Universidade Federal de Roriama
Departamento de Ciência da Computação
Professor: Filipe Dwan Pereira
Código da disciplina: DCC305
Período: 2019.2

Disclaimer

Esta aula é uma adaptação do capítulo 1 do livro:

- Phillips, Dusty. Python 3 Object-oriented Programming - Unleash the power of Python 3 objects. "Packt Publishing", 2015. Second Edition.

Nesta aula aprenderemos:

- Uma visão geral sobre os principais conceitos do paradigma de Orientação a Objetos
- A diferença entre design orientada a objetos e programação orientada a objetos;
- Os princípios básicos de design orientado a objetos;
- Um pouco sobre UML e quando usá-lo;

Aula 01 - Introdução à orientação a objetos

- Orientação a objetos é um paradigma de programação para desenvolvimento de software;
- Veja abaixo os 3 estágios desse paradigma:

1. Análise Orientada a Objetos (OOA)

- Objeto é uma coleção de **dados** com **comportamentos** associados;
- A análise orientada a objetos (OOA) é o processo de examinar um problema, sistema ou tarefa (que alguém deseja transformar em uma aplicação) e identificar os objetos e as interações entre esses objetos.
  - A saída do estágio de análise é um conjunto de requisitos.
  - A OOA transforma uma tarefa como "eu preciso de um site" em um conjunto de requisitos;
  - Veja no exemplo abaixo onde as ações estão em itálico e os objetos em negrito:
    - *revisão* de **histórico**;
    - *procurar*, *comparar*, *encomendar* e *comprar* **produtos**.
  - No desenvolvimento de software, os estágios iniciais de análise incluem entrevistar clientes, estudar seus processos e eliminar possibilidades.

2. Design Orientado a Objetos (OOD)

- Design orientado a objetos é o processo de converter esses requerimentos em especificações de implementação.

    - Especificação de implementação pode ser vista como um conjunto de classes e interfaces que poderiam ser implementadas em (idealmente) qualquer linguagem de programação orientada a objetos.
- O *designer* deve nomear os objetos, definir os comportamentos e especificar formalmente quais objetos podem ativar comportamentos específicos em outros objetos.

3. Programação Orientada a Objetos (OOP)

- Programação Orientada a Objetos é o processo de converter o OOD definido em um sistema funcional;

Sobreposição entre os estágios

- Infelizmente, normalmente encontraremos problemas que precisam de análise adicional enquanto estamos fazendo o design.
- Quando estamos programando, encontramos recursos que precisam de esclarecimento no design.
- Atualmente a maior parte do desenvolvimento acontece iterativamente.
    - No desenvolvimento iterativo, uma pequena parte da tarefa é modelada, projetada, programado
    - O programa é revisado e expandido para melhorar cada recurso e incluir novos recursos em uma série de ciclos de desenvolvimento curtos.

- A seguir veremos um pouco mais sobre OOD:

Objetos e Classes

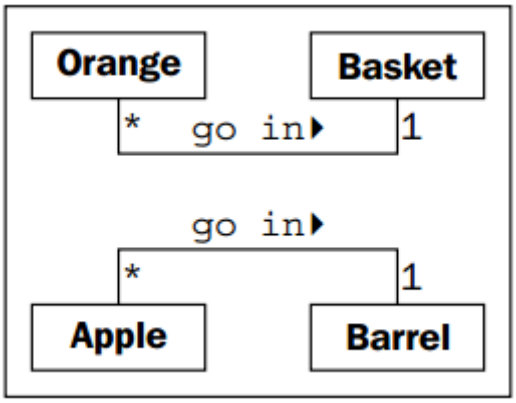
- Objeto é uma coleção de dados com comportamentos associados;
    - Mas como diferenciamos os tipos de objetos?
- Por fins didáticos, vamos fingir que estamos fazendo um aplicação de inventário para uma fazenda de frutas.
- Para facilitar o exemplo, podemos supor que as maçãs são guardadas em barris (*barrels*) e as laranjas em cestos (*basckets*).
    - Agora temos 4 **tipos** de objetos: maça, laranja, barris e cestos;
    - Em modelagem orientada a objetos chamamos os *tipos de objetos* de **classe**;
    - Assim, em termos técnicos, temos 4 classes de objetos;

Qual a diferença entre classes e objetos?

