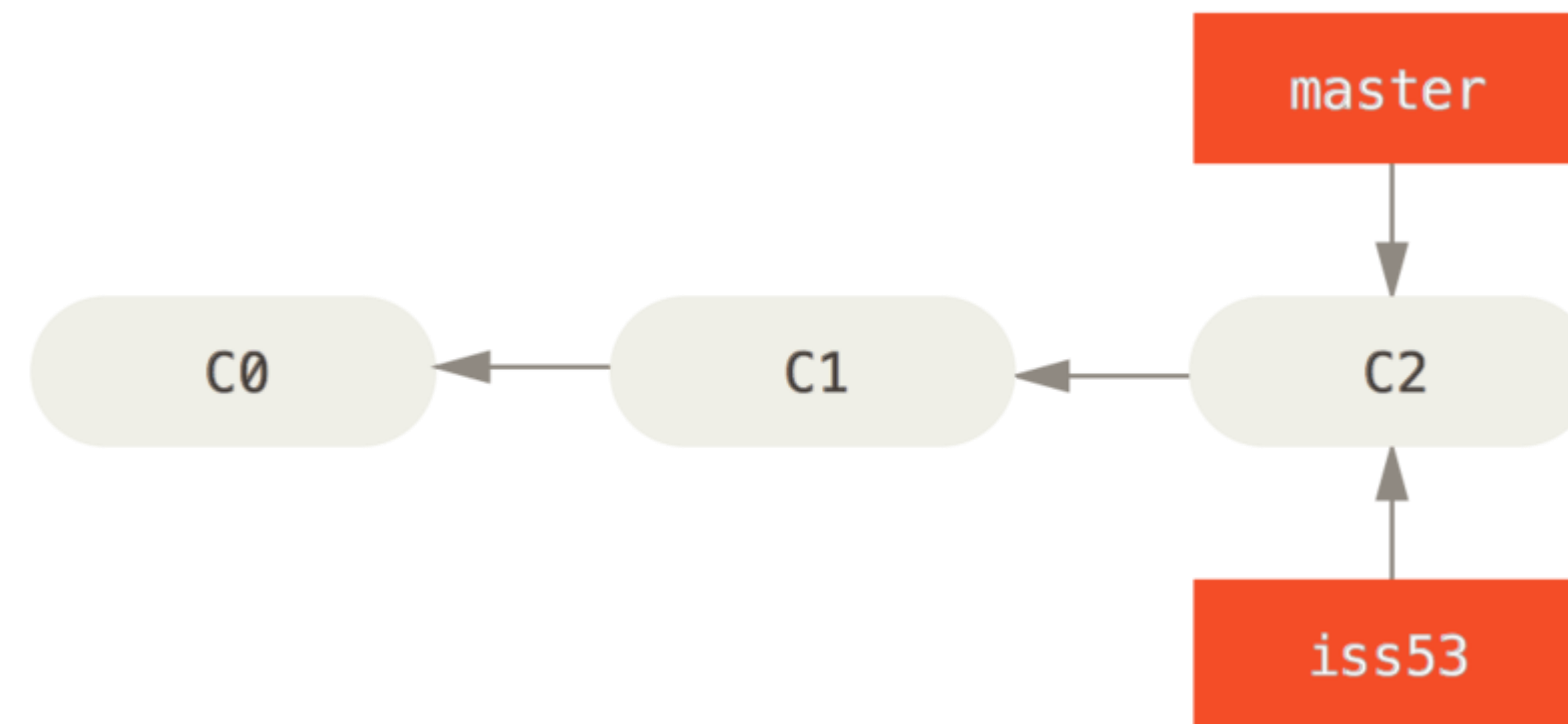
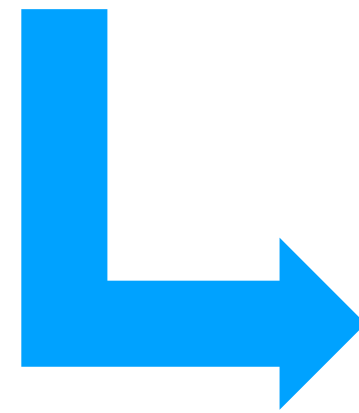
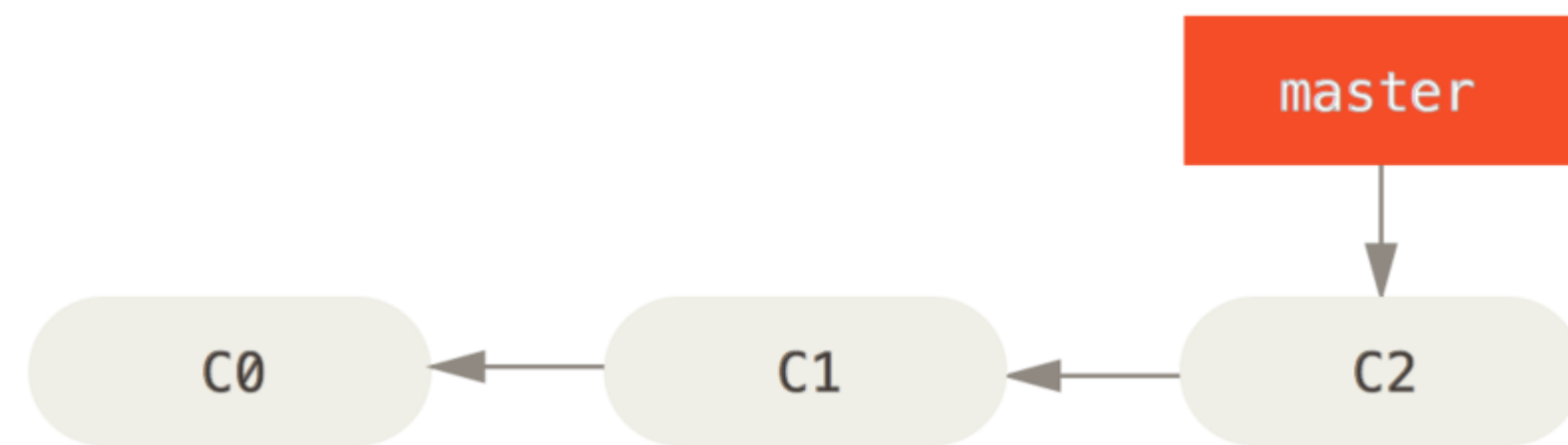
Branches no Git



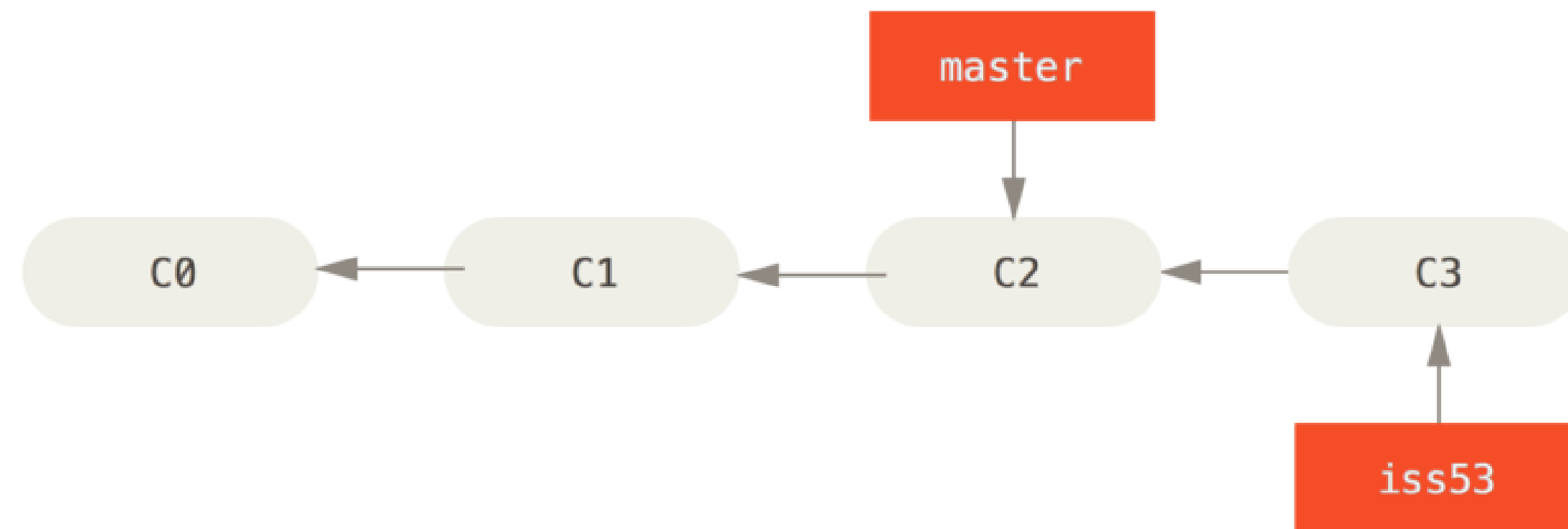
Branches no Git



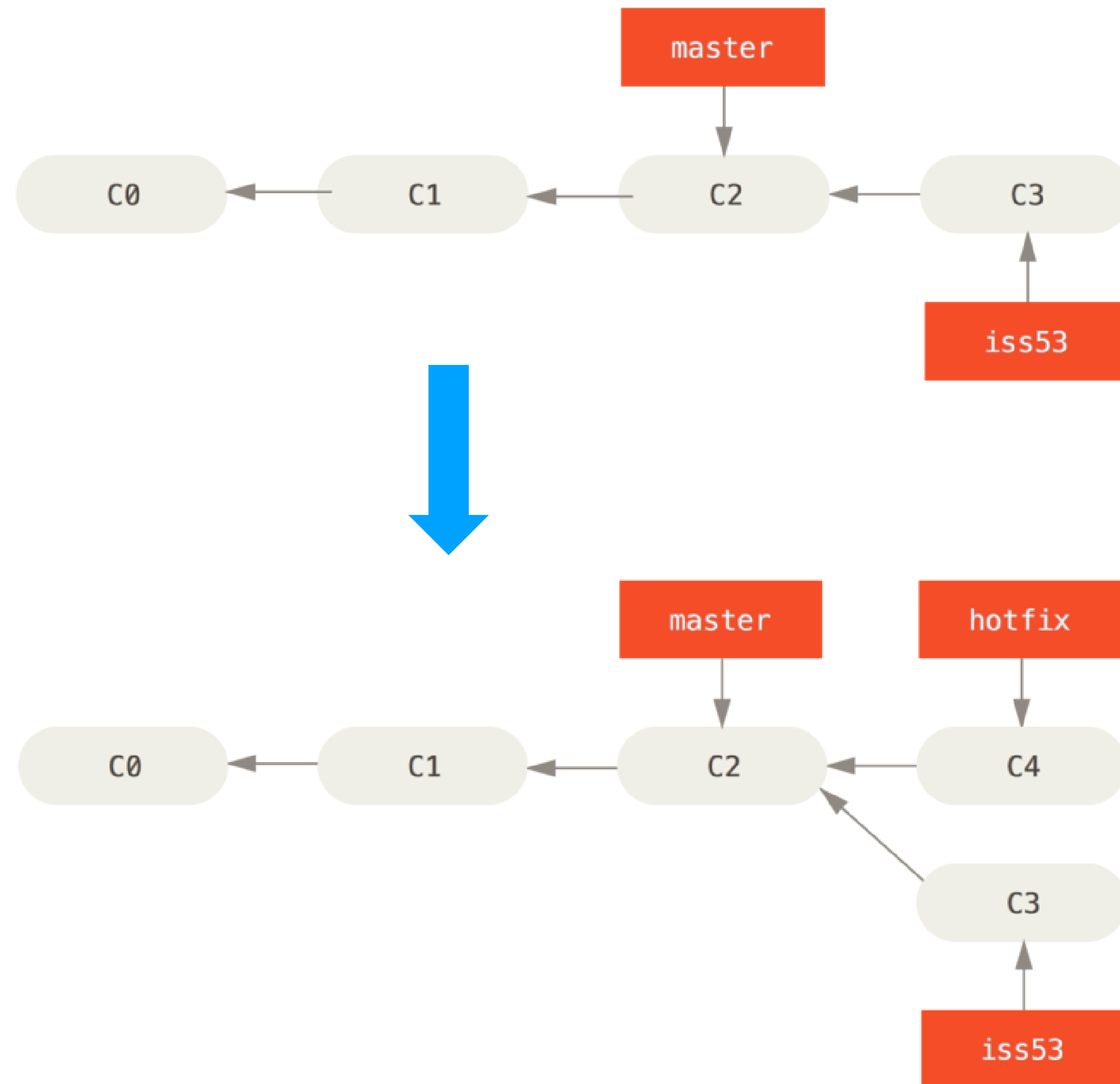
Branches no Git



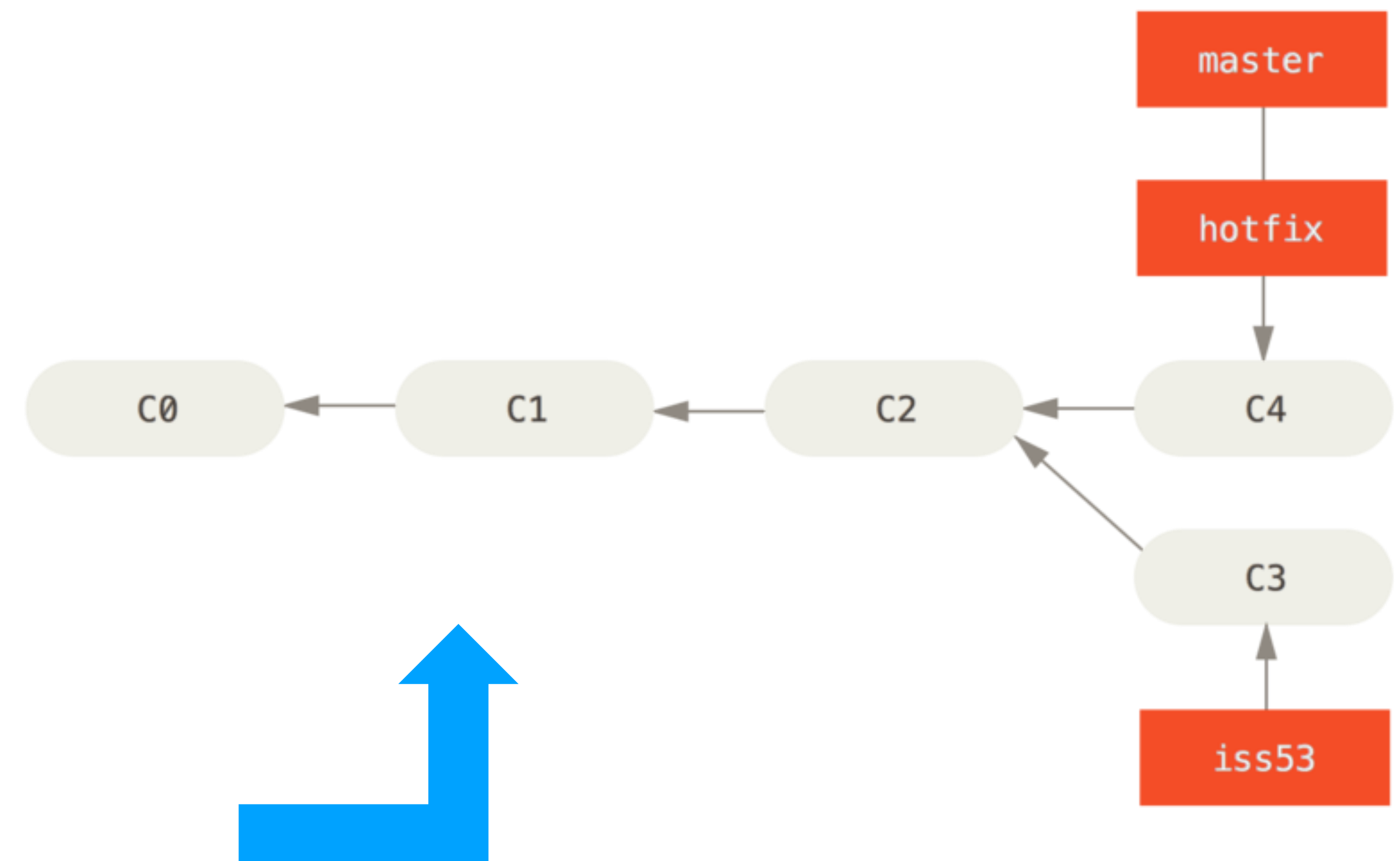
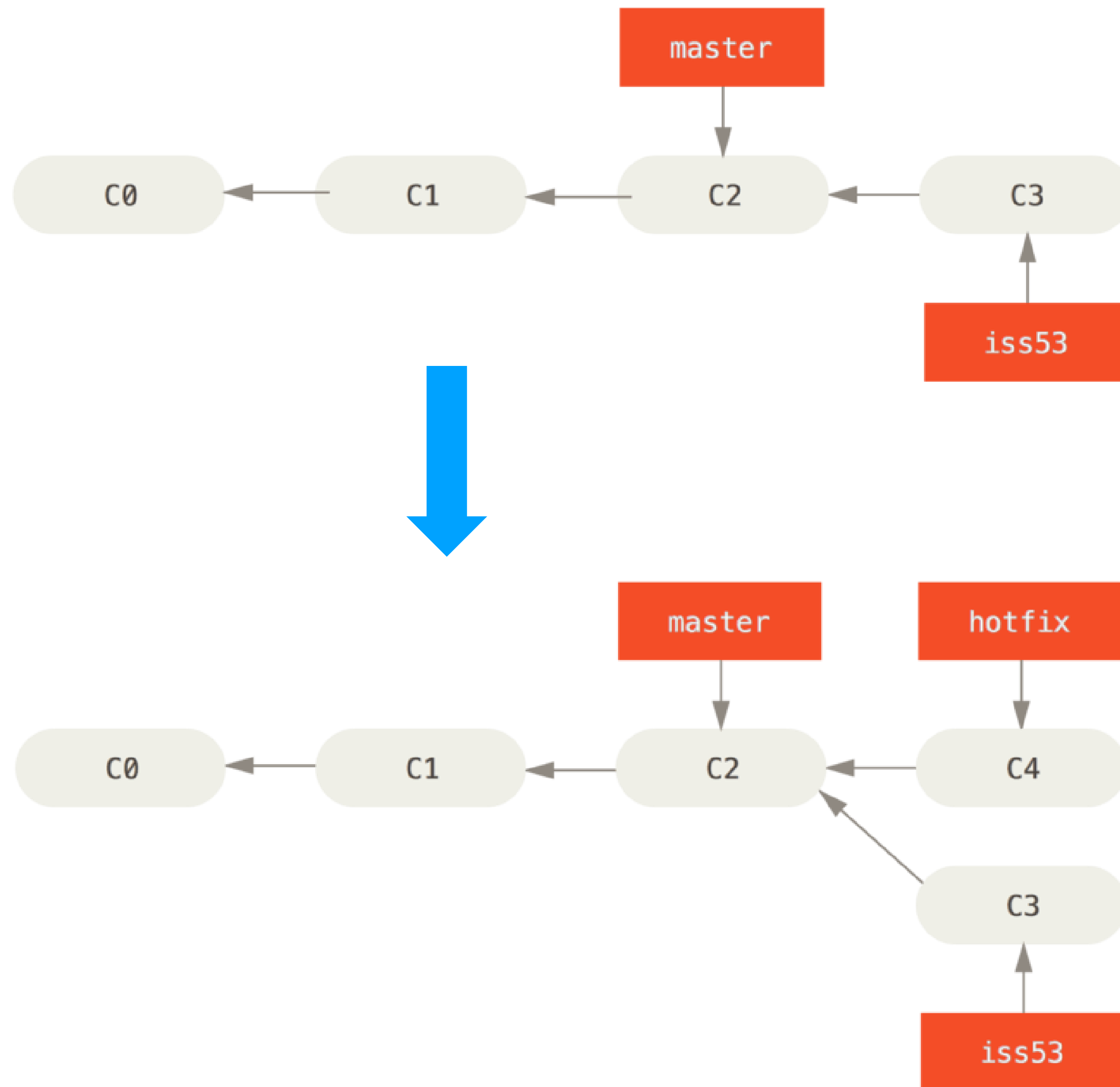
Branches no Git



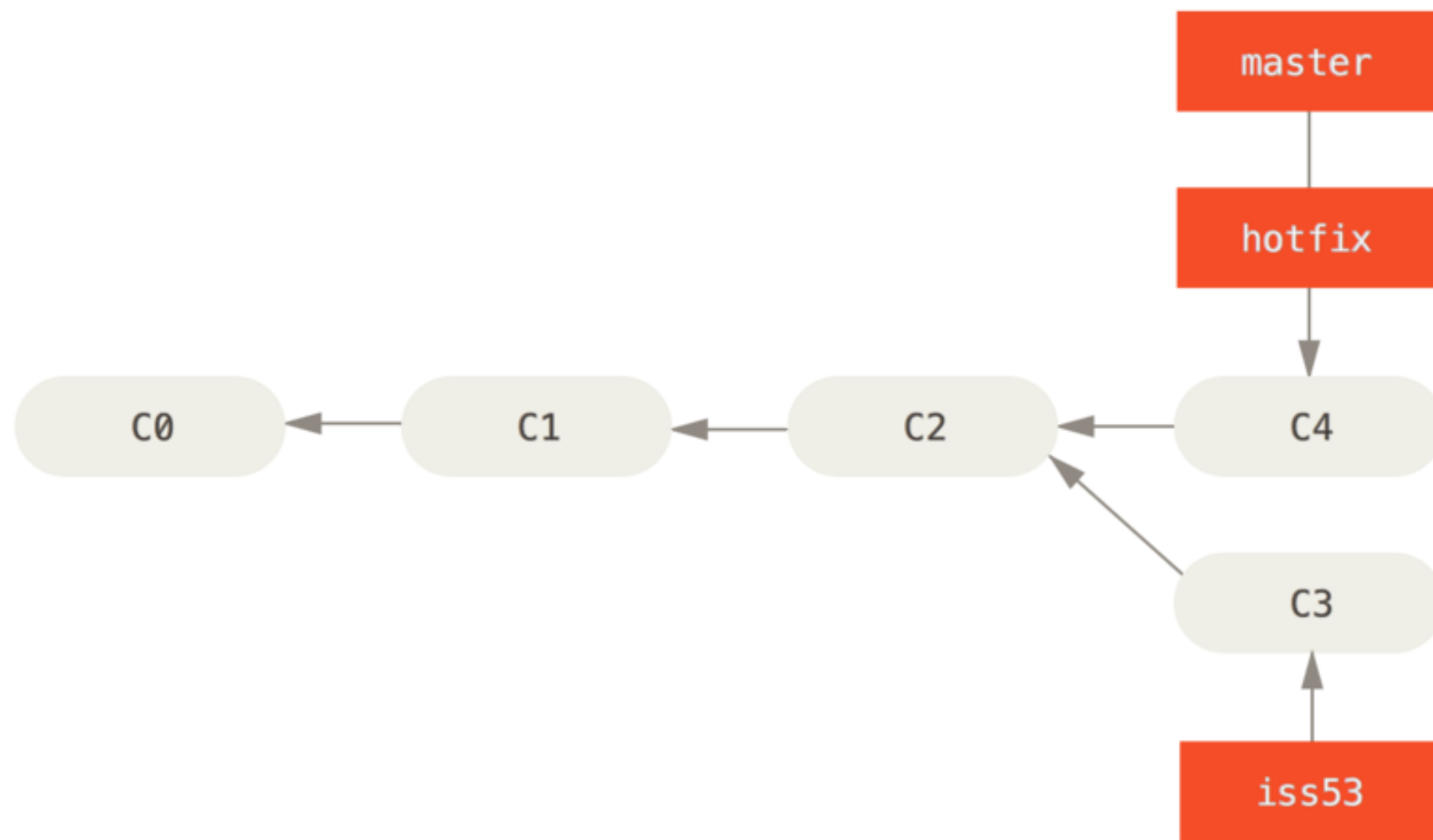
Branches no Git



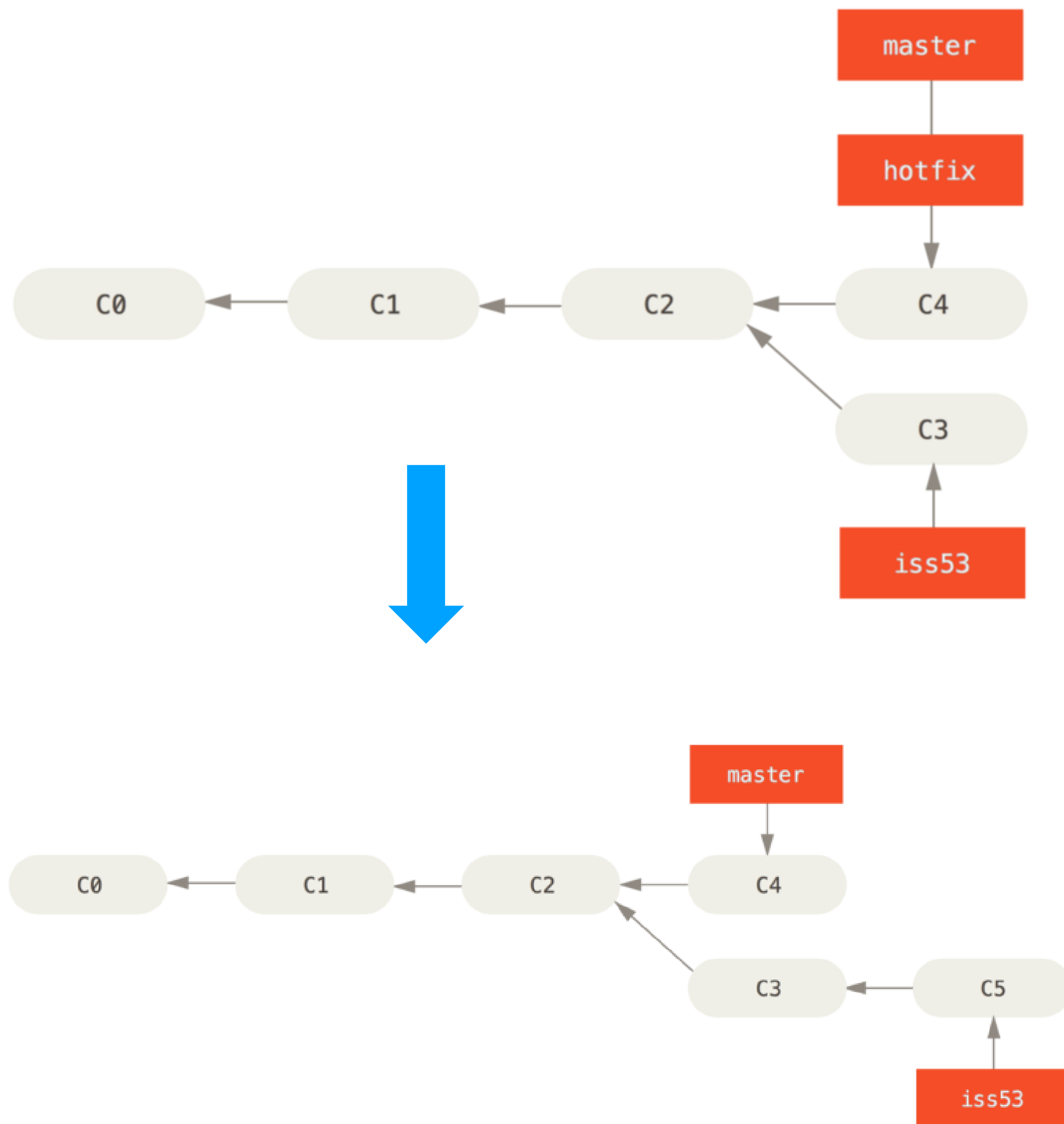
Branches no Git



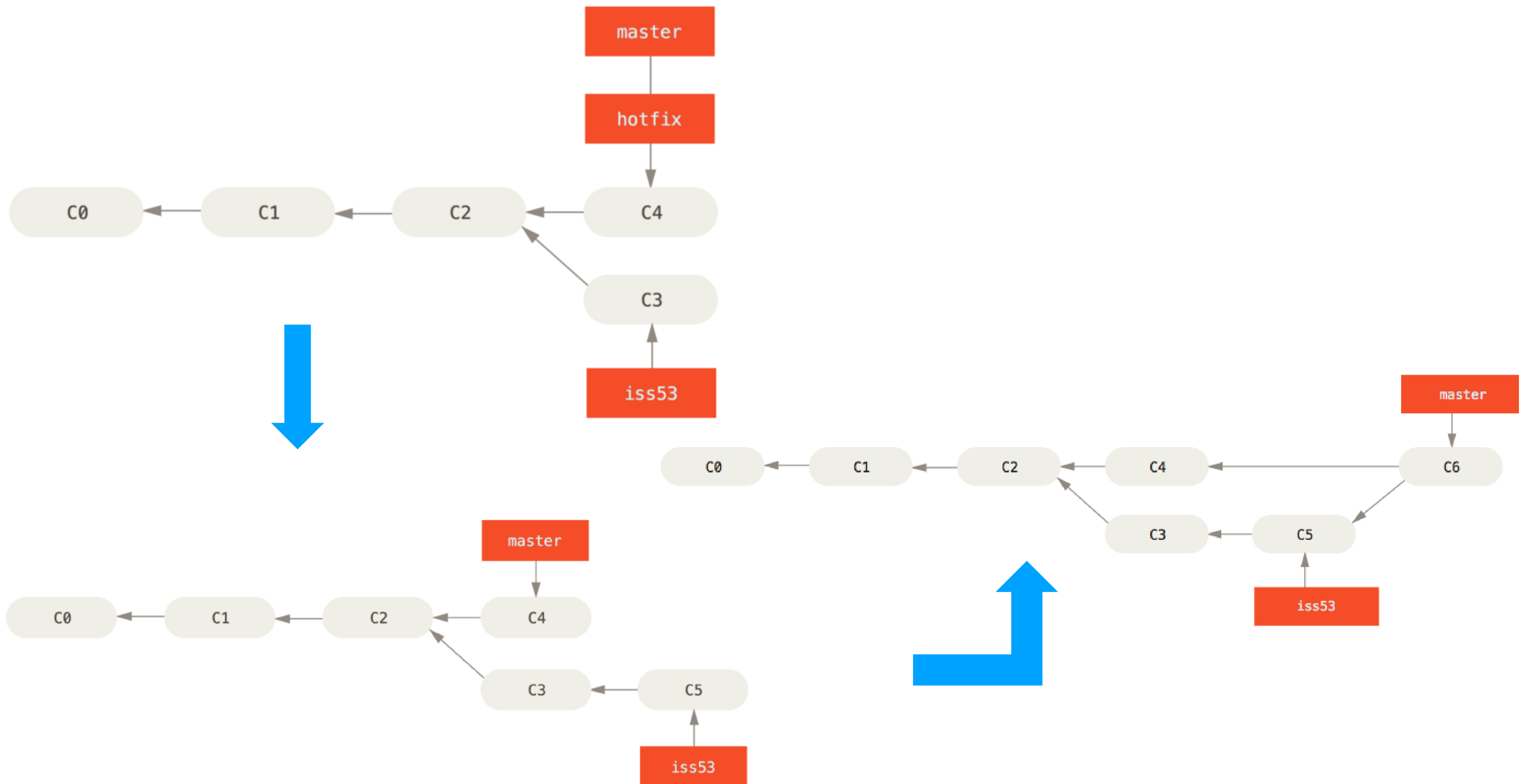
Branches no Git



Branches no Git



Branches no Git



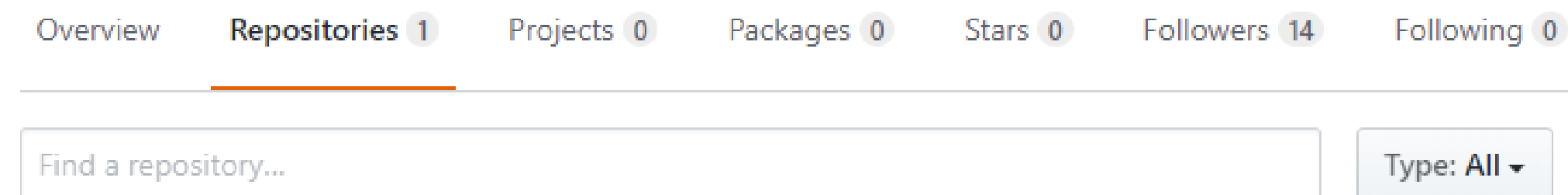
Iniciando o uso de Git com GitHub

- Antes de mais nada, precisamos instalar o sistema Git na nossa máquina.
- <https://git-scm.com/downloads>
- O GitHub (www.github.com) é uma plataforma de versionamento que utiliza Git como base.
- Não é o nosso objetivo ensinar a plataforma a fundo, mas vamos utilizar algumas funcionalidades. Caso tenha interesse em se aprofundar no assunto, recomendo algumas páginas:
 - <https://github.com/culturagovbr/primeiros-passos>;
 - <https://help.github.com/en>;
 - https://rogerdudler.github.io/git-guide/index.pt_BR.html.
- Antes de começar a configurar, crie uma conta no site.

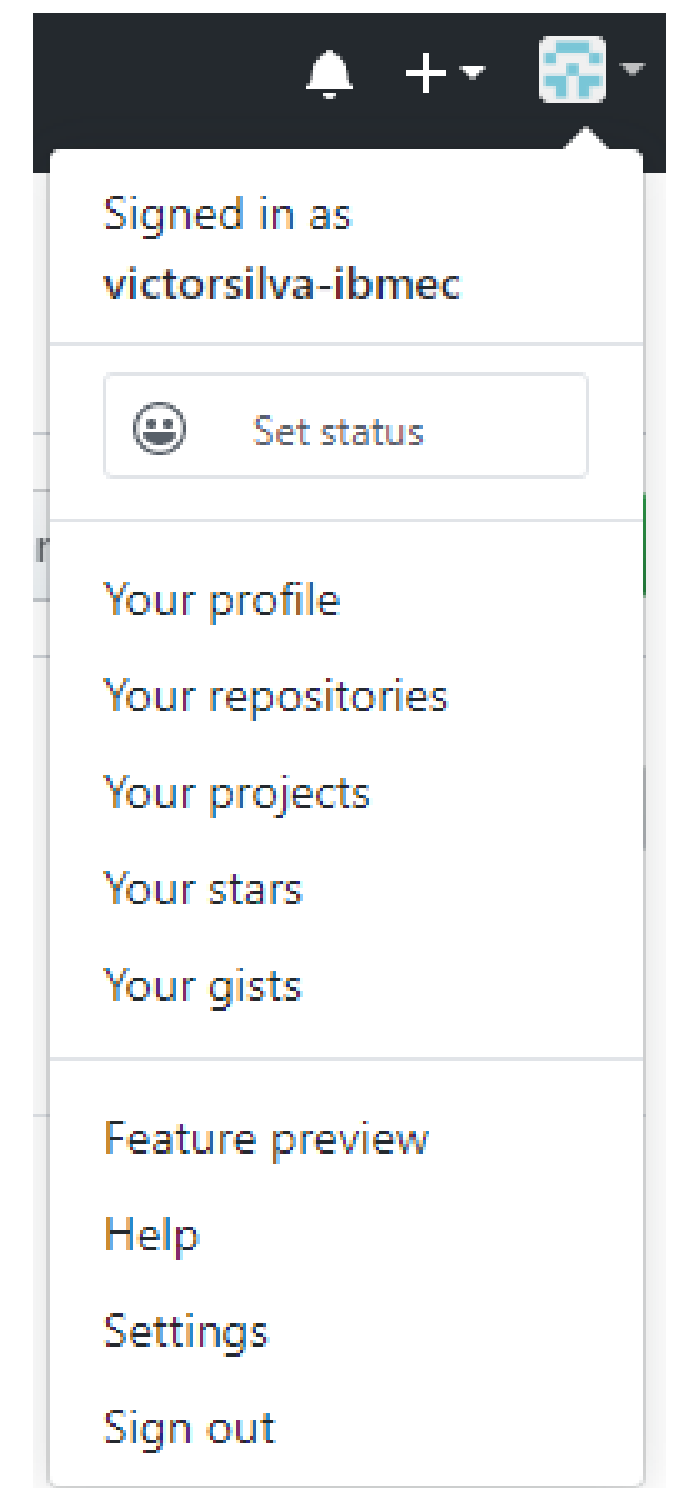
Algumas tarefas no GitHub

- **Criando um novo repositório:**

- No canto superior direito, clique no ícone do seu usuário e, em seguida, em **Your Profile**;
- Na nova janela, clique em **Repositories** e em **New**;

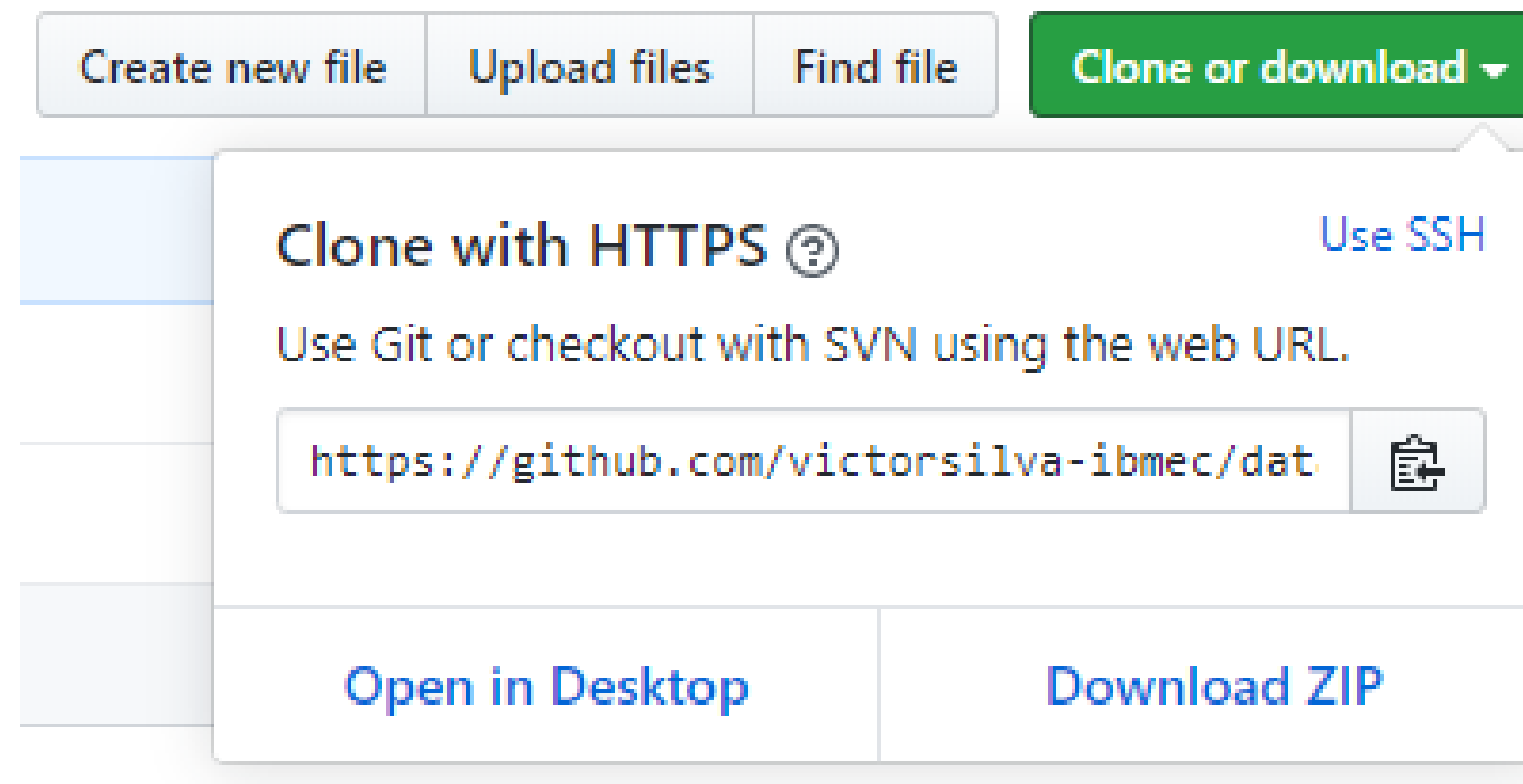


- Defina um nome (evite espaços e números), insira uma descrição se desejar e marque a opção **Public**, para que ele possa ser compartilhado. Por fim, marque a caixa **Initialize this repository with a README**;
- Clique em **Create repository**.

A screenshot of the GitHub "Create new repository" form. The "Owner" is set to "victorsilva-ibmec" and the "Repository name" is "data-mining" with a green checkmark. A note says "Great repository names are short and memorable. Need inspiration? How about upgraded-octo-bassoon?". The "Description (optional)" field contains "Repositório para curso de Data Mining com Python". The "Public" option is selected, with the description "Anyone can see this repository. You choose who can commit." The "Private" option is also visible. A note says "Skip this step if you're importing an existing repository." The "Initialize this repository with a README" checkbox is checked, with the description "This will let you immediately clone the repository to your computer." Below this are dropdowns for "Add .gitignore: None" and "Add a license: None". At the bottom is a green "Create repository" button.

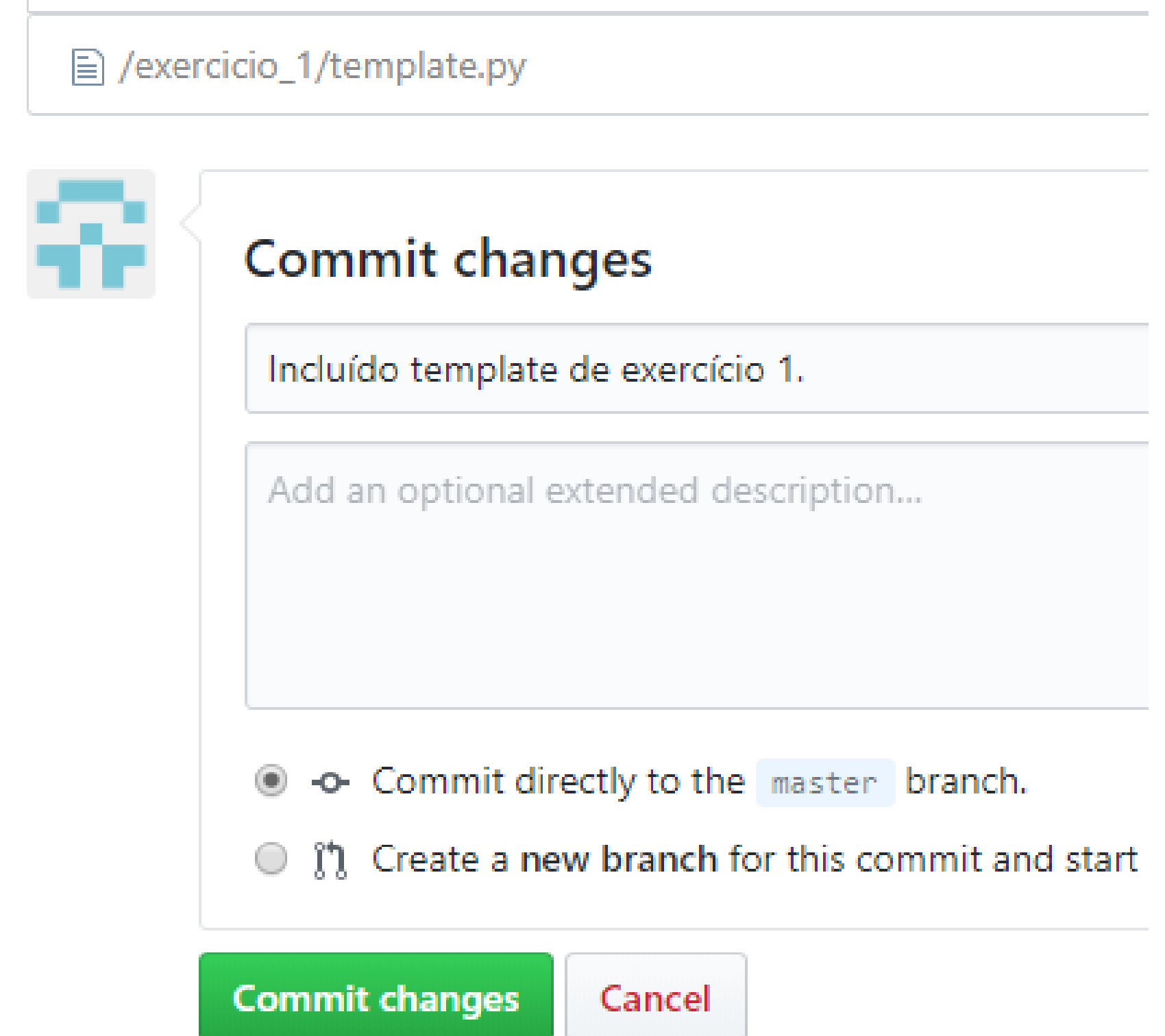
Algumas tarefas no GitHub

- **Baixando manualmente um repositório para a máquina:**
 - Na tela do seu repositório, clique em **Clone or download**;
 - Em seguida, clique em **Download ZIP**;
 - Com o arquivo .zip baixado, descompacte-o na sua pasta de projeto, substituindo arquivos antigos se necessário.



Algumas tarefas no GitHub

- **Fazendo um novo commit no seu repositório:**
 - Commits são alterações feitas no repositório. Podem conter um único arquivo ou vários. Caso um arquivo commitado já exista, o GitHub vai fazer um controle de versão, comparando as alterações entre a versão anterior e a que foi commitada;
 - Na tela do seu repositório, clique em **Upload files**;
 - Arraste para a tela os arquivos que deseja incluir;
 - Insira uma breve descrição do que está sendo commitado;
 - Deixe marcada a opção **Commit directly to the master branch**;
 - Clique em **Commit changes**.



The screenshot shows the GitHub 'Commit changes' interface. At the top, a file path `/exercicio_1/template.py` is displayed. Below it is a blue square icon with a white plus sign. The main section is titled 'Commit changes' and contains a text box with the message 'Incluído template de exercício 1.' Below this is a larger text box with the placeholder 'Add an optional extended description...'. At the bottom, there are two radio button options: 'Commit directly to the master branch.' (which is selected) and 'Create a new branch for this commit and start'. At the very bottom are two buttons: 'Commit changes' (green) and 'Cancel' (grey).

Algumas tarefas no GitHub

- **Submetendo um trabalho para revisão:**
 - Um **pull request** é o ato de submeter para aprovação as alterações ou inserções de código de um ou mais arquivos. A pessoa que abre um **pull request** sinaliza que gostaria de uma aprovação do conteúdo antes de ele ser, de fato, incorporado ao repositório;
 - No repositório desejado, clique na pasta em que você deseja incluir ou atualizar os arquivos;
 - Clique em **Upload files**;
 - Arraste para a tela o(s) arquivo(s) com a sua atualização, e na descrição explique o que está sendo feito. Em seguida, clique em **Commit changes**;
 - Na nova janela, insira um comentário e depois clique em **Create pull request**.

