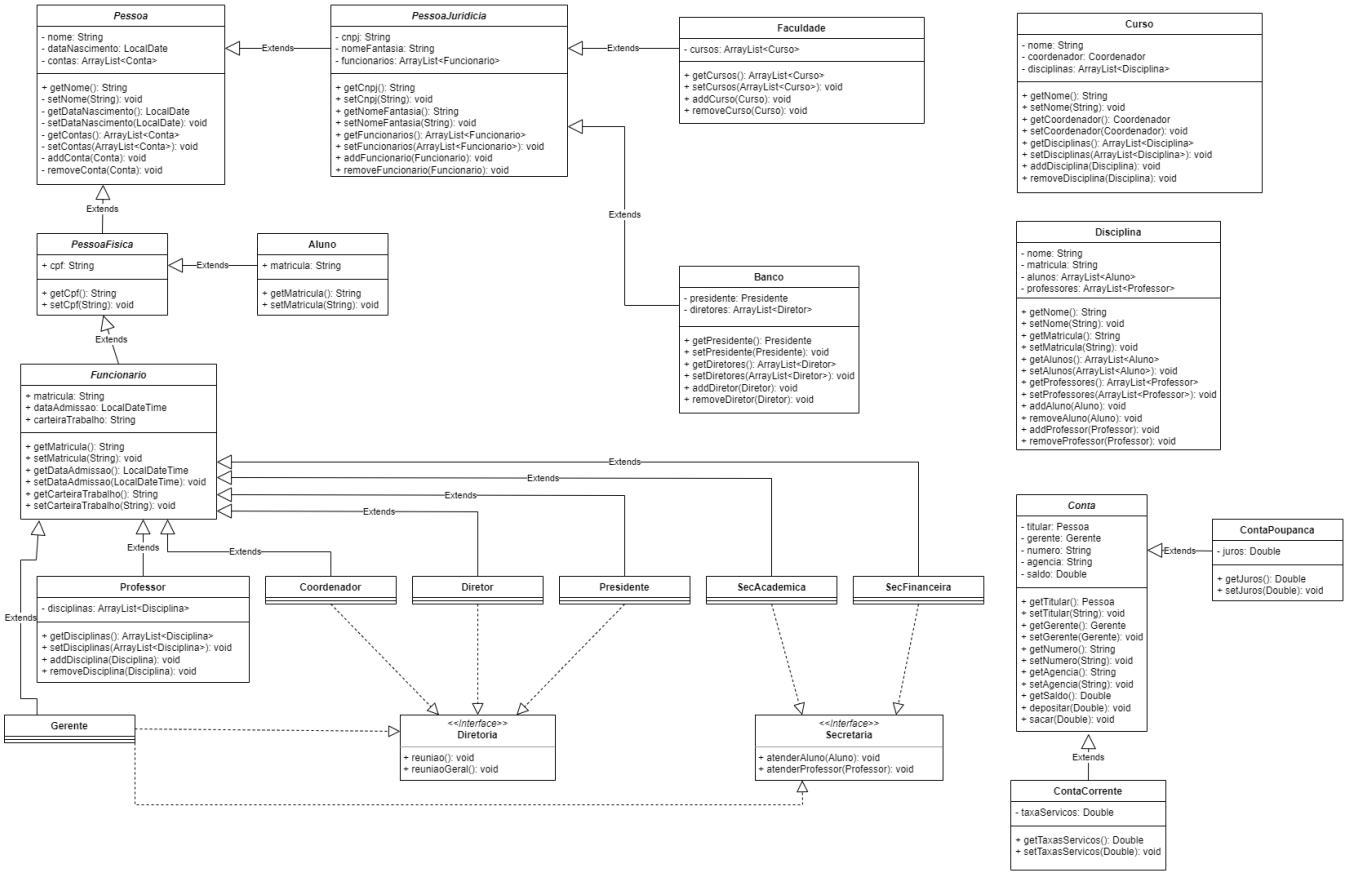
Tutorial

Nesse tutorial procuro explicar, o mais detalhadamente possível, o processo de construção do diagrama de classes e sua posterior implementação.