- Classes descrevem objetos;
- São como plantas para criar um objeto;
- Você pode ter três laranjas à sua frente.
- Cada laranja é um objeto distinto, mas todos os três têm os atributos e comportamentos associados a uma classe: a classe geral de laranjas.
- Em OOP é comum dizer que um objeto é uma **instância** de uma classe;

Unified Modeling Language (UML)

- A relação entre as quatro classes de objetos em nosso sistema de inventário pode ser descrita usando uma Linguagem de Modelagem Unificada;
- Abaixo veremos um exemplo de um primeiro exemplo de **diagrama de classes**:



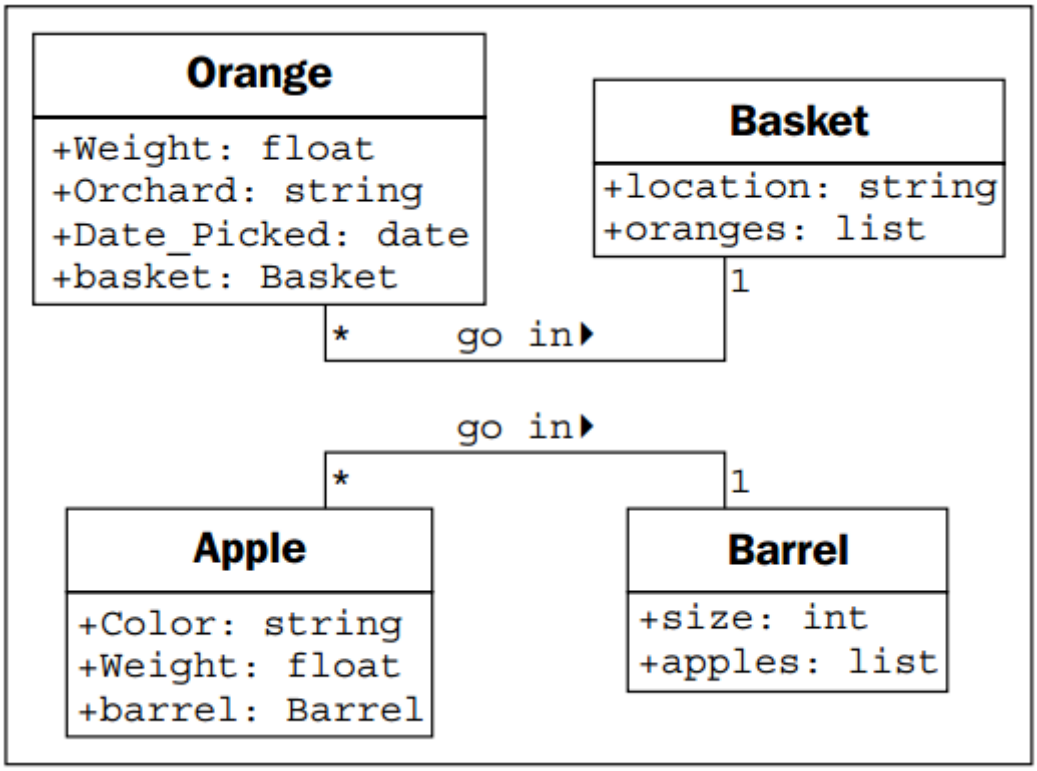
- Note que a laranja está associada ao cesto e a maçã ao barril;
- **Associação** é a forma mais básica de relação entre duas classes;
- Na figura podemos observar também a cardinalidade da relação;

Especificando Atributos e Comportamentos

- Uma classe pode definir conjuntos específicos de características que são compartilhados por todos os objetos dessa classe.
- Qualquer objeto específico pode ter valores de dados diferentes para tais características.
- Para ilustrar, três laranjas da fazenda podem ter diferentes pesos (*weight*);
  - A classe laranja pode então ter o atributo *weight*;
  - Assim, todas as instâncias da classe laranja têm um atributo weight, mas cada laranja tem um valor diferente para esse atributo.
  - No entanto, atributos não precisam ser únicos (duas laranjas quaisquer podem pesar a mesma quantidade);
  - Como um exemplo mais realista, dois objetos representando clientes diferentes podem ter o mesmo valor para o atributo *primeiro nome*;