Alguns comandos do Git

- Apesar do Github fornecer recursos para operar com Git direto pelo navegador, esses recursos são limitados. Por exemplo, fazer checkout de um novo branch apenas pelo navegador pode ser bem complicado.
- Uma forma mais usual de se usar o Git é através de programas específicos para o computador.
- É bem comum utilizar os comandos do Git por linha de comando, porém existem bons programas para uso do Git através de uma interface gráfica, como o [Sourcetree](#).
- No slide a seguir serão apresentados alguns comandos comuns para o uso do Git pela linha de comando. Eles podem ser executados pelo Git Bash (programa instalado junto com o Git), ou pelo terminal.

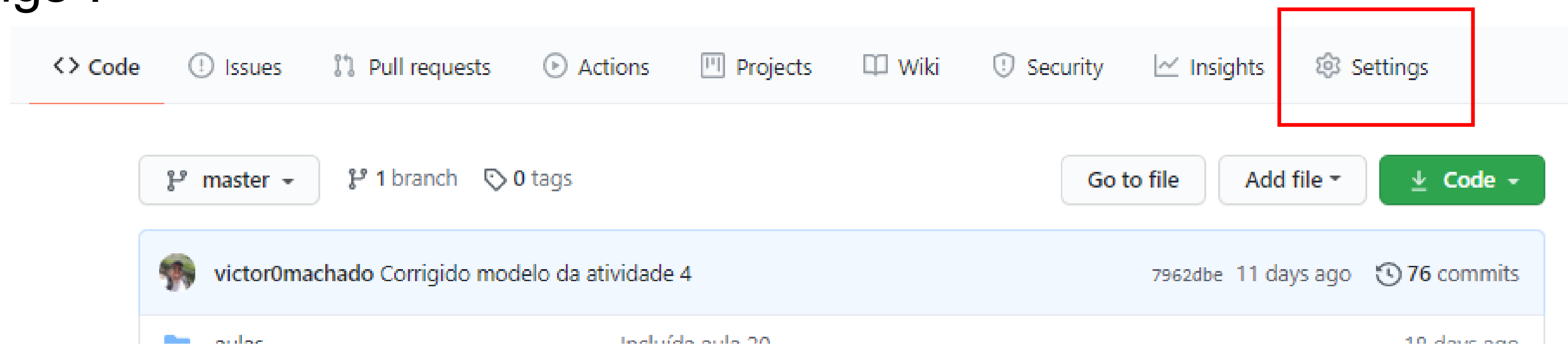
Alguns comandos do Git

- Uma observação, para facilitar as operações, é trabalhar sempre na raiz do repositório.
- Com o terminal na raiz do repositório, insira os seguintes comandos para obter os efeitos apresentados:

Comando	Efeito
git init	Inicia um repositório Git no diretório em questão
git status	Indica os status de arquivos modificados, adicionados ou removidos, além de arquivos preparados (staged)
git add <arquivo>	Prepara o arquivo mencionado
git add -u	Prepara todos os arquivos modificados (porém não faz nada com arquivos novos)
git add .	Prepara todos os arquivos (incluindo arquivos novos)
git commit -m "Mensagem"	Faz um commit dos arquivos preparados, incluindo a mensagem de commit definida
git restore <arquivo>	Desfaz modificações do arquivo que não foi preparado
git restore --staged <arquivo>	Desfaz a preparação do arquivo (porém mantém modificações)
git pull	No branch escolhido, atualiza as informações com o repositório remoto
git branch	Lista todos os branches armazenados localmente
git branch --show-current	Lista o branch atual
git branch -m <novo_nome>	Renomeia o branch atual (cuidado ao fazer isso para branches que já estão no repositório remoto!)
git checkout <branch>	Dá checkout no branch mencionado
git checkout -b <branch>	Cria um novo branch, com o nome mencionado, e dá checkout nele
git push	Envia os commits realizados localmente para o branch remoto

GitHub pages

- Uma boa forma de manter o seu portfólio sempre atualizado é com uma página pessoal, na qual você inclui seu currículo, projetos realizados, trabalhos, interesses pessoais e profissionais, e outras informações que achar pertinente.
- O Github possui uma forma muito simples de se criar um repositório que também serve como página pessoal.
- Para isso, crie um repositório normal, vá na página desse repositório e clique em “Settings”.



GitHub pages

- Nas configurações, desça a página até encontrar a seção “GitHub Pages”.

GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

Source
GitHub Pages is currently disabled. Select a source below to enable GitHub Pages for this repository. [Learn more.](#)

Theme Chooser
Select a theme to publish your site with a Jekyll theme using the gh-pages branch. [Learn more.](#)

- No campo “Source”, indique o branch que você quer que seja a sua página principal (usualmente é o branch master). Se quiser, escolha um tema da lista de temas gratuitos disponíveis e, em seguida, clique em “Save”.

GitHub pages

- A página terá uma atualização que mostrará a URL do site, além de incluir um campo no qual você pode inserir um domínio customizado, caso o tenha.

GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

Your site is ready to be published at <https://victor0machado.github.io/2020.2-logprog/>.

Source

Your GitHub Pages site is currently being built from the master branch. [Learn more.](#)

Branch: master ▾

/ (root) ▾

Save

Theme Chooser

Select a theme to publish your site with a Jekyll theme. [Learn more.](#)

Choose a theme

Custom domain

Custom domains allow you to serve your site from a domain other than victor0machado.github.io. [Learn more.](#)

Save

GitHub pages

- O site funciona como um repositório normal. Todas as páginas devem ser escritas em Markdown, que já vimos ao longo do curso.

```
# Boas-vindas

Vou atualizar essa página com os materiais das disciplinas que leciono no
IBMEC/RJ.

## Disciplinas

* [Algoritmos e Programação de Computadores](/courses/algprog.md)
* [Lógica e Programação de Computadores](/courses/logprog.md)
* [Data Mining com Python](/courses/datamining.md)

## Meus contatos

* E-mail: <victor.silva@professores.ibmec.edu.br>
* [Linkedin](https://www.linkedin.com/in/victormachadodasilva/)
* [Lattes](http://lattes.cnpq.br/1584907276781609)
```

Repositório público do Prof. Victor Machado

Material usado nas minhas disciplinas do
IBMEC/RJ

[View My GitHub Profile](#)

Boas-vindas

Vou atualizar essa página com os materiais das disciplinas que leciono no
IBMEC/RJ.

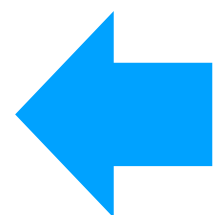
Disciplinas

- Algoritmos e Programação de Computadores
- Lógica e Programação de Computadores
- Data Mining com Python

Meus contatos

- E-mail: victor.silva@professores.ibmec.edu.br
- [Linkedin](#)
- [Lattes](#)

- O único arquivo exigido para o site é o **index.md**, que deve ficar na raiz do repositório. Novos arquivos e pastas podem ser criados se necessário, e a navegação é sempre relativa à raiz do repositório.



Fundamentos da Orientação a Objetos

Por que usar OO?

Segundo o Paradigma Procedural, é possível representar todo e qualquer processo do mundo real a partir da utilização de **apenas** três estruturas básicas:

- Sequência: Os passos devem ser executados um após o outro, linearmente. Ou seja, o programa seria uma sequência finita de passos. Em uma unidade de código, todos os passos devem ser feitos para se programar o algoritmo desejado;
- Decisão: Uma determinada sequência de código pode ou não ser executada. Para isto, um teste lógico deve ser realizado para determinar ou não sua execução. A partir disto, verifica-se que duas estruturas de decisão (também conhecida como seleção) podem ser usadas: a if-else e a switch.
- Iteração: É a execução repetitiva de um segmento (parte do programa). A partir da execução de um teste lógico, a repetição é realizada um número finito de vezes. Estruturas de repetição conhecidas são: for, foreach, while, do-while, repeat-until, entre outras (dependendo da linguagem de programação).

Por que usar OO?

Usar apenas essas três estruturas pode apresentar algumas limitações:

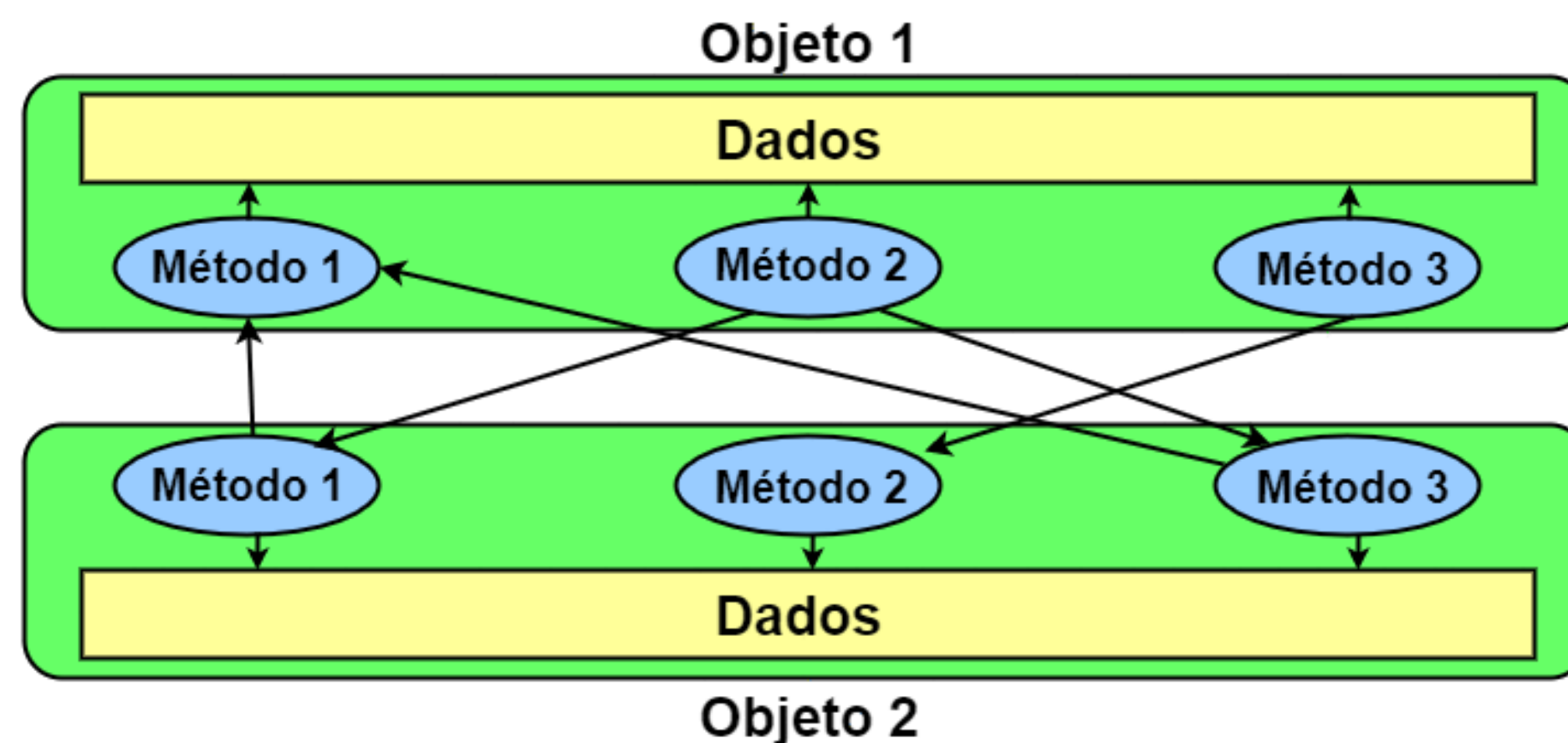
- Quanto mais complexo o programa se torna, mais difícil fica a manutenção de uma sequência organizada de código;
- Com esse paradigma é muito fácil deixar o código extenso e com muitas duplicações;
- Mesmo modularizações providas pelas linguagens podem deixar o código muito complexo.

Em resumo, a simplificação da representação das reais necessidades dos problemas a serem automatizados leva a uma facilidade de entendimento e representação. Porém, isso pode levar a uma complexidade de programação caso o nicho de negócio do sistema-alvo seja complexo.

Por que usar OO?

Ao contrário do paradigma procedural, a OO preconiza que os dados relativos a uma representação de uma entidade do mundo real devem somente estar juntos de suas operações, quais são os responsáveis por manipular - exclusivamente - tais dados.

Assim, há uma separação de dados e operações que não dizem respeito a uma mesma entidade. Todavia, se tais entidades necessitarem trocar informações, farão isto através da chamada de seus métodos, e não de acessos diretos a informações da outra.



Fundamentos da OO

Abstração:

- Processo pelo qual se isolam características de um objeto, considerando os que tenham em comum certos grupos de objetos.
- Não devemos nos preocupar com características menos importantes, ou seja, acidentais. Devemos, neste caso, nos concentrar apenas nos aspectos essenciais. Por natureza, as abstrações devem ser incompletas e imprecisas.



Fundamentos da OO

Abstração:

- Com a abstração dos conceitos, conseguimos reaproveitar, de forma mais eficiente, o nosso “molde” inicial, que pode ser detalhado conforme for necessário para o nosso caso específico.
- Os processos de inicialmente se pensar no mais abstrato e, posteriormente, acrescentar ou se adaptar são também conhecidos como **generalização** e **especialização**, respectivamente.



Fundamentos da OO

Reuso:

- Não existe pior prática em programação do que a repetição de código. Isto leva a um código frágil, propício a resultados inesperados. Quanto mais códigos são repetidos pela aplicação, mais difícil vai se tornando sua manutenção.
- O fato de simplesmente utilizarmos uma linguagem OO não é suficiente para se atingir a reusabilidade, temos de trabalhar de forma eficiente para aplicar os conceitos de **herança** e **associação**, por exemplo.
- Na herança, é possível criar classes a partir de outras classes. A classe filha, além do que já foi reaproveitada, pode acrescentar o que for necessário para si.
- Já na associação, o reaproveitamento é diferente. Uma classe pede ajuda a outra para poder fazer o que ela não consegue fazer por si só. Em vez de simplesmente repetir, em si, o código que está em outra classe, a associação permite que uma classe forneça uma porção de código a outra. Assim, esta troca mútua culmina por evitar a repetição de código.

Fundamentos da OO

Encapsulamento:

- Quando alguém se consulta com um médico, por estar com um resfriado, seria desesperados se ao final da consulta o médico entregasse a seguinte receita:

Receituário (Complexo)

- 400mg de ácido acetilsalicílico
- 1mg de maleato de dexclorfeniramina
- 10mg de cloridrato de fenilefrina
- 30mg de cafeína

Misturar bem e ingerir com água. Repetir em momentos de crise.

- A primeira coisa que viria em mente seria: onde achar essas substâncias? Será que é vendido tão pouco? Como misturá-las? Existe alguma sequência? Seria uma tarefa difícil - até complexa - de ser realizada. Mais simples do que isso é o que os médicos realmente fazer: passam uma cápsula onde todas estas substâncias já estão prontas. Ou seja, elas já vêm encapsuladas.

Fundamentos da OO

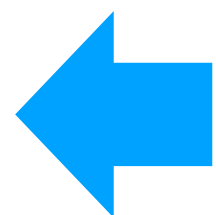
Encapsulamento:

- Com isso, não será preciso se preocupar em saber quanto e como as substâncias devem ser manipuladas para no final termos o comprimido que resolverá o problema. O que interessa é o resultado final, no caso, a cura do resfriado. A complexidade de chegar a essas medidas e como misturá-las não interessa. É um processo que não precisa ser do conhecimento do paciente.

Receituário (Encapsulado)

1 comprimido de Resfriol. Ingerir com água. Repetir em momentos de crise.

- Essa mesma ideia se aplica na OO. No caso, a complexidade que desejamos esconder é a de implementação de alguma necessidade. Com o encapsulamento, podemos esconder a forma como algo foi feito, dando a quem precisa apenas o resultado gerado.
- Uma vantagem deste princípio é que as mudanças se tornam transparentes, ou seja, quem usa algum processamento não será afetado quando seu comportamento interno mudar.



Introdução a Java

O que é Java?

A linguagem Java começou a ser concebida no início da década de 1990, com o objetivo de resolver alguns dos problemas comuns em programação na época, tais como:

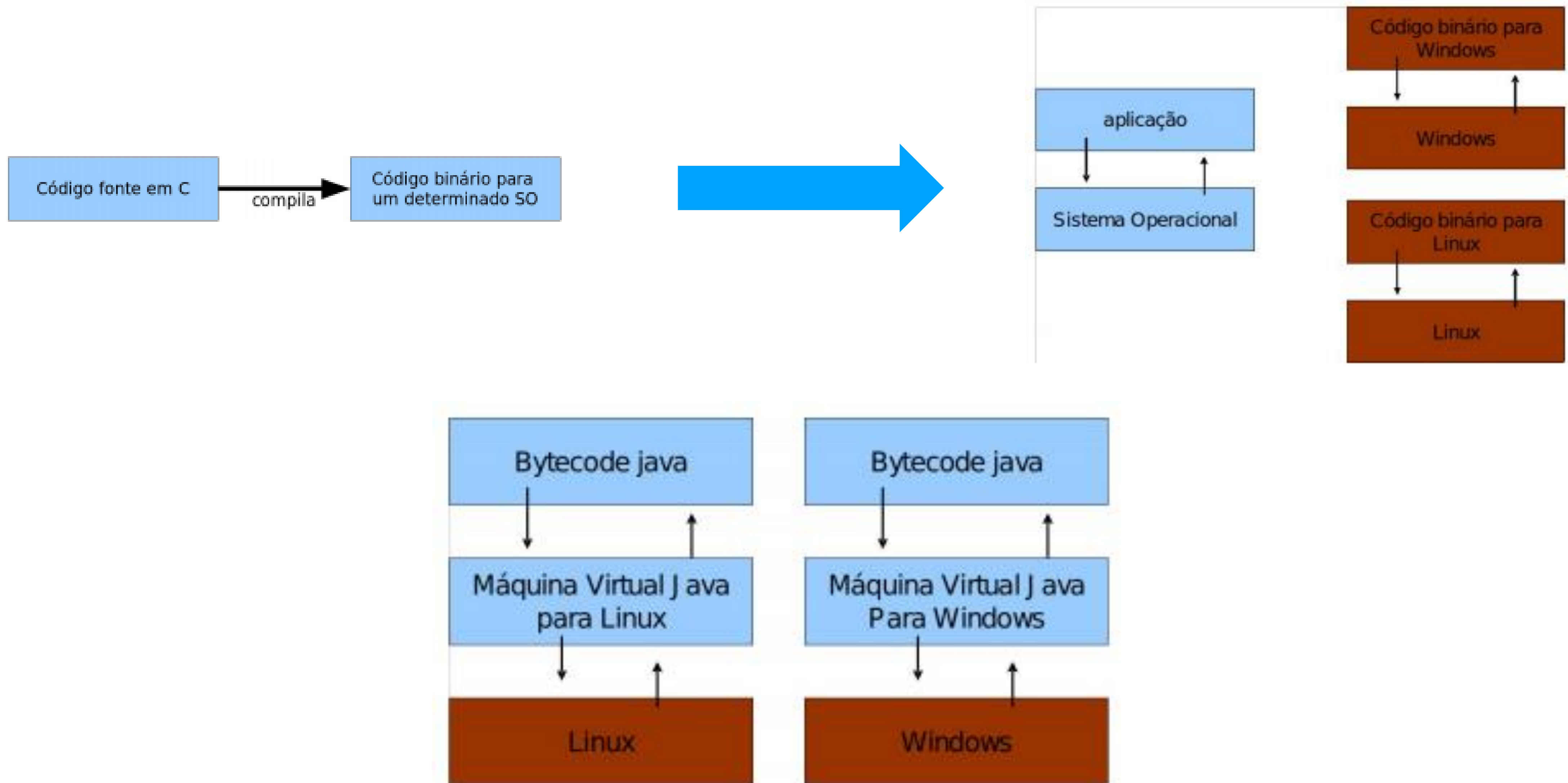
- Uso de ponteiros;
- Gerenciamento de memória;
- Organização;
- Falta de bibliotecas;
- Necessidade de reescrever parte do código ao mudar de sistema operacional;
- Custo financeiro de usar a tecnologia.

Uma das grandes motivações para a criação da plataforma Java era de que essa linguagem fosse usada em pequenos dispositivos, como TVs. Apesar disso a linguagem teve seu lançamento focado no uso em clientes web (browsers) para rodar pequenas aplicações (applets). Hoje em dia esse não é o grande mercado do Java: apesar de ter sido idealizado com um propósito e lançado com outro, o Java ganhou destaque no lado do servidor.

Uma breve história

- Time criado em 1992 na Sun, liderado por James Gosling, considerado o pai do Java.
- Ideia inicial de criar um interpretador para pequenos dispositivos, facilitando a reescrita de software para aparelhos eletrônicos.
- A ideia não deu certo, devido ao conflito de interesses e custos.
- Hoje, sabemos que o Java domina o mercado de aplicações para celulares com mais de 2.5 bilhões de dispositivos compatíveis, porém em 1994 ainda era muito cedo para isso.
- Com o advento da web, a ideia pode ser reaproveitada, já que na internet havia uma grande quantidade de sistemas operacionais e browsers, e com isso seria grande vantagem poder programar numa única linguagem, independente da plataforma.
- O Java 1.0 foi lançado focado em transformar o browser de apenas um terminal “burro” em uma aplicação que possa também realizar operações avançadas, e não apenas renderizar HTML.
- Em 2009 a Oracle comprou a Sun e fortaleceu a marca e a plataforma Java.

Máquina virtual?



Máquina virtual?

O conceito de máquina virtual é bem mais amplo que o de um interpretador:

- Uma VM tem tudo que um computador tem: é responsável por gerenciar memória, threads, a pilha de execução, etc.
- Sua aplicação roda sem nenhum envolvimento com o sistema operacional, então a JVM pode tirar métricas, decidir onde é melhor alocar a memória, entre outros.
- Se uma JVM termina abruptamente, só as aplicações que estavam rodando nela irão terminar.

Onde usar?

É preciso ficar claro que a premissa do Java não é a de criar sistemas pequenos, onde temos um ou dois desenvolvedores, mais rapidamente que linguagens como PHP, Perl, e outras.

O foco da plataforma é outro: aplicações de médio a grande porte, onde o time de desenvolvedores tem várias pessoas e sempre pode vir a mudar e crescer.

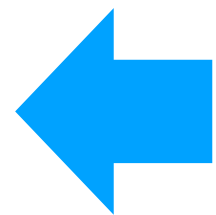
Não tenha dúvidas que criar a primeira versão de uma aplicação usando Java, mesmo utilizando IDEs e ferramentas poderosas, será mais trabalhoso que muitas linguagens script ou de alta produtividade. Porém, com uma linguagem orientada a objetos e madura como o Java, será extremamente mais fácil e rápido fazer alterações no sistema, desde que você siga as boas práticas e recomendações sobre design orientado a objetos.

Além disso, a quantidade enorme de bibliotecas gratuitas para realizar os mais diversos trabalhos é um ponto fortíssimo para adoção do Java: você pode criar uma aplicação sofisticada, usando diversos recursos, sem precisar comprar um componente específico, que costuma ser caro. O ecossistema do Java é enorme.

Onde usar?

Cada linguagem tem seu espaço e seu melhor uso. O uso do Java é interessante em aplicações que virão a crescer, em que a legibilidade do código é importante, onde temos muita conectividade e se há muitas plataformas (ambientes e sistemas operacionais) heterogêneas (Linux, Unix, OSX e Windows misturados).

Você pode ver isso pela quantidade enorme de ofertas de emprego procurando desenvolvedores Java para trabalhar com sistemas web e aplicações de integração no servidor.



Conceitos estruturais

Introdução

Embora a OO tenha vantagens em relação aos paradigmas que a precederam, existe uma desvantagem inicial: ser um modo mais complexo e difícil de se pensar. Isso pode ser atribuído à grande quantidade de conceitos que devem ser assimilados para podermos trabalhar orientado a objetos.

Vamos falar de quatro conceitos estruturais principais:

- A classe
- O atributo
- O método
- O objeto

Classes

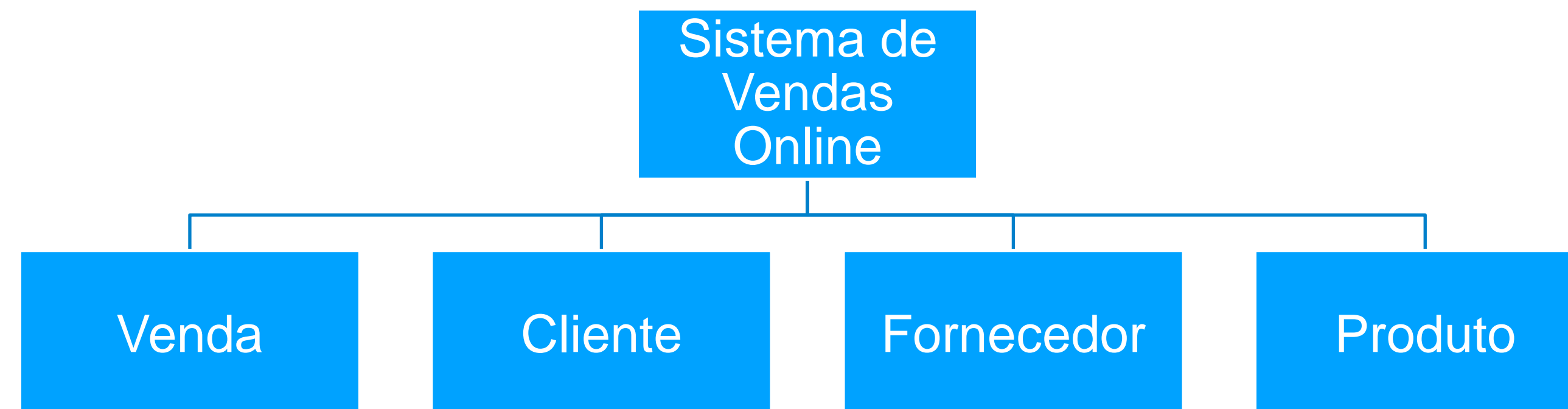
Apesar do paradigma ser nomeado como Orientado a Objetos, tudo começa com a definição de uma classe.

Antes mesmo de ser possível manipular objetos, é preciso definir uma classe, pois esta é a unidade inicial e mínima de código na OO. É a partir de classes que futuramente será possível criar objetos.

O objetivo de uma classe é definir, servir de base, para o que futuramente será o objeto.

- A classe é um “molde” que deverá ser seguido pelos objetos.

Uma classe pode ser definida como uma abstração de uma entidade, seja ela física (bola, pessoa, carro, etc.) ou conceitual (viagem, venda, estoque, etc.) do mundo real.



Classes

O nome de uma classe deve representar bem sua finalidade dentro do contexto ao qual ela foi necessária.

- Em um sistema de controle hospitalar, podemos ter uma classe chamada **Pessoa**;
- Em um sistema de ponto de vendas (PDV), também temos o conceito de **Pessoa**.

Entretanto, nota-se que o termo *pessoa* pode gerar uma ambiguidade, embora esteja correto.

Assim, recomenda-se ser mais específico na nomeação das classes, como **Médico**, **Paciente**, **Cliente**, **Vendedor**.

Embora possa parecer preciosismo, classes com nomes pobremente definidos podem dificultar o entendimento do código e até levar a erros de utilização. Pense bem antes de nomear uma classe.

Atributos

Após o processo inicial de identificar as entidades (classes) que devem ser manipuladas, começa a surgir a necessidade de detalhá-las.

- Quais informações devem ser manipuladas através desta classe?

Devemos, então, caracterizar as classes definidas. Essas características é que vão definir quais informações as classes poderão armazenar e manipular. Na OO, estas características e informações são denominadas de **atributo**.

Essa definição deixa bem claro que os atributos devem ser definidos dentro da classe. É a partir do uso de atributos que será possível caracterizar (detalhar) as classes.

Assim como nas classes, os atributos podem ser representados a partir de substantivos. Além destes, podemos também usar adjetivos. Pensar em ambos pode facilitar o processo de identificação dos atributos.

Atributos

Classe **Paciente** em um sistema hospitalar → quais seriam os possíveis atributos?

- Nome
- CPF
- Data de nascimento
- Histórico médico

Todos estes são substantivos, mas alguns de seus valores poderiam ser adjetivos.

Quanto mais for realizado o processo de caracterização, mais detalhada será a classe e, com isso, ela terá mais atributos.

Porém, é preciso ter parcimônia no processo de identificação dos atributos!

- Hobby?
- Possui carro?

Atributos

Nem sempre uma informação, mesmo sendo importante, deve ser transformada em um atributo. Por exemplo, **idade**. Ter que recalcular a idade toda vez que a pessoa fizer aniversário é custoso e propenso a erros.

Neste caso, seria melhor usar o que é conhecido como **atributo calculado** ou **atributo derivado**. A idade não se torna um atributo em si, mas tem seu valor obtido a partir de um método.

Não diferentemente de linguagens estruturadas, um atributo possui um tipo. Como sua finalidade é armazenar um valor que será usado para caracterizar a classe, ele precisará identificar qual o tipo do valor armazenado em si. Linguagens orientadas a objetos proveem os mesmos tipos de dados básicos - com pequenas variações - que suas antecessoras.

Atributos

A nomeação de atributos deve seguir a mesma preocupação das classes: deve ser o mais representativo possível. Nomes como **qtd**, **vlr** devem ser evitados.

Esses nomes eram válidos na época em que as linguagens de programação e os computadores que as executavam eram limitados, portanto deveríamos sempre abreviar os nomes das variáveis.

Entretanto, atualmente não temos essa limitação, e os editores modernos possuem recursos para reaproveitar nomes de atributos e variáveis sem precisar digitar tudo novamente. Portanto, passe a usar **quantidade** e **valor**.

Utilize nomes claros. Evite, por exemplo, o atributo **data**. Uma data pode significar várias coisas: nascimento, morte, envio de um produto, cancelamento de venda. Escreva exatamente o que se deseja armazenar nesse atributo: **dataNascimento**, **dataObito**, etc.

Métodos

Tendo identificado a classe com seus atributos, as seguintes perguntas podem surgir:

- Mas o que fazer com eles?
- Como utilizar a classe e manipular os atributos?

É nessa hora que o método entra em cena. Este é responsável por identificar e executar as operações que a classe fornecerá. Essas operações, via de regra, têm como finalidade manipular os atributos.

Para facilitar o processo de identificação dos métodos de uma classe, podemos pensar em verbos. Isso ocorre devido à sua própria definição: **ações**. Ou seja, quando se pensa nas ações que uma classe venha a oferecer, estas identificam seus métodos.

Métodos

No processo de definição de um método, a sua assinatura deve ser identificada. Esta nada mais é do que o nome do método e sua lista de parâmetros. Mas como nomear os métodos? Novamente, uma expressividade ao nome do método deve ser fornecida, assim como foi feito com o atributo.

Por exemplo, no contexto do hospital, imagine termos uma classe **Procedimento**, logo, um péssimo nome de método seria **calcular**. Calcular o quê? O valor total do procedimento, o quanto cada médico deve receber por ele, o lucro do plano? Neste caso, seria mais interessante **calcularTotal**, **calcularGanhosMedico**, **calcularLucro**.

Veja que, ao lermos esses nomes, logo de cara já sabemos o que cada método se propõe a fazer. Já a lista de parâmetros são informações auxiliares que podem ser passadas aos métodos para que estes executem suas ações. Cada método terá sua lista específica, caso haja necessidade. Esta é bem livre e, em determinados momentos, podemos não ter parâmetros, como em outros podemos ter uma classe passada como parâmetro, ou também tipos primitivos e classes ao mesmo tempo.

Métodos

Há também a possibilidade de passarmos somente tipos primitivos, entretanto, isto remete à programação procedural e deve ser desencorajado. Via de regra, se você passa muitos parâmetros separados, talvez eles pudessem representar algum conceito em conjunto. Neste caso, valeria a pena avaliarmos se não seria melhor criar uma classe para aglutiná-los.

Por fim, embora não faça parte de sua assinatura, os métodos devem possuir um retorno. Se uma ação é disparada, é de se esperar uma reação. O retorno de um método pode ser qualquer um dos tipos primitivos vistos na seção sobre atributos.

Além destes, o método pode também retornar qualquer um dos conceitos (classes) que foram definidos para satisfazer as necessidades do sistema em desenvolvimento, ou também qualquer outra classe - não criada pelo programador - que pertença à linguagem de programação escolhida.

Dois métodos especiais

Em uma classe, independente de qual conceito ela queira representar, podemos ter quantos métodos forem necessários. Cada um será responsável por uma determinada operação que a classe deseja oferecer. Além disso, independente da quantidade e da finalidade dos métodos de uma classe, existem dois especiais que toda classe possui: o construtor e o destrutor.

Características do construtor:

- Responsável por criar objetos a partir da classe em questão;
- Prover valores iniciais para o objeto;
- Possui nome igual ao da classe;
- Não possui retorno, ao contrário dos outros métodos (omitimos inclusive o **void**);
- Está presente de forma implícita em linguagens como Java e C#;
- Construtores implícitos possuem como assinatura o mesmo nome da classe e não possuem parâmetros.

Dois métodos especiais

Características do destrutor:

- Destrói o objeto criado a partir da classe;
- Seu nome padrão em Java é **finalize**, e retorna o tipo **void**;
- Não possui parâmetros;
- Usar um destrutor libera possíveis recursos do computador;
- O Java fornece um destrutor implícito para toda classe;
- Não devemos utilizar diretamente, para isso o Java tem o **Garbage Collector**.

Garbage Collector

- É um programa usual em toda linguagem orientada a objetos, como Smalltalk 80, Java e C#;
- No caso do Java, é um programa que roda dentro da JVM;
- O Garbage Collector é responsável por automaticamente identificar objetos que não estão sendo mais usados e eliminá-los;
- No momento da eliminação, o Garbage Collector utiliza os destrutores definidos nas classes;
- Ele possui algoritmos de identificação de objetos ociosos e elimina os que não são mais usados, tirando a responsabilidade do desenvolvedor.

Sobrecarga de métodos

Muitas vezes, é preciso que um mesmo método possua entradas (parâmetros) diferentes. Isso ocorre porque ele pode precisar realizar operações diferentes em determinado contexto. Este processo é chamado de **sobrecarga de método**.

Para realizá-la, devemos manter o nome do método intacto, mas alterar sua lista de parâmetros. Podem ser acrescentados ou retirados parâmetros para assim se prover um novo comportamento. Por exemplo, se uma determinada aplicação tivesse uma classe para representar um quadrilátero, ela deveria se chamar **Quadrilátero** e possuir o método **calcularArea**, seguindo as boas práticas já citadas. Mas sabemos que um quadrado, retângulo, losango e trapézio também são quadriláteros.

Mais interessante do que possuir um método para cada figura (**calcularAreaQuadrado**, **calcularAreaLosango**, etc.) é definir o mesmo método **calcularArea** com uma lista de parâmetros que se adeque a cada um desses quadriláteros.

Sobrecarga de métodos

```
class Quadrilatero {
    // área do quadrado
    double calcularArea(double lado) {
        return lado * lado;
    }

    // área do retângulo
    double calcularArea(double baseMaior, double baseMenor) {
        return baseMaior * baseMenor;
    }

    // área do trapézio
    double calcularArea(double baseMaior, double baseMenor, double altura) {
        return ((baseMaior * baseMenor) * altura) / 2;
    }

    // área do losango
    double calcularArea(float diagonalMaior, float diagonalMenor) {
        return diagonalMaior * diagonalMenor;
    }
}
```

Sobrecarga de métodos

Caso haja necessidade, os tipos também podem ser mudados.

Sempre que a lista de parâmetros muda - seja acrescentando ou eliminando parâmetros, mudando seus tipos e até mesmo sua sequência -, estaremos criando sobrecargas de um método. Mas lembre-se de que o nome dele deve permanecer intacto.

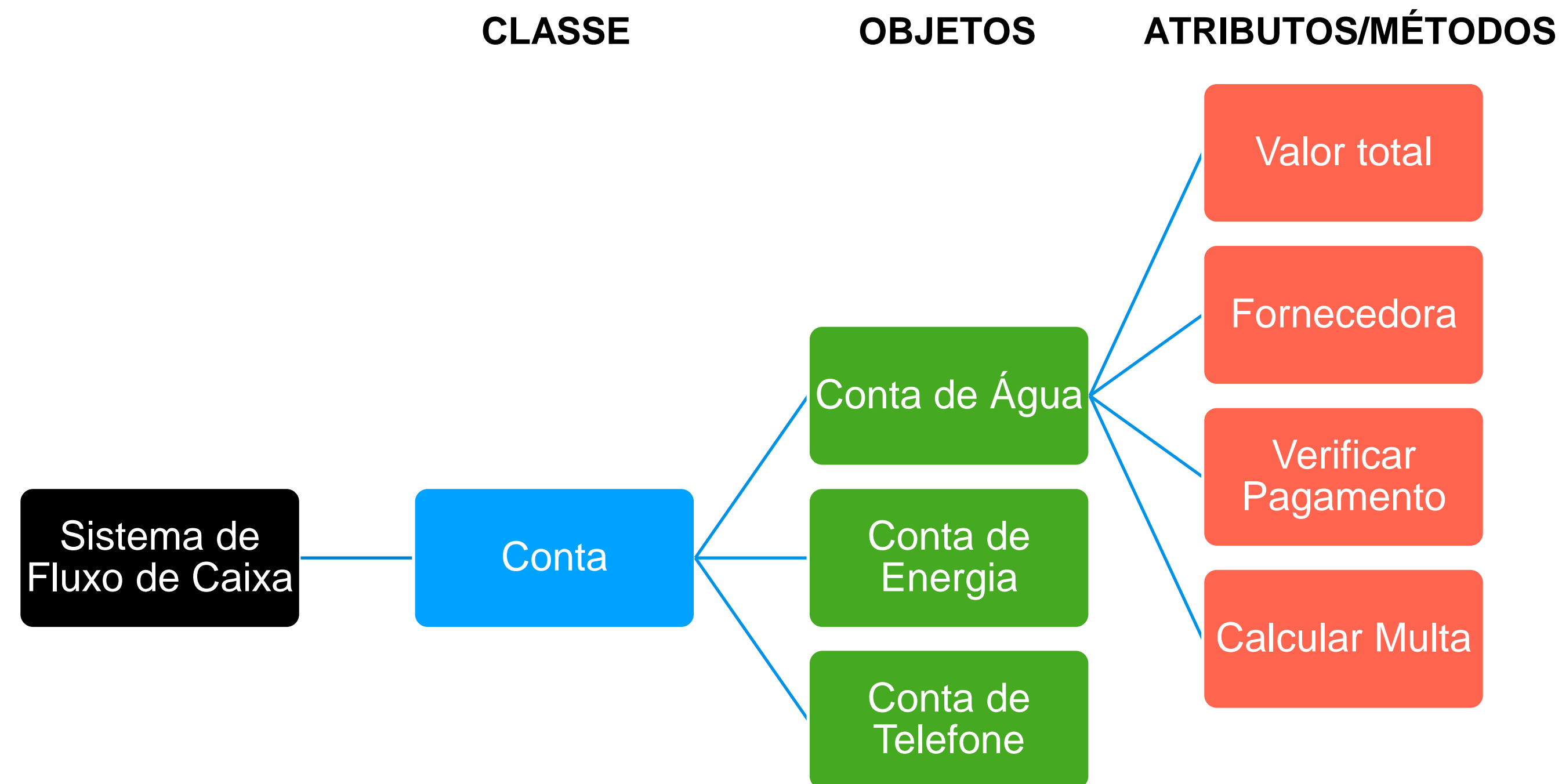
A vantagem de usar a sobrecarga não se limita à "facilidade" de se manter o mesmo nome do método. Na verdade, existe uma questão conceitual, que é manter a abstração.

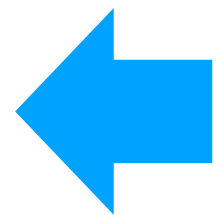
Assim, no caso de nosso exemplo do quadrilátero, a abstração é calcular sua área, independente de sua real forma. Deste modo, a sobrecarga possibilitou que tal cálculo pudesse receber os devidos parâmetros de acordo com a sua necessidade - no caso, a forma do quadrilátero - e, mesmo assim, manteve-se a abstração-alvo, que é calcular a área.

Objetos

Objeto é a instanciação de uma classe.

Como já explicado, a classe é a abstração base a partir da qual os objetos serão criados. Quando se usa a OO para criar um software, primeiro pensamos nos objetos que ele vai manipular/representar. Tendo estes sido identificados, devemos então definir as classes que servirão de abstração base para que os objetos venham a ser criados (instanciados).





Introdução a UML

Origens

A UML, ou *Unified Modeling Language*, nasceu em 1994 a partir da junção das metodologias de Grady Booch, James Rumbaugh e Ivar Jacobson. Os três autores desenvolveram, em anos anteriores, várias metodologias para modelar projetos de software, principalmente aqueles implementados utilizando o paradigma Orientado a Objetos, daí o significado da sigla.

A UML é uma linguagem para especificar, visualizar, construir e documentar os artefatos de software. Trata-se de uma notação que normatiza um conjunto de diagramas. Também pode ser usada como ferramenta para modelagem de negócios.

Atualmente, é mantida como um padrão de indústria da **Object Management Group**, e se encontra na versão 2.5.1. A documentação oficial da UML pode ser encontrada na página <https://www.omg.org/spec/UML/About-UML/>.

Objetivos

A UML tem como objetivos:

- Padronizar a comunicação entre equipe através de uma linguagem visual, independente do processo de desenvolvimento de sistemas utilizado;
- Viabilizar a documentação de ideias para resolver problemas recorrentes, os chamados **design patterns**;
- Estabelecer a associação explícita entre o conceitual e a implementação

Um **design pattern**, ou padrão de projeto, descreve uma solução geral reutilizável para um problema recorrente no desenvolvimento de sistemas de software orientados a objeto.

Não é um código final, é uma descrição - ou modelo - de como resolver o problema do qual trata, que pode ser usada em muitas situações diferentes.

Os **design patterns** normalmente definem as relações e interações entre as classes ou objetos sem especificar os detalhes das classe ou objetos envolvidos - estão em um nível de generalidade mais alto.

Diagramas estruturais

- Diagrama de Classes: é uma representação da estrutura e relações das classes que servem de modelo para objetos
- Diagrama de Objetos: é uma variação do diagrama de classes e utiliza quase a mesma notação. A diferença é que o diagrama de objetos mostra os objetos que foram instanciados das classes. O diagrama de objetos poderia representar o perfil do sistema em um certo momento de sua execução.
- Diagrama de Componentes: ilustra como as classes deverão se encontrar organizadas através da noção de componentes de trabalho. Componente é uma peça física distribuível e substituível de código e que contém elementos que apresentam um conjunto de *interfaces* requeridas e fornecidas.
- Diagrama de Estruturas Compostas: destina-se à descrição dos relacionamentos entre os elementos. Utilizado para descrever a colaboração interna de classes, *interfaces* ou componentes de forma a especificar uma funcionalidade.
- Diagrama de Pacotes: descreve os pacotes ou pedaços do sistema divididos em agrupamentos lógicos mostrando as dependências entre estes, ou seja, pacotes podem depender de outros pacotes.
- Diagrama de Implantação: descreve os componentes de hardware e software e sua interação com outros elemento de suporte ao processamento.
- Diagrama de Artefatos: mostra um conjunto de artefatos e seus relacionamentos com outros artefatos e com classes que implementam.

Diagramas comportamentais

- Diagrama de Casos de Uso: descreve a funcionalidade proposta para um novo sistema que será projetado.
- Diagrama de Sequência: também chamado de diagrama de sequência de mensagens, representa a sequência de processos (mais especificamente, de mensagens transmitidas entre objetos) em um caso de uso.
- Diagrama de Comunicação: dá ênfase à organização estrutural dos objetos que enviam e recebem mensagens, mostrando o conjunto de papéis e as mensagens enviadas e recebidas pelas instâncias.
- Diagrama de Estados: mostra uma máquina de estados, que consiste de estados, transições, eventos e atividades, com a finalidade de modelarem o comportamento de uma interface, classe ou colaboração. Os diagramas de estado dão ênfase ao comportamento de um objeto, solicitado por eventos, que é de grande ajuda para a modelagem de sistemas reativos.
- Diagrama de Atividades: representa os fluxos conduzidos por processamentos. É, essencialmente, um gráfico e fluxo, mostrando o fluxo de controle de uma atividade para outra.
- Diagrama de Temporização: apresenta o comportamento dos objetos e sua interação em uma escala de tempo, focalizando as condições que mudam no decorrer desse período.

Diagramas de Classes

Os diagramas de classes proveem as bases de qualquer metodologia de análise e projeto de sistemas computacionais orientada a objetos. Não há, portanto, um sistema minimamente documentado para o qual não tenhamos desenvolvido um diagrama de classes.