1 Diagrama de Classes

Observando o diagrama de classes é possível ver que temos 19 classes:

1. Pessoa;
2. PessoaJuridica;
3. Faculdade;
4. Curso;
5. PessoaFisica;
6. Aluno;
7. Banco;
8. Disciplina;
9. Funcionario;
10. Professor;
11. Coordenador;
12. Diretor;
13. Presidente;
14. SecAcademica;
15. SecFinanceira;
16. Gerente;
17. Conta;
18. ContaCorrente;
19. ContaPoupanca.



Além das classes existem 2 `Interfaces` :

1. Diretoria;
2. Secretaria.

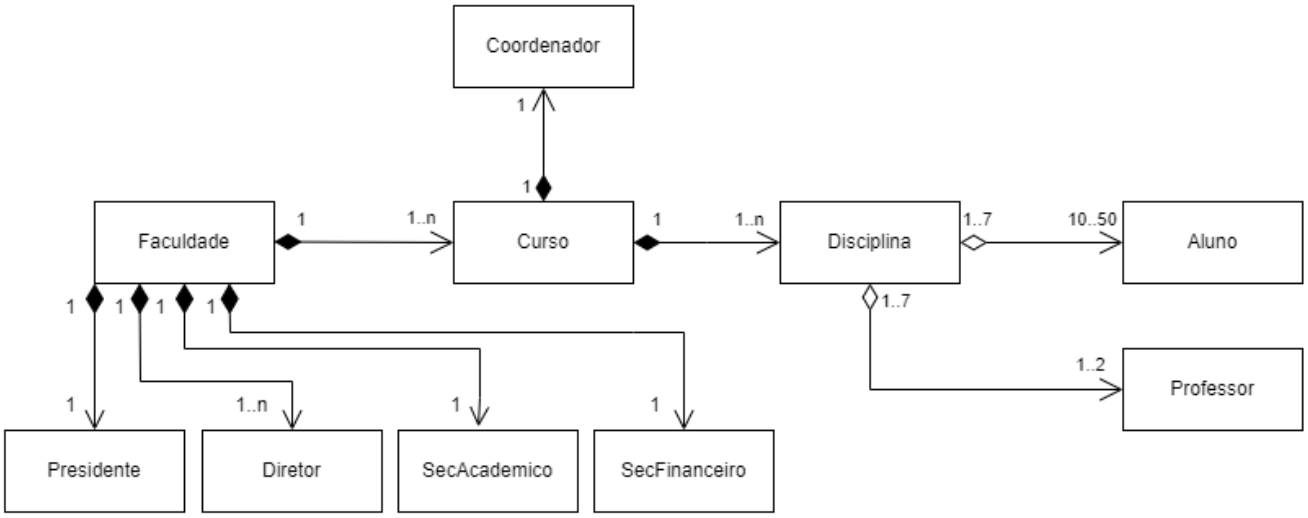
A construção do diagrama em si não é uma tarefa trivial, de forma que as empresas tendem a ter pessoas específicas para lidarem com isso, normalmente em conjunto com a construção do Banco de Dados, uma vez que esse tipo de programa é espelhado em um `Banco de Dados Relacional` .

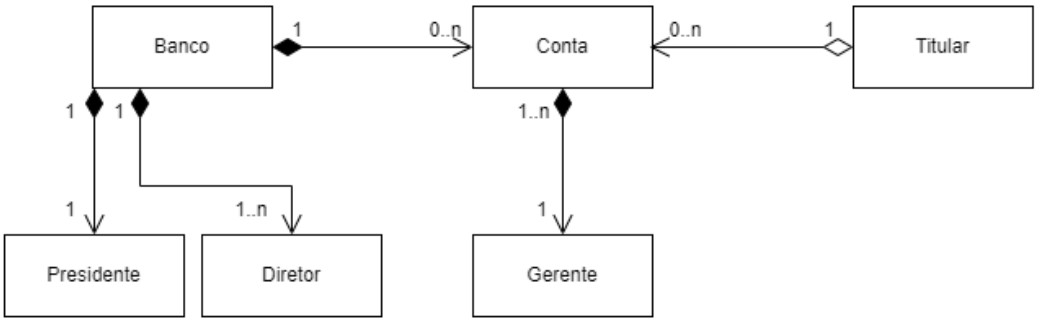
Os desenvolvedores irão receber os diagramas prontos e terão de implementá-los. Caso algum deles perceba que a adição, remoção ou mistura de classes se apresente como uma melhor solução em determinado trecho do programa, o desenvolvedor terá de requisitar à equipe de Banco de Dados, ou equipe de Analistas, a modificação. Se aceita a requisição, então o desenvolvedor receberá a atualização. Senão, terá de continuar estritamente como planejado.