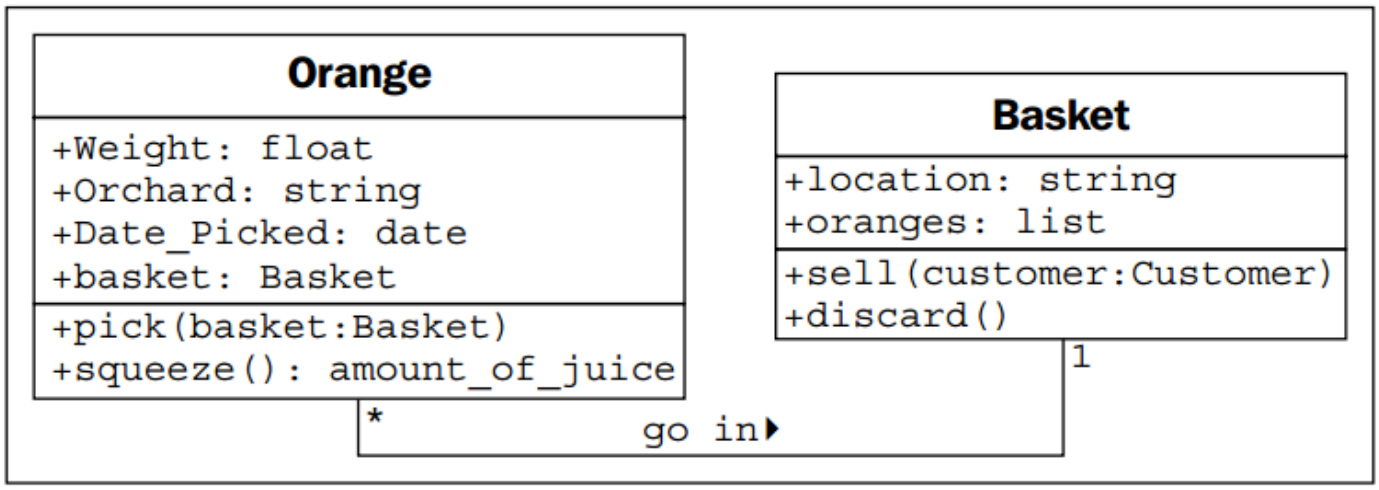
Atributos

- Atributos podem ser vistos como **o que tem** (membros ou propriedades) todos objetos de uma classe e são importantes para escopo do problema:



Comportamentos

- Comportamentos são ações que ocorrem em um objeto, isto é, **o que faz** o objeto ;
- Os comportamentos que podem ser executados em uma classe específica de objetos são chamados de **métodos**;
  - No nível de programação, os métodos são como funções na programação estruturada, com a diferença de que métods têm acesso a todos os dados associados a esse objeto.



- Note que cada objeto pode estar em um **estado** diferente, isto é, os dados em cada objeto podem ser diferentes;
- Assim, cada objeto pode reagir a um método de maneira diferente em função de seu estado;

Escondendo detalhes e criando interfaces públicas

- O objetivo principal da modelagem de um objeto no design orientado a objeto é determinar qual será a **interface** pública desse objeto;

    - A interface é a coleção de atributos e métodos que outros objetos podem usar para interagir com esse objeto;
    - Normalmente não é permitido o acesso ao funcionamento interno do objeto;
        - Um exemplo comum é a televisão.
        - Nossa interface para a televisão é o controle remoto.
    - Esse processo de esconder a implementação (detalhes de funcionamento de um objeto) é conhecido como **encapsulamento**;
        - Note que encapsulamento vai além disso. Iremos discutir isso no momento adequado;