Os diagramas de classes em modelos de sistemas podem especificar as perspectivas conceitual, de especificação e de implementação. Cada perspectiva representa o problema ou a solução com graus diferentes de abstração:

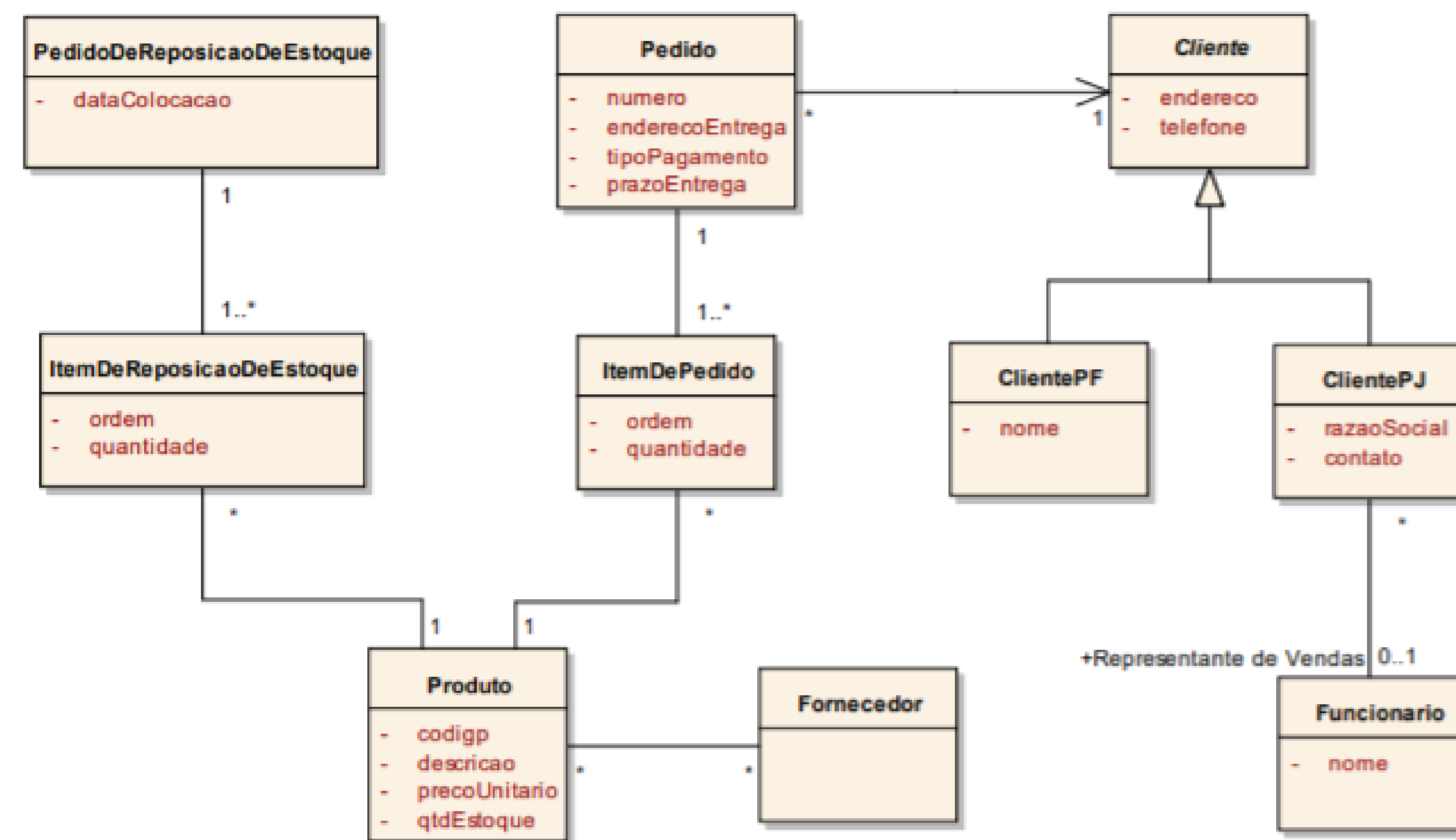
- Um diagrama de classes conceitual contém apenas classes de conceito (daí o nome conceitual), dotando o modelo de alto grau de abstração, ou seja, onde os detalhes são esquecidos. Modelos conceituais especificam parte do problema a ser solucionado pelo sistema.
- No extremo oposto, na perspectiva de implementação, representamos todos os detalhes necessários para a implementação do sistema considerando todas as características das tecnologias escolhidas. Dizemos que, na perspectiva de implementação, estamos no nível de abstração zero, em que nada é esquecido. Esses diagramas são bastante extensos e complexos por detalharem as minúcias da solução que os projetistas conceberam.
- A perspectiva de especificação se situa entre essas duas, ou seja, começa no instante em que adicionamos ao modelo conceitual completo a primeira classe ou detalhe da solução que o projetista está dando para o problema e termina quando obtemos o modelo de implementação. O nome *especificação* está associado à fase em que o projetista especifica a solução que está dando para o problema.

Diagramas de Classes - Classes

Focaremos neste curso apenas no nível conceitual, discutindo assuntos do nível de especificação quando for pertinente. Os diagramas de classes que elaboraremos terão o objetivo de representar os conceitos de negócios, seus relacionamentos e restrições (regras de negócio).

Os diagramas de classes compõem-se de classe, dos relacionamentos entre elas e de restrições do negócio.

A classe é o elemento-chave em um diagrama de classes. No nível conceitual, uma classe representa um conceito do negócio. Assim, como mostrado na figura abaixo, **Pedido**, **Cliente**, etc. são conceitos que fazem parte de um contexto típico em uma empresa fictícia - à qual demos o nome de ZYX - que lida com pedidos feitos por sua clientela.



Diagramas de Classes - Classes

As classes conceituais, também chamadas de classes de entidades, são entidades das quais nos interessa ter suas propriedades armazenadas em um arquivo convencional (pastas suspensas e fichas), em um sistema manual, ou no banco de dados de sistema informatizado.

Os conceitos representados pelas classes conceituais são de pleno conhecimento dos participantes do negócio, ou seja, no exemplo anterior, nosso cliente conhece bem os conceitos de **Pedido**, **Fornecedor**, **Pedido de Reposição de Estoque**, etc.

Nos diagramas de classes, as classes são representadas por retângulos com um ou mais compartimentos, dependendo do nível de detalhamento. O nome da classe é colocado no primeiro compartimento em negrito e centralizado. Recomenda-se que os nomes sejam substantivos no singular ou expressões breves, preferencialmente com base no jargão usado no negócio. Os nomes são únicos em um *espaço de nomes* (*namespace*, em inglês), pois identificam univocamente as classes no modelo.

Um espaço de nomes é um local abstrato que fornece contexto para os itens colocados nele. Um espaço de nomes é um conjunto abstrato de coisas, um container. Em um dado espaço de nomes, cada elemento nele contido precisa ter um identificador - um nome - que deve ser único nesse espaço. Identificadores podem ser repetidos em espaços de nomes distintos, entretanto, quando compostos com o respectivo espaço de nomes, se tornam únicos no domínio.

Diagramas de Classes - Classes

Uma dada classe pode ter mais de uma cópia em um mesmo (ou em outro) diagrama do modelo. A propósito, podemos, sim, ter mais de um diagrama de classes compondo o modelo de classes de um sistema. Isso, por sinal, é até bastante usual, especialmente em sistemas grandes.

Durante o desenvolvimento do modelo de classes, o analista deve ter preocupação com o nome que dará a cada classe; o nome, embora deva ser um substantivo ou uma expressão breve, deve transmitir bem o conceito que a classe representa. E modelos conceituais, o trabalho de dar nomes às classes é, de certa forma, facilitado, já que os nomes devem ser preferencialmente retirados do jargão do negócio.

Uma boa técnica para descobrirmos se determinada classe é ou não conceitual é perguntar sobre o conceito a ela relacionado ao cliente especialista do negócio que está sendo entrevistado. Se ele não souber responder a respeito, não conhece, nunca usou aquele termo no seu dia-a-dia, provavelmente a classe não deverá fazer parte do modelo conceitual.

Objetos são instâncias ou ocorrências das classes. Cada pedido da coleção de pedidos feitos à empresa ZYX, por exemplo, é uma instância da classe **Pedido**. As classes que podemos instanciar, ou seja, das quais podemos solicitar a criação de objetos, são chamadas de **classes concretas**. A classe **Pedido** é, portanto, um exemplo de classe concreta. Outros exemplos de classes concretas na figura anterior são **ItemDePedido**, **Produto**, **Funcionario**, **ClientePF** e **ClientePJ**.

Diagramas de Classes - Atributos

As informações a respeito dos conceitos (por exemplo, o endereço e o telefone do cliente na figura anterior) que gostaríamos de manter em um cadastro são chamadas de **atributos** das classes. A relação de atributos é colocada no segundo compartimento do retângulo da classe, justificada à esquerda. A necessidade de mantermos valores de atributos para as ocorrências de uma determinada classe justifica, como já mencionamos, a existência dessa classe, ou seja, se desejamos armazenar as informações sobre uma categoria de coisas em um negócio, provavelmente essa categoria se tornará uma classe no modelo de classes do sistema.

Os atributos que desejamos relacionar no modelo conceitual são aqueles que os especialistas do negócio mencionam. Não é certo relacionarmos atributos nessa fase além daqueles que os especialistas julgam necessários. Podemos, claro, lembrá-los de alguns atributos que são típicos, mas eles é que dão a palavra final sobre a necessidade ou não. Também não é certo nos preocuparmos com detalhes, como os tipos dos atributos, se cadeias de caracteres, se numéricos e com qual precisão numérica etc. No modelo da figura anterior, a classe **Fornecedor** não possui atributos relacionados, o que sugere que ainda não terminamos o modelo conceitual.

Diagramas de Classes - Atributos

Os nomes dos atributos são suficientes nos modelos conceituais. Mais adiante, no ciclo de desenvolvimento do sistema, mais especificamente na fase de projeto, devemos completar os nomes dos atributos com outros detalhes. Além dos nomes, a notação UML um pouco mais completa para rótulos de atributos (atributos são referenciados pela UML como propriedades) é:

[visibilidade][/]nome:tipo[multiplicidade][= valorDefault]

onde os valores entre "[" e "]" nem sempre ocorrem e:

- *Visibilidade* é o caractere "-" (para privado), "+" (para público) ou "#" (para protegido), que indica se o atributo é visível ou não de outros objetos. Atributos privados só podem ser acessados (consultados diretamente e/ou modificados) pelos objetos que os contêm. Atributos públicos podem ser acessados por outros objetos e atributos protegidos são acessados pelos objetos que os contêm ou por objetos instanciados de classes especializadas. A visibilidade deve ser omitida no modelo conceitual;
- A "/" antes do nome indica que o atributo é derivado, ou seja, seu valor pode ser determinado por um algoritmo a partir de outro(s). Por exemplo, se tivermos os atributos **idade** e **dataDeNascimento**, o atributo idade deve ser precedido da "/", já que a idade de um indivíduo pode ser determinada a partir da sua data de nascimento;
- *Tipo* define o tipo de dados: inteiro, real, cadeia de caracteres, data etc.;
- *Multiplicidade* indica as possíveis cardinalidades para a ocorrência do atributo. Se a multiplicidade é omitida, significa que ela é exatamente 1. Veremos multiplicidades em maiores detalhes um pouco mais adiante;
- O *valor default* é o valor que o atributo assume de início.

Diagramas de Classes - Atributos

A visibilidade à qual nos referimos anteriormente tem a ver com a ideia de encapsulamento, que recomenda deixarmos escondido o que não é preciso ser mostrado. Aumentamos a facilidade com que damos manutenção em um sistema (manutenibilidade), definindo como privado o maior número possível de atributos (e operações, como veremos adiante). Mesmo os atributos que precisam ser "vistos" por outros objetos devem ser definidos como privados e devem ser criadas operações públicas de acesso para leitura e escrita a eles. Por meio dessas operações garantimos acessos mais "policiados" aos atributos.

Diagramas de Classes - Métodos

O terceiro compartimento do retângulo da classe contém a lista de operações que os objetos da classe implementam para realizar suas responsabilidades. É praxe, no entanto, que esses compartimentos fiquem vazios no modelo de análise, pois as operações normalmente só começam a ser descobertas quando iniciamos o nível de especificação, ao elaborar os diagramas de sequência.

Para o propósito de nosso curso, a notação UML suficientemente completa para os rótulos de operações é:

[visibilidade]nome([listaDeParametros])(: tipoDeRetorno)

onde os valores entre "[" e "]" nem sempre ocorrem e:

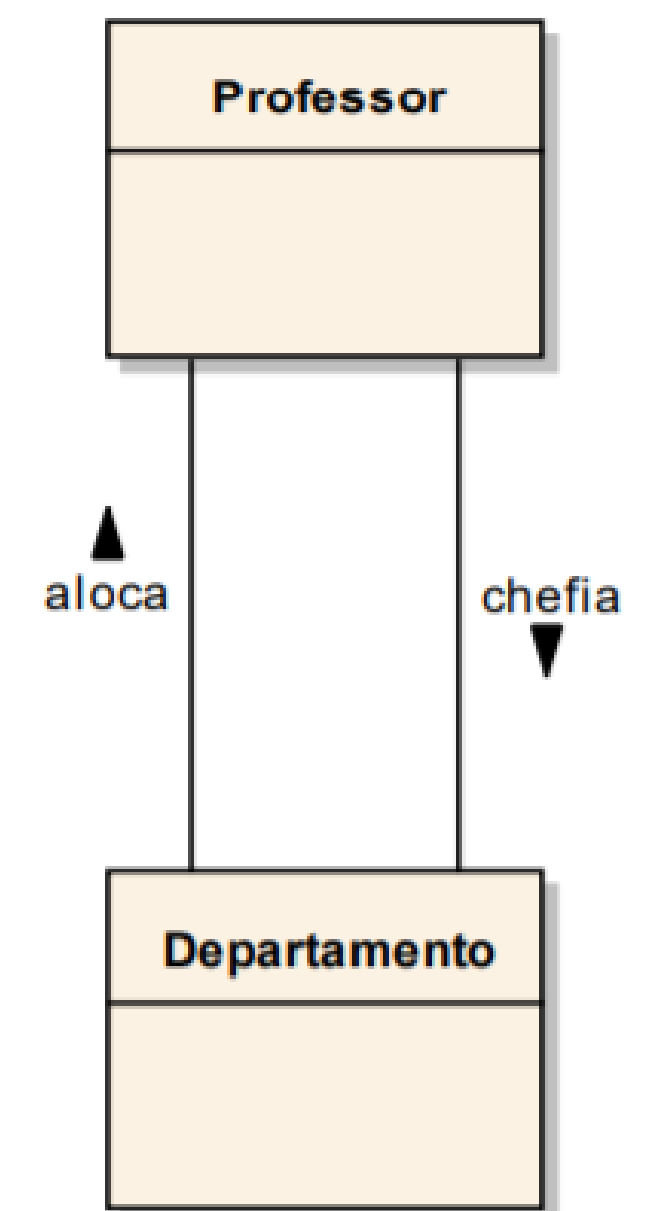
- *Visibilidade* é o caractere "-" (privado), "+" (público) ou "#" (protegido) que indica que a operação é visível ou não de outros objetos, do mesmo jeito que com os atributos;
- o nome da operação também é formado segundo o padrão *CamelCase*;
- Tipo define o tipo de retorno da operação: inteiro, real, cadeia de caracteres, data, etc.;
- A lista de parâmetros é formada pelos parâmetros de entrada e saída separados por vírgulas, da seguinte forma:
[direção]nome: tipo, onde *direção* (opcional) é "in" ou "out" ou "inout", significando parâmetro de entrada, de saída e de entrada e saída, respectivamente. O nome e o tipo são da mesma forma que nos atributos.

Diagramas de Classes - Relacionamentos

Os conceitos identificados em um dado contexto invariavelmente se ligam de alguma forma. Relacionamentos em diagramas de classes expressam essas ligações, que, por também fazerem parte do conhecimento a respeito do negócio, precisam ser capturadas e especificadas no modelo de classes.

Associação entre Classes:

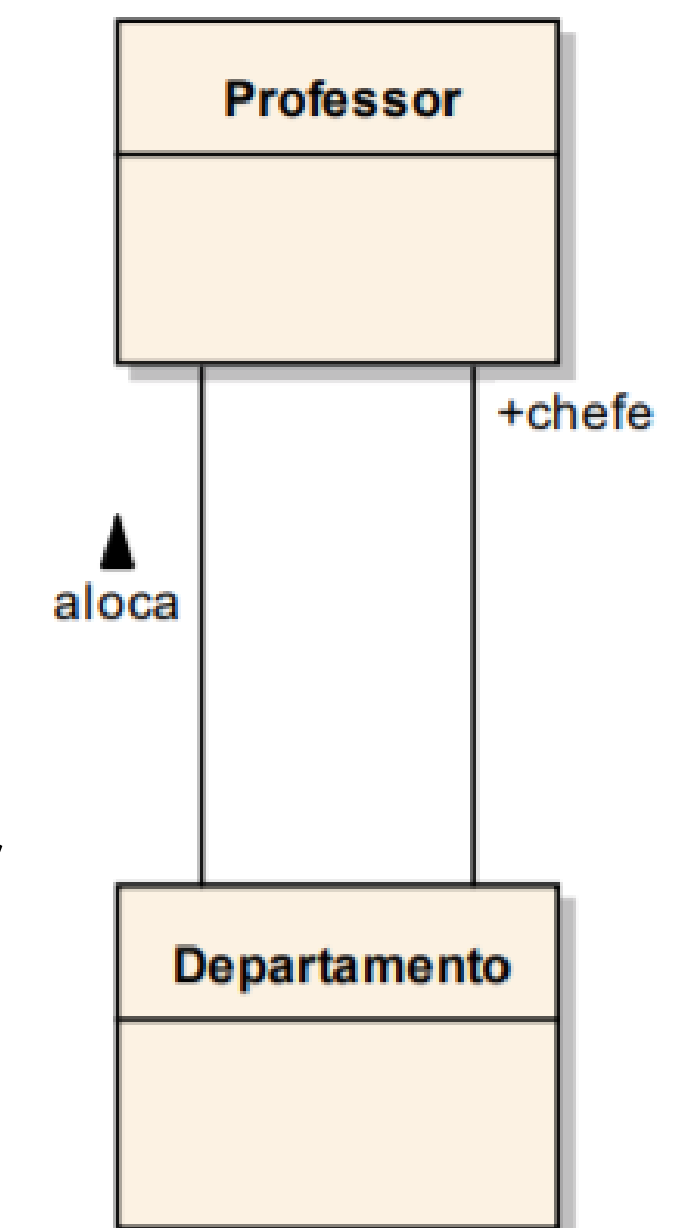
- Associações são o tipo mais comum de relacionamentos entre classes em um diagrama de classes. No diagrama do nosso exemplo, está especificado que um determinado cliente pode estar associado a qualquer número de pedidos (inclusive zero, ou seja, empresas ou pessoas físicas são consideradas clientes mesmo não tendo feito qualquer pedido), e um determinado pedido está associado a somente um cliente.
- As associações são representadas nos diagramas de classes por segmentos de retas, poligonais ou arcos que ligam as classes associadas. Uma associação é opcionalmente rotulada com o nome da associação, que deve ser colocado sempre que o significado da associação não é claro no diagrama.
- O nome da associação deve vir acompanhado do símbolo de sentido de leitura (só a ponta cheia de seta) e deve exprimir bem o significado da associação, sendo preferencialmente um verbo na voz ativa. A vantagem de usarmos verbos na voz ativa em nomes de associações é que também podemos ler o nome da associação no sentido contrário ao da seta, bastando mudar o verbo para a voz passiva (exemplos: "professor é alocado a departamento"; "departamento é chefiado por professor").



Diagramas de Classes - Relacionamentos

Associação entre Classes (cont.):

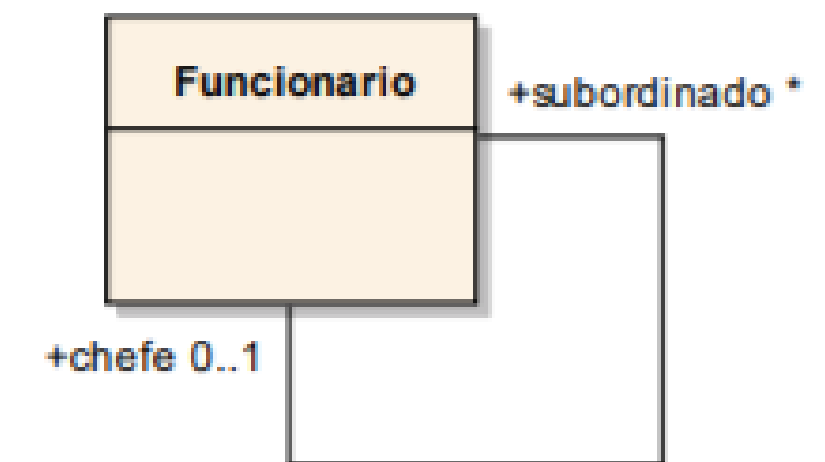
- As pontas das associações (no pontos onde elas se encontram com as caixas das classes) chamam-se papéis, que também podem ser usados para dar nomes aos papéis que as classes representam nas associações. Quando não especificamos o rótulo do papel (que deve ficar bem junto do ponto onde a associação encontra a caixa da classe), este leva o nome da classe.
- As pontas de uma associação entre classes devem especificar também as multiplicidades, que indicam quantas instâncias das classes podem participar da associação. Essa indicação é feita por meio dos valores máximo e mínimo. As multiplicidades podem ser:
 - Obrigatórias, quando especificadas por meio de um número natural diferente de zero;
 - Opcionais, se “0..1”;
 - Multivaloradas, se “*” ou “0..*” (as duas notações têm o mesmo significado).
- Intervalos de multiplicidades podem ser especificados com os “..” (exemplo, “1..3”, para 1, 2 ou 3, ou de 1 a 3). Se houver mais de um intervalo, eles são especificados entre vírgulas (exemplos: “1..3, 5..7”).
- As pontas das associações também podem conter o sinal de *navegabilidade*, indicada por uma seta aberta, que representa a responsabilidade que um objeto tem de localizar os objetos da outra classe com os quais se associa. No nosso exemplo, os objetos da classe **Pedido** “têm a responsabilidade” de localizar os clientes a eles relacionados que, em outras palavras, quer dizer que os objetos da classe **Pedido** devem dispor de recursos para localizar os clientes a eles relacionados. Isso equivale a dizer no negócio que um pedido deve conter informações para a localização do cliente que o colocou, possivelmente o nome dele ou seu código para a localização no cadastro de clientes.



Diagramas de Classes - Relacionamentos

Associação entre Classes (cont.):

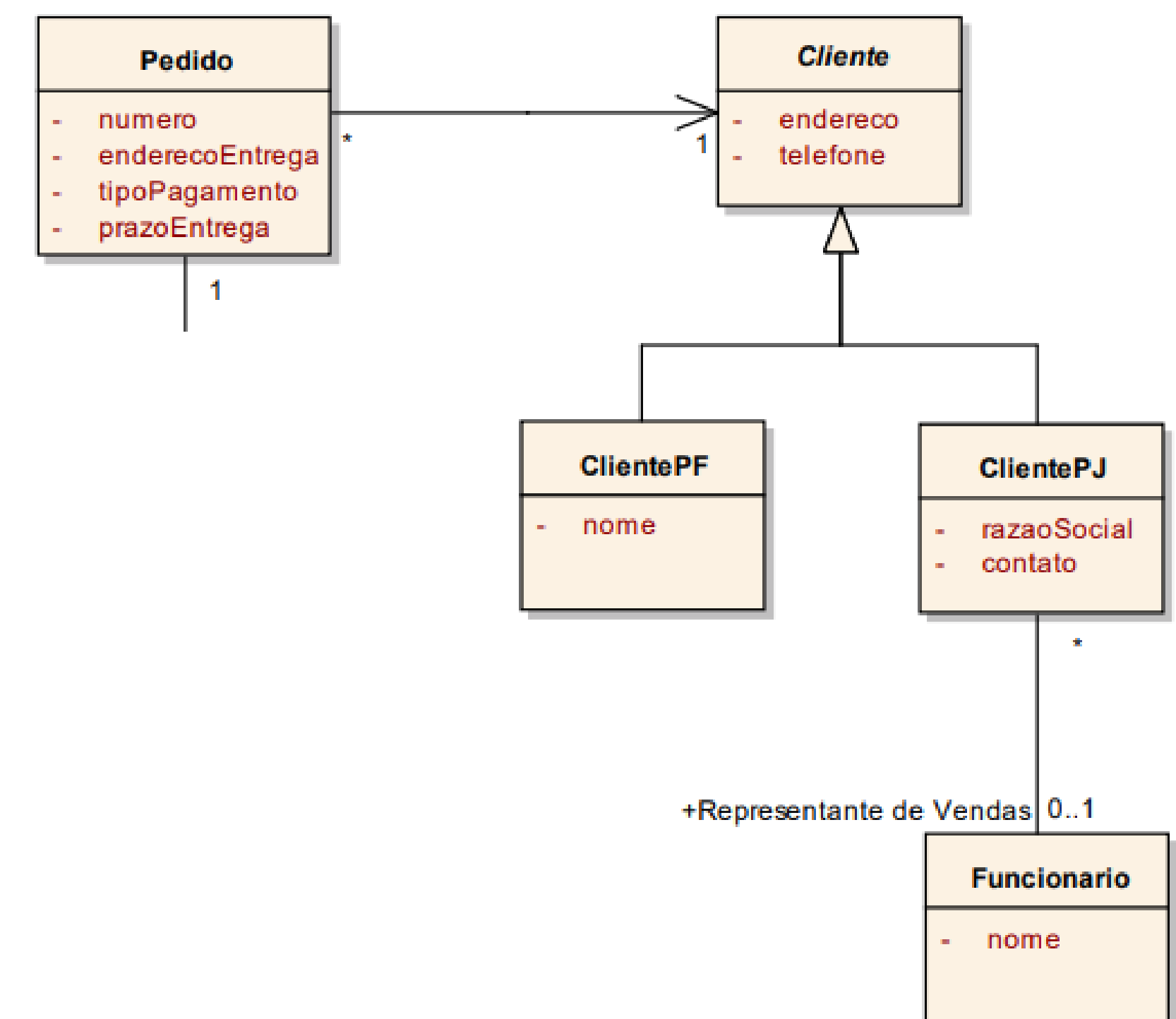
- As navegabilidades podem ser unidirecionais (uma seta), bidirecionais (duas setas ou nenhuma seta, se for assim convencionado) ou indeterminada (nenhuma seta). Navegabilidades são usualmente raras em modelos conceituais, pois normalmente refletem a preocupação dos projetistas com a rapidez de acesso aos objetos na memória e a economia de espaço em disco, preocupações estas associadas à fase de projeto dos sistemas.
- Autoassociações são associações entre objetos da mesma classe. São representadas no diagrama de classes usando-se poligonais ou arcos partindo de uma classe e chegando nela própria.



Diagramas de Classes - Relacionamentos

Especializações-Generalizações:

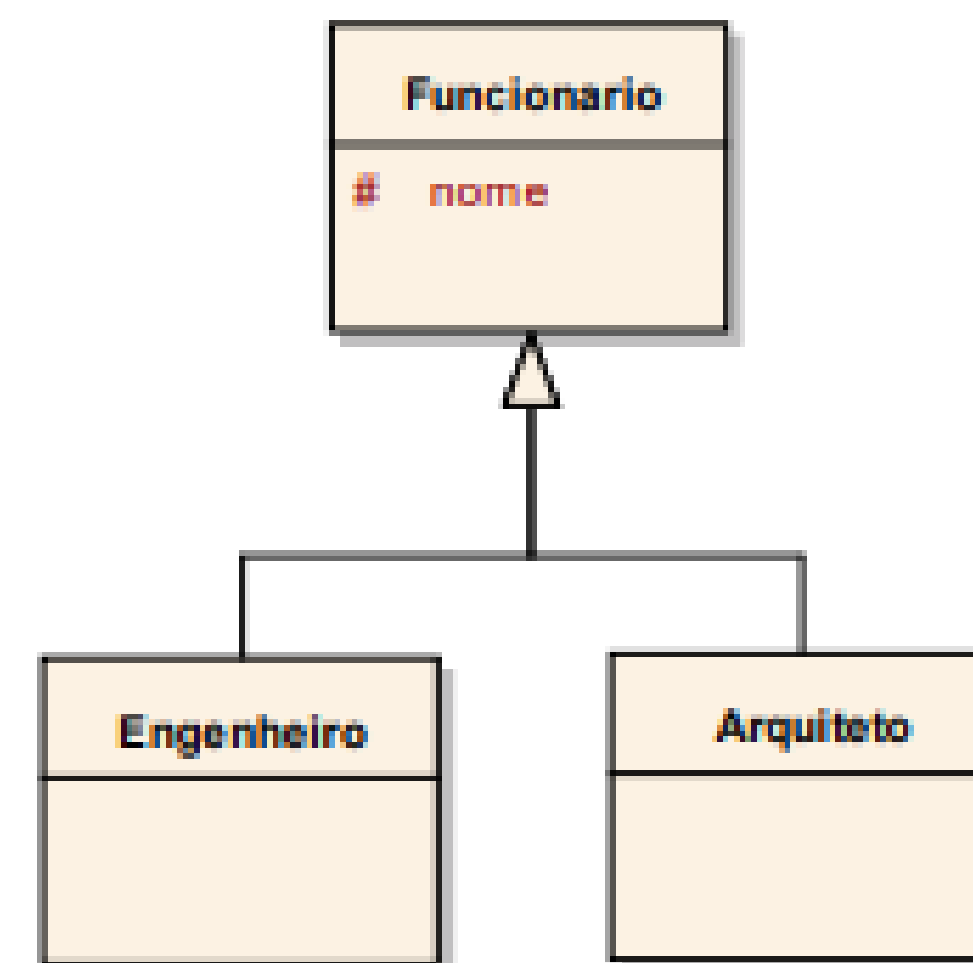
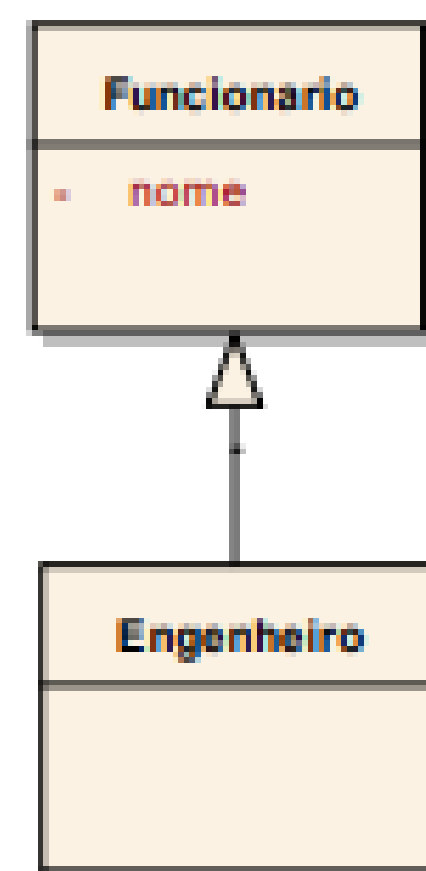
- A nossa empresa fictícia ZYX necessita manter em seu cadastro dois tipos diferentes de clientes: pessoas físicas (classe **ClientePF**) e pessoas jurídicas (classe **ClientePJ**), que possuem semelhanças e diferenças entre si: os endereços e telefones devem ser armazenados nos cadastros para todos os clientes, independentemente se são pessoas físicas ou jurídicas, os nomes são necessários apenas para as pessoas físicas e as razões sociais e nomes de contato são necessários apenas para as pessoas jurídicas.
- Utilizando o relacionamento de especialização-generalização, os atributos, operações e relacionamentos comuns ficam na classe que chamamos de superclasse ou classe-base, enquanto as diferenças vão para as subclasses, também chamadas de classes derivadas. Estas herdam da superclasse os atributos, as operações e os relacionamentos comuns.
- Sendo assim, os clientes pessoas físicas do modelo ao lado têm endereço, telefone e nome como atributos e estão associados a qualquer número de pedidos. Os clientes pessoas jurídicas, além do endereço e telefone, têm como atributos a razão social e o contato e estão associados a qualquer número de pedidos. Podem possuir, ainda, um representante de vendas, que é um funcionário da ZYX. Os relacionamentos de generalização-especialização são representados por setas com pontas triangulares vazadas.



Diagramas de Classes - Relacionamentos

Especializações-Generalizações:

- Cabe aqui explicar um detalhe sobre outro tipo de visibilidade de atributos e métodos que deixamos de explicar anteriormente porque ainda não tínhamos visto os relacionamentos de especialização-generalização: os atributos e métodos protegidos.
- Em uma classe, ser protegido significa que o atributo ou método é protegido do acesso externo de forma geral, mas pode ser acessado pelos métodos das classes que a especializam. Um atributo ou método protegido tem sua visibilidade denotada por um #.
- Como ilustra o diagrama abaixo, o atributo nome da classe **Funcionario** só pode ser acessado diretamente pelo próprio objeto instanciado dessa classe, porque o atributo está marcado como sendo privado. Nesse caso, nem um objeto da classe **Engenheiro** tem acesso direto a esse atributo. Já no diagrama ao lado, o atributo **nome** pode ser acessado diretamente por objetos das classes **Funcionario** e **Arquiteto**.



Diagramas de Classes - Relacionamentos

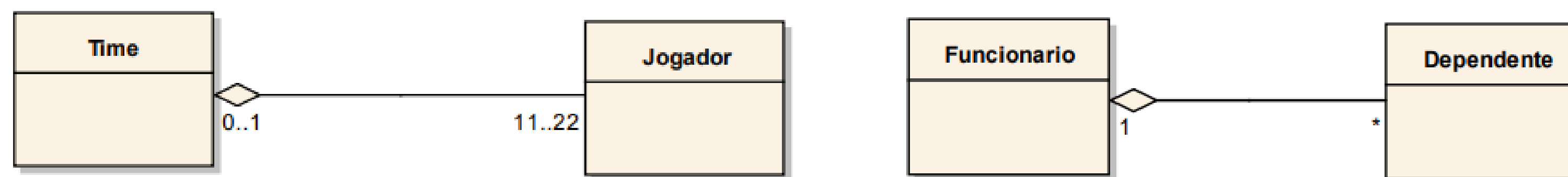
Especializações-Generalizações:

- Há situações em que não queremos que uma classe possa ser instanciada, ou seja, que não possa haver objetos criados dela (ao contrário das classes concretas, que já vimos). Essas classes são chamadas de classes **abstratas**. Por exemplo: no modelo anterior, a classe **Cliente** foi definida como uma classe abstrata porque não pode haver objetos instanciados dessa classe, ou seja, não há nenhum cliente da ZYX que não seja pessoa física nem pessoa jurídica (eles têm de ser, obrigatoriamente, instâncias de **ClientePF** ou **ClientePJ** – quem nos disse isso foi o especialista do negócio na ZYX).
- Classes abstratas existem no modelo somente para agruparmos em uma só classe os atributos, operações e associações comuns a duas ou mais classes. A esse processo de agrupamento, em superclasses, de atributos e operações comuns a duas ou mais classes damos o nome de **fatoração**.
- Classes abstratas são denotadas na UML com seus nomes em itálico ou colocando *abstract* logo abaixo de seus nomes, dentro do compartimento do nome na caixa da classe.
- Por fim, nada impede que façamos especializações de especializações em qualquer quantidade de níveis. A única recomendação é que muitos níveis de especialização (mais do que cinco, conforme cita a literatura) prejudicam o entendimento do modelo.

Diagramas de Classes - Relacionamentos

Agregações:

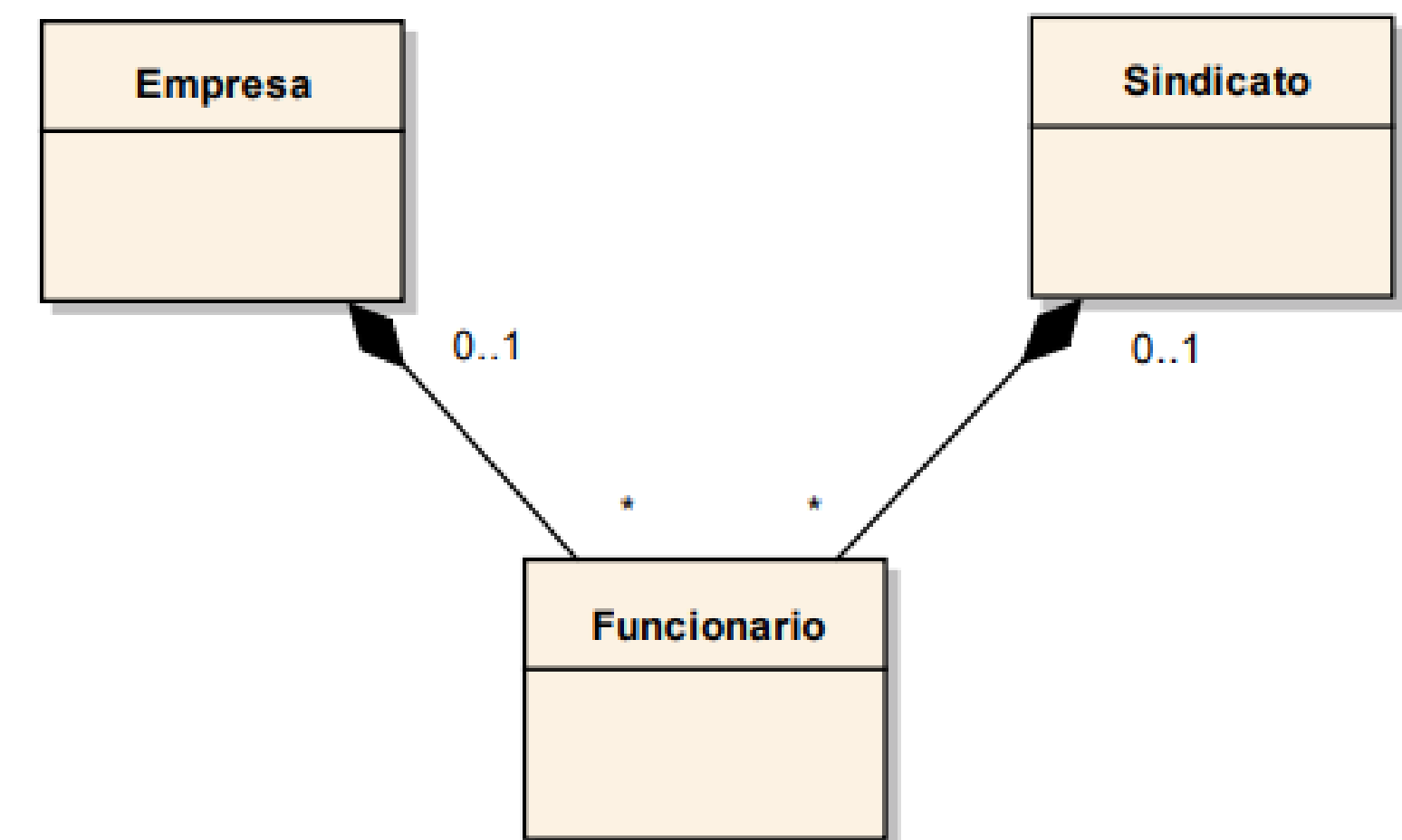
- Há situações em que precisamos especificar conceitos que representam conjuntos de entidades. Nesse caso, além das entidades que correspondem às partes, os conjuntos também são entidades que devem ser representadas em nosso modelo. Os conjuntos e as partes são ligadas entre si por meio de relacionamentos ditos de **agregação** – pois conjuntos agregam suas partes. Exemplos de conjuntos e suas partes são times de futebol e seus jogadores, empregados e seus dependentes, departamentos e suas divisões, divisões e seus colaboradores, etc.
- O relacionamento de agregação é um relacionamento do tipo todo-parte; representamos o "todo" associado às "partes" que o compõem.
- A UML possui uma notação especial para agregações: um losango vazado, conforme ilustrado abaixo. Nessa figura representamos os funcionários em uma organização e seus dependentes: cada dependente está associado a um funcionário específico, enquanto funcionários têm qualquer número de dependentes, eventualmente nenhum.
- Abaixo, à esquerda, é indicada uma situação em que um time é composto de 11 a 22 jogadores e que cada jogador ou não está associado a um time ou está associado a, no máximo, um time.

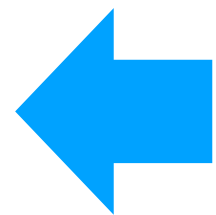


Diagramas de Classes - Relacionamentos

Agregações compostas (ou composições):

- Agregação composta, também chamada composição, é um tipo mais forte de agregação. É também um relacionamento todo-parte representado por um losango cheio (veja abaixo). A principal, e fundamental, diferença entre composições e agregações é que nas composições as partes não podem pertencer, em um mesmo instante, a mais do que um todo. Como consequência, se o todo deixa de existir, as partes também deixam. É importante notar que, quando permitido pela multiplicidade, uma parte pode ser removida da composição antes de ela deixar de existir.
- As composições são indispensáveis quando, no modelo, uma entidade parte está associada a mais de uma entidade todo e queremos especificar que uma instância da parte não pode estar associada, ao mesmo tempo, a mais de uma instância de entidade todo. Por exemplo, segundo a figura abaixo, à direita, se um determinado funcionário está associado a uma determinada empresa, não pode estar associado, ao mesmo tempo, a um sindicato.
- Na figura, uma empresa (o todo) tem qualquer número de funcionários (as partes) associados a ela, da mesma forma que sindicatos (o outro todo). Além disso, um funcionário pode estar associado a nenhuma ou uma empresa ou a nenhum ou a um sindicato. O que o modelo especifica ainda, por se tratar de composições, é que, se um funcionário está associado a uma empresa, não pode estar associado ao mesmo tempo a um sindicato¹, já que uma parte não pode pertencer a mais de um todo nas composições.





Conceitos relacionais

Herança

O conceito de herança nada mais é do que uma possibilidade de representar algo que já existe no mundo real. Um exemplo clássico disto é quando, na escola, estudamos sobre “classificação biológica” na aula de ciências. Nela, é feita a seguinte divisão entre os seres vivos: Reino, Filo, Classe, Ordem, Família, Gênero, Espécie.

Cada divisão mais baixa herda o que for necessário da divisão superior, e isto ocorre porque a mais baixa é um subtipo da divisão acima. Espécie herda de Gênero, que, por sua vez, herda de Família e assim por diante.

Dentro da Orientação a Objetos, quando desejamos usar o conceito de herança, é necessário fazer uma classe herdar de outra.

Herança é o relacionamento entre classes em que uma classe chamada de subclasse (classe filha, classe derivada) é uma extensão, um subtipo, de outra classe chamada de superclasse (classe pai, classe mãe, classe base). Devido a isto, a subclasse consegue reaproveitar os atributos e métodos dela. Além dos que venham a ser herdados, a subclasse pode definir seus próprios membros.

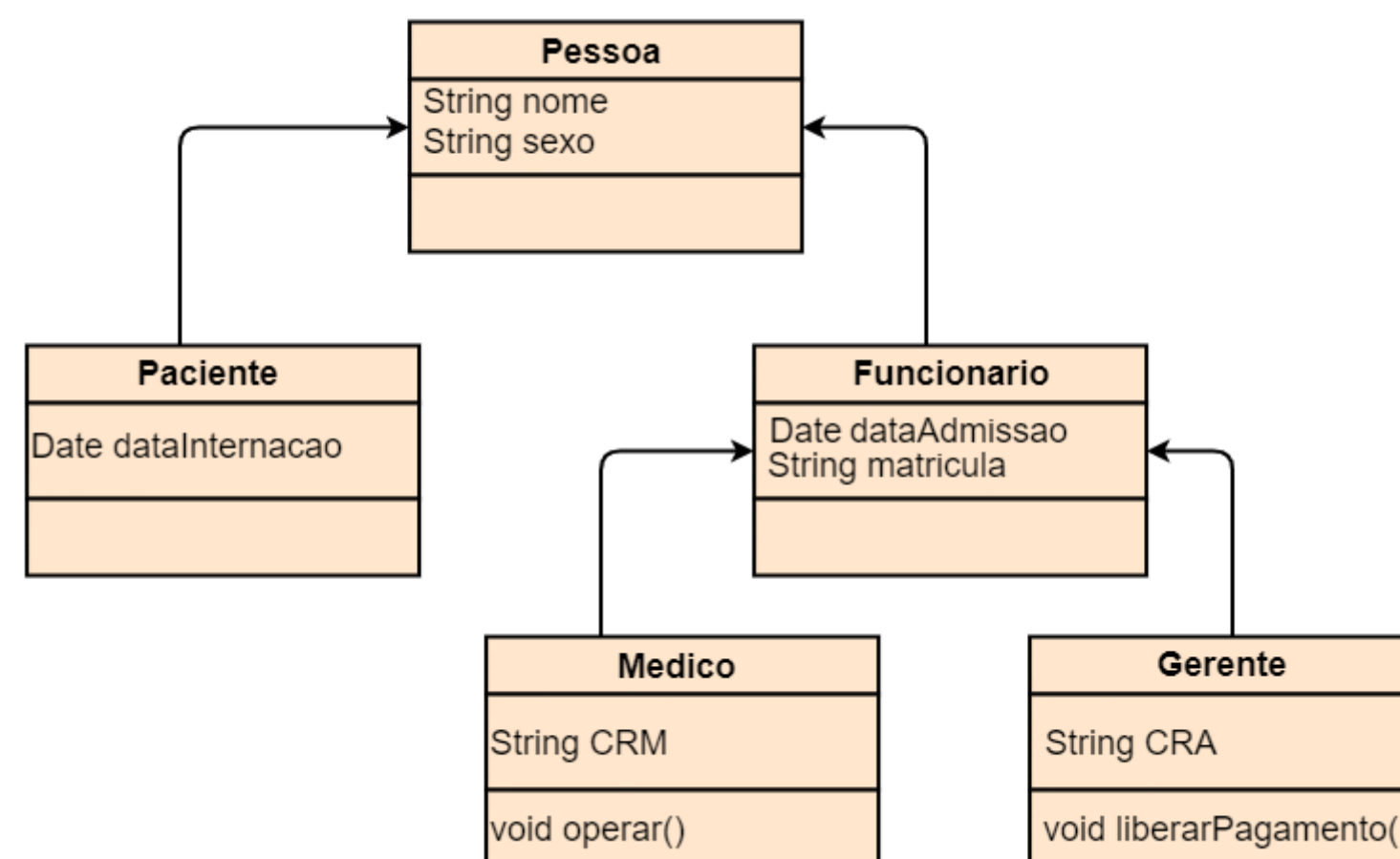
Essa definição deixa bem claro que herança só ocorre entre classes, então é incorreto dizer que “o objeto X herdou de Y”. Isto ocorre porque objetos só existem em tempo de execução, impossibilitando assim sua alteração estrutural. Já as classes, por serem do tempo de desenvolvimento (compilação), poderão definir a estrutura de novas classes e, conseqüentemente, de objetos criados a partir destas.

Herança

A herança pode ocorrer em quantos níveis forem necessários. Porém, uma boa quantidade de níveis é de, no máximo, 4. Quanto mais níveis existirem, mais difícil de entender o código será, pois cada vez mais é gerado um distanciamento do conceito base. Esses níveis são chamados de Hierarquia de Classe.

O fundamento de reuso que já vimos é intrinsecamente ligado à herança e também à abstração. Quando definimos uma classe da forma mais abstrata possível, é porque necessitamos reusar seu conceito e seus membros em outros conceitos similares. A herança deve ser aplicada para isso.