Observando de forma mais detalhada o diagrama, é possível notar algumas padronizações:

- **Retângulos**: o diagrama é permeado de retângulos, onde cada retângulo é dividido em duas ou três seções. Cada retângulo representa uma `classe`, ou outra entidade, como `interface`, por exemplo.
  - A primeira seção de um retângulo possui o nome da classe, ou entidade. Classes "normais", têm seus nomes escritos de forma normal. Classes `abstratas` têm seus nomes escritos em *itálico*. As `interfaces` recebem uma indicação dentro de dois chevrons `<< >>`
  - A segunda seção contém os atributos de uma classe. Os atributos são descritos da seguinte forma: [encapsulamento] : .
    - O encapsulamento é opcional. Porém se estiver indicado como `-`, significa que o atributo é `privado`. Se estiver indicado com `+`, significa que o atributo é `público`, e se estiver indicado com `#`, é porque o atributo deve ser `protegido` (*protected*). Não havendo qualquer indicação terá por consequência a implementação do atributo como `default`.
  - A terceira seção contém os métodos, os quais são indicados da seguinte forma: [encapsulamento] ([tipo]): .
    - O encapsulamento aqui funciona da mesma forma que nos atributos.
    - Dentro dos parênteses é opcional o preenchimento dos tipos dos parâmetros. Por exemplo, na classe `Pessoa` o método `setNome()` deve receber um parâmetro do tipo `String`.
    - O tipo indicado após os dois pontos `:` é o tipo de retorno do método. Por exemplo, na classe `Pessoa` o método `getNome()` retorna um valor do tipo `String`.
- **Setas**: as setas indicam algum relacionamento entre as classes. Uma seta cheia, porém com ponta vazia, significa `herança`. Ou seja, temos aí a indicação de uma superclasse e uma subclasse. Quando a seta é tracejada, temos o conceito de `implementação` de `interface`. Lembrando, em `Java`, uma classe só pode ter uma superclasse, mas pode implementar várias interfaces.

No diagrama seguinte temos um detalhamento da relação entre algumas classes. São duas as relações mostradas: **composição** e **agregação**.


Coordenador
1
1
Faculdade
1
1..n
Curso
1
1..n
Disciplina
1..7
10..50
Aluno
1
1
1
1
1..7
1..2
Professor
1
1..n
1
1
Presidente
Diretor
SecAcademico
SecFinanceiro


Banco
1
0..n
Conta
0..n
1
Titular
1
1
1..n
1
1..n
1
Presidente
Diretor
Gerente

1.1 Composição

Em uma relação de **composição**, indicada pela seta com uma das pontas como um losango preenchido, a classe que está sendo apontada é uma classe "menor", e que faz parte da classe maior. Na prática, na implementação, deve ser garantido que a classe menor só será instanciada em um contexto que envolva a classe maior. Por exemplo, a classe `Curso`, só pode ser instanciada se houver um objeto da classe `Faculdade`. O `tempo de vida` de um objeto `Curso` estará completamente atrelado ao `tempo de vida` do objeto maior `Faculdade`. Ou seja, se o objeto `Faculdade` for excluído, **obrigatoriamente** todos os objetos `Curso` atrelados a ele deverão ser excluídos também.