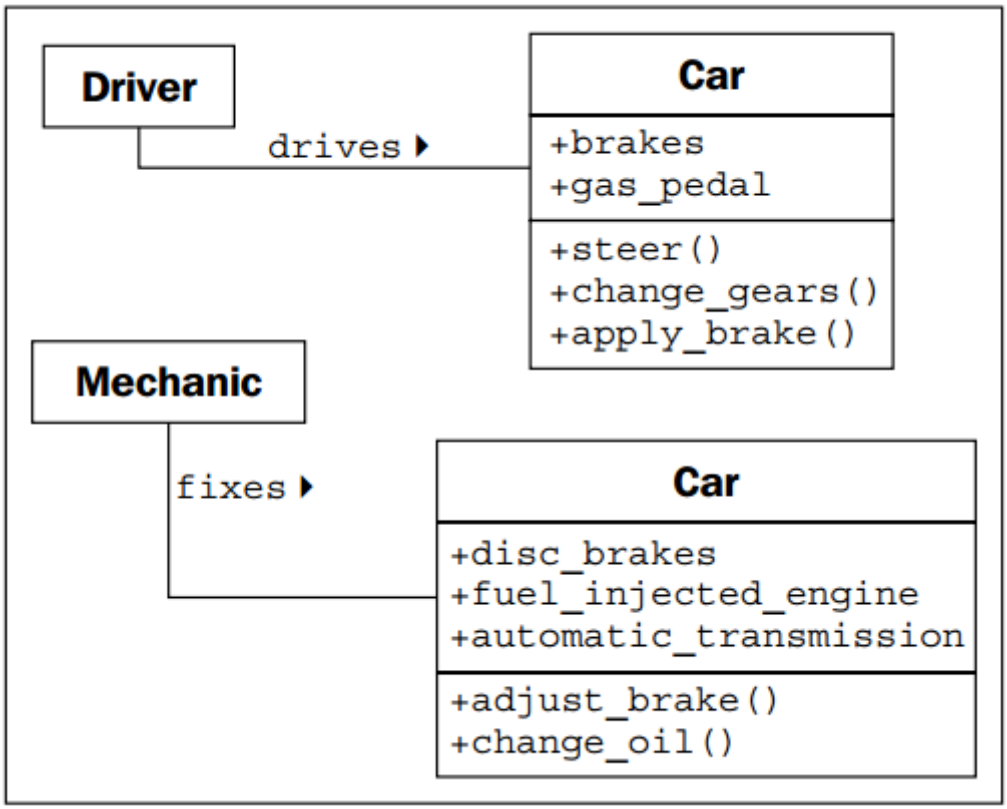
- A interface pública deve ser cuidadosamente modelada, uma vez que é difícil modificá-la no futuro;
    - Mudar uma interface vai mudar a forma como o cliente do objeto faz uso dela;
    - Note que você pode mudar o funcionamento interno sem nenhuma preocupação (o que é uma vantagem do encapsulamento);
    - É importante projetar uma interface de um objeto pensando no quão fácil será usá-la, e não quão difícil será codificá-la.

Abstração

- Programa-se objetos para que eles representem objetos reais, mas eles não são reais ;-)
    - Eles são modelos!
- Um dos pontos mais importantes em modelagem é ignorar detalhes irrelevantes;
    - O modelo é uma **abstração** do conceito real;
- Abstração é um conceito relacionado com encapsulamento.
    - Abstração significa lidar com o nível de detalhe que é mais apropriado para uma determinada tarefa;
    - Abstração é o processo de extrair a interface pública dos detalhes internos;

Exemplo de Abstração

A Figura abaixo mostra dois níveis de abstração diferentes de um carro (a abstração de um carro sob a perspectiva de um motorista e um mecânico):



- Observe que o funcionamento interno do carro não importa para o motorista, apenas a interface pública;
- Por outro lado, um mecânico trabalha em nível de abstração diferente;

Resumindo

- Abstração é o processo de encapsulamento de informação que separa interfaces públicas e privadas;
- A lição importante a ser tirada de todas essas definições é **tornar nossos modelos compreensíveis para outros objetos** que precisam interagir com eles.
  - Assegure-se de que métodos e propriedades tenham nomes sensatos.
  - Ao analisar um sistema, os objetos geralmente representam substantivos no problema original, enquanto os métodos são normalmente verbos.
  - Não tente modelar objetos ou ações que possam ser úteis no futuro.
    - Isso não quer dizer que a aplicação não possa ser alterada (pelo contrário, o app deve ser open-ended);
    - Não deixe objetos terem acesso a dados que não são necessário;
      - Imagine que objetos tem prioridade por privacidade;

Composição

- Mesmo os padrões de projeto mais complexo dependem de dois famosos princípios da OOP: composição e herança.
- Composição é o ato de coletar vários objetos juntos para criar um novo.
  - A composição geralmente é uma boa escolha quando um objeto faz parte de outro objeto.
  - Ex.: Um carro é composto de um motor, transmissão, motor de arranque, faróis, pára-brisa, entre inúmeras outras partes.
    - O motor, por sua vez, é composto de pistões, um eixo de manivela e válvulas.