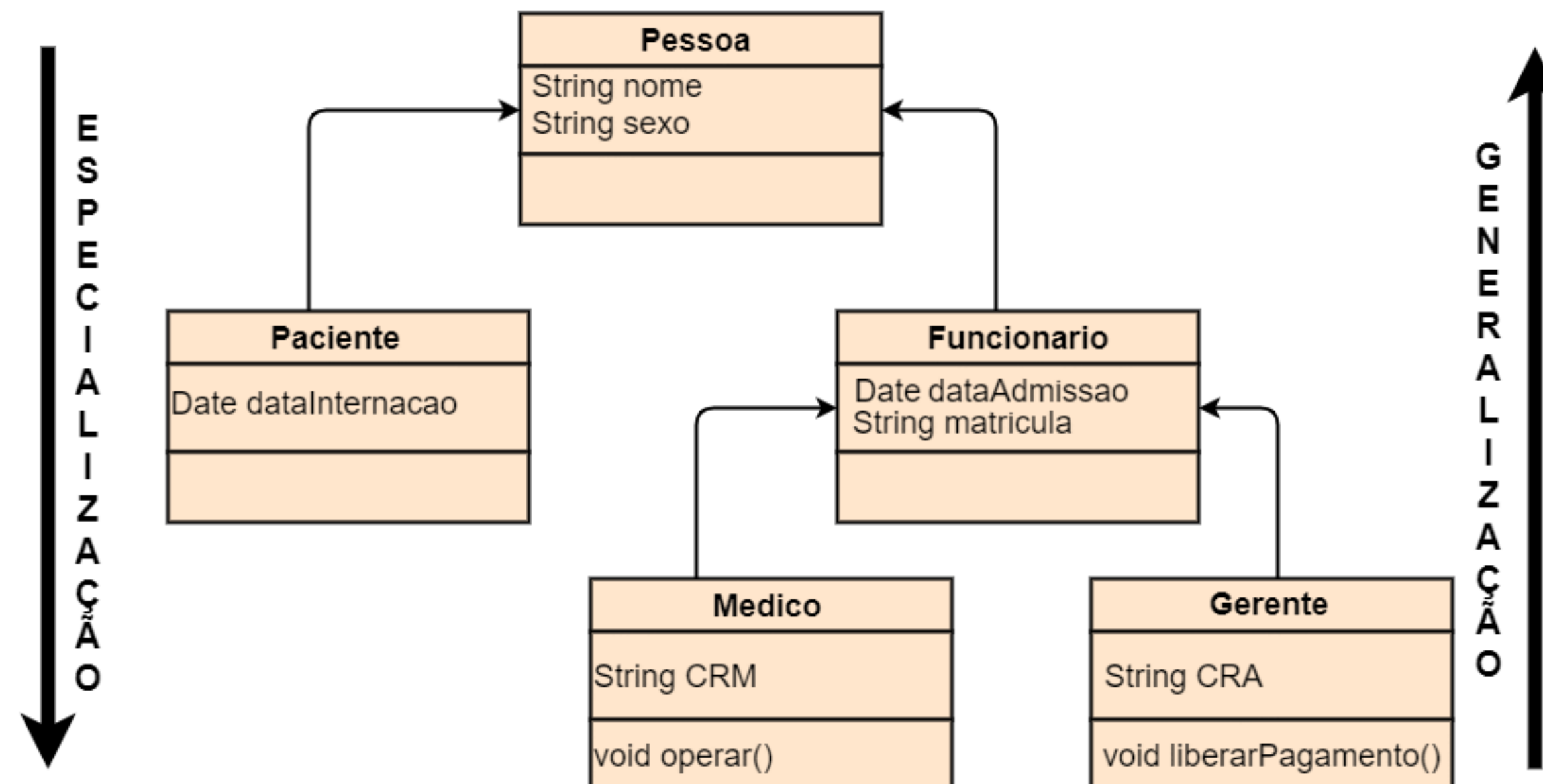
Quando uma classe herdar de outra, ela poderá acrescentar novos membros, mas não excluir. Ora, se a ideia é reusar para evitar repetição, não teria lógica excluir código. Além disto, a grande vantagem da herança é a definição de subtipos. Embora o reuso seja importante, na verdade ele é uma consequência da herança, já que é possível também termos reuso através de outros relacionamentos.



Herança

O processo de definir o mais genérico nas classes bases e ir acrescentando nas filhas o mais específico é conhecido como *Generalização* e *Especialização*, respectivamente. Quando mais se sobe na Hierarquia de Classe, mais genérico fica, e quanto mais desce, mais específico.

Alguns cuidados devem ser tomados quando usamos a herança, como onde colocar os atributos e métodos, e quando realmente devemos usá-la. Caso os membros sejam definidos na classe errada, situações estranhas podem ocorrer, pois não representarão a realidade.



Herança

Classes abstratas:

- Uma classe abstrata tem como principal função ser a implementação completa do conceito de abstração. São classes que representam conceitos tão genéricos que não vale a pena trabalhar com elas diretamente. Elas são incompletas e devem ser completadas pelas classes que herdarem delas, ou seja, seus subtipos.
- Por não valer a pena trabalhar diretamente com elas, essas classes têm uma característica importante: não podem ser instanciadas. Ou seja, não podemos criar objetos diretamente a partir delas. Ao tentarmos usar o operador **new** com uma classe abstrata, um erro do compilador informará que classes abstratas não podem ser instanciadas.
- Por serem de uso indireto, geralmente classes abstratas estão no topo da hierarquia de classe. Por exemplo, em uma especialização da superclasse **Pessoa** em classes **Funcionário** e **Paciente** em um sistema hospitalar, talvez não seja útil utilizar diretamente objetos do tipo **Pessoa**, afinal é importante distinguir quem é funcionário e quem é paciente. Cada um executará uma tarefa diferente dentro do hospital e deverá ser tratado da forma adequada.
- Além de definir a classe como abstrata - que servirá de molde para outras classes -, podemos também definir métodos como abstratos. A ideia de definir um método como abstrato é para que ele também sirva de molde. Para isso, ele não deve possuir uma implementação, mas sim apenas a definição de sua assinatura.
- Métodos abstratos só podem ser definidos em classes abstratas. Porém, classes abstratas podem também possuir métodos não abstratos, ou seja, que possuam sua implementação.
- É importante destacar que não existe “atributo abstrato”!

Herança

Classes concretas:

- Quando uma classe não é abstrata, ela só pode ser concreta. Ao contrário das abstratas, estas não são genéricas, e sim bem específicas. Elas representam o conceito de uso direto que deve ser trabalhado e, por isso, não só podem, como devem, ser instanciadas. Manipulá-las é vital para o bom funcionamento da aplicação.
- Para definir classes como concretas, basta definir as classes como já vínhamos fazendo, antes de explicar o conceito das abstratas. No caso, basta usar a palavra **class** seguida no nome da classe. O que separa uma classe abstrata de uma concreta é apenas uma questão conceitual, que deve ser bem entendida.
- Então, o uso da palavra **abstract** para a definição de classes abstratas deve ser utilizado quando houver a necessidade de aplicar esse conceito de abstração, ou seja, quando for de grande valor conceitual e relevante para nossa situação. Caso isso não se aplique, é só não usar essa palavra, assim estaremos criando classes concretas que deverão ser manipuladas diretamente.
- Quando classes concretas herdam a partir de uma classe abstrata que possua métodos abstratos, elas terão a obrigatoriedade de prover a implementação para tais métodos. Porém, se uma classe abstrata herdar de outra abstrata, essa obrigatoriedade não é válida.
- Para finalizar, embora seja possível fazer uma classe concreta herdar de outra concreta, isto deve ser desencorajado ou mesmo nunca realizado.

Herança

Tipos de herança:

- Existem dois tipos de herança: a simples e a múltipla. A simples ocorre quando uma subclasse tem apenas uma superclasse. Neste caso, a classe filha precisou apenas especializar e reutilizar membros de apenas um conceito, uma classe mãe da aplicação.
- A herança múltipla ocorre quando uma subclasse necessita não de apenas uma, mas duas ou mais superclasses. Assim, essa classe filha poderá especializar mais de um conceito de uma aplicação. Podemos pensar no seguinte como exemplo:
 - Um sistema hospitalar possui como classes **Gerente** e **Médico**. Pode existir uma situação na qual um médico assume um papel de chefe de departamento, e ele pode executar ações gerenciais, como aprovação de férias. Sendo assim, poderíamos criar uma subclasse **ChefeDeDepartamento**, que herdaria das classes **Gerente** e **Médico**.
- A linguagem Java não possui suporte para heranças múltiplas. Esta foi uma decisão de projeto dos criadores dessa linguagem, e o principal motivo para isso é para evitar *conflito de nomes*, o que é um risco nas linguagens que suportam. Esse conflito se dá quando as duas superclasses possuem atributos ou métodos com os mesmos nomes. Por exemplo, tanto **Gerente** quanto **Médico** poderiam ter o atributo **cargaHorária**, com significados diferentes.
- Dentre as linguagens atuais que possuem suporte a heranças múltiplas, podemos citar C++, Python, Perl, Eiffel e R. Java possui formas de emular a herança múltipla, que vamos ver logo mais.

Herança

Upcast e Downcast:

- Quando trabalhamos com herança, podem surgir duas operações realizadas com os objetos que foram criados a partir das classes envolvidas em uma hierarquia: *upcast* e *downcast*.
- O upcast é uma operação de conversão, na qual subclasses são promovidas a superclasses. Como uma classe filha é do tipo de sua classe mãe, esta conversão é permitida.
- Quando falamos sobre “cast” em linguagens estruturadas, logo lembramos dos tipos primitivos, de como realizar o “cast” de um *long* para um *int*, de um *double* para um *int*. Quando fazemos uma conversão (cast) de um *int* para *float*, a simples codificação é feita: *float = int*. Isso ocorre porque um *int* cabe dentro de um *float*.
- Essa ideia de “caber” também se aplica aos objetos, só que com outro nome (no caso, subtipo). Se uma subclasse é subtipo de sua classe mãe, então ela “cabe” nela. Por isto é permitido fazer upcast de forma implícita entre objetos.
- O downcast é a operação inversa, assim superclasses são convertidas em subclasses. Porém, embora seja um conceito válido, este deve ser desencorajado. Isto ocorre porque podem ocorrer várias especializações distintas a partir de uma generalização.

```
Pessoa pessoa;
```

```
pessoa = new Medico();
```

```
pessoa = new Paciente();
```

```
pessoa = new Funcionario();
```

Herança

Polimorfismo:

- Em determinados momentos em uma hierarquia de classes, precisamos que um mesmo método (nome e lista de parâmetro, ou seja, assinatura) se comporte de forma diferente dependendo do objeto instanciado a partir de uma classe de uma hierarquia qualquer. Isto surge devido à necessidade de flexibilidade que a hierarquia de classe deseja fornecer.
- Por exemplo, sabemos que cada tipo de médico pode ter uma forma diferente de realizar sua ação de operar um paciente de acordo com o procedimento. Em um parto, por exemplo, o anestesiista aplica uma injeção anestésica, o obstetra realiza a preparação e a retirada da criança, o pediatra realiza um conjunto de verificações para atestar a saúde do recém-nascido, etc.
- Cada médico realiza suas determinadas ações dependendo de sua função no parto, mas todos estão “operando” naquele momento. Esta possibilidade de uma mesma ação poder se moldar de acordo com o objeto em questão é chamado de polimorfismo. Em cada subclasse, a ação de operar é realizada de forma distinta, pois cada uma tem suas peculiaridades. Mas mesmo assim, a ação é a mesma em sua forma mais íntima: operar. Esta foi herdada a partir da superclasse **Medico**.
- A grande vantagem do uso do polimorfismo é que podemos utilizar objetos distintos e continuar executando a mesma ação, sendo que esta se moldará ao objeto corrente. Essa é a “flexibilidade” citada anteriormente.
- A melhor forma de possibilitarmos o uso de polimorfismo é trabalhar com classes e métodos abstratos. Assim, podemos apenas definir a assinatura do método (ação) e deixar para a subclasse realmente definir o comportamento desta operação.
- Em Java, é necessário o uso da instrução `@Override` antes da definição do método na subclasse.
- Para existir polimorfismo, é necessário que se tenha uma herança. Só assim será possível prover o comportamento para um método abstrato herdado, com o intuito de que este tenha um comportamento diferente de acordo com o objeto. Porém, ao usarmos a herança, não precisamos necessariamente utilizar o polimorfismo.

Herança

Sobrescrita:

- Como o próprio nome sugere, sobrescrita é quando uma “escrita”, uma implementação de um método, sofre uma “escrita por cima”, ou seja, é redefinida. A sobrescrita é utilizada quando é necessário modificar um comportamento herdado. Essa alteração pode acrescentar ou eliminar algo do comportamento herdado.
- Em Java, assim como no polimorfismo, a utilização do `@Override` deve ser feita. Assim, a subclasse redefine o método herdado e a sobrescrita é realizada.
- Entretanto, em alguns casos, o método sobrescrito na subclasse precisa utilizar integralmente o comportamento do método da superclasse, e depois realizar seus passos específicos. Para realizar esta tarefa, as linguagens orientadas a objeto proveem sintaxes específicas: em Java, é a palavra `super`.
- Do ponto de vista de implementação, a sobrescrita é idêntica ao polimorfismo. No entanto, conceitualmente são diferentes. A sobrescrita “sobrescreve” algo existente - no caso, um comportamento padrão da superclasse. De acordo com a necessidade, podemos muda-lo ou não. Já no polimorfismo, não há necessidade de se ter um comportamento padrão.

```
class ResidenteAnestesista extends Anestesista {  
  
    @Override  
    void operar() {  
        super.operar();  
        // passos específicos para a sobrescrita  
    }  
}
```

Associação

Até o momento, foi visto apenas um tipo de relacionamento: a herança. Este é útil para quando precisamos definir subtipos e, conseqüentemente, obter reuso de membros. Embora estas situações sejam comuns e úteis em aplicações orientadas a objetos, não é a única necessidade de relacionamento.

Por exemplo, dentro de um hospital, existem vários tipos de médicos. Para podermos aplicar o reuso e a especialização, podemos ter uma superclasse **Medico** e subclasses **Anestesista**, **Obstetra**, **Pediatra**, etc.

Mas e se for necessário representar, no modelo do hospital, os endereços dos médicos? Tanto um anestesista quanto um obstetra precisarão de um endereço. Este teria um nome de rua, bairro, cidade, entre outros atributos que seriam necessários para representa-lo.

É comum iniciantes aplicarem a seguinte solução: fazer as classes **Anestesista** e **Obstetra**, ou mesmo **Medico**, herdarem da classe **Endereco**. Assim, todos os atributos de um endereço seriam compartilhados com todas essas classes. À primeira vista, isso parece uma solução aceitável, mas infelizmente não é, afinal de contas, um endereço não é um médico.

Associação possibilita um relacionamento entre classes/objetos, no qual estes possam pedir ajuda a outros e representar de forma completa o conceito no qual se destinam. Neste tipo de relacionamento, as classes e os objetos interagem entre si para atingir seus objetivos.

Associação

A associação pode ser realizada de duas formas: estrutural e comportamental. A primeira possui dois tipos: agregação e composição. A segunda, somente um: dependência.

A estrutural tem como característica a associação ocorrer na estrutura de dados da classe, mais precisamente em seus atributos. Assim, um dos atributos de uma classe é do tipo de outra classe.

No exemplo anterior, temos uma associação estrutural, já que **Medico** tem um de seus atributos que é do tipo **Endereco**.

A associação estrutural do tipo composição ocorre quando um relacionamento da forma “parte todo” ocorre. Ou seja, a parte não pode existir sem a existência do todo. Utilizando o exemplo anterior, o endereço “Rua 123 de Oliveira 4, nº 10” só pode existir se pertencer a um (e unicamente um) médico. Não teria finalidade alguma esse endereço existir sem estar ligado a um médico, empresa, entre outros.

Neste caso, notamos uma forte relação entre a parte (o endereço) e o todo (o médico). Assim, o médico é composto por um endereço, e este pertence somente a esse médico.

Já a associação estrutural do tipo agregação ocorre quando o relacionamento “parte todo” não ocorre. Ou seja, a parte pode ser compartilhada entre vários objetos (todos) distintos. Por exemplo, o procedimento **parto** será executado na “Sala 02” no período da manhã. Já o procedimento **revascularizacaoMiocardio** também será executado na “Sala 02”, só que pela tarde.

Associação

Do ponto de vista de implementação, composição e agregação são idênticos, mas conceitualmente são diferentes. Essa divisão surgiu principalmente por conta da UML, pois a visualização das duas associações é diferente. No entanto, quando estamos falando de implementação do código, a execução é a mesma.

Por fim, temos a associação comportamental, no caso, a dependência. Muitas vezes precisamos passar objetos como parâmetros para os métodos, ou mesmo instanciar objetos dentro do corpo dos métodos. Com isso, temos acessos aos membros desses objetos/classes para nos “ajudar” a realizar as atividades necessárias. Isto nada mais é do que um outro exemplo de associação, porém essa agora não está ligada à estrutura da classe/objeto, já que não é um atributo.

Associação

Características de uma associação:

- As associações possuem algumas características que visam facilitar a sua usabilidade e também o seu entendimento. A seguir, serão demonstradas situações que visam expor essas características.
- Geralmente, os hospitais atendem pacientes por meio de um plano de saúde. Nestes planos, existe um conceito que é o beneficiário, uma pessoa que possui um plano para cobrir suas necessidades médias. Esse beneficiário termina se transformando no paciente quando ele é atendido no hospital.
- É comum também que ele possua dependentes, tipo uma mãe que paga o plano de saúde de seu filho, juntamente com seu próprio plano. Assim, define-se um autorrelacionamento, pois tanto a mãe quanto o filho são beneficiários. A diferença é que o filho está relacionado com a mãe (depende dela).
- Para identificar separadamente o titular e o dependente, poderíamos definir um atributo. A codificação a seguir ilustra essa situação.
- O atributo **dependente** na classe **Beneficiario** é do seu próprio tipo. Isto é uma associação *unária*, pois somente uma classe/objeto foi usada, no caso, a classe **Beneficiario**. Para diferenciar quem é o titular e quem é o dependente, foi criado um atributo **tipoBeneficiario**, no qual possíveis valores poderiam ser “titular” ou “dependente”.

```
class Beneficiario {  
  
    String nome;  
    Date dataNascimento;  
    String tipoBeneficiario;  
    Beneficiario dependente;  
  
    // gets/sets  
  
    // métodos afins  
}
```


Associação

Características de uma associação (cont.):

- Já na classe **Parto**, temos uma associação múltipla, pois vários tipos de classes são usados nas associações. **Parto** tem o atributo **sala** do tipo **Sala**, e tem um vetor de médicos do tipo **Medico**. Ou seja, teve mais de um tipo de classe envolvida na associação.

```
class Parto extends Procedimento {  
  
    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetra(), new Pediatra()};  
  
    Sala sala;  
}
```

- Ainda na classe **Parto**, notamos que são exatamente 3 médicos e 1 sala envolvidos neste procedimento. Esta quantidade de médicos e sala corresponde à cardinalidade destas associações. As cardinalidades podem ter quantidades fixas, como a deste exemplo, ou não ter uma quantidade definida - ou seja, terá quantos objetos forem necessários. Ela serve para identificar quantos objetos a associação possui.
- Por fim, vamos ver a navegabilidade. Ela pode ser unidirecional ou bidirecional. A primeira determina que a associação acontece somente de um lado. No caso da classe **Parto**, o tipo é unidirecional, pois só é relevante saber a sala na qual o parto será executado. Assim, criou-se um atributo em **Parto** do tipo **Sala**.
- Caso fosse necessário saber a qual procedimento uma sala pertencesse, deveríamos então ter um vetor de **Parto** na classe **Sala**, pois só assim seria possível obter essa rastreabilidade. Ao fazer isso, a navegabilidade seria bidirecional, pois as duas classes envolvidas tinham uma a referência da outra.

```
class Parto extends Procedimento {  
  
    Medico[] medicos = new Medico[]{new Anestesista(), new Obstetra(), new Pediatra()};  
  
    Sala sala;  
}  
  
class Sala {  
  
    Parto[] partos;  
  
}
```

Interface

Em algumas aplicações orientadas a objetos que necessitam de uma modelagem um pouco mais elaborada, muitas vezes é preciso determinar um conjunto de métodos que devem obrigatoriamente ser usados. Porém, como eles são realmente implementados, não importa a quem definiu tal conjunto. Essa obrigatoriedade de definição de métodos é chamada de interface.

Interface define um contrato que deve ser seguido pela classe que a implementa. Quando uma classe implementa uma interface, ela se compromete a realizar todos os comportamentos que a interface disponibiliza.

Por exemplo, imagine que o hospital que estamos usando como exemplo terá que prestar contas ao Ministério da Saúde. Ele sabe que deve informar ao ministério quanto faturou no mês corrente, quais procedimentos foram executados, entre outras necessidades.

O próprio ministério sabe que precisa dessas informações, mas não sabe como obtê-las, afinal, elas estão em poder do hospital. É para possibilitar essa troca de informações entre o hospital e o ministério, que deve ser definida uma interface.

Assim, o ministério deve disponibilizar um conjunto de métodos (no caso, a interface), para que o hospital seja obrigado e tenha como fornecê-las. Para o ministério, não importa quais atividades foram realizadas para se chegar a tais informações, apenas importa as informações em si. Como estas foram obtidas é de responsabilidade do hospital.

Interface

Quando um outro hospital for repassar suas informações, este também deverá implementar a mesma interface. Entretanto, a sua implementação poderá ser completamente diferente em relação ao primeiro hospital.

Essa situação reforça a definição de interface: é obrigatório prover o comportamento, mas como este será realizado para a interface é irrelevante.

Assim como existe uma palavra reservada para criar uma classe, existe uma para a interface - no caso, **interface**. Em Java, devemos usar a palavra reservada **implements**.

O exemplo demonstra a situação que, quando a classe **TransmissaoDadosMinisterio** implementou a interface **IDemonstrativoOperacional**, ela necessitou realizar a implementação dos métodos da interface. A prova disto é que os métodos estavam sem corpo na interface, isto é, havia um “;” logo após os parênteses, e não havia chaves delimitando seu corpo.

Todavia, quando a classe implementou a interface, os métodos tiveram seu corpo definido. Temos um detalhe a mais: é uma boa prática colocar a letra **I** no início do nome das interface, para assim diferenciarmos o que é uma classe e o que é uma interface.

```
interface IDemonstrativoOperacional {  
  
    double disponibilizarFaturamentoMensal();  
  
    Procedimento[] informarProcedimentoExecutados();  
}  
  
class TransmissaoDadosMinisterio implements IDemonstrativoOperacional {  
  
    @Override  
    public double disponibilizarFaturamentoMensal() {  
        // implementação específica para o hospital  
        // conseguir informar seu faturamento mensal  
    }  
  
    @Override  
    public Procedimento[] informarProcedimentoExecutados() {  
        // implementação específica para o hospital  
        // conseguir informar os procedimentos executados  
    }  
}
```

Interface

Mais alguns detalhes sobre interfaces:

- Por padrão, todo método em uma interface é abstrato, portanto não precisamos colocar a palavra **abstract**.
- A interface se comporta como uma classe abstrata, só que mais restritiva. Em classes abstratas vimos que, se necessário, podemos ter métodos não abstratos. Mas em uma interface isso não é possível.
- Usualmente, interfaces não possuem atributos. No entanto, se necessário, podemos definir atributos, e eles serão sempre públicos, estáticos e constantes. Usamos o termo “estático” quando queremos definir um atributo que é compartilhado entre as instâncias de uma classe ou interface.
- Um atributo “constante” não varia durante a execução do código. Em Java utilizamos a expressão **final** para indicar constantes.
- Como vimos, Java não disponibiliza a possibilidade de se implementar herança múltipla. Entretanto, em relação às interfaces, ela permite implementação múltipla. Ou seja, uma classe pode implementar mais de uma interface e, para isso, basta separá-las por vírgula. Portanto, utilizando o recurso de interface, podemos emular, de certa forma, a possibilidade de heranças múltiplas em Java.

```
//Java
class Classe1 extends Classe0 implements IUm, IDois {

//Códigos

}
```

```
//Java
interface IUm {

//Códigos

}

interface IDois {

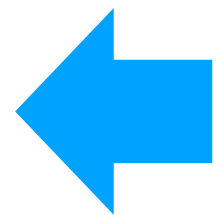
//Códigos

}

class Classe1 implements IUm, IDois {

//Códigos

}
```

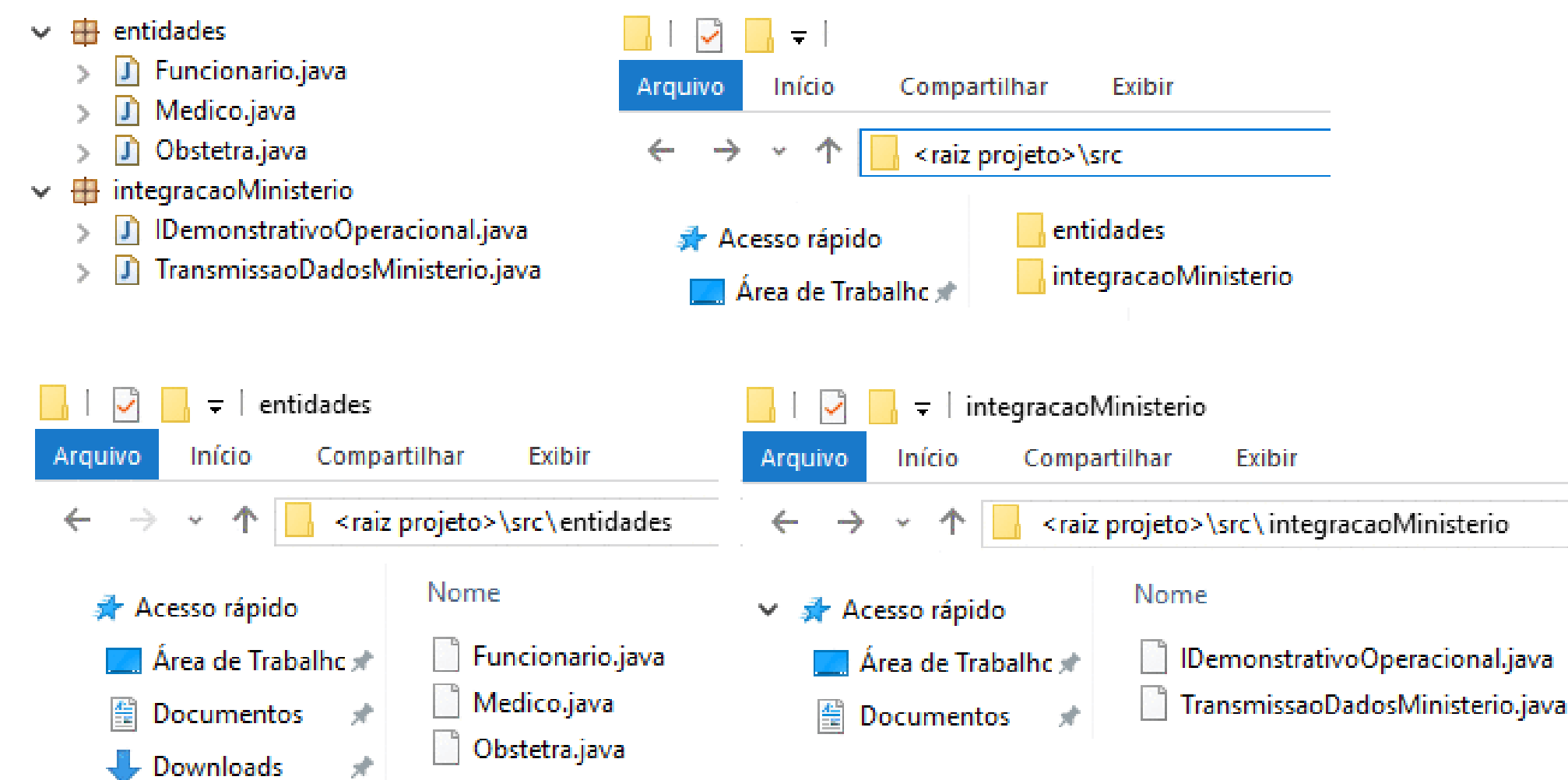
Conceitos organizacionais

Pacotes

É comum surgir a seguinte situação: o sistema terá dezenas de classe que representarão os conceitos do domínio da aplicação, como classes utilitárias, classes de acesso a bancos de dados, entre outros tipos possíveis. Porém, se simplesmente deixarmos todas estas juntas, ficará difícil de achar uma classe quando desejado. A mistura de classes com finalidades e conceitos diferentes dificulta a organização e pesquisa. É para isso que existem os pacotes.

Um pacote é uma organização física ou lógica criada para separar classes com responsabilidades distintas. Com isso, espera-se que a aplicação fique mais organizada e seja possível separar classes de finalidades e representatividades diferentes.

Em Java, utilizamos a palavra reservada **package** para criar pacotes. Ao usarmos essa palavra, o Java cria uma pasta no sistema de arquivos do computador com o intuito de juntar as classes que possuam representatividades semelhantes. Para possibilitar isso, cada classe deve declarar a definição de pacote de forma igual.



Pacotes

Caso precisemos criar pacotes dentro de pacotes (no caso, subpacotes), estes devem ser separados por um ponto. Poderíamos ter, por exemplo, **package integracaoMinisterio.saude** ou **package integracaoMinisterio.fazenda**.

Existe um padrão, em Java, para a definição de nomes dos pacotes. O padrão é a URL da empresa, de trás para frente, que está desenvolvendo a aplicação. Logo após isto, temos o nome do projeto e, ao final, os reais pacotes que se deseja criar.

No nosso caso, usando a URL do IBMEC como base (br.ibmec), teríamos o seguinte: **br.ibmec.progoo.organizacionais**, **br.ibmec.progoo.relacionais**, e por aí vai.

Para podermos usar classes dentro de outros pacotes, precisamos utilizar a palavra reservada **import**, para que uma classe “enxergue” a outra.

```
//Java
package integracaoMinisterio;

import entidades.Medico;

class TransmissaoDadosMinisterio implements IDemonstrativoOperacional {

    ...

}
```

Visibilidades

Também chamadas de *modificadores de acesso*, as visibilidades têm como finalidade controlar o acesso (manipulação) de classes, atributos e métodos.

Um modificador de acesso tem como finalidade determinar até que ponto uma classe, atributo ou método pode ser usado. A utilização de modificadores de acesso é fundamental para o uso efetivo da Orientação a Objetos. Algumas boas práticas e conceitos só são atingidos com o uso correto deles.

A OO provê três visibilidades, que são: privada, protegida e pública, sendo respectivamente as palavras **private**, **protected** e **public**, utilizadas para indicar tais visibilidades. Apesar de Java utilizar essas palavras, nem toda linguagem implementa completamente o conceito de visibilidade. A linguagem Python é um exemplo disto, pois nela todos os atributos e métodos são públicos, por exemplo.

Embora todas as visibilidades possam ser aplicadas a classes, os conceitos de classes privadas e classes protegidas são mais avançados e não serão abordados nesse curso.

Veremos então as visibilidades para atributos e métodos.

Visibilidades

Privada:

- Essa visibilidade define que atributos e métodos só podem ser manipulados no local de sua definição. Ou seja, se definirmos membros com essa visibilidade, eles só poderão ser manipulados dentro da classe onde foram estipulados.
- No código abaixo, teríamos erros em **//1** e **//2**, pois, como os atributos e os métodos foram definidos como **private**, só serão acessíveis dentro da classe **Beneficiario**. Na classe **TestePrivate**, é impossível acessá-los.
- Em Java é necessário estipular essa visibilidade de forma explícita. Dada a sua importância, outras linguagens, como C#, consideram-na como sendo a visibilidade padrão caso o programador não especifique.

```
public class Beneficiario {  
  
    private String nome;  
    private Date dataNascimento;  
    private String tipoBeneficiario;  
    private Endereco endereco;  
  
    // gets/sets  
    private void idade() {  
        // cálculo da idade a partir da data de nascimento  
    }  
}  
  
public class TestePrivate {  
    public static void main(String[] args) {  
  
        Beneficiario beneficiario = new Beneficiario();  
        //1  
        String nome = beneficiario.nome;  
        //2  
        beneficiario.idade();  
    }  
}
```


Visibilidades

Protegida:

- Essa visibilidade define que atributos e métodos podem ser manipulados apenas no local de sua definição e nas classes que herdaram da classe na qual foram definidos. Ou seja, se forem feitos membros com essa visibilidade, eles só poderão ser manipulados dentro da classe e nas subclasses desta classe.
- No exemplo ao lado, são apresentadas as marcações **//1** e **//2** na classe **Medico**, e nelas é possível acessar o atributo **nome** e o método **metodo1()**, respectivamente. Isso porque a classe **Medico** é uma subclasse de **Funcionario**, e ela definiu estes membros como *protected*. Entretanto, a classe **Paciente**, que não é uma classe filha de **Funcionario**, apresentará erros nas marcações **//3** e **//4**.
- Caso se precisasse de algum método de **Medico** ou **Funcionario** seria preciso criar um objeto destas classes para se ter acesso aos seus membros. A linha **//5** evidencia isso e reforça que a classe **Paciente** não tem acesso direto a membros de **Funcionario** ou **Medico**, pois não é uma subclasse destas.

```
package entidades;

public class Funcionario {

    protected String nome;

    protected void metodo1() {
        //codigo
    }
}

package entidades;

public class Medico extends Funcionario {

    private void metodo() {

        //1
        String texto = nome;

        //2
        metodo1();
    }
}

package entidades;

public class Paciente {

    private void metodo() {

        //3
        String texto = nome;

        //4
        metodo1();

        Medico medico = new Medico();

        //5
        medico.metodo1();
    }
}
```


Visibilidades

Pública:

- Todos os membros definidos com essa visibilidade são acessíveis em qualquer lugar, independente de qualquer relacionamento entre as classes. À primeira vista, pode parecer a melhor visibilidade, mas não é.
- Tornar todos os membros de uma classe públicos pode possibilitar acessos indevidos de atributos e uso indevido de métodos. O uso da visibilidade **public** deve ser feito com cuidado para não ferir alguns dos preceitos da Orientação a Objetos.
- Por ser uma visibilidade de uso livre, mostramos ao lado apenas a exemplificação da definição. Não é necessário exemplificar seu uso, já que, por ser de acesso livre, basta utilizá-lo da forma que já vinha sendo exposta nas outras visibilidades. Entretanto, agora nenhum erro ocorrerá.

```
public class Endereco {  
  
    public String logradouro;  
    public int numero;  
    public String bairro;  
  
    public String getLogradouro() {  
        return this.logradouro;  
    }  
  
    public void setLogradouro(String logradouro) {  
        this.logradouro = logradouro;  
    }  
  
    // demais get/set  
}
```

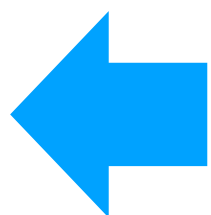
Visibilidades

Em cada situação, um modificador de acesso diferente deve ser usado.

Via de regra, todo atributo deve ser privado. Em casos esporádicos protegidos e em casos excepcionais, podem ser públicos. Já os métodos, via de regra são todos públicos. Em casos esporádicos protegidos e em poucos casos, podem ser privados.

Pela Orientação a Objetos, existem apenas esses três modificadores de acesso. Entretanto, algumas linguagens oferecem visibilidades a mais, que não fazem parte da teoria de OO.

Em Java, por exemplo, existe a visibilidade **default**, em que membros e classes definidos com esta podem ser usados por classes dentro de um mesmo pacote, independente de qualquer relacionamento entre elas. Apesar de existir, o seu uso em um projeto OO é desencorajado, principalmente porque podem contrariar alguns dos preceitos do paradigma.



Estudo de caso

O problema

Os sistema do hospital deve possibilitar a manipulação de pacientes e médicos. Cadastrar, atualizar e excluí-los deve ser possível. Também devemos poder marcar e cancelar consultas e procedimentos.

O paciente deve conseguir visualizar suas consultas e os médicos consultarem seus procedimentos. Tanto a consulta como o procedimento terão um valor total, dependendo do que for realizado. Os tipos de procedimento são: faringoplastia e neurocirurgia.

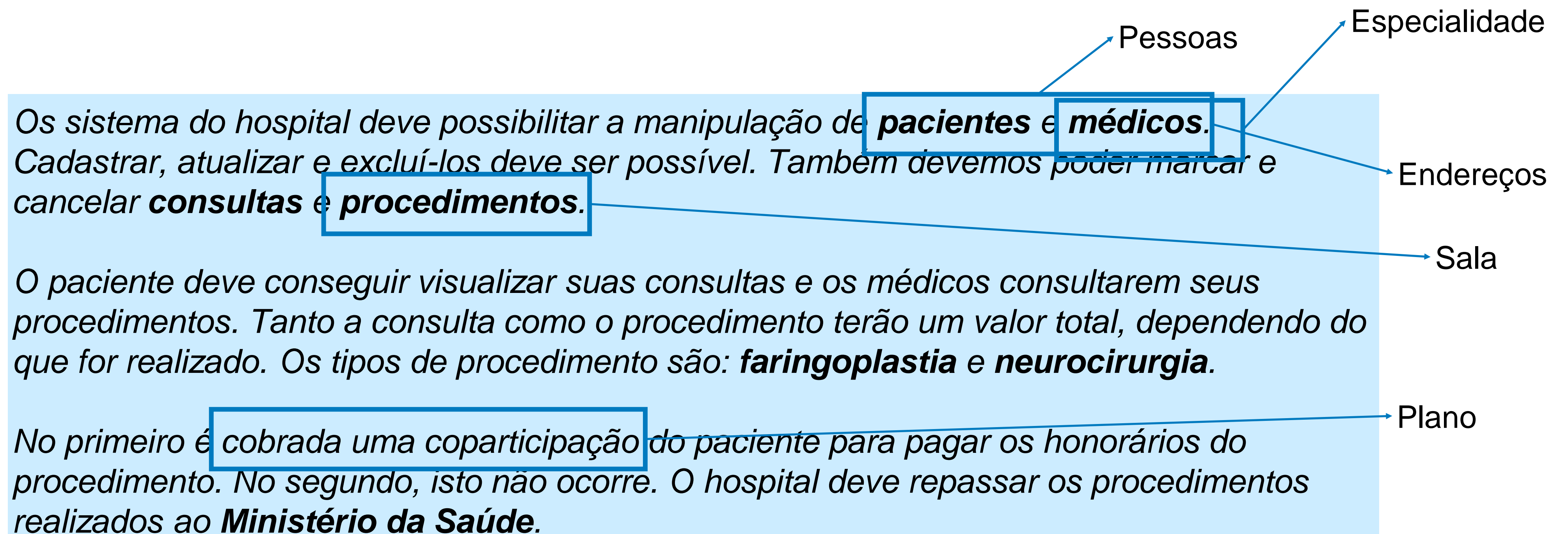
No primeiro é cobrada uma coparticipação do paciente para pagar os honorários do procedimento. No segundo, isto não ocorre. O hospital deve repassar os procedimentos realizados ao Ministério da Saúde.

Apesar das especificações de requisitos tentarem cobrir todo o problema a ser resolvido, sempre é necessário um conhecimento do domínio a ser implementado. Esse conhecimento pode vir de entrevistas adicionais com os clientes, dinâmicas ou consultores especializados no domínio.

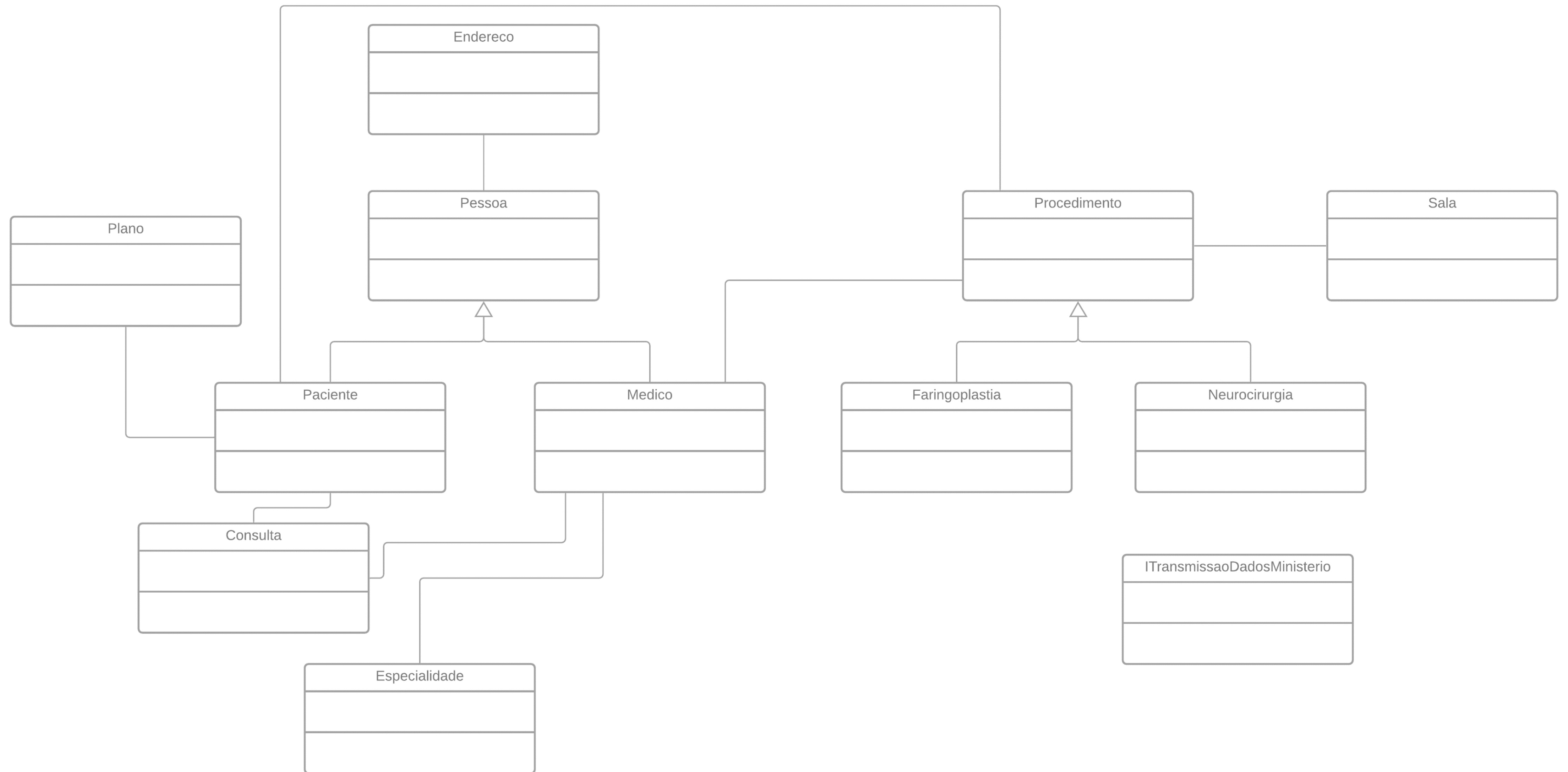
O primeiro passo de qualquer projeto é levantar as classes, procurando nas especificações por substantivos que possam indicar possíveis entidades qualificadas para classes. Também é interessante buscar por entidades que possuam características em comum e que possam compartilhar uma superclasse, indicando heranças.

Por fim, é necessário observar as associações entre as classes.

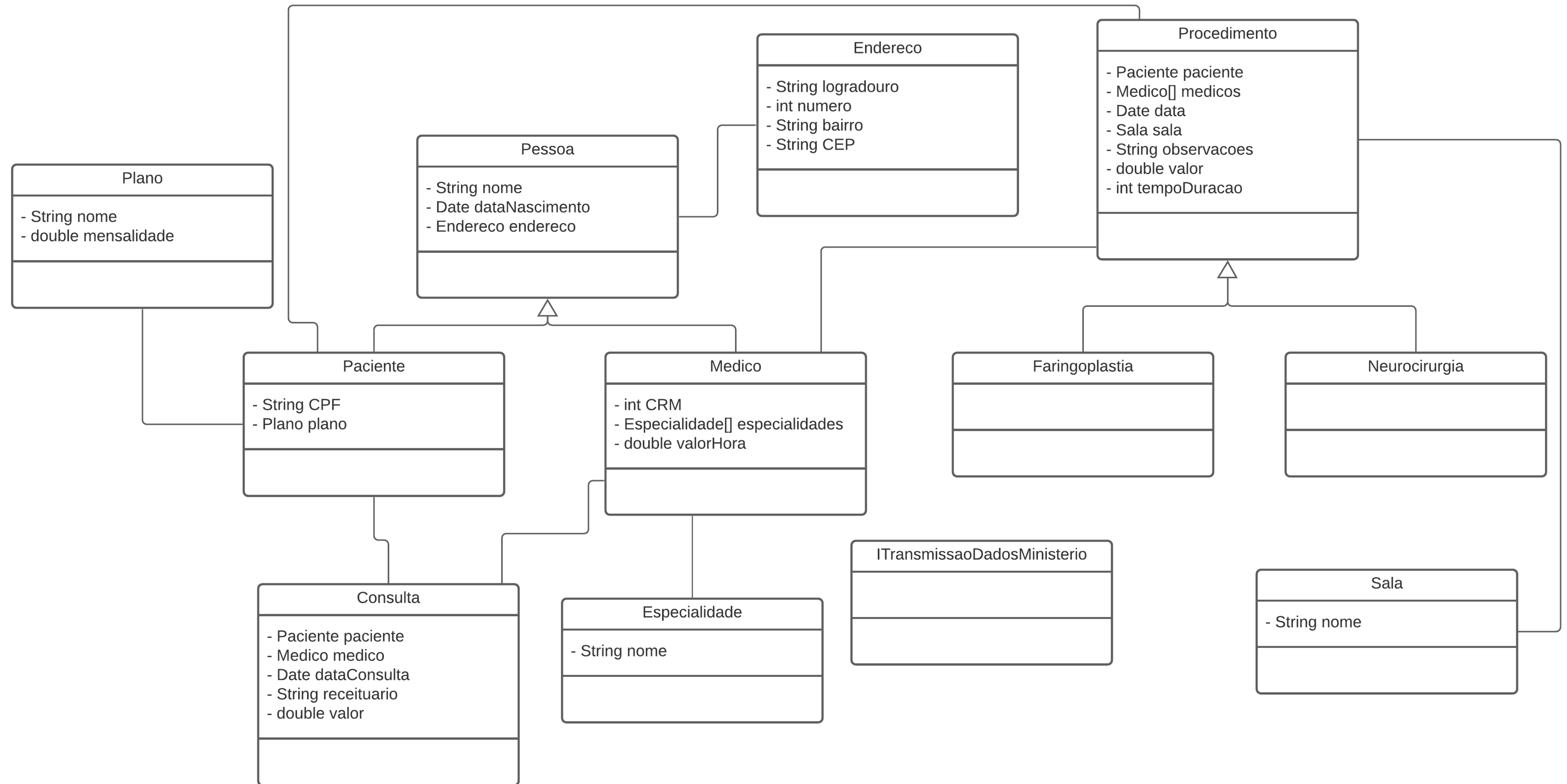
Levantando as classes



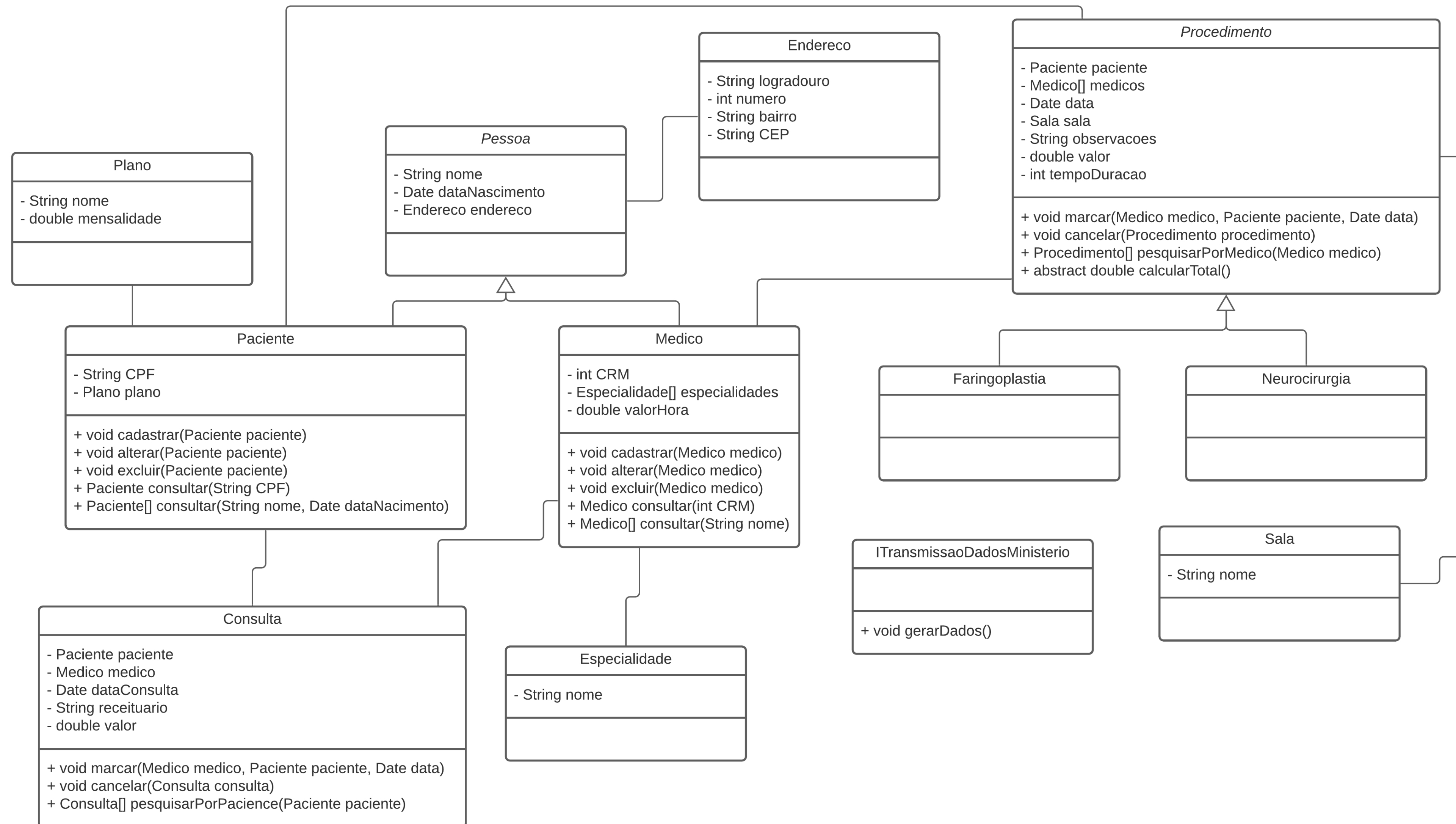
Modelo



Modelo



Modelo



Processo de codificação

Tendo definido todo o modelo de domínio da aplicação, é hora de codificá-lo. Neste momento, algumas decisões devem ser tomadas, e talvez a principal seja: *qual vertente da programação orientada a objetos deveremos seguir?*

Existem duas grandes vertentes de codificar as aplicações: uma que usa o padrão *Business Object*, e outra que não o usa e, assim, evita o chamado *Modelo Anêmico*. Esta abordagem também é conhecida como *Domain Model*. Cada caminho será explicado, começando pelo *Business Object* e, depois, como evitar o *Modelo Anêmico*.