Existem formas diferentes de implementar uma **composição**. Uma delas é instanciando o objeto menor na classe maior. Um exemplo disso pode ser visto na classe `Faculdade`. Outra forma é garantir através da implementação `regras de negócio`, o que pode incluir a manipulação de `exceções` as quais se seguidas pelos desenvolvedores fará com que o programa tenha a relação de composição na prática, ou seja, a classe menor sempre será manipulada no contexto da classe maior.

1.2 Agregação

Em uma relação de **agregação**, indicada pela seta com uma das pontas como um losango, a classe que está sendo apontada é uma classe menor, que faz parte de uma classe maior, porém é independente. Ou seja, a instanciação da classe menor não requer a instanciação da classe maior. Da mesma forma, a exclusão da classe maior não significa a exclusão da classe menor.

Por exemplo, a classe `Disciplina` tem uma relação de **agregação** com a classe `Aluno`. Os objetos da classe `Aluno` deverão fazer parte de `Disciplina`, porém serão independentes, ou seja, é possível instanciar `Aluno` sem que haja um objeto `Disciplina`. De forma similar, caso um objeto `Disciplina` seja excluído, os objetos `Aluno` relacionados a ele, não precisarão ser excluídos.

As formas de se implementar uma **agregação** é fazer com que a classe maior tenha entre seus atributos um ou mais objetos da classe menor, porém o preenchimento desse

atributo acontecerá em `tempo de execução` (*runtime*).

1.3 Cardinalidade

Nas pontas das setas são perceptíveis alguns números. Esses números indicam a `cardinalidade` da relação. Normalmente as cardinalidades são:

- 1 para 1;
- 1 para n;
- n para 1;
- n para n;
- 0,n para 1;
- 1,n para 1;
- etc.

A cardinalidade vai indicar quantos objetos de uma classe devem estar relacionadas à outra classe. Por exemplo, a classe `Faculdade` deve ter 1 `Presidente` . A classe `Disciplina` pode ter entre 1 e 2 professores. A partir disso é possível entender que uma relação pode possuir uma quantidade mínima de objetos de cada classe, e também uma quantidade máxima.

2 Implementação

Cada programador irá implementar as classes na ordem que lhe convém. Entretanto, faz mais sentido começar pelas superclasses e em seguida as subclasses.

Outra estratégia, aproveitando as ajudas provenientes de uma IDE, é declarar logo todas as classes:

```
public class nome (){

        }
```

Após declarar todas as classes, então o desenvolvedor poderá ir implementando cada uma, sucessivamente.

Outra estratégia que pode ser seguida é a separação das classes em "pacotes lógicos". Por exemplo, as classes que são subclasses de `PessoaJuridica` foram agrupadas no pacote `empresas`. Aquelas que são subclasse de `Funcionario` ficaram no pacote `funcionarios`. As classes abstratas foram agrupadas também, assim como as interfaces. Essa separação de classes entre pacotes ajuda a deixar o código mais organizado.

É importante lembrar que, uma vez que as classes estarão em pacotes diferentes, será necessário o uso dos `imports`. Por exemplo, a classe `Professor` é subclasse de `Funcionario`. Para que o compilador saiba que classe `Funcionario` é essa, foi necessário indicar onde ela está na seção de `imports`, indicando justamente o pacote `abstratas`.

A utilização de uma boa IDE para a implementação de códigos mais complexos é de fundamental importância, pois te ajudará a não deixar passar vários desses detalhes.

No pacote `app` estão duas classes principais. A primeira mostra a criação e preenchimento de alguns objetos. Você pode continuar a partir dali. A segunda classe mostra a instanciação e preenchimento dos objetos a partir de uma interação com o usuário. Acada sendo um pouco mais trabalhoso, entretanto, é um meio possível.

Por fim, devo lembrar que, na "vida real", toda seção de código repetitivo (por exemplo, os *getters* e *setters*, *formulários*, etc.) passam a ser criados automaticamente por `frameworks`, o que deixa aos desenvolvedores a necessidade de se preocuparem, o máximo possível, apenas com a criação de código novo.