No *Business Object*, uma das principais características da Orientação a Objetos é quebrada: aglutinar dados e comportamentos na mesma unidade de código. Ou seja, os atributos e métodos são separados. Esta atitude é tomada quando desejamos obter uma alta reusabilidade dos comportamentos, mas não desejamos que isto leve a uma interferência no modelo de entidades (conceitos) da aplicação.

Levando em consideração a modelagem para o sistema hospitalar de exemplo, para o conceito de paciente, duas classes seriam criadas: a **Paciente**, que representaria somente o conceito (entidade) a ser manipulado e, assim, teria somente os atributos; e a **PacienteBO** ou **PacienteBusiness**, que conteria somente os métodos para manipular os pacientes.

Note que esta abordagem termina por obrigar a criar **gets** e **sets** para possibilitar a manipulação dos atributos fora da entidade. Isto, embora possa parecer natural, termina por ferir uma outra característica da Orientação a Objetos: o encapsulamento.

Processo de codificação

Ter acesso direto ao atributo por meio de um **get**, mesmo este sendo definido como privado e principalmente possibilitar muda-lo diretamente com um **set**, pode resultar em comportamentos adversos futuramente. Essa abordagem é dita na literatura orientada a objetos como *programar de forma estruturada usando Orientação a Objetos*.

Mesmo com essas ressalvas, esta forma de programação é muito usada, e isso ocorre porque ela facilita o processo de codificação, tornando o código menos complexo e facilitando o seu entendimento. Quanto mais relacionamentos existirem entre as entidades da aplicação, mais essa abordagem mostrará o seu valor.

```
public class CarrinhoDeCompras {  
  
    private String codigo;  
    private Produto[] produtos;  
  
    public void setCodigo(String codigo) {  
        this.codigo = codigo;  
    }  
  
    public String getCodigo() {  
        return this.codigo;  
    }  
  
    public void setProdutos(Produto[] produtos) {  
        this.produtos = produtos;  
    }  
  
    public Produto[] getProdutos() {  
        return this.produtos;  
    }  
}
```

```
public class CarrinhoDeComprasBO {  
  
    public Produtos[] listarProdutos() {  
        //lógica de obter todos os produtos a partir de um  
        //repositório de dados  
    }  
  
    public void adicionarProduto(Produto produto) {  
        //lógica de adicionar um novo produto. Esta de se  
        //preocupar se o produto novo já não existe.  
    }  
  
    public void removerProduto(Produto produto) {  
        //lógica de remover o produto do carrinho  
    }  
  
    public void esvaziar(Produto produto) {  
        //lógica de remover todos os produtos de uma vez  
    }  
  
    public void finalizarPedido() {  
        //lógica de gerar uma venda a partir do carrinho  
    }  
}
```


Processo de codificação

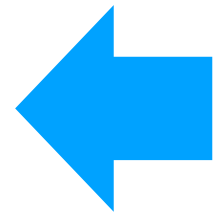
Não usando o *Business Object*, e assim evitando o *Modelo Anêmico*, terminamos seguindo 100% os preceitos da OO: juntar dados e comportamentos. Com isso, os métodos de manipulação dos atributos e os atributos estão juntos na mesma classe. Dessa forma, apenas uma classe é criada, no caso a **Paciente**.

Esta abordagem preza que não se deve criar **gets** e **sets** de forma indiscriminada, mas que estes sejam uma situação de exceção. Métodos de negócio, que expressam as necessidades, são os que devem ser utilizados para acessar os atributos diretamente.

Esta forma de programação é a mais defendida por grandes gurus da Orientação a Objetos, como Martin Fowler. Entre os motivos de defesa dessa abordagem é que ela gera um menor acoplamento entre as classes da aplicação, já que diminuímos a quantidade de classes e, conseqüentemente, de relacionamentos também. Além disto, não temos um modelo de domínio pobre, limitando-nos a um simples punhado de **gets**, **sets** e atributos.

De fato, um *Modelo Anêmico* fere os preceitos da OO. Mas não se deve tirar o mérito dessa abordagem, pois a complexidade de codificação e entendimento são perceptíveis para aplicações com grande quantidade de entidades e, conseqüentemente, de relacionamentos - principalmente para iniciantes.

```
public class CarrinhoDeCompras {  
  
    private String codigo;  
    private Produto[] produtos;  
  
    public Produtos[] listarProdutos() {  
  
        // lógica de obter todos os produtos a partir de um  
        // repositório de dados  
    }  
  
    public void adicionarProduto(Produto produto) {  
  
        // lógica de adicionar um novo produto. Esta deve se  
        // preocupar se o produto novo já não existe.  
    }  
  
    public void removerProduto(Produto produto) {  
  
        // lógica de remover o produto do carrinho  
    }  
  
    public void esvaziar(Produto produto) {  
  
        // lógica de remover todos os produtos de uma vez  
    }  
  
    public void finalizarPedido() {  
  
        // lógica de gerar uma venda a partir do carrinho  
    }  
}
```



UML: Diagramas de Casos de Uso

Introdução

A qualidade de um sistema está fortemente associada ao atendimento das necessidades do cliente, dos patrocinadores e dos usuários desse sistema, no que diz respeito, em última instância, às funções que o sistema precisa executar. Assim sendo, talvez a atividade mais importante durante a análise de um sistema seja a conversa com o cliente.

Na conversa, precisamos compreender e registrar corretamente as necessidades do cliente com respeito ao sistema que será desenvolvido, de forma que os demais membros da equipe entendam e não tenham dúvida sobre o que foi registrado.

A atividade de compreensão e registro das necessidades do cliente é conhecida como *levantamento (e captura) dos requisitos do sistema*. Poderíamos, claro, fazer os registros em linguagem coloquial, em formato livre, mas textos escritos dessa forma são susceptíveis a ambiguidades, e nós devemos evitar isso.

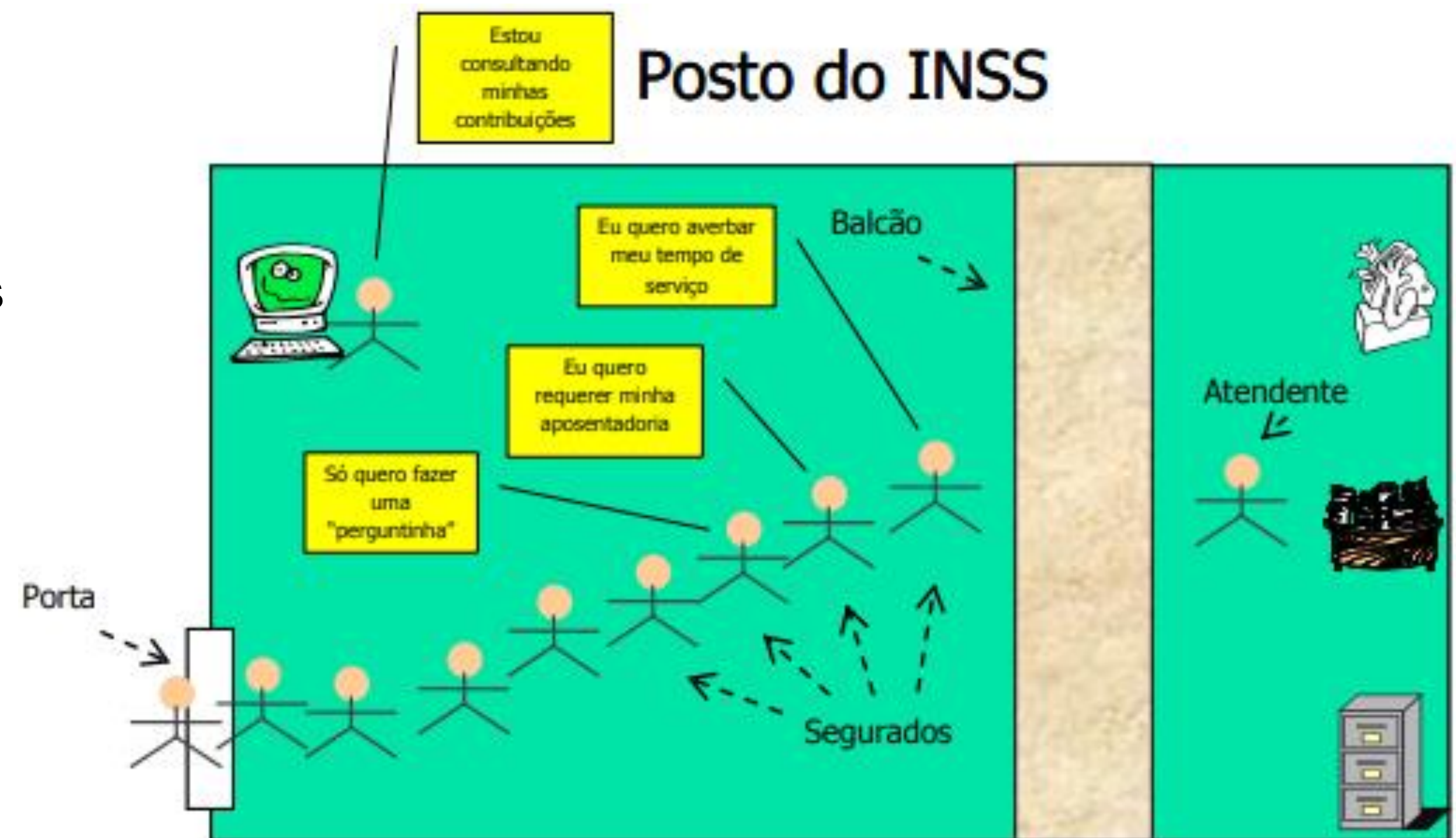
Iniciaremos, agora, o estudo dos diagramas de casos de uso da UML, que são usados para especificar os requisitos funcionais de um sistema. Esses diagramas são também usados para confirmar com o cliente o que ele disse durante o levantamento e para passar essas informações, de forma precisa, sem ambiguidade, à equipe de projeto e construção do sistema.

Enfoques dos Diagramas de Casos de Uso

Os diagramas de casos de uso da UML têm dois enfoques: o de negócios e o de sistemas. Os dois enfoques são úteis, mas precisamos distingui-los porque, durante o trabalho de análise, eles são usados em tempos diferentes.

Imagine um atendente de balcão do INSS que atende segurados em suas diversas necessidades. O primeiro segurado da fila deseja executar o processo de negócio **Averbar Tempo de Serviço**. Para isso, o atendente consulta um arquivo convencional, pega um ou mais formulários sobre a mesa e, possivelmente, consulta o tempo de contribuição do segurado no sistema.

Durante a realização desse processo de negócio, o atendente e o segurado trocam informações e documentos, ambos participando, portanto, do processo. Os dois são ditos *atores do negócio*, pois cada um deles desempenha seu papel específico no processo de negócio. O atendente também consulta o tempo de contribuição do segurado no sistema. Por usar o sistema, o atendente é *ator do sistema*, além de ator do negócio. O segurado, no entanto, não interagem com o sistema e, portanto, não é ator do sistema.

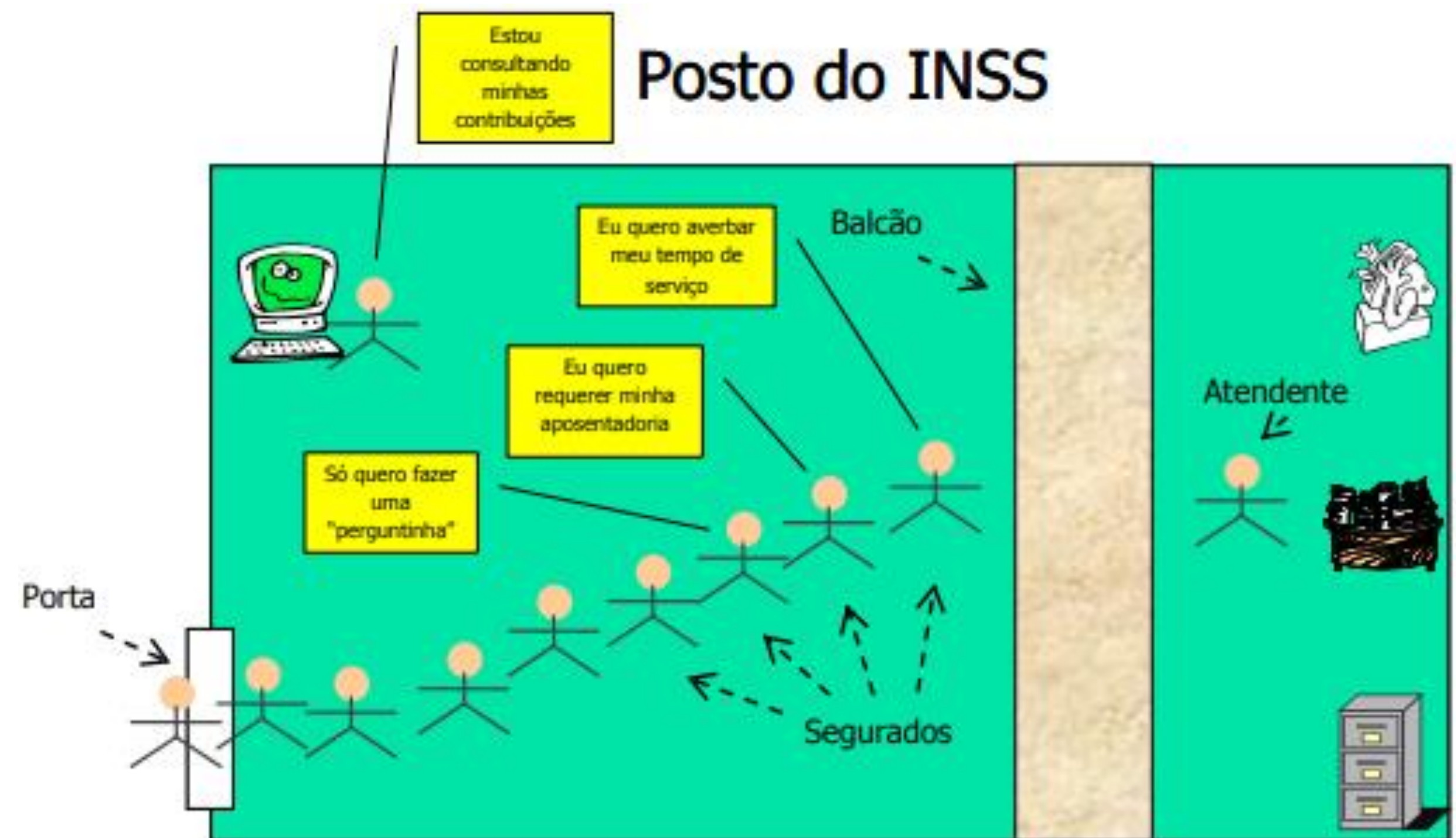


Enfoques dos Diagramas de Casos de Uso

Um indivíduo só é considerado ator de um sistema se ele é usuário desse sistema, ou seja, se ele insere um dado, pressiona um botão ou tecla, passa um cartão, toca a tela e seleciona uma opção, rola uma tela ou se identifica biometricamente, interagindo com o sistema.

Situação análoga pode acontecer com os demais processos de negócio dos quais o atendente do INSS participa: **Registrar Requerimento de Aposentadoria e Retirar Dúvidas.**

A figura ao lado também ilustra a situação em que um segurado precisa consultar o seu tempo de contribuição e acessa diretamente outra funcionalidade do sistema do INSS: **Consultar Tempo de Contribuição Via Terminal do Cidadão.** Nesse caso, o processo de negócio do INSS **Informar Tempo de Contribuição** é realizado totalmente pela funcionalidade **Consultar Tempo de Contribuição Via Terminal do Cidadão.** O segurado é ator de um processo de negócio e de uma funcionalidade do sistema.



Enfoques dos Diagramas de Casos de Uso

De maneira geral, processos de negócio envolvem relações entre organizações, entre organizações e seus cliente e fornecedores, entre colaboradores de uma organização e entre todas as demais entidades que precisam colaborar de alguma forma para que as organizações cumpram seus objetivos. Esse é o enfoque de negócio e, segundo esse enfoque, os casos de uso de negócio representam os processos realizados e os atores de casos de uso de negócio são todos os que participam deles.

Eventualmente, um processo em uma organização é informatizado, parcial ou totalmente. As funcionalidades, demais características desse sistema e seus usuários dizem respeito ao enfoque do sistema. Segundo esse enfoque, os casos de uso de sistema representam as funcionalidades do sistema; os atores dos casos de uso de sistema são seus usuários.

Os processos de negócio podem ser modelados com o uso de diagramas de casos de uso de negócio e as funcionalidades de um sistema podem ser modeladas com o uso de diagramas de casos de uso de sistema. A notação usada nos dois pode ser exatamente a mesma, contanto que mencionemos no diagrama a que enfoque corresponde.

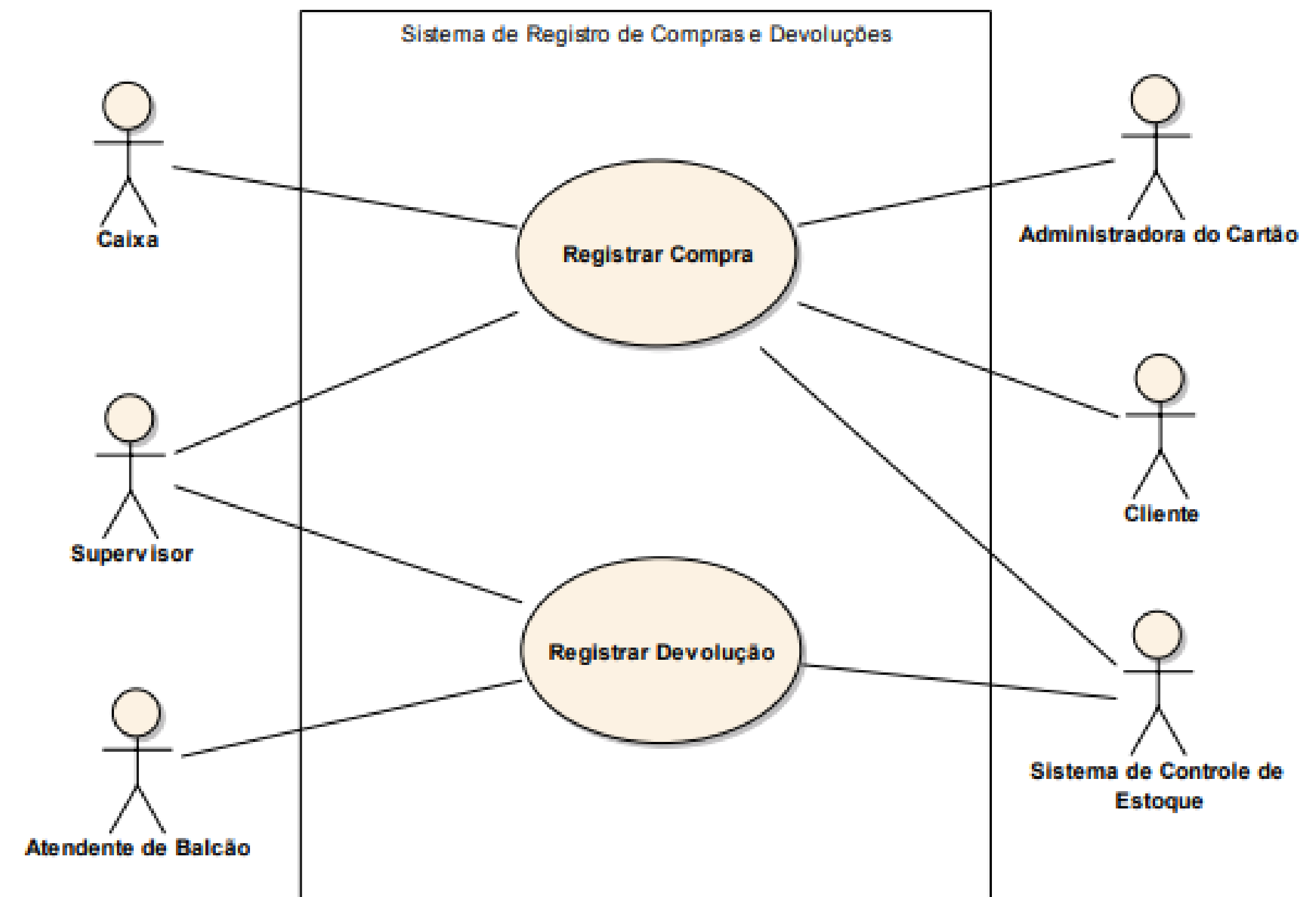
Enfoques dos Diagramas de Casos de Uso

Casos de uso de negócio não são o foco deste texto, apesar de ser uma área de grande relevância para a especificação dos requisitos do negócio.

Elaboramos o diagrama de casos de uso de sistema para representar os requisitos funcionais, ou seja, as funções que deverão estar disponíveis no sistema para que as necessidades que motivaram a sua construção sejam satisfeitas.

Este é o enfoque que teremos no curso. Por isso, ao mencionarmos simplesmente “caso de uso” e “ator” estaremos nos referindo neste contexto a “caso de uso de sistema” e “ator de caso de uso de sistema”, respectivamente.

A seguir apresentaremos a notação gráfica usada nos diagramas de casos de uso e os conceitos de cada elemento da notação. A figura ao lado ilustra seis atores e dois casos de uso de um Sistema de Registro de Compras e Devoluções de um supermercado hipotético.

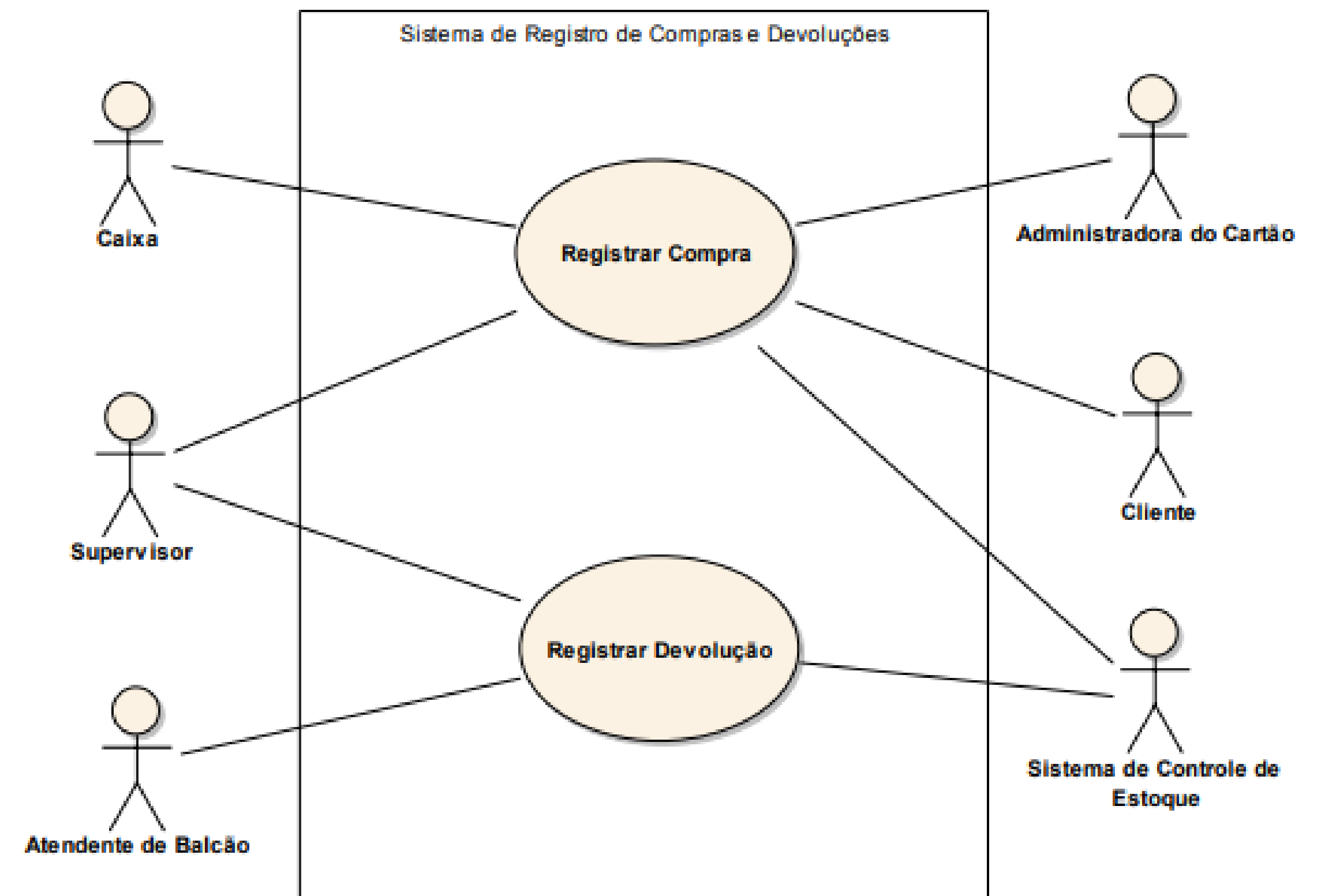


Os Atores

O termo *ator do sistema* se refere ao papel que alguém ou alguma coisa interpreta enquanto interage com o sistema sendo modelado. No diagrama da figura ao lado, os "bonequinhos" representam atores do Sistema de Registro de Compras e Devoluções.

A UML se refere à representação gráfica como sendo de *stick men*. Embora, em boa parte das vezes, atores sejam seres humanos, eles também podem ser outras coisas, como dispositivos eletrônicos ou outros sistemas computacionais que se relacionam com o sistema em estudo.

Um único indivíduo pode interpretar o papel de vários atores (por exemplo, Joel, além de ser – ou interpretar o papel de – caixa, pode atuar como o atendente de balcão); vários indivíduos podem interpretar o papel de um único ator (por exemplo, Joel e Pedro podem ser, ambos, atores caixa). Atores podem participar de um ou mais casos de uso; no nosso exemplo, os supervisores podem participar de registros de compras e de devoluções.



Os Atores

O nome do ator, idealmente uma expressão breve no singular, deve sugerir claramente o papel que o ator representa, dentro do jargão do negócio, ou seja, não deve ser, por exemplo, uma expressão de uso restrito ao ambiente da equipe de modelagem. A tabela abaixo mostra alguns exemplos do que usar e do que não usar como nomes de atores.

<i>Usar</i>	<i>Não Usar</i>
Contador	Contadores
Sistema de Controle de Estoque ou simplesmente SCE	João
Sensor de Presença	Usuário

A versão atual da UML permite representarmos um ator de uma forma gráfica mais sugestiva quanto ao seu tipo, ou seja, atores sistemas podem ser representados por figuras de computadores. Outra representação alternativa é com a notação de classes, ou seja, retângulos com a palavra-chave «*actor*» em seu topo, já que atores também podem ser entendidos como categorias ou classes de usuários dos sistemas.

Quando desenhamos o retângulo que representa os limites do sistema, os atores são colocados fora dele. Isso significa que, para o propósito do modelo a ser desenvolvido, não interessa saber como eles agem, qual a lógica de funcionamento e como são seus detalhes internos; o que interessa é apenas o que eles fazem durante a interação com o sistema que está sendo estudado.

Os Atores

Eles podem aparecer repetidos em um mesmo diagrama. Para muitos analistas, isso só tem efeito cosmético, pois possibilita eliminar cruzamentos entre relacionamentos, o que não se justifica pelo aspecto prático e, na maioria das vezes, de clareza do modelo. Isso, pelo contrário, só adiciona complexidade visual ao modelo.

Resta, agora, identificar os atores do sistema. Essa atividade parece, em princípio, simples, mas devemos ter em mente as diferenças entre participar do processo de negócio e interagir com o sistema. Os atores são descobertos classificando-se os indivíduos que efetivamente usarão o sistema ou identificando-se o software – tipicamente outros sistemas – ou hardware externo que inicia um caso de uso do sistema ou que é necessário durante a execução desse caso e uso.

Não se pode ter certeza de que todos os atores foram descobertos antes de descrevermos em detalhes todos os casos de uso do sistema, pois durante a especificação é que entendemos quem faz o que no sistema.

Um ator não é uma pessoa específica. Quase sempre é possível achar pelo menos duas pessoas cujas responsabilidades e atividades se encaixam no perfil de um mesmo ator, isso para cada ator do modelo. Em outras palavras: se você não achou mais do que uma pessoa que interpreta o papel de determinado ator, é bem provável que você tenha modelado a pessoa, e não o ator, embora exista a possibilidade de um papel ser interpretado por somente uma pessoa. Um exemplo disso é em um Sistema de Controle de Clientes de uma padaria, em que o dono, o *Seu Manoel*, é o único que executa o caso de uso **Manter Lista Negra de Clientes**.

Os Casos de Uso

Um caso de uso corresponde a um conjunto de ações executadas durante a realização de uma funcionalidade do sistema. Casos de uso concentram-se nas relações entre as funções do sistema e os usuários que delas participam de alguma forma. Um caso de uso de sistema tem as seguintes características:

- Captura as ações para a realização de uma função do sistema, enfocando as interações entre os usuários e o sistema;
- É uma unidade coerente de passos, expressa como uma transação entre os atores e o sistema, compondo-se tipicamente de várias ações dos atores e respostas do sistema;
- É uma sequência de ações que produzem resultados observáveis de valor para os usuários;
- Expressa o que acontece quando um caso de uso é executado, incluindo suas possíveis variações. Não há preocupação em como os participantes executam suas ações, embora a ordem delas seja relevante.

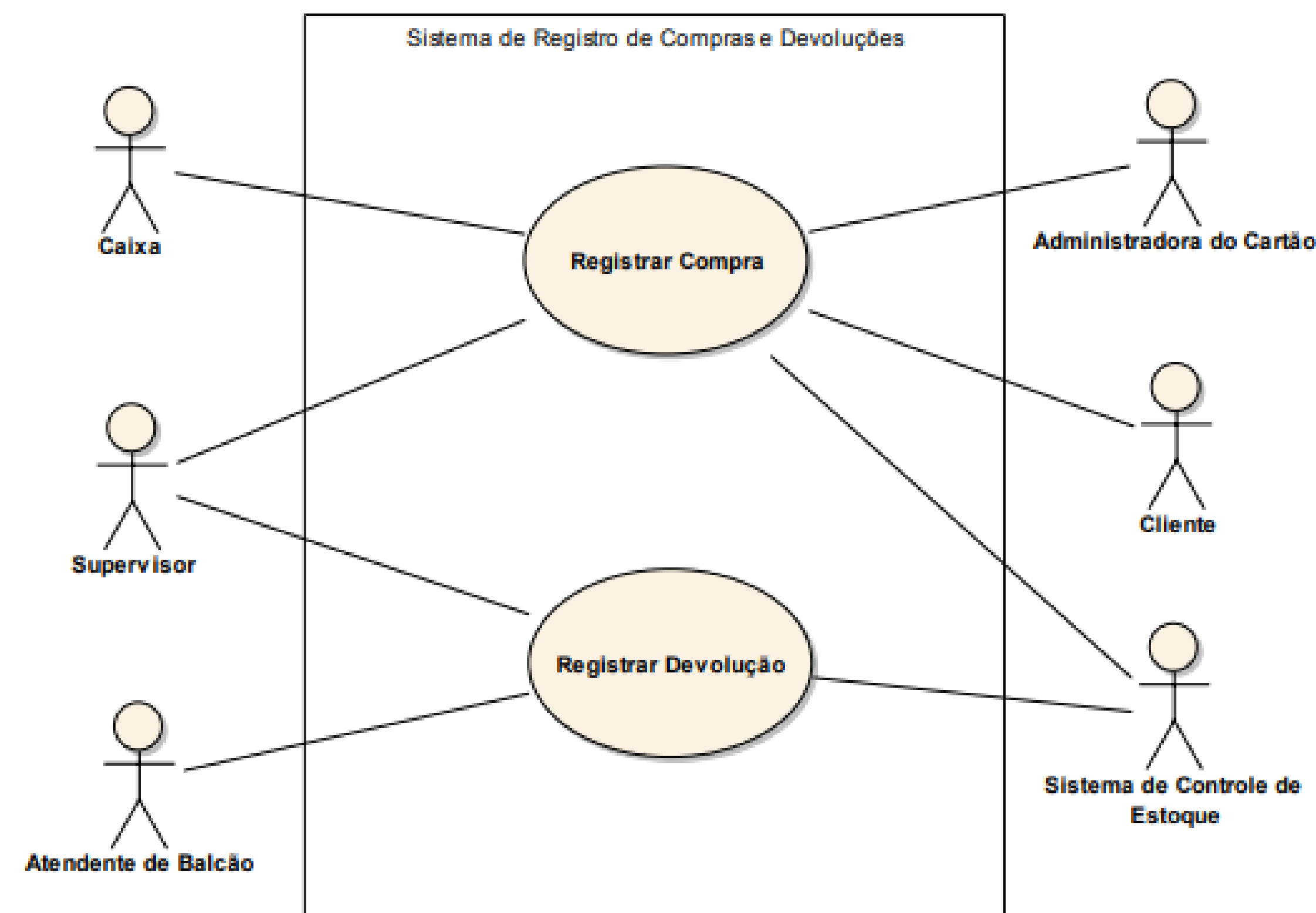
Nos diagramas, os casos de uso são denotados por ovais ou elipses que representam as funcionalidades do sistema. Casos de uso têm nomes que devem ser ativos, ou seja, um verbo no infinitivo concatenado a um substantivo. Exemplos: **Aprovar Crédito**, **Registrar Venda de Automóvel** e **Aprovar Fatura**. Os nomes são colocados dentro ou abaixo das ovais. De certa forma, a colocação do nome sob a oval traz mais complexidade visual ao diagrama, já que nome e oval passam a constituir dois elementos visuais distintos.

Os Casos de Uso

Por exemplo: do caso de uso **Registrar Compra** participam o **Caixa** (que registra os itens de compra no sistema), o **Cliente** (que digita a senha do cartão de crédito), a **Administradora do Cartão** (que aprova o débito do valor no cartão), o **Sistema de Controle de Estoque** (que é informado da compra para que possa controlar o estoque), e o **Supervisor** (que eventualmente retira um item de venda da lista de compras). A forma como eles participam não é especificada no diagrama.

Usamos, basicamente, duas técnicas para descobrir as funcionalidades do novo sistema:

- Iniciando a partir da relação dos atores: para cada ator, identificar as funcionalidades de que necessita.
- Iniciando a partir da relação dos eventos. Isso é feito em três etapas, conforme segue:
 - Identificar os eventos externos aos quais o sistema deve responder;
 - Associar os eventos aos atores que atuam para tratá-los;
 - Identificar as funcionalidades que eles necessitam executar em resposta aos eventos.



Os Casos de Uso/Fronteira do Sistema

A primeira técnica é a mais simples e a mais comumente usada. Apenas para ilustrar a segunda técnica, tomemos como exemplo o evento de recebimento de uma nota fiscal de entrega pelo setor de controle de estoque de uma organização. O encarregado do estoque que recebe a nota (o ator) precisa de uma funcionalidade (o caso de uso) para registrar a chegada dessa fatura, que indica a chegada de material para reposição de estoque. Essa funcionalidade poderia, por exemplo, adicionar os novos itens no estoque e indicar ao sistema de contas a pagar da organização que há um novo compromisso a ser pago.

Fronteira do Sistema:

A fronteira do sistema, também chamada limite ou escopo do sistema, é representada pelo retângulo que contém os casos de uso. A representação da fronteira é opcional, segundo a UML; colocamos a fronteira quando queremos e podemos, já que, às vezes, não conseguimos definir uma fronteira retangular com os casos de uso dentro e os atores fora. A fronteira é colocada para salientarmos o que é o sistema e, portanto, é nosso interesse estudar. Estar fora da fronteira, em contrapartida, significa que não estamos interessados, para efeito de nosso estudo, nos detalhes internos e na lógica de seu funcionamento. Por essa razão, os atores do sistema ficam necessariamente fora da fronteira.

No topo do retângulo, internamente a ele, centralizado, colocamos o nome do sistema.

Relacionamentos

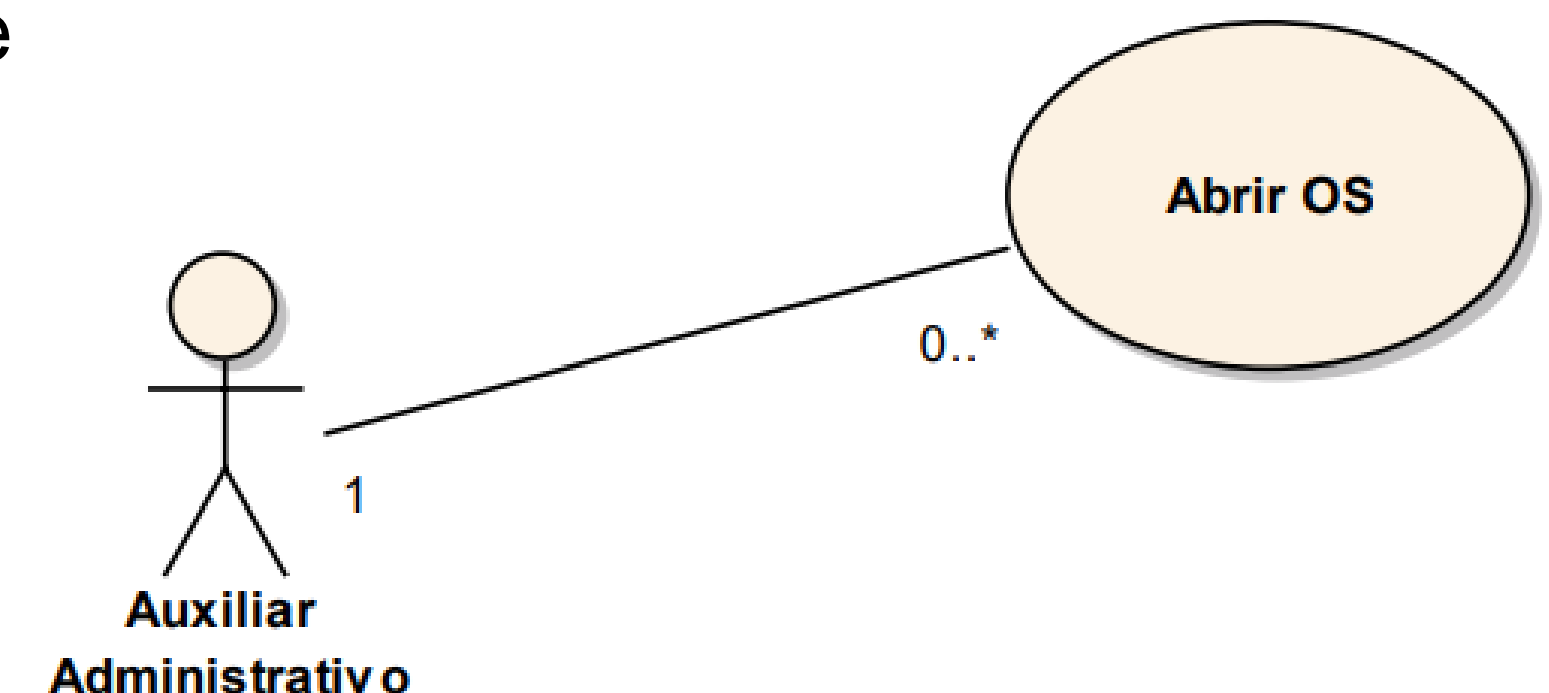
Associação:

O único relacionamento possível entre um ator e um caso de uso leva o nome particular de associação. As associações especificam quais atores participam de quais casos de uso, sendo representadas nos diagramas de casos de uso por meio de segmentos de retas, curvas ou poligonais que ligam os atores aos casos de uso de que participam.

A UML admite que ambas as pontas das associações possuam multiplicidades. Quando um ator tem uma associação com um caso de uso com uma multiplicidade que é maior que um na ponta da associação do lado do caso de uso, isso significa que o dado ator pode estar envolvido com múltiplas execuções (instâncias) do caso de uso. A natureza específica desse relacionamento múltiplo depende do contexto e não é definida na UML.

Dessa forma, um ator pode iniciar uma ou mais instâncias de um mesmo caso de uso concorrentemente ou elas podem ser mutuamente excludentes no tempo. Por exemplo, um colaborador pode iniciar ao mesmo tempo um determinado tipo de atendimento a vários clientes no balcão ou instituir uma fila para atendimento um a um.

Quando essa multiplicidade é maior do que um do lado do ator, significa que mais de uma instância daquele ator participa do caso de uso. A maneira como os atores participam depende do contexto (concorrentemente, de forma colaborativa ou numa sequência) e também não é definida na UML.



Relacionamentos

Associação:

Quando não há multiplicidades nas pontas das associações é porque elas ainda não foram determinadas ou essa informação não é relevante para o propósito do modelo. Nesse caso, deve-se admitir que um ator participa de um número qualquer de instâncias do caso de uso e participa do caso de uso qualquer número de instâncias do ator (usuários daquela classificação).

As formas segundo as quais os atores participam dos processos de negócio não são capturadas nos diagramas de casos de uso. Casos de uso precisam ser descritos (especificados) para que os detalhes fiquem esclarecidos. Especificações de casos servem ainda para validação junto ao usuário, funcionando com um "contrato" entre o usuário e o desenvolvedor.

Especialização-Generalização de Atores e Casos de Uso:

Os relacionamentos de especialização-generalização podem ocorrer entre dois atores e entre dois casos de uso. Esses relacionamentos são representados por setas com pontas triangulares vazadas, que indicam o sentido da generalização; o sentido oposto é o da especialização.

Relacionamentos

Especialização-Generalização de Atores e Casos de Uso:

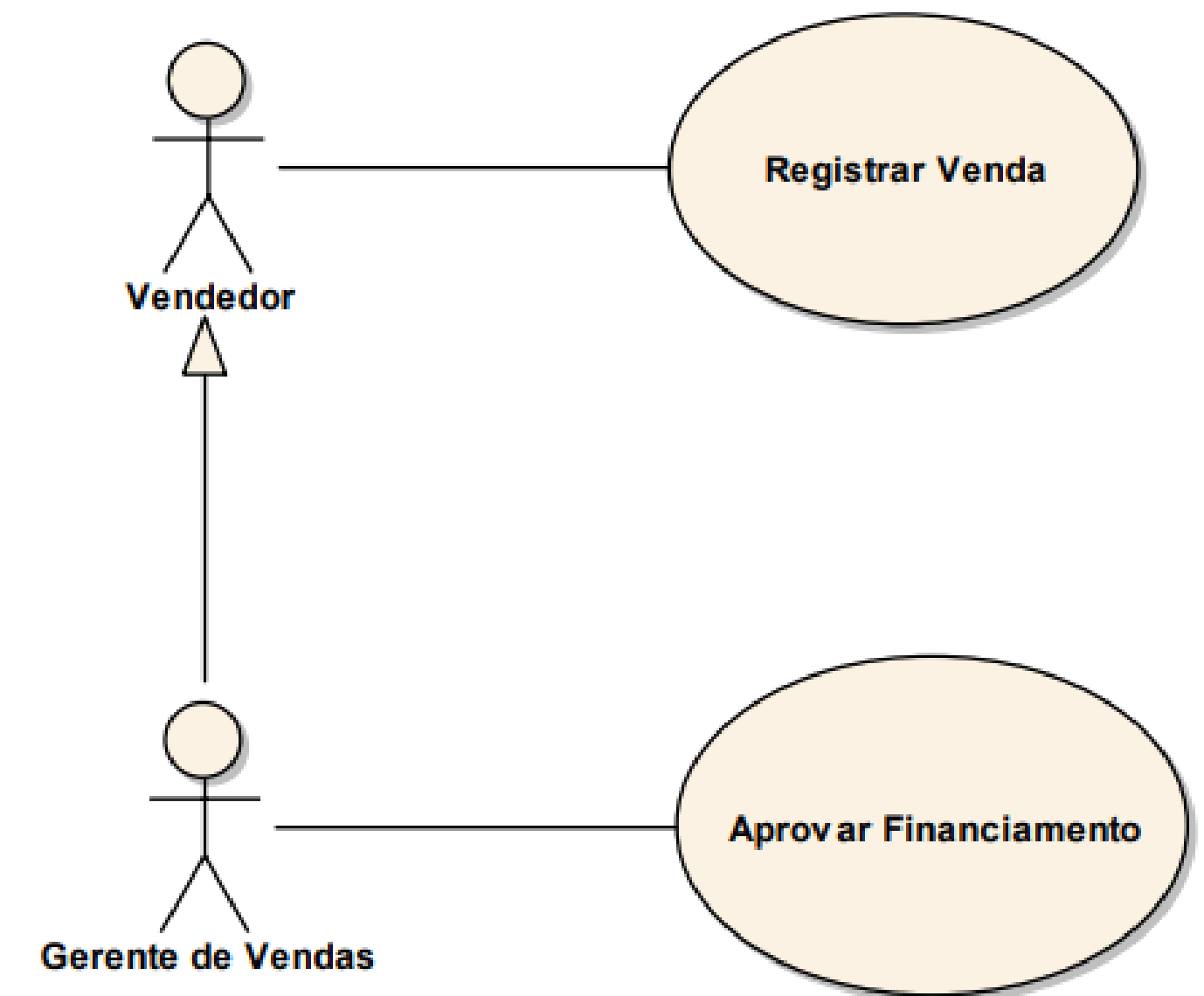
Muitos atores podem interpretar o mesmo papel em um determinado caso de uso. A figura ao lado ilustra a situação em que o ator Gerente de Vendas aprova financiamentos, mas também pode atuar como Vendedor, vendendo automóveis.

De fato, um gerente de vendas vai querer atuar como um vendedor (e ganhar toda a comissão pela venda) quando, por exemplo, um cliente antigo seu chega à loja de automóveis para trocar os cinco automóveis Mercedes-Benz da família, como costumeiramente faz todo início de ano.

Essas participações específicas, por sinal, são as diferenças de comportamento dos atores que justificam suas especializações. Isso quer dizer que o modelo da figura ao lado só se justifica se houver pelo menos um caso de uso associado a Gerente de Vendas.

Note que a recíproca não é verdadeira, ou seja, um vendedor não poderia entrar no sistema para registrar a aprovação de um financiamento.

ATENÇÃO: Especializações-generalizações de atores são os únicos relacionamentos possíveis entre atores em um diagrama de casos de uso.

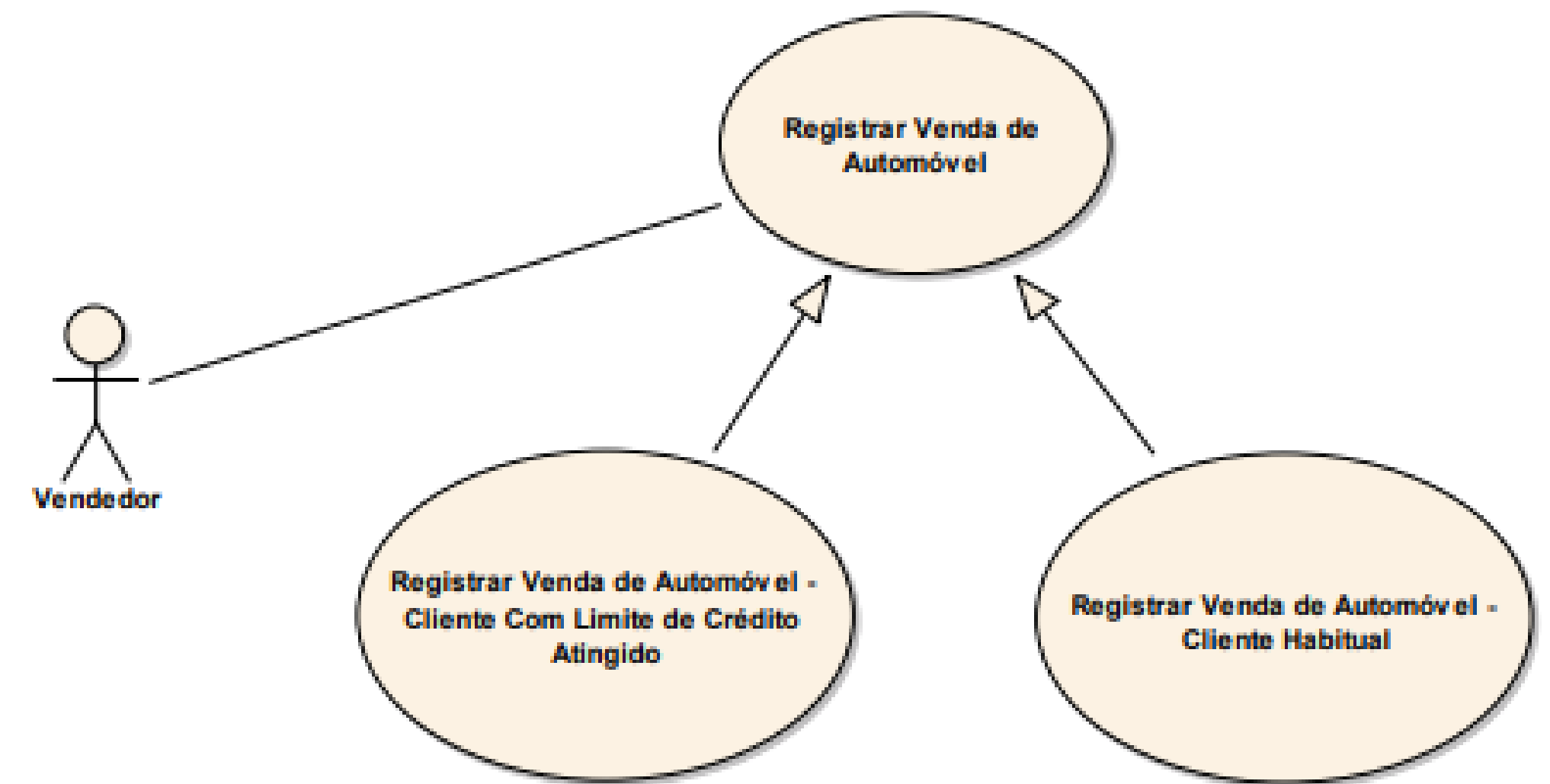


Relacionamentos

Especialização-Generalização de Atores e Casos de Uso:

A figura ao lado ilustra uma situação em que o vendedor registra a venda de um automóvel; esse processo pode ser feito de três maneiras ligeiramente diferentes:

- Para o caso em que o limite de crédito do comprador se encontra ultrapassado (caso de uso Registrar Venda de Automóvel – Cliente Com Limite de Crédito Atingido);
- Para o caso em que o cliente é um cliente habitual (caso de uso Registrar Venda de Automóvel – Cliente Habitual);
- Para o caso em que ocorre o processo básico, normal, que corresponde ao caso de uso base Registrar Venda de Automóvel. Esse caso de uso é chamado de caso de uso base.



A especialização de um caso de uso significa, portanto, que o caso de uso que especializa (por exemplo, o caso de uso Registrar Venda de Automóvel – Cliente Com Limite de Crédito Atingido) é um pouco diferente do caso de uso especializado (o caso de uso Registrar Venda de Automóvel), podendo acrescentar ou remover passos deste.

O processo de venda de automóvel para clientes com limite de crédito atingido é realizado com base no processo básico para clientes que compram a crédito, possivelmente demandando outras informações do cliente, outras garantias e a anuência do gerente de vendas. O processo de venda para um cliente habitual possivelmente altera o prazo de entrega e agiliza o processo de coleta de informações do cliente.

Relacionamentos

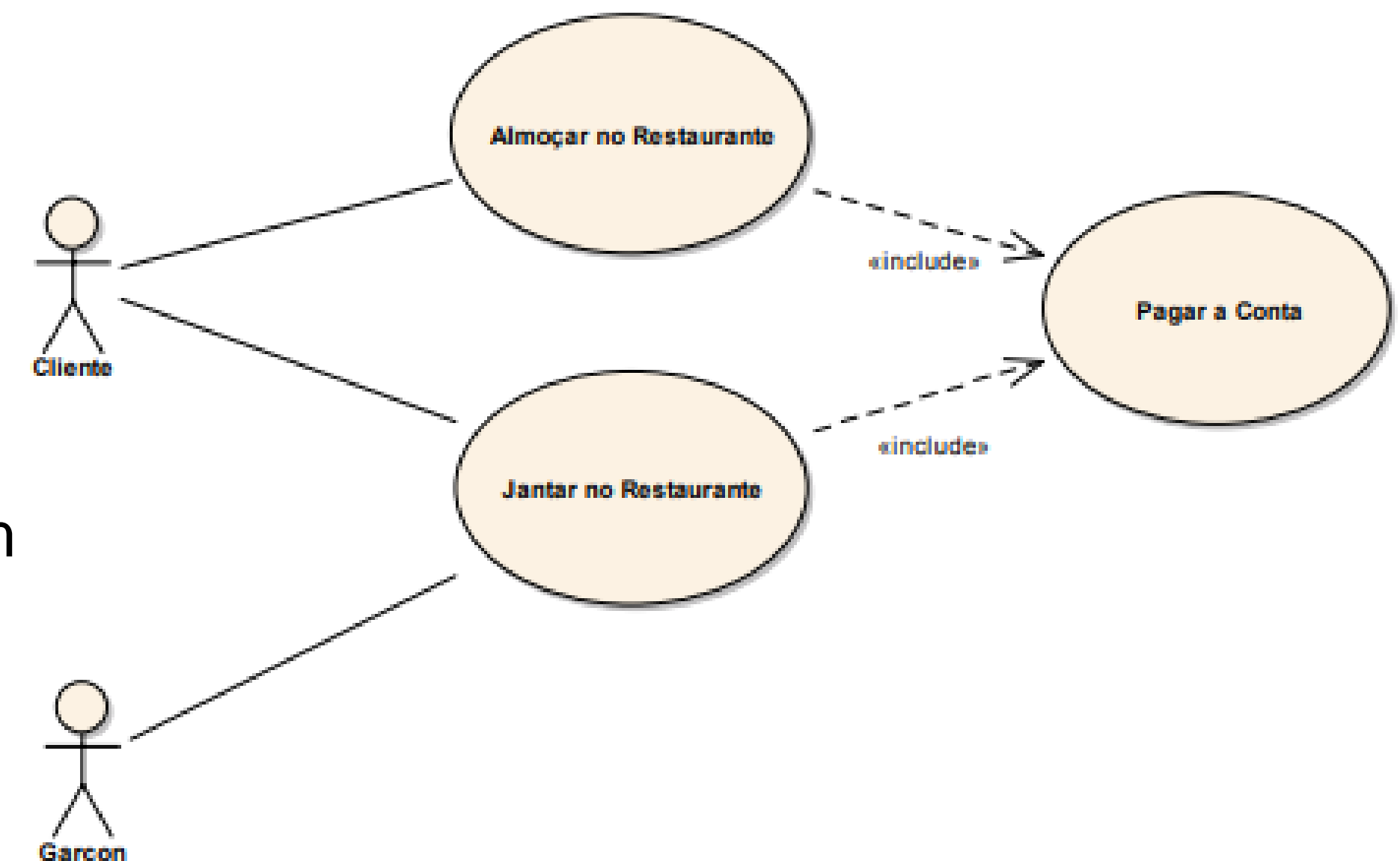
Inclusão de um Caso de Uso em Outro:

Inclusão é um relacionamento somente usado entre dois casos de uso. Uma inclusão ocorre em um caso de uso quando é necessária a invocação do comportamento especificado em outro; em determinado momento da especificação de um processo "se chama" (se faz referência a) o outro.

Uma inclusão se faz necessária quando se tem um conjunto relevante de ações que é comum a dois ou mais casos de uso. Nesse caso, se fatora esse comportamento em um novo caso de uso, ou seja, se retira dos casos de uso esse conjunto de passos que é repetido, colocando-o em um caso de uso à parte.

A figura ao lado ilustra uma situação em que determinado restaurante opera de formas diferentes no almoço e na janta: no almoço ele oferece refeição por quilo e à noite a refeição é oferecida à la carte. Existe um conjunto de ações comum a ambos os casos de uso, que ocorrem obrigatoriamente e de forma idêntica segundo as regras do restaurante: o pagamento pela refeição.

O pagamento da conta, em geral, não é um processo trivial, pois envolve a escolha do meio de pagamento e o cumprimento das formalidades correspondentes. Não ser trivial significa demandar um número relevante de passos para a sua especificação, o que justifica a criação de um caso de uso para representá-lo no diagrama.



Relacionamentos

Inclusão de um Caso de Uso em Outro:

Outro aspecto importante é que o pagamento da conta é obrigatório, no almoço e no jantar... Pelo menos nesse restaurante, pois a inclusão está associada à obrigatoriedade; se o pagamento fosse facultativo, por alguma razão ou forma (se a despesa pudesse ser "pendurada" ou "deixada pra lá", por exemplo), não seria uma inclusão, e sim uma extensão.

Extensão de um Caso de Uso por Outro:

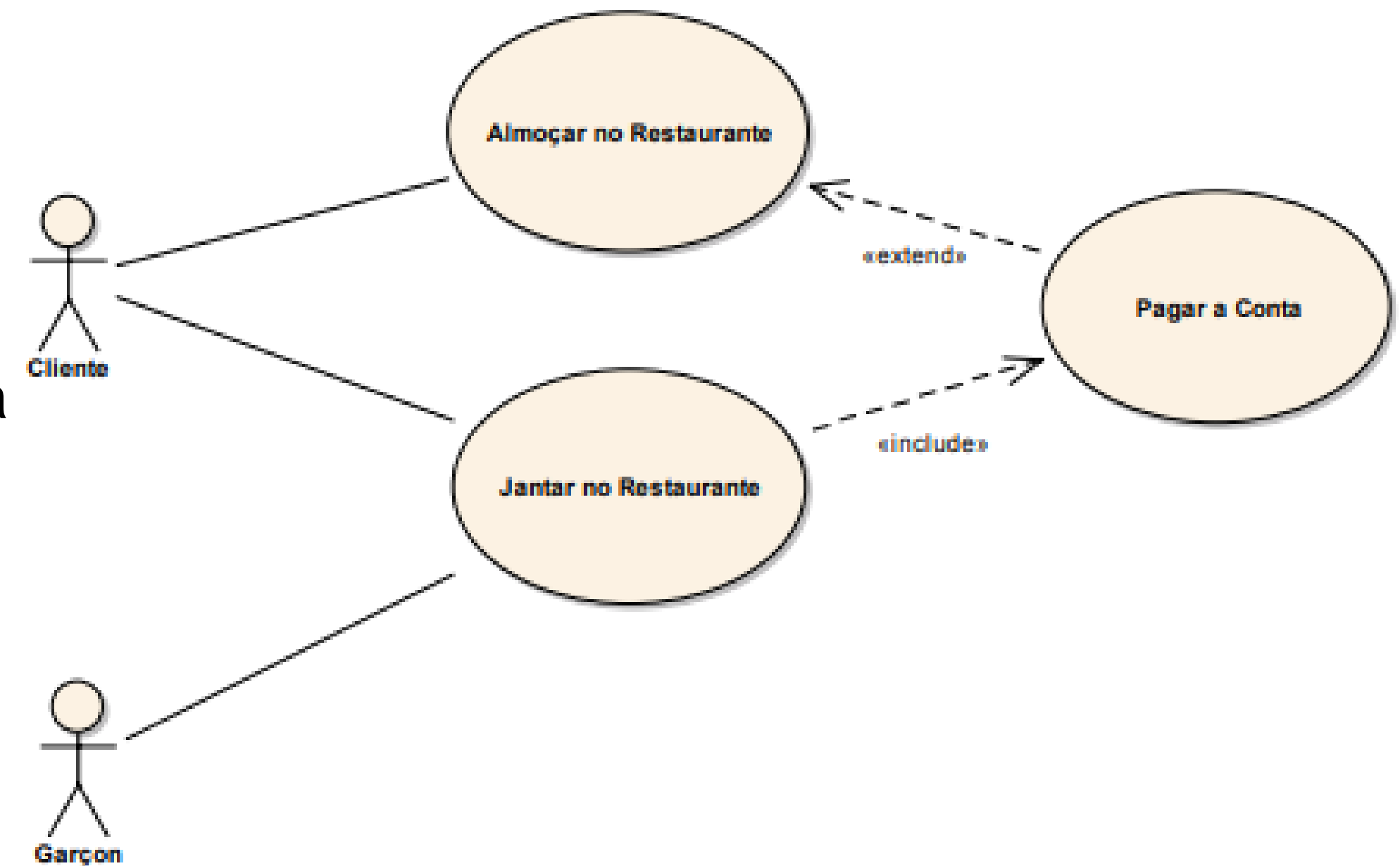
A extensão também é um relacionamento que só ocorre entre dois casos de uso. Extensões são essencialmente generalizações, só que possuem regras mais explícitas. Quando tratamos de generalizações/especializações, mencionamos que elas não perduram por muito tempo no ciclo de modelagem porque, mais cedo ou mais tarde, precisamos especificar as diferenças entre as especializações e entre elas e o caso de uso base, sob pena do nosso modelo ficar incompleto. É recomendado que se use extensão "quando você estiver descrevendo uma variação do comportamento normal de um caso de uso e deseja utilizar a forma mais controlada". Essa forma mais controlada é feita especificando-se pontos de extensão, que não veremos neste texto. Cremos que modelar extensões sem os pontos de extensão, ao invés de generalizações, seja suficiente para cumprir os objetivos da modelagem de casos de uso no nível conceitual.

A extensão também significa a invocação de um caso de uso por outro, mas difere da inclusão na situação em que, de acordo com as regras do negócio, a invocação feita ao outro caso de uso não ocorre obrigatoriamente. Difere ainda porque a invocação ocorre no sentido inverso da seta.

Relacionamentos

Extensão de um Caso de Uso por Outro:

Voltando à situação ilustrada pelo último exemplo, imagine a situação em que o restaurante faz promoções para os almoços, de modo que os clientes habituais não pagam a refeição a cada determinado número de idas ao restaurante. Isso significa que, de acordo com as regras do restaurante, o pagamento pelo almoço pode não ocorrer. Nesse caso, o modelo fica conforme ilustra a figura ao lado (lê-se Pagar Conta estende Almoçar no Restaurante).



Extensão significa que, em determinada posição da execução da funcionalidade representada pelo caso de uso, outro caso de uso é invocado opcionalmente. O nome extensão significa que o caso de uso invocado estende (aumenta) o caso de uso que invoca, acrescentando novas ações a ele.

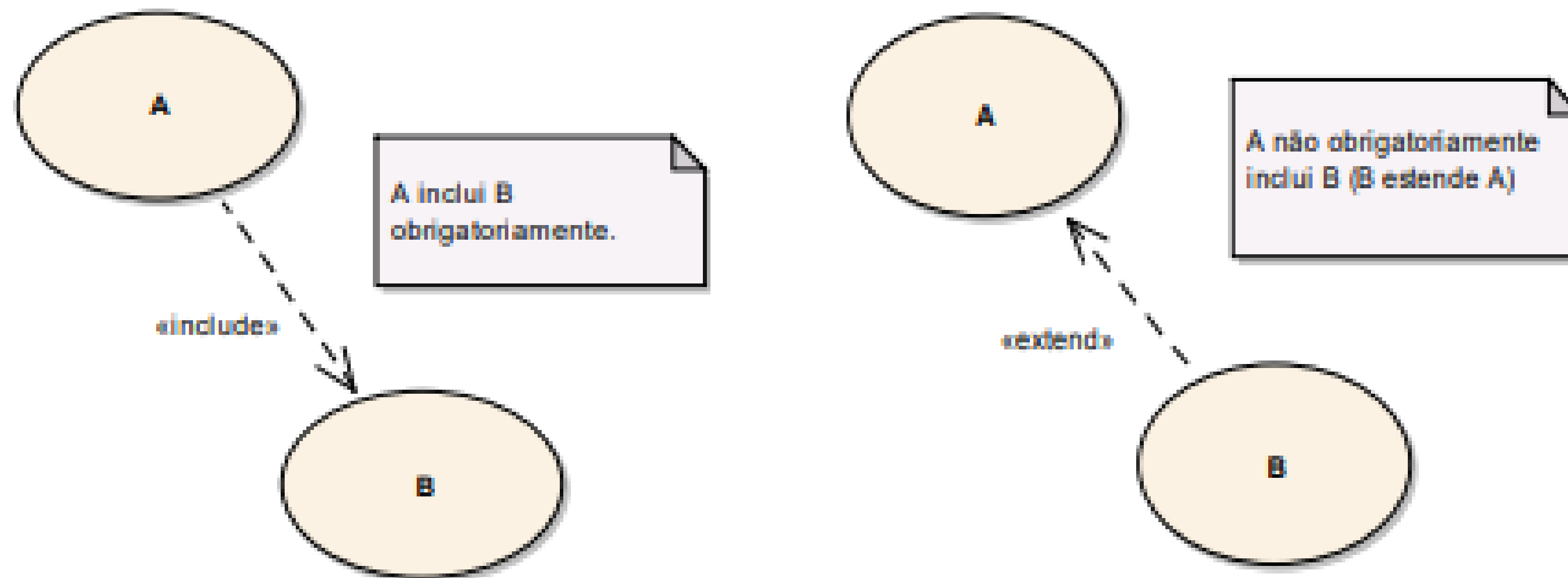
Fechando a ideia sobre inclusões e extensões, os relacionamentos são sempre lidos no sentido da seta (que é tracejada, segundo a notação). Inclusão significa que o caso de uso apontado pela seta é incluído no caso de uso que o aponta. Extensão significa que o caso de uso que aponta estende (agrega mais passos a) o caso de uso apontado. Adicionalmente, a extensão ocorre opcionalmente e a inclusão ocorre obrigatoriamente, segundo as regras do negócio.

Itens Anotacionais

Na figura ao lado há um novo elemento da notação da UML: o "cartãozinho" com a ponta dobrada, que é usado para colocar comentários, qualquer texto que elucide algum aspecto importante do modelo ou um requisito não funcional, como, por exemplo, uma restrição.

Esse é o chamado item anotacional da UML, que é associado ao diagrama ou opcionalmente a um elemento comentado por meio do segmento de reta tracejado.

Qualquer elemento do modelo pode ser associado a um comentário: um ator, um caso de uso, uma associação, uma classe, um objeto, uma atividade, ou seja, itens anotacionais podem aparecer em qualquer diagrama da UML.



Descrições de Casos de Uso

Os diagramas de casos de uso não capturam todas as características das funções que o sistema precisará executar; por exemplo, por não contemplar a dimensão temporal, não há como especificar sequências de execução das ações dos usuários ou do sistema. Pelos diagramas também não podemos inferir como atores e sistema interagem para a realização das funções do sistema; sequer podemos inferir se todos os atores relacionados a um caso de uso no diagrama precisam atuar colaborativamente ou se cada um pode, isolada e independentemente, executar todo o processo. Esses detalhes só são capturados quando descrevemos os casos de uso.

As descrições dos casos de uso, também chamadas de especificações, não são padronizadas pela UML. Dessa forma, as organizações definem seus próprios padrões usualmente baseados em padrões disponíveis na Internet ou em livros.

Outro aspecto importante a ser observado é a dependência da especificação produzida com respeito à ferramenta que a mantém (que a visualiza, altera, elimina e imprime a descrição), seja ela gratuita ou paga. Documentos em formato proprietário estabelecem um vínculo de longo prazo com a ferramenta e com a empresa que a produziu, além de estarem sujeitos a corrompimento. O ideal é armazenar a documentação em formato textual aberto (por exemplo, DocBook), visualizável por qualquer editor de textos e intercambiável entre ferramentas de fabricantes e de versões diferentes.

Descrições abreviadas vs detalhadas

As descrições podem ser feitas de forma abreviada, em alto nível de abstração, ou detalhada. Quando estamos no início do processo de levantamento, é normal optar por fazer uma aproximação "em largura", abordando o quanto antes o maior número de casos de uso do sistema, em detrimento dos detalhes de cada caso de uso. Nessa situação, elaboramos as descrições abreviadas. Nas demais etapas do levantamento fazemos o refinamento dos casos de uso, quando elaboramos as descrições detalhadas. Se determinada função do sistema estiver associada a algum tipo de risco maior, é recomendado detalhar o quanto antes a descrição dessa função.

Independentemente do padrão adotado, a descrição em alto nível deve conter o nome do caso de uso, a relação dos atores e, em poucas linhas, a descrição do processo adotado.

É usual que descrições em alto nível contenham, também:

- *Pré-condições*, ou seja, tudo que precisa ser verdade para que o caso de uso inicie. Um exemplo de pré-condição é quando, para a execução de um caso de uso, é necessário que seus usuários estejam autenticados no sistema. O mais conciso, nesse caso, é colocar como pré-condição a informação "Usuário autenticado no sistema", ao invés de apontar com o relacionamento de inclusão o caso de uso Autenticar Usuário de todos casos de uso no diagrama que demanda autenticação;
- *Pós-condições*, ou seja, o que se torna verdade quando o caso de uso termina. Como isso usualmente depende de como o caso de uso termina, eu evito usar pós-condições ou as restrinjo ao final que tipicamente acontece. Acho importante deixar isso bem claro ao usar-se pós-condições;
- *Garantias*. Essas, diferentemente das pós-condições, são o que se torna verdade quando o caso de uso termina, independentemente da forma como ele termina.

Descrições abreviadas vs detalhadas

As descrições abreviadas são aproveitadas nas descrições detalhadas, pois estas refinam aquelas. A descrição detalhada deve conter, além do que já foi relacionado, as seguintes informações:

- Os passos que compõem o curso típico;
- Os passos que compõem os cursos alternativos;
- A relação de regras de negócio que devem ser observadas. Há situações em que determinadas informações fornecidas pelos usuários ou produzidas pelo sistema precisam estar de acordo com uma ou mais regras do negócio. Nesses casos, o que se costuma fazer para que as descrições fiquem concisas é referenciar a regra ou regras de negócio onde elas precisam ser observadas.

Regras de negócio (RNs) são usualmente numeradas para que possam ser referenciadas mais facilmente nas descrições dos casos de uso. Usualmente, as regras compõem uma tabela de regras ou um capítulo à parte na documentação. Assim sendo, a tabela de RNs precisa ser colocada em um ponto específico da documentação; no capítulo (ou item) Regras de Negócio, por exemplo.

Descrições abreviadas vs detalhadas

Para ilustrar como referências a regras de negócio podem simplificar a descrição de casos de uso, imagine, por exemplo, uma situação em que os dados fornecidos por um usuário do sistema dizem respeito ao dependente de um funcionário que, por regra do negócio, não pode ter mais do que vinte e quatro anos. Poderíamos ter algo do tipo (mostrando apenas um trecho da descrição):

- ...Sistema exibe formulário para fornecimento dos dados do dependente do funcionário;
- Usuário preenche os campos com os dados do dependente;
- Sistema verifica idade do dependente com respeito à RN02...

<i>Tabela de Regras de Negócio</i>	
<i>Regra de Negócio</i>	<i>Descrição</i>
RN01	Todo dependente estará associado a um único funcionário.
RN02	Dependentes de funcionários devem ter idade inferior a 24 anos.
...	...

Fluxos principais e alternativos

Cada caso de uso tem o seu curso típico ou fluxo principal, em que relacionamos os passos da interação usuário(s)-sistema que compõem a situação que típica ou idealmente acontece. É comum relacionarmos o curso típico à "situação dos sonhos". Por exemplo, em um sistema de registro de compras de um supermercado, a situação típica é o código de barras poder ser lido pelo leitor ótico, ou seja, muito eventualmente o/a caixa precisa entrar com o código pelo teclado.

Quando não conhecemos o que tipicamente acontece, por estarmos lidando com uma situação nova, podemos considerar como fluxo principal os passos que comporão a situação que deverá, idealmente, acontecer.

Um caso de uso também pode possuir fluxos alternativos, em que descrevemos as possíveis variações – em geral muitas – de execução do caso de uso. Situações alternativas são comumente – e erradamente – associadas a situações ruins. Na realidade, nem todas as situações alternativas são situações ruins, pois também podem estar associadas a escolhas feitas pelos usuários.

A descrição do fluxo principal e dos cursos alternativos é feita de forma não procedimental, em linguagem coloquial e usando o jargão do negócio, pois a especificação é normalmente usada para validar os casos de uso junto aos especialistas do negócio e demais envolvidos no processo.

Fluxos principais e alternativos

ATENÇÃO: Na descrição de um caso de uso se deve informar o que é feito pelo sistema em cada ação, sem haver a preocupação de relatar como a ação é realizada. Por não estarmos interessados, no momento da descrevermos o casos de uso, em como as ações são realizadas, referimo-nos à descrição como sendo não procedimental.

Considerando a necessidade de se ter, numa especificação de caso de uso, as suas informações básicas (nome, pequena descrição, relação de atores, pré e pós-condições, dentre outras), e sendo o conjunto de passos para a realização de um processo dividido em duas partes (o fluxo principal e o conjunto de fluxos alternativos), os formulários usados nas organizações para descrição dos casos de uso são normalmente organizados conforme a figura ao lado. O número/nome do caso de uso, a relação de atores, a descrição abreviada e, possivelmente, as pré e pós-condições ficam no cabeçalho.

UC01 - Receber Notificação de Defeito dos Painéis Voltaicos

Objetivo: Notificar o morador administrador quando uma peça de um painel voltaico apresentar defeito

Funcionalidades relacionadas: RF45

Atores: Morador administrador

Prioridade: Alta

Precondições:

Frequência de uso: Baixa

Criticalidade: Média

Gatilho: Defeito em peça em algum dos painéis voltaicos

Fluxo principal:

- 1 - Sistema notifica o Morador Administrador de que uma das peças dos painéis voltaicos apresenta defeito.
- 2 - Morador Administrador seleciona a opção "Ver malha de painéis".
- 3 - Sistema apresenta o status de funcionamento do painel, indicando em qual dos painéis está o defeito e quão crítico é este defeito, ou seja, o quanto afeta a produção de energia. Caso seja crítico, o sistema alertará o Morador Administrador.
- 4 - Morador Administrador seleciona a opção "Ver peça defeituosa". (A1)
- 5 - Sistema apresenta a descrição da peça defeituosa e contatos de manutenção e/ou fornecedores.
- 6 - O Sistema apresenta a opção "Ligar Agora".
- 7 - Morador Administrador seleciona a opção "Ligar Agora" e escolhe o contato que deseja ligar.
- 8 - Fim do caso de uso.

Fluxo alternativo:

(A1) Alternativa ao Passo 4 - Usuário não seleciona a opção "Ver peça defeituosa".

1.a Usuário seleciona a opção "Ok"

1.b Fim do caso de uso.

Extensões:

Pós-condições:

Regras de negócio:

Fluxos principais e alternativos

As práticas comumente adotadas para a descrição dos casos de uso recomendam:

- Tratar as alternativas mais frequentes primeiro e as menos frequentes em fluxos alternativos subsequentes, ou seja, sempre procurando descrever antes o comportamento mais típico (daí o nome de curso típico para o primeiro bloco de passos) e depois a(s) variação(ões) mais típica(s) desse comportamento. Essa regra deve ser aplicada recursivamente, ou seja, se existe um comportamento mais frequente dentro de um comportamento alternativo, tratamos o mais frequente primeiro.
- Evitar o uso de expressões do tipo "Se... então...", ou seja, devemos informar que ocorre a situação mais típica e, nos fluxos alternativos, tratamos a(s) outra(s) possibilidade(s). Por exemplo, se um participante do processo pode responder "sim" ou "não" a uma pergunta do sistema e se, por exemplo, ele responde "sim" mais frequentemente a tal pergunta, assumir inicialmente que ele responderá "sim" e, em um fluxo alternativo seguinte, assumir que ele responderá "não".
- Usar, na especificação de repetições (loops) em um caso de uso, blocos "Para cada..." ou "Para todos...".
- Numerar os passos do fluxo típico e de cada curso alternativo, iniciando de 1 (um) para que possam ser facilmente referenciados em outras partes da especificação. É mais usual numerar os passos dos fluxos alternativos sempre iniciando de 1 (um). Outra maneira é prefixando a numeração com o número do curso alternativo (exemplos: 1.1 e 1.1.1, para passo 1 do fluxo alternativo 1 e passo 1 do fluxo alternativo 1.1, respectivamente).
- Referenciar um outro caso de uso quando ocorre uma inclusão ou uma extensão, explicitando a "chamada" no ponto da descrição do caso de uso que chama. Normalmente se usa a expressão "Executar caso de uso tal" no passo onde essa chamada deve ocorrer. No caso de uma inclusão, o caso de uso que chama é o de onde parte o relacionamento de inclusão; no caso de uma extensão, a chamada é feita no caso de uso aonde chega o relacionamento de extensão. Uma regra básica é a de que inclusões são especificadas como chamadas nas descrições do curso típico e extensões como chamadas em um dos cursos alternativos.

Fluxos principais e alternativos

Existem também os fluxos de exceção. Esses fluxos são fluxos alternativos normalmente associados a possíveis falhas no sistema. Entendemos que esses fluxos só precisam ser mencionados se suas ocorrências acarretarem impacto apreciável, seja no negócio sendo automatizado, seja no restante da execução do sistema. Por exemplo, se um erro do sistema demandar a execução de um outro caso de uso ou uma forma diferente de interação com o usuário, esse erro precisará ser tratado como um curso de exceção. Costumo exemplificar esse impacto citando o exemplo do fim da fita de impressão de um sistema de caixa de supermercado, quando ela ocorre no meio de uma lista de compras, o que precisa ser feito? Reiniciar a impressão do primeiro item quando a fita for trocada? Imprimir uma observação na nova fita ou simplesmente continuar a imprimir como se nada tivesse acontecido? Quando esse evento ocorrer, o supervisor de vendas precisará executar alguma outra funcionalidade do sistema?

Curiosidade: Em determinadas situações não é tão trivial definir se um curso é alternativo ou de exceção. Nesses casos acho que não precisamos perder muito tempo tentando descobrir se um curso é alternativo ou é de exceção. Existe, no entanto, uma técnica que pode ser útil para você estabelecer essa diferença, se você está muito preocupado com ela: cursos alternativos normalmente são trilhados por opções que os atores fazem durante a interação com o sistema; cursos de exceção são as "opções" que o sistema faz.

Como já dissemos, a UML não define um padrão de descrição de casos de uso e a forma de especificar casos de uso que foi apresentada acima se alinha com o que se observa na prática e com o que se encontra na bibliografia que trata desse assunto. Nada impede, no entanto, que definamos um novo padrão dentro da organização ou para um sistema em particular.

Erros frequentes

Associação entre atores:

A comunicação entre dois ou mais atores só é de interesse para o negócio quando é necessária para a realização colaborativa de alguma funcionalidade do sistema. É um tanto frequente encontrarmos diagramas contendo associações diretas entre dois atores em um diagrama numa tentativa de se especificar um caminho (que eventualmente existe) de comunicação entre os atores. Se indivíduos trocam informação necessária para a realização colaborativa de algum caso de uso, esse processo deve ser modelado como um caso de uso, e a comunicação entre os atores deve ser feita não diretamente, mas sim por meio desse caso de uso. Não há, portanto, sentido em haver associação entre dois atores sem que haja um processo representado por um caso de uso entre eles.

Casos de uso muito pequenos ou muito grandes:

Outra prática incorreta é identificar casos de uso que representem passos individuais, operações ou ações dentro de um processo. Casos de uso representam funções do sistema e, tipicamente, envolvem um número significativo de passos. Além dos casos de uso que representam funcionalidades disponíveis para os usuários do sistema, novos casos de uso podem ser criados nas seguintes situações:

- Quando existe um comportamento comum a dois ou mais casos de uso, contanto que o número de passos comuns justifique essa fatoração. O novo caso de uso passa a constar do diagrama, sendo incluído pelos casos de uso originais;
- Extraíndo comportamentos complexos (com números significativos de passos na especificação), obrigatórios ou variantes de dentro de casos de uso. Assim, os novos casos de uso aparecerão no diagrama, sendo incluídos pelos casos de uso originais ou os estendendo.

Erros frequentes

Casos de uso muito pequenos ou muito grandes (cont.):

A segunda situação está relacionada a casos de uso muito grandes, correspondendo a descrições muito longas. Devemos lembrar que as descrições servem, em um primeiro momento, para validar com os usuários o nosso entendimento a respeito de suas necessidades. Se a descrição é longa demais, o usuário "dorme" ao lê-la, fica com preguiça de ler, valida de qualquer maneira... E isso não é bom!

Em decorrência disso, procure evitar o uso de casos de uso do tipo "manter", que tratam da inclusão, exclusão, alteração e pesquisa de informações em um cadastro. Esses casos de uso são conhecidos como casos de uso "CRUD", de Create, Read, Update e Delete. Se tratarmos essas quatro, diríamos, subfuncionalidades em um único caso de uso, provavelmente produziremos uma descrição muito grande, principalmente porque as regras de negócio e de integridade aplicáveis são normalmente muito diferentes para cada uma delas. Por exemplo:

- Só se pode incluir um item em um cadastro se ele ainda não consta do cadastro;
- Só se pode excluir um item do cadastro desde que ele exista e não seja referenciado por um item de outro cadastro;
- Só se pode detalhar um item em uma consulta se ele consta do cadastro;
- Só se pode atualizar um item se ele consta do cadastro e não estiver sendo referenciado por um item de outro cadastro.

Sempre considere, portanto, a possibilidade de dividir esses casos de uso em quatro, usando casos de uso "manter" unicamente em situações de cadastros bem simples. Em caso de dúvida, a melhor métrica é o tamanho da descrição: a meu ver, mais do que cinco páginas de descrição já é muito.

Erros frequentes

Atores “voadores” e atores “gordinhos”:

Atores não "voam" no modelo. Cada ator está associado a, pelo menos, um caso de uso do modelo. Se isso não ocorre para algum ator, remova-o do modelo. Um determinado ator tipicamente não interage com o sistema de muitas e distintas formas. Se a descrição do papel que interpreta é longa e compreende atividades independentes entre si, quase certamente há mais de um papel sendo descrito. Nesse caso, devemos separar os atores, um para cada papel.

Complexidade visual de modelos:

Modelos visualmente complexos dificultam seu entendimento. O mesmo acontece com modelos contendo elementos e textos diminutos. Quando o número de elementos do modelo é tal que somos obrigados a diminuir a escala para que ele caiba em uma página da documentação impressa, é hora de pensarmos em dividir o modelo em partes. Existe uma regra empírica que estabelece que o número ideal de elementos de um modelo em uma mesma página deve variar de 5 a 9 ("regra do 7 +/- 2" – sete mais ou menos dois) para uma boa compreensão. Na prática, observamos que, contrariando a regra, modelos de casos de uso com até quinze casos de uso ainda são bem compreensíveis se não há muitos cruzamentos de associações. No caso de isso não ser possível em função da complexidade do negócio, somos obrigados a usar pacotes(agrupamentos) de atores e de casos de uso.

Fim do caso de uso:

O indicativo de "fim de caso de uso" devem significar o encerramento do processo e o término do que virá a ser o programa que implementará o caso de uso, e não o fim da descrição de um fluxo principal ou alternativo.

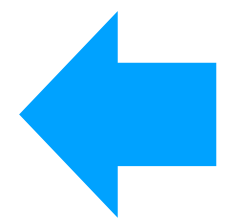
Erros frequentes

Consistência diagrama-descrições:

Embora diagramas de casos de uso e as descrições dos casos de uso sejam partes distintas do modelo, eles possuem uma coerência mútua que deve ser observada e preservada. Essa coerência consiste de diversos aspectos. Dentre eles:

- Todos os casos de uso desenhados nos diagramas precisam ter suas descrições feitas. Se existe uma descrição sem caso de uso ou vice-versa, é sinal de que algo está errado, no diagrama ou nas descrições;
- Casos de uso que incluem ou são estendidos não possuem referência das respectivas inclusões nas especificações (como "Executar caso de uso tal"). Se uma inclusão ou extensão não aparece em algum ponto da descrição do caso de uso que inclui, seja no fluxo principal, seja em um alternativo, algo está errado no diagrama ou na descrição;
- Atores nos diagramas não podem "desaparecer" ou mudar de nome nas relações de atores nas especificações, ou seja, atores associados a casos de uso devem fazer parte da lista de atores dos respectivos casos de uso;
- Atores associados a um caso de uso em um diagrama (e, portanto, também na relação de atores da descrição desse diagrama) devem ser mencionados de alguma forma, em algum ponto da descrição.

Esses e outros lembretes, dicas e orientações sobre casos de uso podem ser encontrados do artigo *How to avoid use-case pitfalls*, de Suzan Lilly ([9]), disponível gratuitamente na Internet (em <http://www.drdobbs.com/184414560>, por exemplo).



Exceções e Tratamento de Erros

Motivação

Vamos pensar em uma potencial classe **Conta**, com um método **sacar**. O que aconteceria ao chamar esse método com um valor fora do limite? O sistema poderia mostrar uma mensagem de erro, mas quem chamou o método **sacar** não saberá que isso aconteceu.

Em Java, os métodos dizem qual o **contrato** que eles devem seguir. Se, ao tentar sacar, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário que o saque não foi feito.

Veja no exemplo abaixo: estamos forçando uma **Conta** a ter um valor negativo, isto é, estar num estado inconsistente de acordo com a nossa modelagem.

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
minhaConta.saca(1000);
//      o saldo é -900? É 100? É 0? A chamada ao método saca funcionou?
```

Em sistemas reais, é muito comum que quem saiba tratar o erro é aquele que chamou o método, e não a própria classe. Portanto, nada mais natural do que a classe sinalizar que um erro ocorreu.

A solução mais simples utilizada antigamente é a de marcar o retorno de um método como **boolean** e retornar **true**, se tudo ocorreu da maneira planejada, ou **false**, caso contrário.

Motivação

```
Conta minhaConta = new Conta();  
minhaConta.deposita(100);  
minhaConta.setLimite(100);  
if (!minhaConta.saca(1000)) {  
    System.out.println("Não saquei");  
}
```

Repare que tivemos de lembrar de testar o retorno do método, mas não somos obrigados a fazer isso. Esquecer de testar o retorno desse método teria consequências drásticas: a máquina de autoatendimento que está rodando esse programa poderia vir a liberar a quantia desejada de dinheiro, mesmo que o sistema não tivesse conseguido efetuar o método **sacar** com sucesso.

Mesmo invocando o método e tratando o retorno de maneira correta, o que faríamos se fosse necessário sinalizar quando o usuário passou um valor negativo como **quantidade**? Uma solução seria alterar o retorno de **boolean** para **int** e retornar o código do erro que ocorreu. Isso é considerado uma má prática, no geral.

Além de você perder o retorno do método, o valor devolvido é “mágico” e só legível perante extensa documentação, além de não obrigar o programador a tratar esse retorno e, no caso de esquecer isso, seu programa continuará rodando já num estado inconsistente. Para completar, você “rouba” um possível retorno útil do seu método.

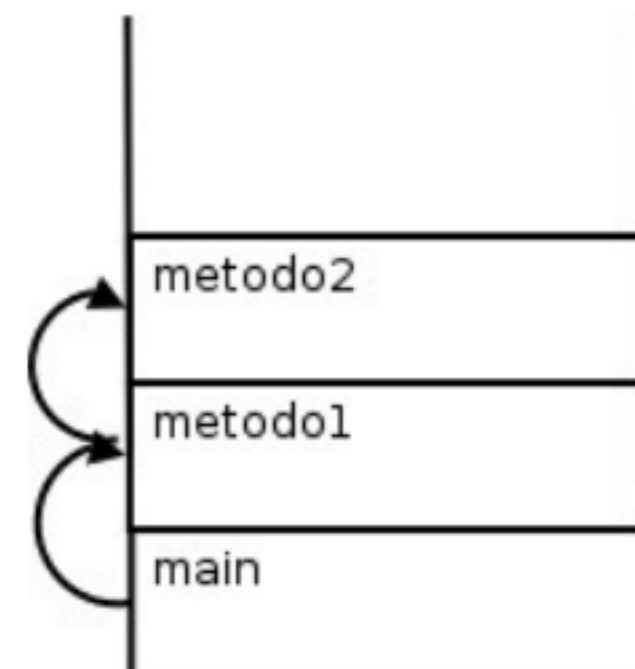
Conceitos básicos

Por esses e outros motivos, utilizamos um código diferente em Java para tratar aquilo que chamamos de exceções: os casos onde acontece algo que, normalmente, não iria acontecer. O exemplo do argumento do saque inválido ou do **id** inválido de um cliente é uma *exceção à regra*.

Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema.

Repare o método **main** chamando **metodo1** e esse, por sua vez, chamando o **metodo2**. Cada um desses métodos pode ter suas próprias variáveis locais, isto é, o **metodo1** não enxerga as variáveis declaradas dentro do **main** e aí por diante.

Como o Java (e muitas outras linguagens) faz isso? Toda invocação é empilhada em uma estrutura de dados que isola a área de memória de cada um. Quando um método termina (retorna), ele volta para o método que o invocou. Ele descobre isso através da **pilha de execução** (stack): basta remover o marcador que está no topo da pilha.



```
class TesteErro {  
    public static void main(String[] args) {  
        System.out.println("inicio do main");  
        metodo1();  
        System.out.println("fim do main");  
    }  
  
    static void metodo1() {  
        System.out.println("inicio do metodo1");  
        metodo2();  
        System.out.println("fim do metodo1");  
    }  
  
    static void metodo2() {  
        System.out.println("inicio do metodo2");  
        ContaCorrente cc = new ContaCorrente();  
        for (int i = 0; i <= 15; i++) {  
            cc.deposita(i + 1000);  
            System.out.println(cc.getSaldo());  
            if (i == 5) {  
                cc = null;  
            }  
        }  
        System.out.println("fim do metodo2");  
    }  
}
```

Conceitos básicos

Porém, o nosso **metodo2** propositadamente possui um enorme problema: está acessando uma referência nula quando o índice for igual a 6!

```
início do main
início do metodo1
início do metodo2
1000.0
2001.0
3003.0
4006.0
5010.0
6015.0
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null
    at TesteErro.metodo2(exemplo.java:18)
    at TesteErro.metodo1(exemplo.java:10)
    at TesteErro.main(exemplo.java:4)
```

Essa saída é o conhecido *rastro da pilha* (stacktrace). É uma saída importantíssima para o programador - tanto que, em qualquer fórum ou lista de discussão, é comum os programadores enviarem, juntamente com a descrição do problema, essa stacktrace. Mas por que isso aconteceu?

O sistema de exceções do Java funciona da seguinte maneira: quando uma exceção é lançada (*throw*), a JVM entra em estado de alerta e vai ver se o método atual toma alguma precaução ao **tentar** executar esse trecho de código. Como podemos ver, o **metodo2** não toma nenhuma medida diferente do que vimos até agora.

Conceitos básicos

Como o **metodo2** não está *tratando* o problema, a JVM para a execução dele anormalmente, sem esperar ele terminar, e volta um *stackframe* para baixo, onde será feita nova verificação: “o **metodo1** está se precavendo de um problema chamado **NullPointerException**?” “Não...” Volta para o **main**, onde também não há proteção, então a JVM encerra o programa.

Obviamente, aqui estamos forçando esse caso e não faria sentido tomarmos cuidado com ele. É fácil arrumar um problema desses: basta verificar antes de chamar os métodos se a variável está com referência nula.

Porém, apenas para entender o controle de fluxo de uma **Exception**, vamos colocar o código que vai *tentar* (**try**) executar o bloco perigoso e, caso o problema seja do tipo **NullPointerException**, ele será *pego* (**caught**).

```
try {
    for (int i = 0; i <= 15; i++) {
        cc.deposita(i + 1000);
        System.out.println(cc.getSaldo());
        if (i == 5) {
            cc = null;
        }
    }
} catch (NullPointerException e) {
    System.out.println("erro: " + e);
}
```

```
inicio do main
inicio do metodo1
inicio do metodo2
1000.0
2001.0
3003.0
4006.0
5010.0
6015.0
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null
fim do metodo2
fim do metodo1
fim do main
```


Conceitos básicos

Repare que, dependendo de onde a estrutura **try/catch** é inserida, o programa se comporta de forma diferente, mas ele sempre será concluído (a não ser que outra exceção seja pega).

```
for (int i = 0; i <= 15; i++) {  
    try {  
        cc.deposita(i + 1000);  
        System.out.println(cc.getSaldo());  
        if (i == 5) {  
            cc = null;  
        }  
    } catch (NullPointerException e) {  
        System.out.println("erro: " + e);  
    }  
}
```

```
inicio do main  
inicio do metodo1  
inicio do metodo2  
1000.0  
2001.0  
3003.0  
4006.0  
5010.0  
6015.0  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
fim do metodo2  
fim do metodo1  
fim do main
```

```
static void metodo1() {  
    System.out.println("inicio do metodo1");  
    try {  
        metodo2();  
    } catch (NullPointerException e) {  
        System.out.println("erro: " + e);  
    }  
  
    System.out.println("fim do metodo1");  
}
```

```
inicio do main  
inicio do metodo1  
inicio do metodo2  
1000.0  
2001.0  
3003.0  
4006.0  
5010.0  
6015.0  
erro: java.lang.NullPointerException: Cannot invoke "ContaCorrente.deposita(int)" because "<local0>" is null  
fim do metodo1  
fim do main
```

Checked exceptions

Um outro tipo de exceção em Java obriga a quem chama o método ou construtor a tratar essa exceção. Chamamos esse tipo de exceção de *checked*, pois o compilador checará se ela está sendo devidamente tratada, diferente das anteriores, conhecidas como *unchecked*.

```
class Teste {  
    public static void metodo() {  
        new java.io.FileInputStream("arquivo.txt");  
    }  
}
```

O código acima não compila, e o compilador avisa que é necessário tratar o **FileNotFoundException** que pode ocorrer.

Para compilar e fazer o programa funcionar, temos duas maneiras que podemos tratar o problema. O primeiro é trata-lo com o **try** e **catch** do mesmo jeito que usamos no exemplo anterior. A segunda forma de tratar esse erro é delegar ele para quem chamou o nosso método, isto é, passar para frente.

```
public static void metodo() throws java.io.FileNotFoundException {  
  
    new java.io.FileInputStream("arquivo.txt");  
  
}  
  
public static void metodo() {  
  
    try {  
        new java.io.FileInputStream("arquivo.txt");  
    } catch (java.io.FileNotFoundException e) {  
        System.out.println("Nao foi possível abrir o arquivo para leitura");  
    }  
  
}
```

Checked exceptions

No início, existe uma grande tentação de sempre passar o problema para frente para outros o tratarem. Pode ser que faça sentido, dependendo do caso, mas não até o main, por exemplo. Acontece que quem tenta abrir um arquivo sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa pode não saber ou, pior ainda, tentar abrir cinco arquivos diferentes e não saber qual deles teve um problema!

Não há uma regra para decidir em que momento do seu programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, você provavelmente vai preferir delegar a responsabilidade para o método que te invocou.

Checked exceptions

No início, existe uma grande tentação de sempre passar o problema para frente para outros o tratarem. Pode ser que faça sentido, dependendo do caso, mas não até o main, por exemplo. Acontece que quem tenta abrir um arquivo sabe como lidar com um problema na leitura. Quem chamou um método no começo do programa pode não saber ou, pior ainda, tentar abrir cinco arquivos diferentes e não saber qual deles teve um problema!

Não há uma regra para decidir em que momento do seu programa você vai tratar determinada exceção. Isso vai depender de em que ponto você tem condições de tomar uma decisão em relação àquele erro. Enquanto não for o momento, você provavelmente vai preferir delegar a responsabilidade para o método que te invocou.

Tratando mais de um erro

É possível tratar mais de um erro quase que ao mesmo tempo:

```
try {
    objeto.metodoQuePodeLancarIOeSQLException();
} catch (IOException e) {
    // ..
} catch (SQLException e) {
    // ..
}

public void abre(String arquivo) throws IOException, SQLException {
    // ..
}

public void abre(String arquivo) throws IOException {
    try {
        objeto.metodoQuePodeLancarIOeSQLException();
    } catch (SQLException e) {
        // ..
    }
}
```

Lançando exceções

Lembre-se do método **sacar** da nossa potencial classe **Conta**. Uma possível solução para o problema é:

```
public boolean sacar(double valor) {  
    if (this.saldo < valor) {  
        return false;  
    } else {  
        this.saldo -= valor;  
        return true;  
    }  
}
```

Podemos, também, lançar uma **Exception**, o que é extremamente útil. Dessa maneira, resolvemos o problema de alguém poder esquecer de fazer um **if** no retorno de um método.

A palavra-chave **throw**, que está no imperativo, lança uma **Exception**. Isto é bem diferente de **throws**, que está no presente do indicativo, e que apenas avisa da possibilidade daquele método lança-la, obrigando o outro método que vá utilizar deste de se preocupar com essa exceção em questão.

```
public void sacar(double valor) {  
    if (this.saldo < valor) {  
        throw new RuntimeException();  
    } else {  
        this.saldo -= valor;  
    }  
}
```

No nosso caso, lança uma do tipo *unchecked*. **RuntimeException** é a **Exception** mãe de todas as exceções *unchecked*. A desvantagem, aqui, é que ela é muito genérica; quem receber esse erro não sabe dizer exatamente qual foi o problema.

Lançando exceções

Podemos então usar uma **Exception** mais específica:

```
public void sacar(double valor) {  
    if (this.saldo < valor) {  
        throw new IllegalArgumentException();  
    } else {  
        this.saldo -= valor;  
    }  
}
```

Essa exceção diz um pouco mais: algo foi passado como argumento e seu método não gostou. Ela é uma exceção *unchecked* pois estende de **RuntimeException** e já faz parte da biblioteca do Java. Para pegar um erro desse tipo, ao invés de usarmos um **if/else**, usaremos um **try/catch**, porque faz mais sentido já que a falta de saldo é uma exceção (e aí podemos lembrar das exceções em um caso de uso!).

Podemos melhorar ainda mais e passar para o construtor da exceção o motivo dela ocorrer:

```
public void sacar(double valor) {  
    if (this.saldo < valor) {  
        throw new IllegalArgumentException("Saldo insuficiente");  
    } else {  
        this.saldo -= valor;  
    }  
}
```

Lançando exceções

O método **getMessage()** definido na classe **Throwable** (mãe de todos os tipos de erros e exceções) vai retornar a mensagem que passamos ao construtor da **IllegalArgumentException**.

```
try {  
    cc.sacar(100);  
} catch (IllegalArgumentException e) {  
    System.out.println(e.getMessage());  
}
```


Criando suas próprias exceções

É bem comum criar uma própria classe de exceção para controlar melhor o uso de suas exceções. Dessa maneira, podemos passar valores específicos para ela carregar, que sejam úteis de alguma forma. Vamos criar a nossa:

```
public class SaldoInsuficienteException extends RuntimeException {  
  
    public SaldoInsuficienteException(String message) {  
        super(message);  
    }  
}
```

Em vez de lançar um **IllegalArgumentException**, vamos lançar nossa própria exceção, com uma mensagem que dirá “Saldo Insuficiente”:

```
public void sacar(double valor) {  
    if (this.saldo < valor) {  
        throw new SaldoInsuficienteException("Saldo insuficiente, " +  
                                              "tente um valor menor");  
    } else {  
        this.saldo -= valor;  
    }  
}
```

Podemos transformar essa exceção de *unchecked* para *checked*, obrigando a quem chama esse método a agir (usar **try/catch** ou **throws**):

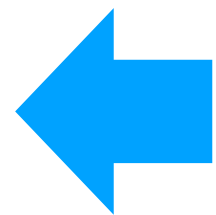
```
public class SaldoInsuficienteException extends Exception {  
  
    public SaldoInsuficienteException(String message) {  
        super(message);  
    }  
}
```

Expressão **finally**

Os blocos **try** e **catch** podem conter uma terceira cláusula chamada **finally** que indica o que deve ser feito após o término do bloco **try** ou de um **catch** qualquer.

É interessante colocar algo que é imprescindível de ser executado, caso o que você queira fazer tenha dado certo, ou não. O caso mais comum é o de liberar um recurso no **finally**, como um arquivo ou conexão com banco de dados, para que possamos ter a certeza de que aquele arquivo (ou conexão) vá ser fechado, mesmo que algo tenha falhado no decorrer do código.

```
try {  
    // bloco try  
} catch (IOException ex) {  
    // bloco catch 1  
} catch (SQLException sqlex) {  
    // bloco catch 2  
} finally {  
    // bloco que será sempre executado, independente  
    // se houve ou não exception e se ela foi tratada ou não  
}
```



Boas Práticas no Uso da OO

Introdução

Nas aulas anteriores, vimos por que usar a OO, seus conceitos e como utiliza-los. Entretanto, é preciso usar com cuidado todos eles. Devemos ter prudência ao aplica-los, pois se forem utilizados de forma impensada, manutenções futuras na aplicação podem se tornar onerosas, ou até mesmo inviáveis. Esse tipo de situação extrema é o que leva a insucessos de projetos orientados a objeto.

Para tentar evitar tais situações adversas, podemos usar formas avançadas de se trabalhar com OO, como por Padrões de Projeto. Porém, para uma utilização correta que traga os ganhos esperados, é necessária uma certa experiência em programação orientada a objetos. Só assim, estas e outras formas avançadas serão entendidas de forma eficaz e, conseqüentemente, serão aplicadas no momento certo e de forma correta.

Todavia, não podemos esperar até essa “experiência” chegar para assim tentar solucionar falhas anteriores, ou mesmo evita-las. Nesta aula, veremos um pequeno catálogo de boas práticas que ajudam a iniciar o uso da OO.

Com isso, espera-se prover um atalho a este longo caminho de se obter sucesso e assim encurtar o tempo para uma melhor aplicabilidade da OO. São boas práticas simples, mas que ajudam no amadurecimento e entendimento de como e quando usar os conceitos apresentados no curso.

Se preocupe com coesão e acoplamento

Esta é a mais básica, mas também a mais importante das boas práticas. No contexto de OO, criar classes não coesas e com acoplamento forte é um fator preponderante para o insucesso de projetos. Existe uma máxima que dita esta boa prática: **“trabalhe com alta coesão e baixo acoplamento”**.

Relembrando, **coesão** é a relação existente entre as responsabilidades de uma determinada classe e de cada um de seus métodos e atributos.

A falta de coesão leva a classes inchadas, que misturam responsabilidades.

Veja o caso de uma classe não coesa ao lado. Note que ela tem muitas características que não dizem respeito diretamente a ela mesma. O que comprova isso é o fato de os atributos possuírem nomes mais detalhados

para conseguirem representar suas utilidades e representatividades que terminam explicitando outros conceitos.

Uma classe mais coesa seria:

```
public class Venda {  
  
    private Cliente cliente;  
    private Endereco endereco;  
    private Debito pagamento;  
    private Produto[] produtos;  
    private Vendedor vendedor;  
}
```

```
public class Venda {  
  
    private String nomeCliente;  
    private String cpfCliente;  
    private String enderecoEntrega;  
    private int cep;  
    private Debito pagamento;  
    private Produto[] produtos;  
    private String nomeVendedor;  
    private double comissaoVendedor;  
}
```

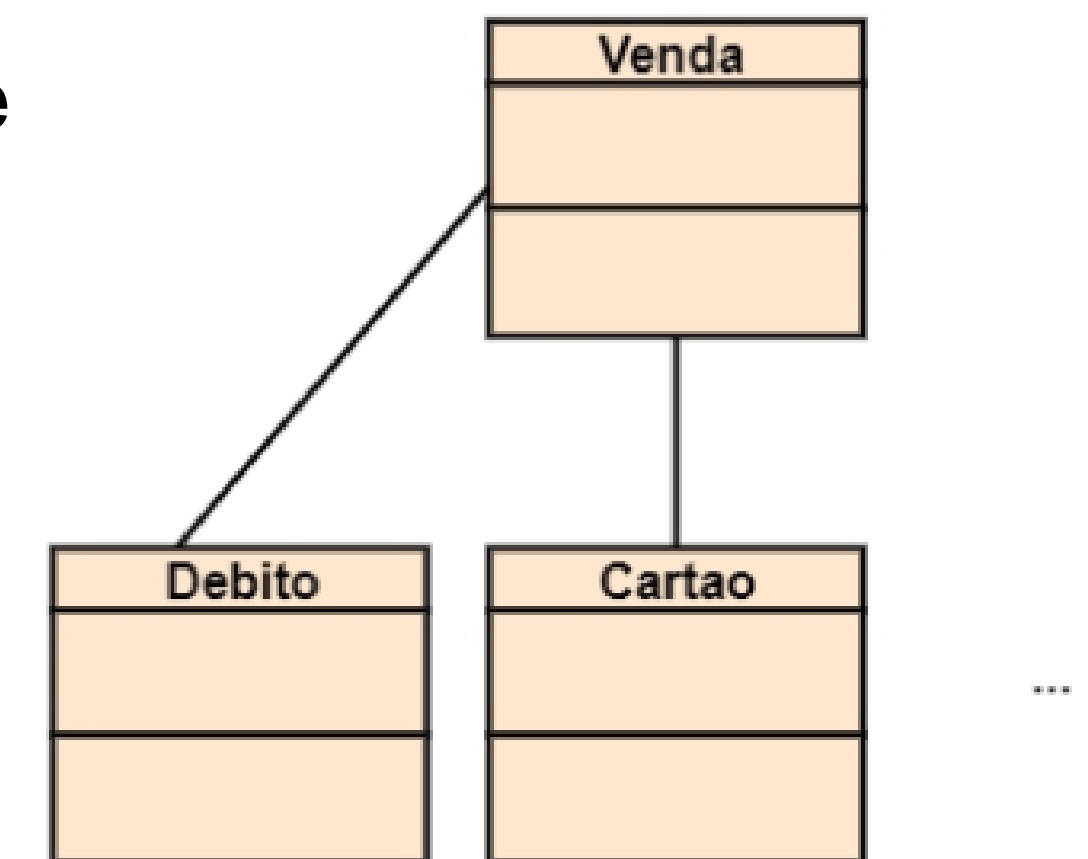
Se preocupe com coesão e acoplamento

Perceba que algumas características sumiram, pois novas classes foram criadas para armazenar tais informações, e assim retirá-las da classe **Venda**. Neste caso, as associações ajudaram a criar uma classe mais coesa.

Entretanto, ainda existe algo a ser melhorado neste exemplo: o acoplamento. **Acoplamento** é o grau de dependência entre dois artefatos, sejam eles entidades, métodos, componentes, etc. Note que existe um acoplamento muito alto, ou seja, **Venda** depende de outra classe de uma forma muito forte. Isto acontece com a classe **Debito**. Caso a venda precise ter outra forma de pagamento, uma grande alteração teria de ser realizada para poder aceitar essa nova modalidade (por exemplo, cartão).

Inicialmente, podemos até pensar em acrescentar um novo atributo, **private Cartao pagamentoCartao**, e resolver o problema com isso. Mas e se um financiamento passar a ser aceito? Um novo atributo? Logo percebemos que essa abordagem não resolve; na verdade, gera novos acoplamentos e cada vez mais vai ficando difícil solucioná-los.

Cada uma dessas formas de pagamento trabalha isoladamente, e depender diretamente dela termina culminando em uma forte dependência da classe **Venda** com esses pagamentos, pois alterações no pagamento acabam refletindo também na classe **Venda**.

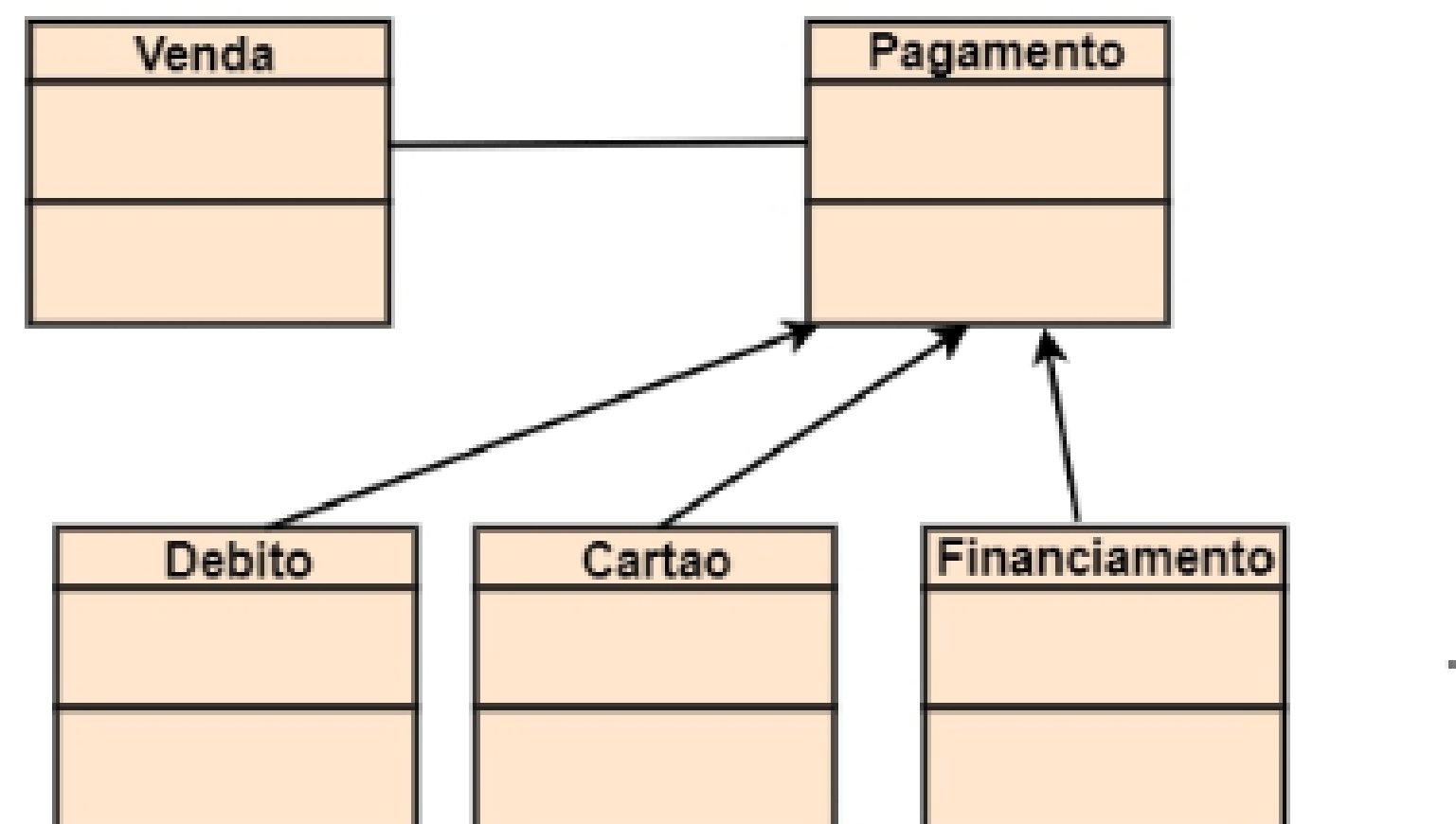


Se preocupe com coesão e acoplamento

Acoplamentos devem existir, pois uma das características básicas da Orientação a Objetos é a troca de mensagens, e isso só é possível pelo acoplamento. Então, como resolver a situação apresentada? Tornando os acoplamentos fracos, flexíveis.

Um bom acoplamento é aquele que possibilita manutenções sem grandes impactos, sem efeitos colaterais. Para esta situação, a melhor forma seria criar uma classe abstrata, ou interface, e os tipos de pagamento herdariam ou implementariam a classe ou interface. Com isso, a classe **Venda** não dependeria diretamente de **Debito**, **Cartao**, etc., mas sim de um pagamento genérico que poderia se moldar à necessidade corrente.

Desta forma, poderíamos mudar a forma de pagamento sem que grandes alterações na venda fossem necessárias.



Use strings com parcimônia

Infelizmente, é muito comum o uso indiscriminado de strings na definição de atributos. Isto pode ocorrer devido a string ser um tipo de dado muito comum no dia a dia, e que naturalmente termina tornando-a amplamente utilizada. Porém, cuidados devem ser tomados para evitar situações adversas.

À primeira vista, esta classe não possui problema algum. No entanto, existem erros graves na sua definição, que iniciantes comumente cometem: o uso excessivo do tipo texto, no caso string. Isso geralmente ocorre devido a esta suportar todo tipo de valor. Entretanto, essa “facilidade” termina gerando problemas futuros.

```
public class Cliente {  
    private String nome;  
    private String dataAniversario;  
    private String sexo;  
    private String endereco;  
}
```

Um exemplo seria **private String dataAniversario**. Podemos pensar: “precisa ser um texto, pois a data é no padrão dd/mm/aaaa, e só textos suportam isso”. Mas e se for preciso calcular a idade do cliente? Como um texto poderá ser utilizado para determinar tal valor? Certamente o resultado será obtido, mas o caminho traçado para encontra-lo será extremamente árduo.

Neste caso, podemos usar o tipo de dado **Date**. Embora este não guarde a data no formato **dd/mm/aaaa**, várias outras facilidades são obtidas, como adição e subtração de dias, meses ou até anos.

Use strings com parcimônia

Já no caso do atributo **private String sexo**, podemos pensar “agora sim é texto! Só pode ser Masculino ou Feminino. Não tem o que discutir”. Por ser um texto livre, o que impede de alguém colocar um valor diferente destes dois? Mais uma vez, é possível notar que o tipo string termina não sendo a melhor opção.

Neste caso, podemos utilizar um dado do tipo **enum**. Esse tipo é uma facilidade que algumas linguagens orientadas a objetos fornecem. Ele é usado para criar um conjunto fixo e limitado de valores. Caso tenhamos a situação citada, é melhor criar **enum** do que criar toda uma estrutura de programação para manter a “fixação” e “limitação” de valores.

```
public enum Sexo {  
    MASCULINO,  
    FEMININO;  
}
```

Para utilizar o **enum**, basta usar seu nome e um de seus valores disponíveis: **Sexo.MASCULINO** ou **Sexo.FEMININO**. Para mais informações, você pode consultar a [documentação do Java](#).

Assim como **dataAniversario** e **sexo**, podemos fazer o mesmo com o atributo **endereco**, e nossa classe ficaria assim:

```
public class Cliente {  
  
    private String nome;  
    private Date dataAniversario;  
    private Sexo sexo;  
    private Endereco endereco;  
}
```

Seja objetivo, não tente prever o futuro

É muito comum iniciantes na OO pensarem da seguinte forma: “vou criar uma classe bem genérica, para ser a superclasse de todas as outras. Ou, se não servir de classe mãe, poderá ser usada em qualquer situação. Assim, consigo um bom reaproveitamento de código”.

Embora possa parecer certa essa forma de raciocínio, na verdade ela não é. Isto ocorre devido a um princípio chamado **KISS**, que na verdade pode ser aplicado a qualquer área e significa **Keep It Simple, Stupid**.

Quando são criadas classes genéricas demais, torna-se muito difícil entendê-las. Elas podem ficar sem sentido algum, mas, mesmo assim, estarão presentes em todo lugar. Além disso, um acoplamento muito alto será criado, pois todas as classes de sua aplicação dependerão dela, sendo subclasses ou se associando a ela. Se algum dia for necessário fazer uma modificação nessa classe, todo o sistema pode ser afetado.

Usar herança somente com o intuito de reuso é um equívoco. O reuso é uma boa consequência da sua utilização. Por não ser a real aplicabilidade da herança, se pensarmos somente em reuso ao utilizá-la, podemos gerar acoplamentos muito fortes sem a mínima necessidade. Isto ocorre devido às subclasses só existirem e funcionarem corretamente a partir da sua superclasse.

Mudanças na superclasse inevitavelmente afetam a subclasse. Talvez, a classe em questão - a subclasse - pudesse “funcionar sozinha”, sem a necessidade de uma classe mãe. A grande vantagem do uso de herança é a criação de subtipos, que são conceitos reais do dia a dia. O uso de herança é bom e deve ser encorajado, mas no momento certo.

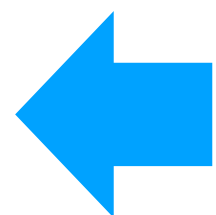
Seja objetivo, não tente prever o futuro

Por exemplo, se o sistema só tratar de vendas para pessoas físicas, então não precisamos criar uma classe chamada **Pessoa** e depois uma chamada **PessoaFisica**, que herda de **Pessoa**. Não serão realizadas vendas para pessoas jurídicas, então porque se preocupar em reuso, se ele na verdade não será necessário?

Caso um dia a situação mude e a venda para pessoa jurídica torne-se realidade, então uma evolução deve ser feita no sistema para tratar essa situação. Neste caso, a classe **Pessoa** começa a mostrar seu valor.

Da mesma forma, usar associação apenas para reuso também pode ser um equívoco. É preciso existir um relacionamento real entre os conceitos para que eles possam se associar. Em ambos os casos, deve existir uma ligação conceitual entre os conceitos para que eles se relacionem, seja por herança ou associação.

Uma modelagem eficiente é aquela que supre as necessidades do momento, mas que pode ser evoluída facilmente. Modelagens extremamente “flexíveis” terminam por tornar o modelo complexo, com acoplamento forte, difícil de entender e manter.



Princípios SOLID

Introdução

É comum muitos iniciantes na área de desenvolvimento de software ou mesmo alguns profissionais com certa experiência não se preocuparem o suficiente com a qualidade do código que é produzido. Em muitos casos, a frase “tá rodando direitinho” é utilizada para subjugar as preocupações que se deveriam ter com a qualidade do código que é produzido.

Embora a OO tenha um conjunto de conceitos que visam facilitar a representação do mundo real e isso termine por possibilitar - de forma inicial - a criação de códigos com uma qualidade maior que linguagens do paradigma estruturado, esta “qualidade” não é seu principal motivo de existência. Devido a isso, muitas vezes a qualidade é deixada de lado e esse desleixo pode cobrar um preço muito caro ao longo do processo de desenvolvimento de uma aplicação.

Introdução

Em uma determinada aplicação existe um método responsável por calcular o preço total de uma venda.

```
public double calcularTotalVenda(Venda venda) {  
    double total = 0.0;  
  
    for (Produto produto : venda.getProdutos()) {  
        total += produto.getValor();  
    }  
  
    return total;  
}
```

Porém, um dia, chega uma manutenção evolutiva. Agora é necessário - além de se calcular o preço sem imposto - calcular o total de acordo com determinado imposto. Neste caso, ele deve ser aplicado sobre o valor do produto. É comum inicialmente se pensar da forma a seguir:

```
public double calcularTotalVenda(Venda venda, double aliquotaImposto) {  
    double total = 0.0;  
  
    for (Produto produto : venda.getProdutos()) {  
        if (aliquotaImposto != 0.0) {  
            total += produto.getValor() * (1 + aliquotaImposto);  
        } else {  
            total += produto.getValor();  
        }  
    }  
  
    return total;  
}
```

Introdução

Logicamente, esse novo código vai executar e retornar o valor desejado. Entretanto, nota-se que o método não mais simples como o inicial. O que antes era apenas um **for**, agora já é uma estrutura com um **if-else** interno. Além disto, agora já são 2 parâmetros e não somente 1.

Para piorar ainda mais a situação, pouco tempo depois chega uma nova demanda: agora a venda pode ou não ter um acréscimo de um seguro contra perdas ou danos durante o processo de envio. Esse valor poderá ou não ser adicionado ao valor final da venda e será de 10% sobre o preço total da venda. Tal seguro é independente da aplicação ou não dos impostos. Assim, a seguinte alteração poderia ser feita:

```
public double calcularTotalVenda(Venda venda, double aliquotaImposto, boolean usaSeguro) {
    double total = 0.0;

    for (Produto produto : venda.getProdutos()) {
        if (aliquotaImposto != 0.0) {
            total += produto.getValor() * (1 + aliquotaImposto);
        } else {
            total += produto.getValor();
        }
    }

    if (usaSeguro) {
        total = (total * 1.1);
    }

    return total;
}
```

Introdução

Essas “manutenções evolutivas” já são o suficiente para notarmos que alterar de forma indiscriminada um mesmo método nem sempre é a melhor prática, embora às vezes seja mais fácil e rápida. Os principais problemas dessa abordagem estão expostos a seguir:

- **Aumento da complexidade do método:** como o método foi absorvendo todas as alterações, ele foi inchando. Internamente, ele foi ficando cada vez mais carregado de estruturas de controle para satisfazer os fluxos desejados. Inevitavelmente, isso culmina em uma maior dificuldade de entendimento, devido à grande quantidade de fluxos que vão surgindo;
- **Quebra da interface:** toda vez que um novo parâmetro é acrescentado no método, todos os pontos dentro da aplicação onde ele é utilizado devem sofrer uma manutenção. Mesmo que neste ponto a passagem de tal parâmetro não seja necessária, a alteração deve ser feita. Isso ocorre devido à mudança da assinatura do método.

Introdução

Para evitar tais problemas, poderíamos simplesmente fazer sobrecargas do método **calcularTotalVenda** e mover a informação da aplicação do seguro para a entidade **Venda**. Assim, tais evoluções deveriam ter culminado nos seguintes códigos:

```
public double calcularTotalVenda(Venda venda) {
    double total = 0.0;

    for (Produto produto : venda.getProdutos()) {
        total += produto.getValor();
    }

    if (venda.possuiSeguro()) {
        total = (total * 1.1);
    }

    return total;
}

public double calcularTotalVenda(Venda venda, double aliquotaImposto) {
    double total = 0.0;

    for (Produto produto : venda.getProdutos()) {
        total += produto.getValor() * (1 + aliquotaImposto);
    }

    if (venda.possuiSeguro()) {
        total = (total * 1.1);
    }

    return total;
}
```

Introdução

Os motivos - e válidos, por sinal - para as alterações serem feitas desta forma são:

- **Quem já usava o método de venda sem imposto não foi afetado:** como novos métodos foram criados, quem usava a versão anterior do método (sem imposto) não foi afetado pelas novas demandas. Além disto, quem usa vendas com imposto ou sem imposto poderá trabalhar de forma isolada;
- **Quem tem um seguro é a venda em si e não o seu total:** o método é responsável por calcular o total final da venda. A informação da aplicação do seguro é uma característica da entidade **Venda**. Esta informação é independente do cálculo do imposto. Sendo assim, a representação mais alinhada com a realidade - e isto é um dos pilares da OO - é transformar o parâmetro **usaSeguro** do método em um atributo da entidade **Venda**.

Ao realizar esta pequena e simples alteração (a sobrecarga em vez da adição de parâmetros e fluxos num método já existente) já começamos a evitar situações que poderiam degradar a qualidade do código. Nota-se que o código em cada método ficou menos complexo e que dessa forma podemos evitar o acréscimo indiscriminado de **if's**. Embora tenhamos escrito mais métodos, eles ficaram mais simples de entender. Além disso, as lógicas de cada tipo de venda (com ou sem imposto) ficaram isoladas e não misturadas no mesmo método. Lógico que existem situações muito mais complexas do que a apresentada, mas o mais importante é perceber que, antes de realizar alterações no código, deve-se pensar bem em como fazê-las.

Para possibilitar o desenvolvimento de aplicações sem grandes problemas, deve-se seguir alguns princípios que visam tornar o código mais fácil de entender, flexível e manutenível. Só assim a robustez desejada poderá ser atingida.

Introdução

Tais princípios são conhecidos pelo acrônimo **SOLID**. Cada letra representa um princípio que é fundamental para a criação de códigos de alta qualidade e que possibilitem a aplicação de padrões e refatorações. A primeira vez que esses princípios foram apresentados à comunidade foi quando Robert C. Martin publicou o artigo intitulado *Design Principles and Design Patterns*, um apanhado de estudos diversos sobre qualidade de código, dentre os quais se destacavam mais os estudos de Michael Feathers.

Os princípios SOLID são listados abaixo, e veremos cada um com mais detalhes logo em seguida:

- **S:** Single Responsibility Principle (Princípio da Responsabilidade Única)
- **O:** Open/Closed Principle (Princípio do Aberto/Fechado)
- **L:** Liskov Substitution Principle (Princípio de Substituição de Liskov)
- **I:** Interface Segregation Principle (Princípio da Separação de Interfaces)
- **D:** Dependency Inversion Principle (Princípio da Inversão de Dependência)

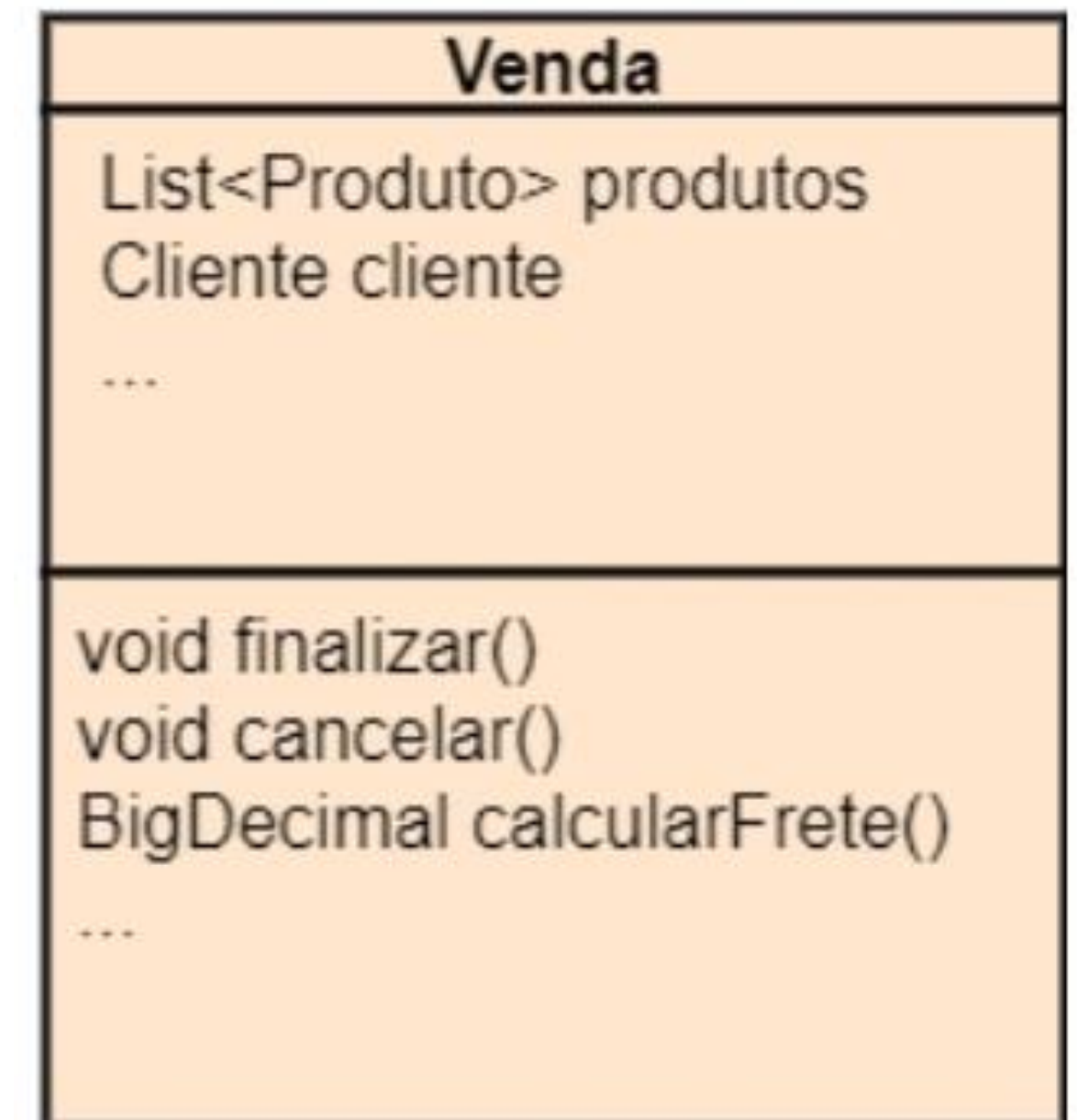
Single Responsibility Principle

Uma classe deve ter apenas uma razão para mudar

A ideia que está por detrás dessa frase é que uma classe não deve possuir mais de uma responsabilidade. Ela não deve representar nem operar sobre mais de um conceito do mundo real. Caso tal situação venha a ocorrer, o código desta classe se tornará não coeso.

A baixa coesão leva a um código mais frágil e propenso a erro, pois como existem responsabilidades misturadas, muitas vezes teremos que alterar a classe devido a demandas que nem pertencem realmente a ela, mas que de forma errônea foram definidas dentro dela. A chance de inserção de erros é aumentada porque, ao alterar uma classe para manter algo que não pertence a ela, podemos provocar efeitos colaterais em conceitos que realmente pertencem à classe.

Para ilustrar tal situação, apresentaremos um cenário de uma venda. É natural uma venda possibilitar diversas operações sobre si. Entre elas, podemos citar: finalizar, cancelar e calcular frete. Assim sendo, uma UML simplificada seria a apresentada ao lado:

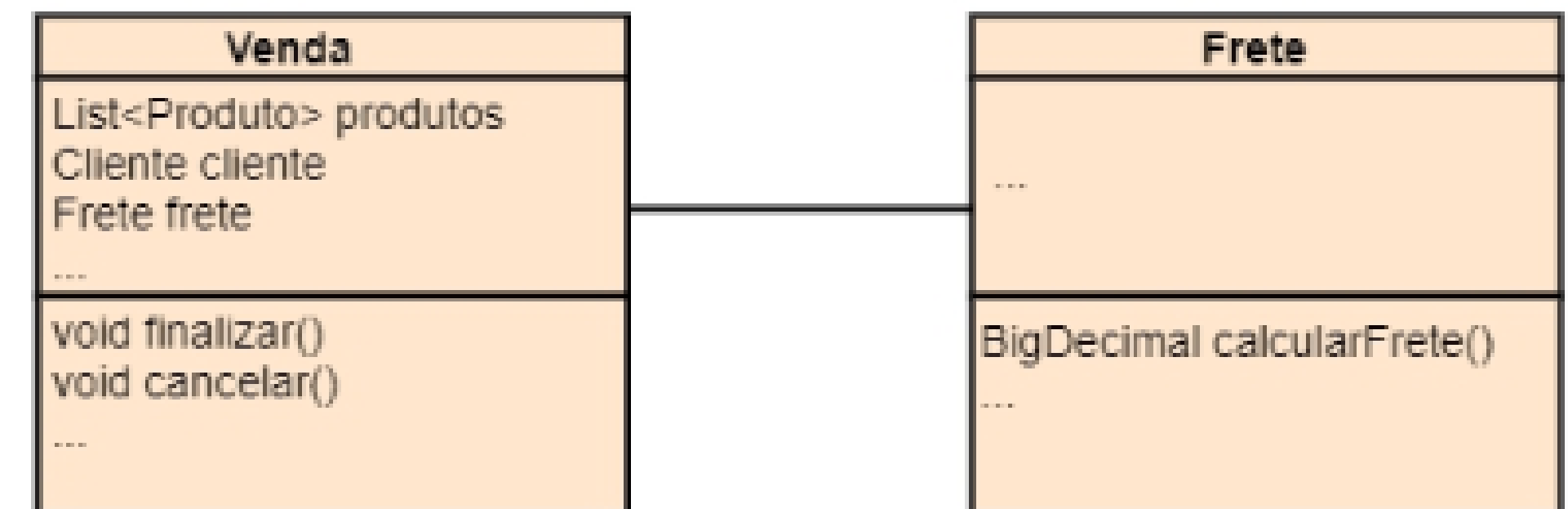


Single Responsibility Principle

Entretanto, ao analisar minuciosamente a entidade **Venda**, podemos notar que existe uma operação que não deveria ser definida dentro dela, mesmo sendo necessária: o cálculo do frete.

Embora seja comum dizermos “quanto é o frete desta venda”, o conceito de **Frete** é separado do de **Venda**. Uma venda usa um **Frete**, mas este não deve ser definido dentro dela. Qual seria a real necessidade de se alterar uma classe de **Venda** devido a uma mudança na forma de cálculo de frete? Assim, nota-se que o princípio da SRP não foi corretamente aplicado nesta classe.

A melhor forma de definir esse cenário seria o exposto ao lado. Nesta nova modelagem, a **Venda** não define em si o método de calcular frete. Foi criada uma outra classe chamada **Frete**. É nela que o método **calcularFrete** agora reside. Todavia, a classe **Venda** ainda precisa do valor do frete para poder ser finalizada, pois ele ainda é uma de suas características. Assim, uma associação foi criada, para tal método estar disponível para a classe **Venda**.



Agora o SRP foi aplicado, pois agora a classe de **Venda** só possui operações que dizem respeito à sua natureza. De acordo com sua necessidade, ela se relaciona com outras classes para poder realizar suas operações, neste caso, a classe **Frete**.

Single Responsibility Principle

Embora a solução dessa situação tenha sido simples, a detecção de que o SRP não está sendo aplicado é difícil. É natural e comum - diria até habitual - misturar as responsabilidades, mesmo que isso seja de forma involuntária. Somente com muita atenção, entendimento do negócio que a aplicação se propõe a solucionar e principalmente experiência, é que se detecta falhas de SRP.

Então, mantenha a atenção: toda vez que modificações forem feitas em classes para reparar códigos que conceitualmente não pertençam a ela, isso é um grande indício de que o SRP está sendo violado.

Open/Closed Principle

As classes, módulos, etc. de um software devem ser abertas para extensão e fechadas para modificação.

A frase anterior tem como intuito transmitir a ideia de que um software deve permitir extensões quando necessário, mas que tais atualizações gerem o mínimo de efeitos colaterais possível. É natural que software sofram manutenções - tanto evolutivas quanto corretivas - e, quando tais atividades são realizadas, é de se esperar que seja necessário efetuar o mínimo de modificações. Todavia, nem sempre esse objetivo é alcançado, dependendo de como o software veio a ser projetado. Quanto mais atômicas e isoladas as modificações forem, melhor.

Para podermos entender melhor e conseguir atingir tal princípio, é importante entender o que o *Aberto* e *Fechado* significam:

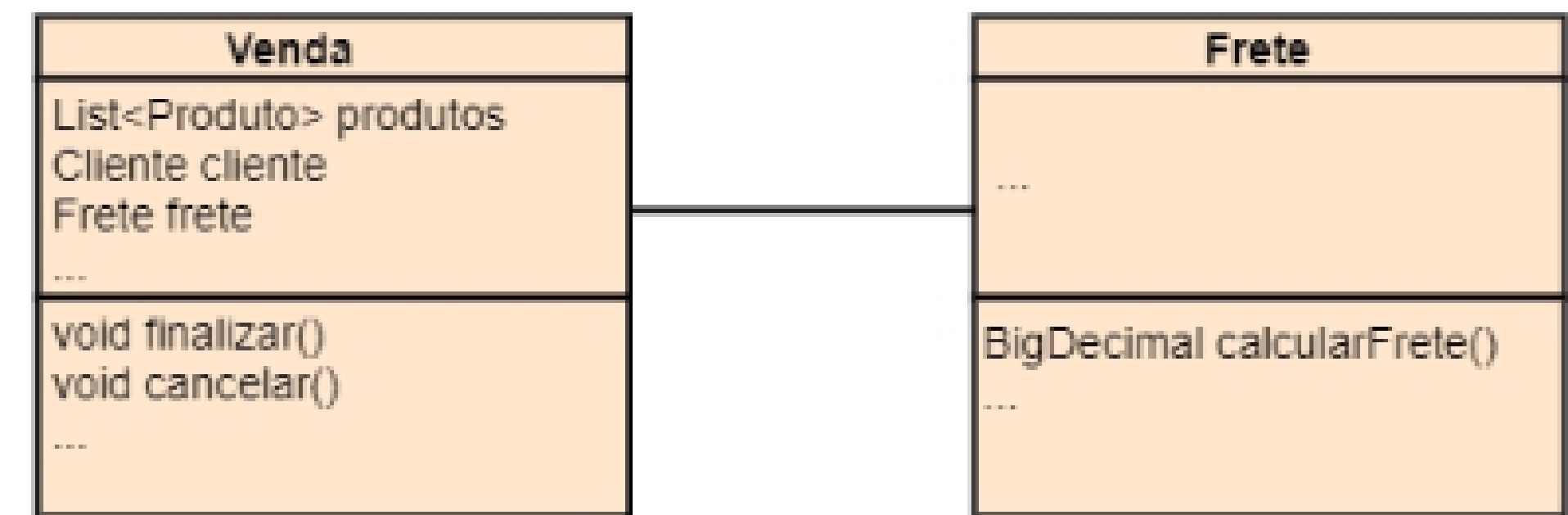
- **Aberto para extensão:** significa que, em determinados pontos do software (classes, módulos, etc.), seja possível estender seu comportamento de forma segura e isolada;
- **Fechado para modificação:** significa que, ao realizar extensões nesses determinados pontos do software, tais pontos não devem ter seus comportamentos originais afetados. As modificações devem ser transparentes, levando em consideração o ponto de vista de quem consome os serviços que estão sendo modificados.

Open/Closed Principle

Vamos revisitar o exemplo de “Frete e Venda”. Embora o SRP tenha sido aplicado e as responsabilidades estejam adequadamente definidas em suas respectivas classes, tal solução ainda pode apresentar futuros problemas.

Vamos imaginar que a **Venda** possui atualmente uma forma de calcular o frete e este é realizado através da chamada do método **calcularFrete()** a partir de algum método da classe **Venda**. Mas e se uma demanda evolutiva for possibilitar que a **Venda** tenha diversas formas de cálculo de fretes? O projeto de classe atualmente adotado não possibilita tal evolução. Ou seja, o OCP não foi aplicado.

O que comprova tal situação é que ambas as classes, **Venda** e **Frete**, são concretas. **Venda** depende exatamente do **Frete** definido. Se for necessário realizar evoluções para possibilitar diversos tipos de fretes, **Venda** inevitavelmente será afetada. Possivelmente **if's** ou **switch's** serão acrescentados em **Venda** para poder determinar qual frete calcular. Assim, este projeto não está “aberto para extensão”, pois as modificações não são atômicas - **Venda** e **Frete** devem ser modificadas - e também não está “fechado para modificações”, pois quem depende de **Frete** - no caso **Venda** - será afetado.

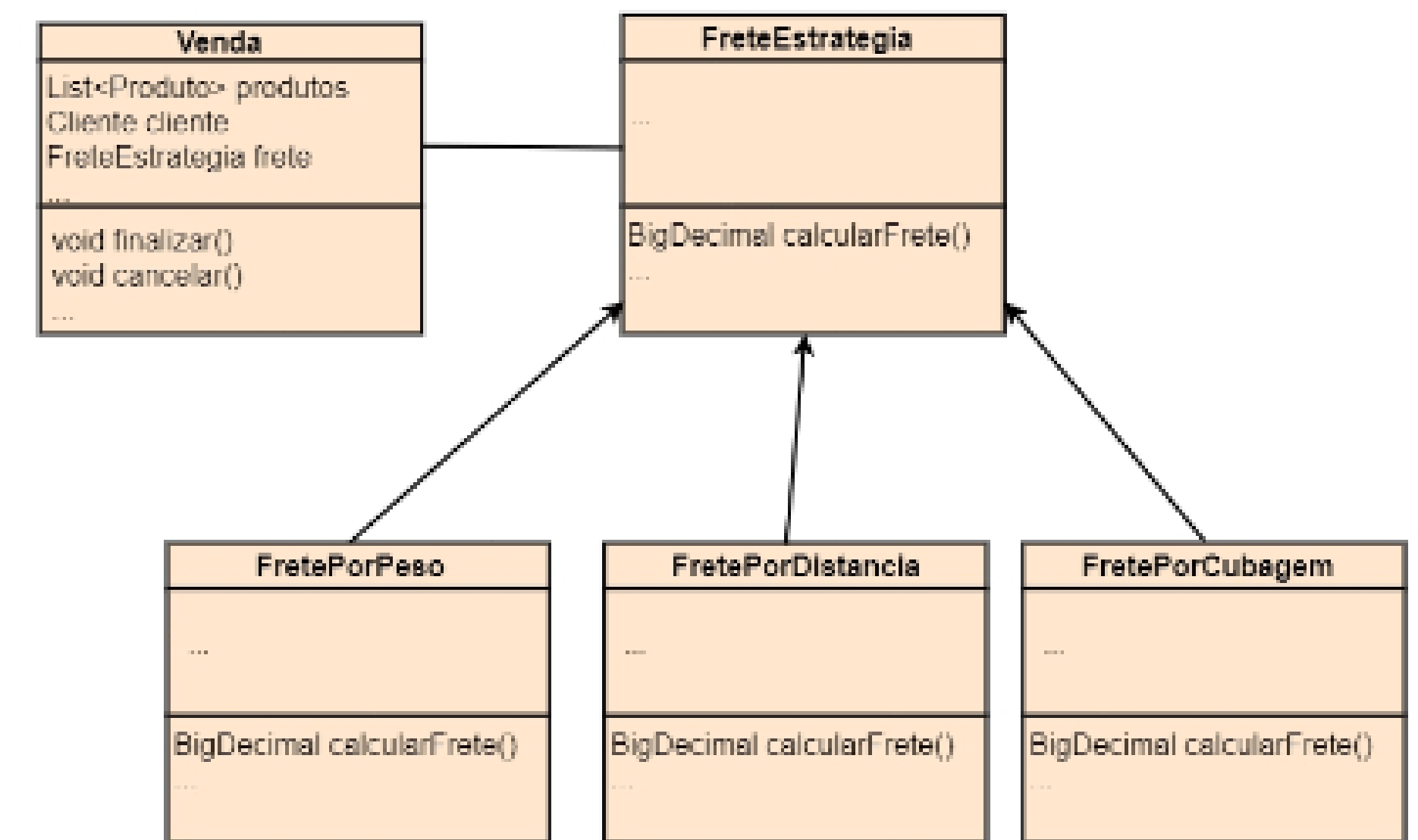


Open/Closed Principle

Embora este cenário possa parecer inevitável e incorrigível, felizmente existe uma solução: trabalhar com Abstrações! Como já é de nosso conhecimento, trabalhar com abstrações traz mais extensibilidade ao código. Usufruir dessa característica que a OO nos propicia é vital para aplicarmos OCP. Para tornar isso mais claro, vejamos o novo projeto, onde agora **Venda** depende de uma abstração de **Frete**, que disponibiliza várias estratégias de cálculo.

A partir do diagrama ao lado, podemos notar que agora **Venda** não está fixada a uma única forma de cálculo de frete. Esta agora pode, dinamicamente, usufruir das diversas formas de frete disponíveis, isto é, as estratégias podem ser plugadas dinamicamente em **Venda**. Nessas situações, devemos aplicar o polimorfismo para poder determinar qual cálculo efetuar.

Este, que já é o grande trunfo da OO em relação aos outros paradigmas, agora é o principal fator que possibilita a aplicação de OCP. Assim, o entendimento de tal característica da OO é de vital importância, pois esta e muitas outras boas práticas e padrões só podem ser utilizadas com a aplicação de polimorfismo.



Open/Closed Principle

Embora a OCP possibilite uma grande flexibilidade se seja vital para facilitar o projeto e codificação de aplicações, infelizmente somente a aplicação deste princípio não torna o software completamente imune a alterações. Inevitavelmente, podem surgir demandas que degradarão a arquitetura do software ou, pelo menos, impactar drasticamente uma funcionalidade de tal maneira que nem mesmo a OCP possa ser a solução.

Para exemplificar isso, basta imaginar que a solução de termos uma abstração de **Frete** para podermos plugar fretes diferentes não seria suficiente, caso a **Venda** precisasse possuir mais de um tipo de **Frete**. Então, caso a venda permitisse entregas em endereços diferentes e consequentemente fretes diferentes, somente OCP não seria o suficiente.

Em **Venda** temos apenas uma estratégia de teste disponível, através do atributo **frete**. Para possibilitar mais de uma estratégia, teríamos que mudar tal atributo para uma lista de estratégias. Neste caso, teríamos que mudar para **List<FreteEstrategia> fretes**. Devido a isso, todas as classes que manipulavam o atributo **frete** seriam afetadas e consequentemente tal alteração não estaria “fechada para modificações”.

O importante sobre o princípio OCP é entender que a aplicação dele resolverá a grande maioria dos problemas, mas que não resolverá 100% dos casos. Contudo, isso não tira a importância de entendê-lo e aplicá-lo sempre que for possível e necessário.

Liskov Substitution Principle

Subtipos podem ser substituídos pelos seus supertipos, sem que isso altere comportamentos

A definição anterior apresenta uma nova perspectiva sobre a herança. Embora seja comum pensarmos em criar subtipos para que estes possam substituir supertipos genéricos, temos que criá-los de modo que, em determinadas situações, eles possam ser substituídos por seus supertipos. Contudo, tais substituições não devem gerar impactos comportamentais na aplicação. Ou seja, em determinados momentos podemos trabalhar tanto com o supertipo ou com os subtipos, mas isso não deve fazer com que a aplicação gere resultados adversos.

Para elucidar este princípio, vamos avaliar um exemplo fictício de um processador de pagamentos, onde existe uma classe responsável por manipular Contas Correntes para, a partir delas, realizar os pagamentos. Inicialmente temos as classes **ContaCorrente** e **ProcessadoraPagamento**. A primeira possui um nome, saldo e um limite especial, para quando o saldo se esgotar e o cliente precisar de mais dinheiro. A segunda, basicamente, é responsável por criar as contas e processar os pagamentos.

Liskov Substitution Principle

```
public class ContaCorrente {
    private String nome;
    private double limiteEspecial;
    private double saldo;

    public ContaCorrente() {
        this.limiteEspecial = 100;
        this.saldo = 0;
    }

    //gets e sets

    public void saque(double valor) {
        this.saldo = saldo - valor;
    }
}

public class ProcessadoraPagamento {
    public static void main(String[] args) {
        List<ContaCorrente> contas = new ArrayList<>();

        ContaCorrente conta1 = new ContaCorrente();
        conta1.setNome("ContaCorrente1");
        conta1.setSaldo(150);
        contas.add(conta1);

        ContaCorrente conta2 = new ContaCorrente();
        conta2.setNome("ContaCorrente2");
        conta2.setSaldo(60);
        contas.add(conta2);

        double valorFinanciamento = 80;

        for (ContaCorrente conta : contas) {
            if (conta.getSaldo() > 0) {
                conta.saque(valorFinanciamento);
            } else if (conta.getLimiteEspecial() > valorFinanciamento) {
                conta.saque(valorFinanciamento);
            }

            System.out.println("O saldo atual da conta " + conta.getNome() + " é " + conta.getSaldo());
        }
    }
}
```


Liskov Substitution Principle

O resultado desta execução seria:

- O saldo atual da conta ContaCorrente1 é 70.0
- O saldo atual da conta ContaCorrente2 é -20.0

Avaliando os resultados, vemos que a primeira conta ficou com um saldo de 70 após o pagamento do financiamento. Já a segunda ficou negativo em 20 e para isto usou o limite especial.

Entretanto, um dia a empresa decide também processar os pagamentos dos financiamentos a partir de poupanças. Já existe uma estrutura pronta para processar contas correntes e uma poupança é um tipo de “conta corrente”, mas que possui um rendimento. Partindo deste princípio, a nova classe **Poupanca** pode herdar de **ContaCorrente** e só acrescentar o rendimento que é particular a si. Desta forma, a classe seria:

```
public class Poupanca extends ContaCorrente {  
    private double rendimento;  
  
    //gets e sets  
}
```

Liskov Substitution Principle

Sendo assim, a **ProcessadoraPagamento** poderia agora também processar pagamentos a partir de poupanças, e seu método **main** ficaria da seguinte forma:

```
public class ProcessadoraPagamento {
    public static void main(String[] args) {
        List<ContaCorrente> contas = new ArrayList<>;

        ContaCorrente conta1 = new ContaCorrente();
        conta1.setNome("ContaCorrente1");
        conta1.setSaldo(150);
        contas.add(conta1);

        ContaCorrente conta2 = new ContaCorrente();
        conta2.setNome("ContaCorrente2");
        conta2.setSaldo(60);
        contas.add(conta2);

        Poupanca poupanca = new Poupanca();
        poupanca.setNome("Poupança1");
        poupanca.setSaldo(20);
        contas.add(poupanca);

        double valorFinanciamento = 80;

        for (ContaCorrente conta : contas) {
            if (conta.getSaldo() > 0) {
                conta.saque(valorFinanciamento);
            } else if (conta.getLimiteEspecial() > valorFinanciamento) {
                conta.saque(valorFinanciamento);
            }

            System.out.println("O saldo atual da conta " + conta.getNome() + " é " + conta.getSaldo());
        }
    }
}
```

Liskov Substitution Principle

O resultado desta execução seria:

- O saldo atual da conta ContaCorrente1 é 70.0
- O saldo atual da conta ContaCorrente2 é -20.0
- O saldo atual da conta Poupança1 é -60.0

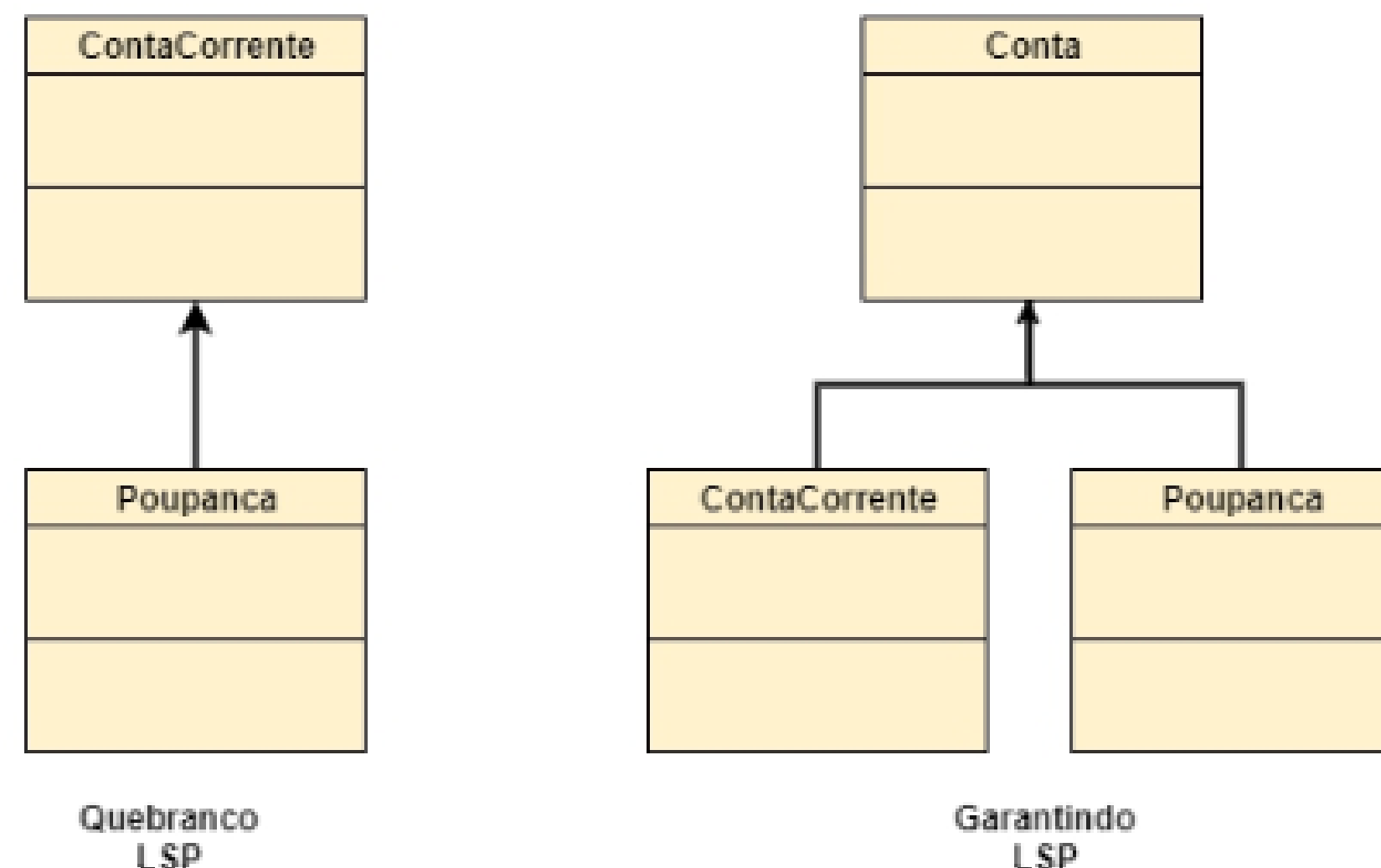
Após a execução, podemos ver que a poupança ficou negativa. Infelizmente, esse é um resultado errado, pois poupanças não possibilitam saldo negativo. Neste caso, houve um erro de substituição. Quando o **for** itera sobre uma lista de **ContaCorrente** e esta possui uma **Poupanca**, a aplicação se comporta de forma errada. Ou seja, o LSP não foi garantido, pois o subtipo **Poupanca**, ao ser substituído pelo seu supertipo **ContaCorrente**, gerou um resultado inesperado.

O que causou a quebra do LSP aqui, e que causa quebras em outros contextos, foi a errônea decisão de fazer **Poupanca** herdar de **ContaCorrente**. Geralmente, quando se realizam decisões erradas no momento de criar hierarquias de classes, elas terminam por violar o LSP. Neste contexto, uma “poupança” se parece muito com uma “conta corrente”, pois pode se fazer um saque, possuir um saldo, pertence a uma pessoa, entre outras coisas, então, é comum pensar em fazer “poupança” herdar de “conta corrente”, usar a relação “é um”. Entretanto, para garantir o LSP é necessário pensar de forma um pouco mais cuidadosa. Muito mais que a relação “é um”, devemos nos preocupar com a relação “se comporta como”.

Liskov Substitution Principle

Isso dá uma nova perspectiva sobre a herança. Devemos também pensar se um subtipo poderá se comportar como seu supertipo. Neste exemplo, inicialmente parecia que sim, mas infelizmente, não. Uma “poupança” não pode se comportar como uma “conta corrente”, pois esta possui um “limite especial”, que possibilita saldos negativos. Poupanças não possuem esse tipo de comportamento. Sendo assim, uma poupança jamais pode ser um subtipo de uma conta corrente. Então, como resolver a situação?

Para garantir a aplicação do LSP, geralmente uma remodelagem da hierarquia deve ser feita. Neste exemplo e geralmente em outros casos, a relação de pai e filho é substituída por uma relação de irmandade. Assim, **ContaCorrente** e **Poupanca** passariam a ser irmãs e filhas de uma nova classe, que seria supertipo para ambas. A classe **Conta** poderia ser criada para armazenar o que fosse realmente comum a ambas e em cada classe separadamente teríamos o que fosse pertinente a cada tipo de conta. A imagem a seguir apresenta a hierárquica antes e depois da melhoria.



Liskov Substitution Principle

```
public abstract class Conta {
    private String nome;
    private double saldo;

    public Conta {
        this.saldo = 0;
    }

    //gets e sets

    public void saque(double valor) {
        if (this.getSaldo() - valor > 0) {
            this.setSaldo(this.getSaldo() - valor);
        }
    }
}

public class ContaCorrente extends Conta {
    private double limiteEspecial;

    public ContaCorrente() {
        this.limiteEspecial = 100;
        this.setSaldo(0);
    }

    //gets e sets

    @Override
    public void saque(double valor) {
        //[...]
    }
}

public class Poupanca extends Conta {
    private double rendimento;

    public Poupanca() {
        this.setSaldo(0);
    }

    //gets e sets

    @Override
    public void saque(double valor) {
        //[...]
    }
}
```

Liskov Substitution Principle

Após a remodelagem, a linha **contas.add(poupança);** da classe **ProcessadoraPagamento** indica um erro de compilação. Isso ocorre porque a lista **contas** é do tipo **ContaCorrente**. Logo, uma “poupança” não pode mais ser adicionada, pois agora elas são irmãs. Ou seja, estamos agora conseguindo evitar a quebra de LSP em tempo de execução e já estamos a detectando em tempo de compilação. Entretanto, o problema persiste. O LSP ainda não está garantido, pois não estamos conseguindo substituir “poupança” por “conta corrente”. Sendo assim, as melhorias devem prosseguir.

O próximo passo é alterar a lista **contas** para aceitar o tipo **Conta** e não mais **ContaCorrente**. Após isso, o teste lógico **if (conta.getLimiteEspecial() > valorFinanciamento)** apresenta um erro em **conta.getLimiteEspecial()**, pois este método agora só pertence a **ContaCorrente**. Neste ponto poderia se pensar em usar um **if** para determinar qual classe usar, utilizando algo como **if (conta instanceof ContaCorrente)** para usar o limite especial.

Contudo, isso ainda ocasionaria uma quebra do LSP, pois não seria possível criar subtipos e eles serem substituídos sem que uma manutenção nesta estrutura de **if** seja feita. Logo, esta abordagem não é o melhor caminho. É melhor usar polimorfismo, que conseguirá gerar um código flexível e transparente à adição de novos tipos de contas.

```
for (ContaCorrente conta : contas) {  
    conta.saque(valorFinanciamento);  
  
    System.out.println("O saldo atual da conta " + conta.getNome() + " é " + conta.getSaldo());  
}
```

Liskov Substitution Principle

Desta forma, Toda vez que novos tipos de contas forem criados, o código em **ProcessadoraPagamento** não sofrerá alteração. Para finalizar, pode não ter ficado claro, mas todas as alterações feitas para possibilitar a aplicação de LSP no final terminaram por aplicar OCP. Ou seja, para garantir a aplicação de LSP, o uso de OCP é primordial. Ambos os princípios estão intimamente ligados: a falta de OCP leva a erros de LSP. Outro ponto também importante é que a quebra de LSP começou depois de herdarmos de uma classe concreta, que já falamos que é algo perigoso a se fazer.

Interface Segregation Principle

Clientes não devem ser forçados a depender de métodos que não usem.

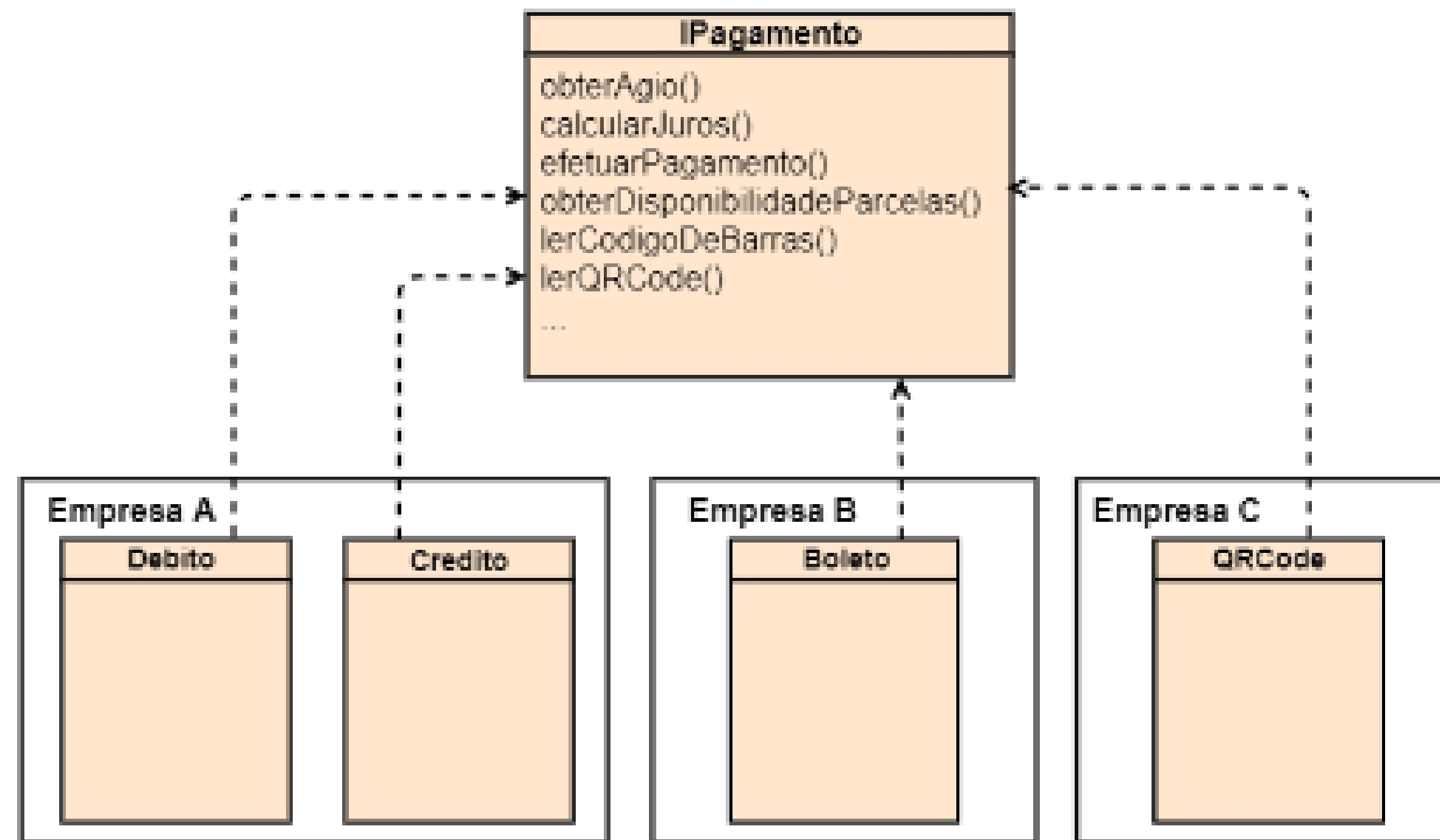
A frase anterior expõe uma situação muito comum de se encontrar, mas que não deveria ocorrer: classes herdando de superclasse, métodos desnecessários e/ou classes implementando métodos de interfaces, mesmo que de forma “bypass”, também desnecessários. Tudo isso só para satisfazer um projeto de classes mal definido.

Infelizmente, é comum vermos classes herdando e/ou implementando o que se chama de “Interface Gorda” ou, em inglês, “Fat Interface”. Tal “Interface Gorda” (classe abstrata ou interface) possui muitos métodos, que geralmente foram definidos de forma não coesa, isto é, as responsabilidades de tais métodos estão misturadas. Dessa forma, a classe ou interface fica “gorda”, cheia de métodos que foram definidos para resolver atividades completamente díspares.

Além desta baixa coesão, a definição de uma “interface gorda” gera um forte acoplamento. Isso ocorre devido ao fato de que, se tais métodos foram definidos de forma não coesa, há uma grande probabilidade de que tal classe ou interface seja base para muitas classes distintas, que realizam diferentes atividades e têm, consequentemente, responsabilidades diferentes. Logo, alterações nesta “interface gorda” terminarão impactando em diversos pontos da aplicação. Isso claramente deixa o código rígido e complexo.

Para ilustrar essa situação, imaginemos um software de pagamentos, que aceita pagamentos via débito, crédito, boleto e QR Code. Quando uma empresa adere a tal software, ela deve se comunicar com ele através de uma interface que a empresa disponibiliza, onde constam as operações necessárias para realizar a comunicação. Cada empresa tem a liberdade de optar pelas formas de pagamento que desejar.

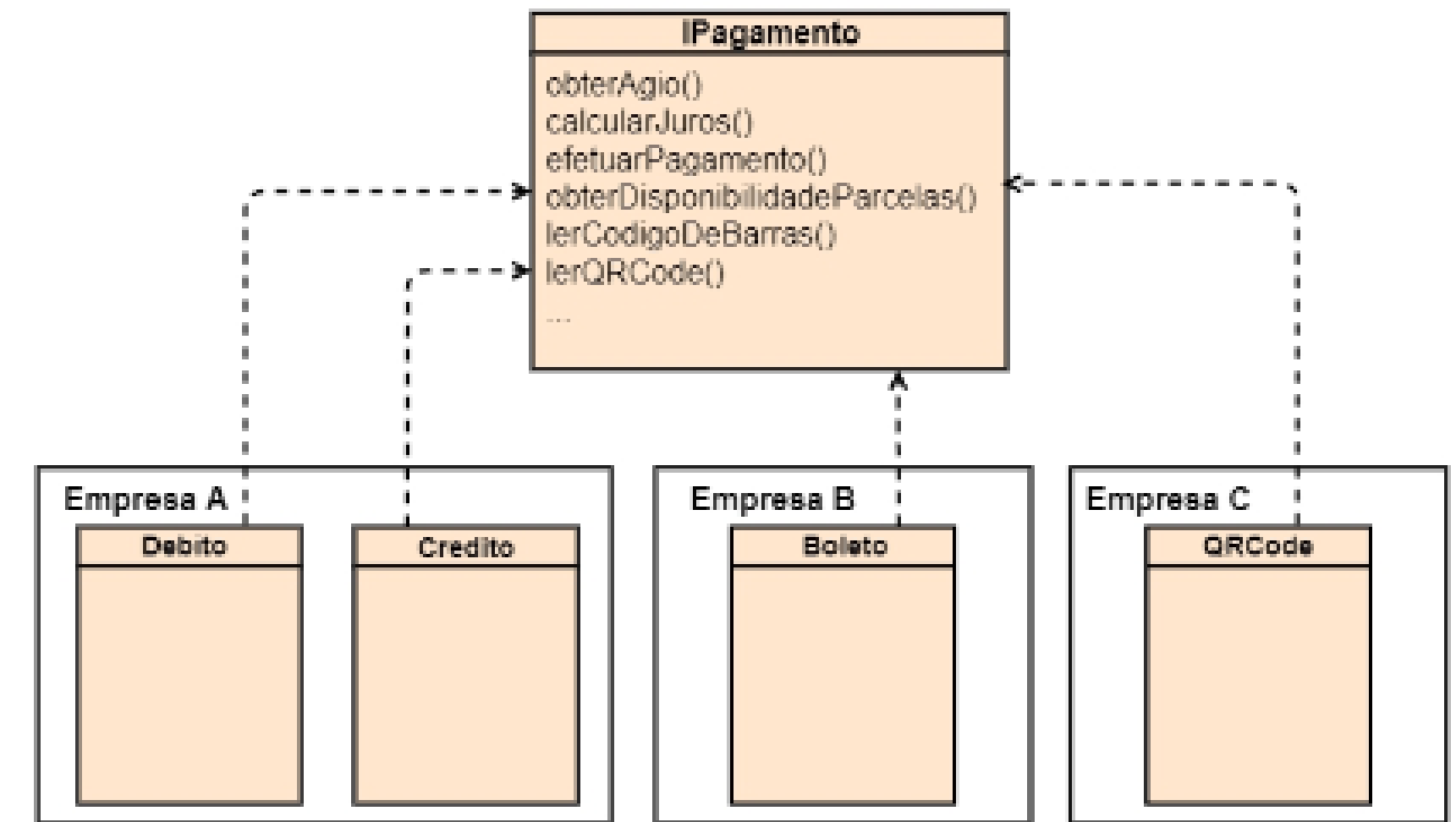
Interface Segregation Principle



A imagem anterior é um claro exemplo de uma “interface gorda”. Embora a empresa tivesse objetivo de facilitar a comunicação, da forma que foi feita ela não conseguirá atingir seu objetivo em toda plenitude. Nota-se que temos métodos para formas de pagamentos distintas em uma mesma interface. Logo, quando uma empresa escolhe apenas um subconjunto das formas de pagamento, ela é obrigada a tratar as outras formas, que não lhe interessam.

Interface Segregation Principle

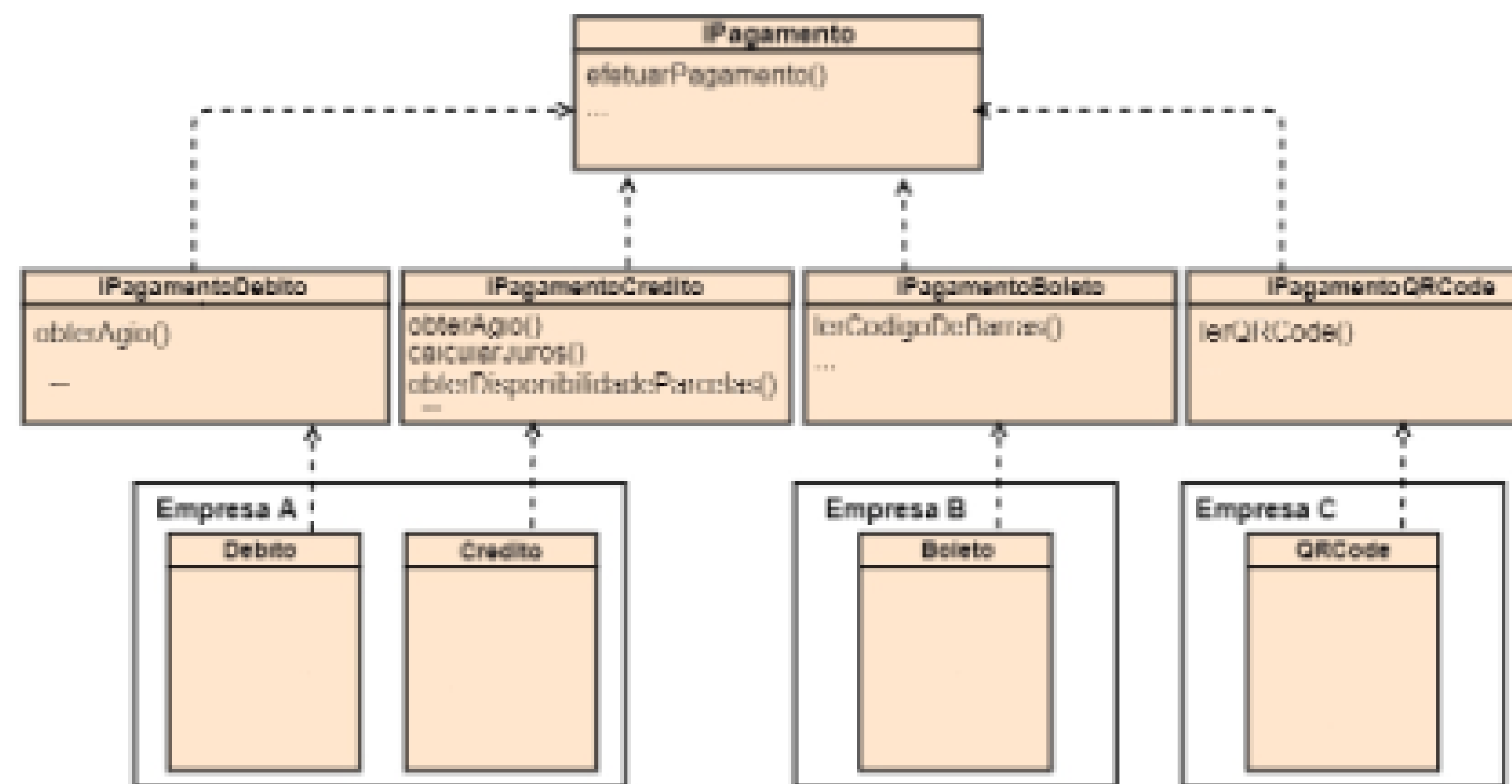
Na **Empresa A**, por exemplo, que só optou por débito e crédito, ela teve que implementar de forma falsa as outras operações, só para satisfazer a interface. Situações similares ocorrem com **Empresa B** e **Empresa C**. Este é um exemplo clássico e comum de quebra do ISP. Métodos que são obrigatoriamente implementados, mas que não são necessários a um negócio específico. Esse processo de quebra do princípio ISP também é conhecido como “vazamento de interface”, pois algumas operações “vazaram” para determinadas classes, que não precisavam desses métodos.



Mesmo que uma determinada empresa optasse por todas as opções de pagamento e assim evitasse o “vazamento”, esta interface ainda assim ocasionaria um outro problema: a baixa coesão. A classe criada para implementar tal interface possuiria uma mistura das formas de pagamento, logo, não estaria coesa. Essa baixa coesão termina por levar à quebra de um outro princípio, o SRP. Portanto, “interfaces gordas” definitivamente não são uma boa decisão de projeto.

Interface Segregation Principle

Ao aplicar ISP neste projeto, o resultado seria o seguinte:



Nota-se que agora cada empresa tem a possibilidade de trabalhar sua forma de pagamento isolada, sem ter que se preocupar com as formas pelas quais não optou. Isso gera uma maior independência entre as classes clientes das interfaces, pois não há vazamentos. Além disto, as operações comuns a todas as formas de pagamento foram isoladas em uma interface de nível mais alto, para assim poder ser compartilhada com as demais interfaces.

Por fim, podemos ressaltar que a aplicação do ISP torna, mais uma vez, o código mais extensível e flexível, além de coeso e manutenível. Embora o uso de interface e classe abstrata possam, por si só, tornar o código mais flexível, se infelizmente estas forem definidas de forma “gorda”, minaremos as vantagens de seus usos.

Dependency Inversion Principle

Deve-se depender de abstrações e não de conceitos concretos.

A sentença anterior determina que em um projeto de classes devidamente bem modelado, segundo os preceitos da OO, módulos (classes, componentes, etc.) não devem se relacionar diretamente com outros módulos, mas sim com abstrações - interfaces - que devem expor o que se deseja e podem ser compartilhadas. Em outras palavras, módulos de alto nível não devem depender diretamente dos módulos de baixo nível, assim como os módulos de baixo nível não devem depender diretamente dos de alto nível. Ambos devem depender entre si através de abstrações.

Toda boa aplicação orientada a objetos deve ser construída em várias camadas. Deve-se ter uma separação lógica entre as diversas classes existentes, para que os devidos agrupamentos em pacotes sejam feitos e as camadas possam ser facilmente visualizadas. Assim sendo, as camadas trocarão informações via suas interfaces, as quais serão responsáveis por expor pontos de entrada e saída de dados. Embora esse modo de trabalho seja a melhor forma de se trabalhar com a OO, nem sempre é aplicado.

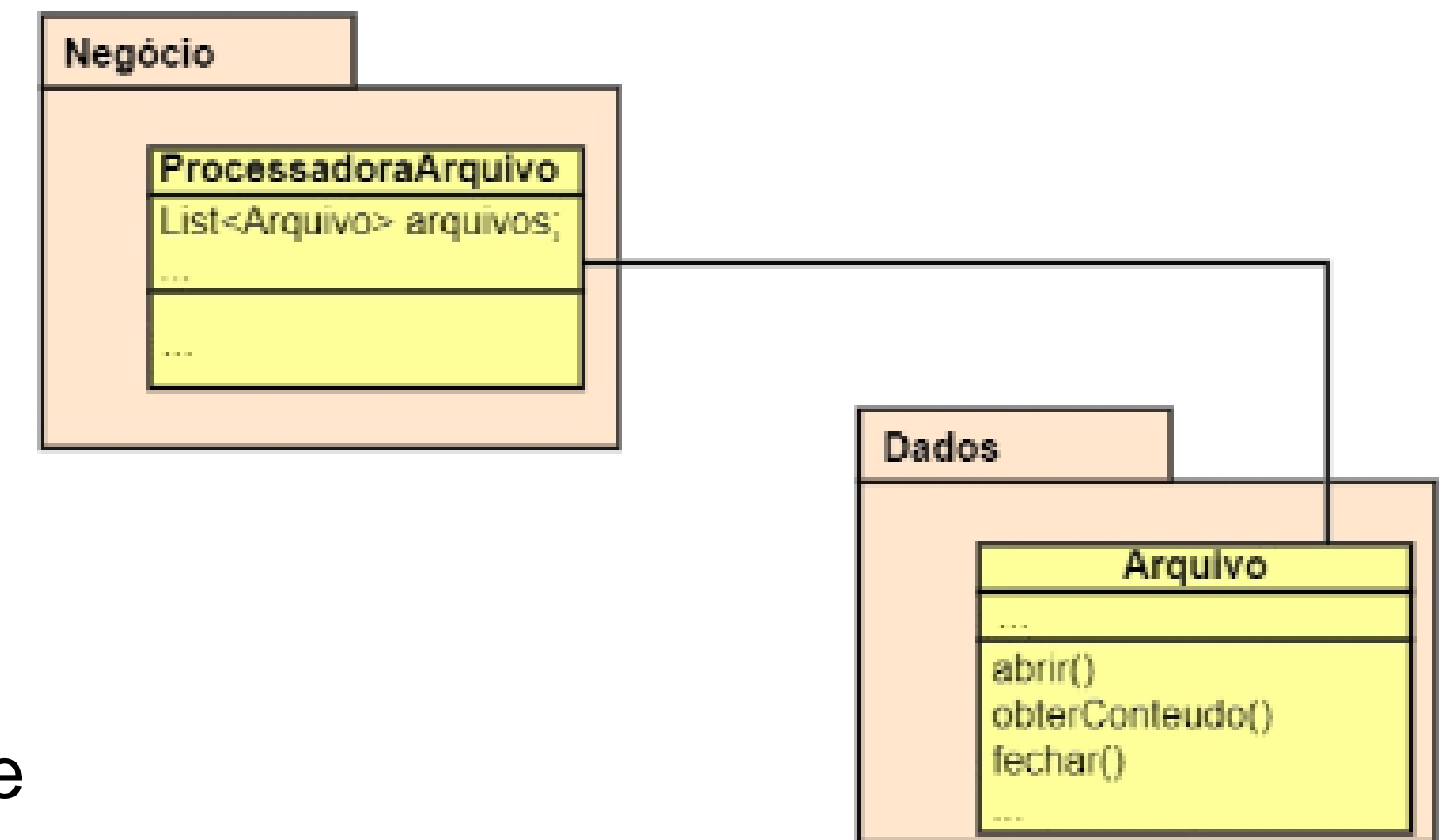
A ausência desta identificação e aplicação de camadas leva a códigos mais rígidos - com forte/alto acoplamento - que minam a flexibilidade de evolução. Para tornar mais clara esta situação, vejamos o exemplo a seguir.

Dependency Inversion Principle

Imaginemos que uma aplicação é responsável por processar arquivos, por exemplo, arquivos .pdf. Assim sendo, temos uma classe de negócio chamada **ProcessadoraArquivo** e uma classe de dados chamada **Arquivo**. A relação entre tais classes é exposta na imagem ao lado.

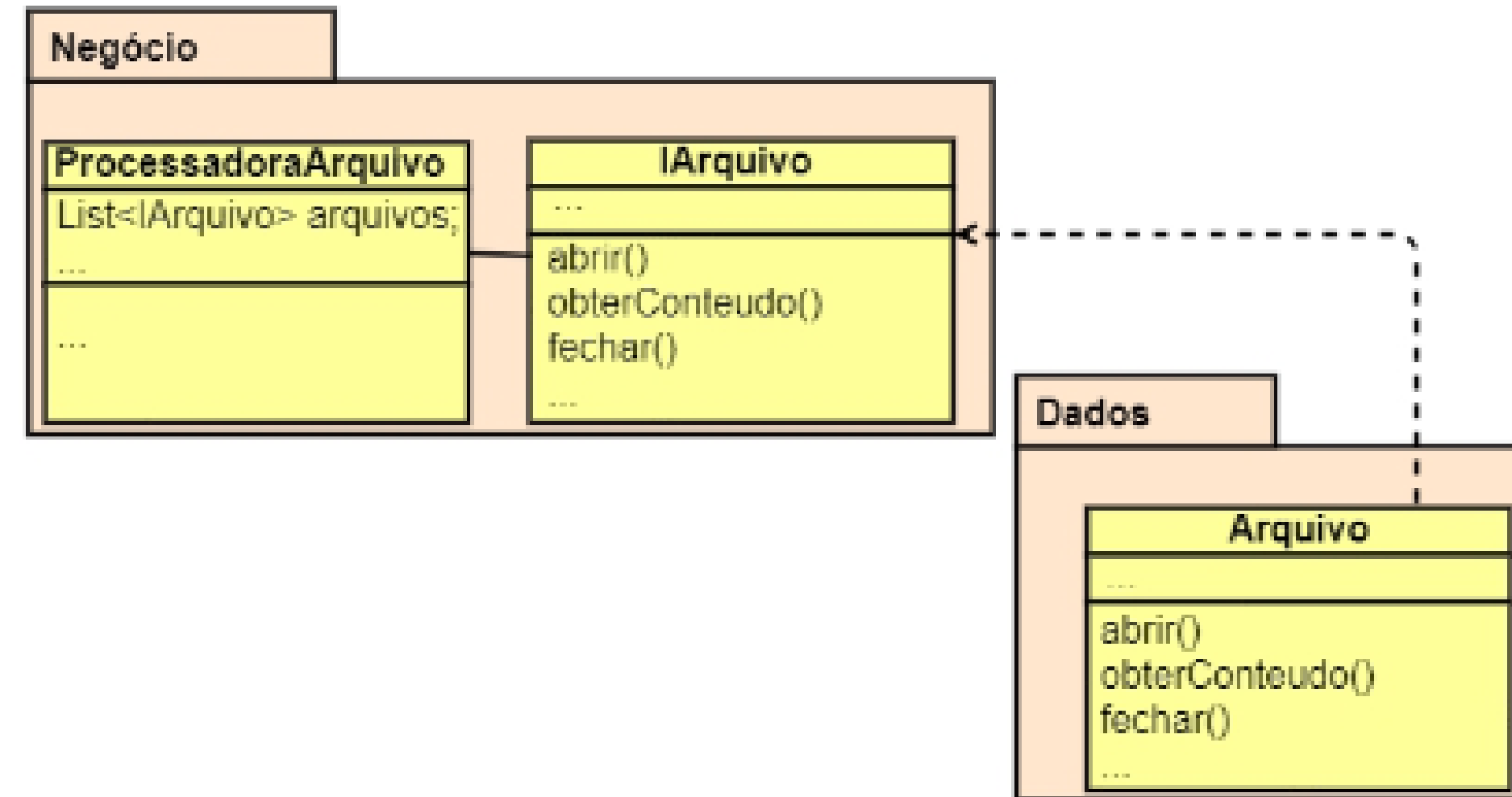
Essa figura demonstra que **ProcessadoraArquivo** - da camada de alto nível - tem uma dependência direta com **Arquivo** - da camada de baixo nível -, que é uma classe concreta. **ProcessadoraArquivo** tem um atributo do tipo **List<Arquivo>** e inicialmente o processamento é de apenas .pdf. Ou seja, a classe **ProcessadoraArquivo** utiliza métodos de **Arquivo** que possuem códigos para tratar apenas esse tipo de arquivo.

Porém, se houver a demanda por novos tipos de arquivo, como .xml, .html, etc., como serão tratados? Infelizmente uma manutenção evolutiva terá que ser feita em **ProcessadoraArquivo**, pois esta depende somente e diretamente de **Arquivo**, o qual terá uma manutenção evolutiva maior ainda, pois é ela que proverá os métodos para **ProcessadoraArquivo** processar os arquivos.



Dependency Inversion Principle

A partir do exposto, nota-se que depender diretamente de classes concretas pode gerar um acoplamento muito forte/alto, o que termina por dificultar manutenções e diminuir a flexibilidade do código. Para solucionar esse problema, deve-se aplicar DIP. A figura a seguir demonstra como realizar isso.



Agora, **ProcessadoraArquivo** não depende diretamente da classe concreta **Arquivo**, mas sim de uma interface definida em sua própria camada, que a classe **Arquivo** deve implementar. Cada tipo de arquivo terá a sua classe, como **ArquivoPDF**, **ArquivoXML**, etc. Elas terão as implementações necessárias para tratar cada tipo de arquivo. Assim, realizamos a alteração de dependência entre as camadas e tornamos o código mais extensível e flexível. Podemos processar arquivos de vários tipos de forma transparente, em relação a **ProcessadoraArquivo**.

Dependency Inversion Principle

Para finalizar sobre DIP, podemos dizer que o projeto de depender de classes concretas parece mais com programação procedural/estruturada do que com OO. Embora seja comum encontrar códigos assim em programas que usam linguagens OO, eles estão pobremente definidos. Depender de abstrações é a melhor forma de projetar sistemas orientados a objetos. Uma prova disto é que frameworks como Hibernate, JSF, entre outros, usam e abusam deste princípio.

Isso vem do fato de que, por natureza, todo framework deve ser incompleto, mas extremamente extensível, para poder se adaptar a situações diversas. Então, a melhor forma de criar tais pontos de extensão é usar DIP.

OBRIGADO!



www.ibmec.br

 /ibmec

 ibmec

 @ibmec_oficial

 ibmec