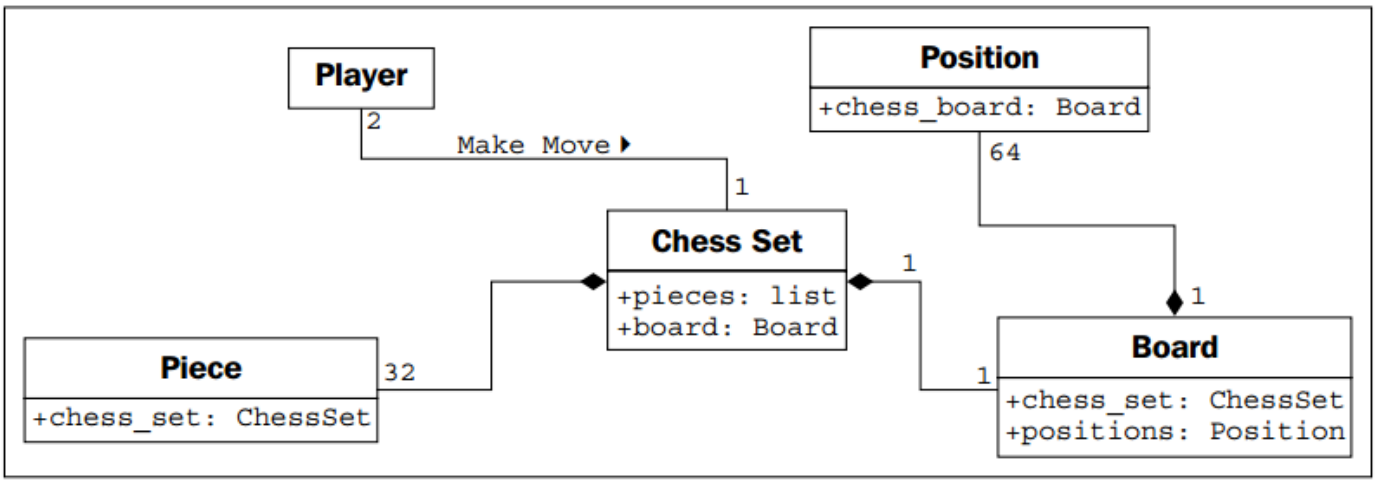
Exemplo de Composição

Pense em um jogo de xadrez, onde temos dois jogadores que jogam em um tabuleiro com peças (**Chess Set**):

- Vamos pensar somente em como seria feita uma composição no **Chess Set**;
  - tabuleiro possui 32 peças e 64 posições (uma matriz 8x8);
- Note que as peças não fazem parte do tabuleiro, o que nos leva ao conceito de **agregação**;
- Em OOP, agregação tem quase que o mesmo significado que composição;
  - A diferença é que na agregação os objetos podem existir independentemente.
- Note que seria impossível uma posição existir independentemente do tabuleiro;
  - Assim, dizemos que o tabuleiro é composto por posições;
- Mas as peças, que podem existir independentemente do tabuleiro, são vistas com uma relação de agregação com o tabuleiro;

- Outro exemplo:
  - Se o objeto composto (externo) controla quando os objetos relacionados (internos) são criados e destruídos, a composição é mais adequada.
  - Se o objeto relacionado for criado independentemente do objeto composto ou puder durar mais que esse objeto, um relacionamento agregado fará mais sentido.
  - Além disso, tenha em mente que composição é agregação;
  - Entretanto, a recíprico pode não ser verdadeira;

- A Figura abaixo mostra como poderia ficar a composição no nosso Chess Set:



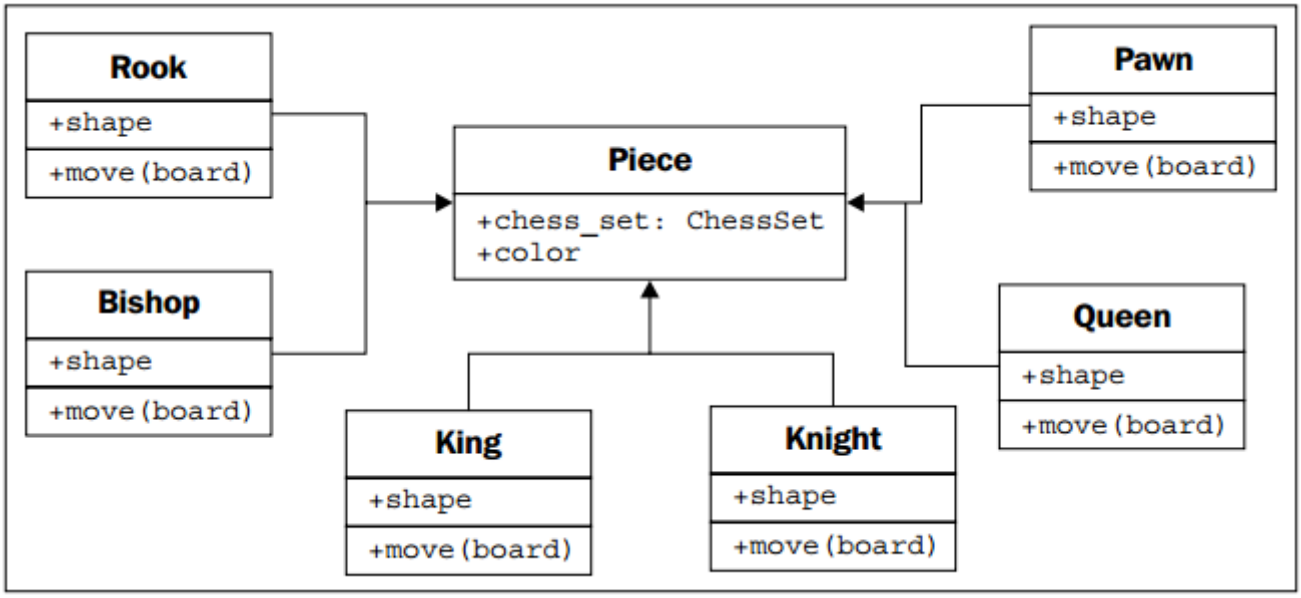
Herença

- Observe que um jogador de xadrez pode ser um humano ou uma implementação de inteligência artificial (rob);
  - Não parece sensato dizer que um hamano está associado a um jogador ou mesmo que a IA está associado a um jogador;

    - Na verdade, precisamos de um relacionamento que permita afirmar que o "Gary Kasparov" é um jogador e o robô (com a IA) é um jogador.
    - Quando existe esse relacionamento de "É UM", dizemos que existe uma relação de herança.
- A herança funciona como uma árvore genealógica.
    - Um filho herda de seu pai um sobrenome e uma carga genética;
- Em OOP, uma classe herda atributos e métodos de uma outra classe;

- Para ilustrar, existem 32 peças de xadrez no nosso *chess set*;
    - Entretanto, existem apenas 6 tipos diferentes de peças, onde cada uma se comportamenta diferentemente em relação aos seus movimentos no tabuleiro;
    - Todas as classes possuem propriedades em comum como cor, forma, qual chess set que ela faz parte e etc.

A Figura abaixo mostra como essas 6 peças herdam da classe mais genérica "Piece", onde as setas indicam a relação de herança:



- Cada subclasse possui automaticamente os atributos **chess_set** e **color**.
    - Tais atributos foram herdados da superclasse Piece.

- Observe que todas as classes possuem o atributo *shape* e o método *move*;
    - Assim sendo, hipoteticamente ambos poderiam ir para a superclasse;
    - O problema é que cada peça tem uma maneira única de movimento;
    - Assim, existe um recurso que permite que se sobrescreva (**override**) o comportamento de um método na subclasse;
- Observe ainda que se usássemos apenas herança colocando o método move, o programador seria obrigado a sobreescrever o método *move* em todas as subclasses.
    - E se o programador não sobreescrever?
        - Resposta: teríamos uma inconcistência em nosso sistema (ex.: um cavalo que não se movimenta em L ou uma torre que não se movimenta apenas horizontalmente ou verticalmente);

- Quando não queremos que as subclusses herdem um comportamento padrão, podemos especificar na superclasse que as subclasses precisarão implementar o método **move**.
    - Nesse sentido, a subclasse herda uma obrigação ao invés de um comportamento padrão.

    - Isso pode ser feito fazendo com que a classes Piece seja uma **classe abstratas** e o método **move** seja um **método abstrato**.
    - Métodos abstratos basicamente diz: "exigimos que este método exista em qualquer subclasse não abstrata, mas não especificamos uma implementação nesta classe".

Herança fornece abstração

- Um dos conceitos mais importantes em OOP é o **polimorfismo**;
- **Polimorfismo** é a habilidade de tratar diferentemente um objetos de uma superclasse, dependendo de qual subclasse ele foi implementado;
- Para ilustrar, quando é executado um movimento (método *move*) em um objeto *Piece* (uma peça) o movimento será diferente dependendo da subclasse (King, Bishop, Knight e etc), mas a chamada ao método será a mesma (*move* da superclasse *Piece*);
- No python esse conceito vai um pouco além.
    - O tabuleiro pode mover qualquer objeto que tenho o método *move* (seja ele peça, um carro ou um pato);
    - Quando o método *move* é invocado, o Bispo (*Bishop*) irá se mover em diagonal, o carro irá para algum lugar e o pato irá nadar (ou voar...);
    - Esse tipo de polimorfismo no python é tipicamente referenciado como **duck typing**:
        - "If it walks like a duck or swims like a duck, it's a duck".
        - Não é importante se ele É UM pato (herança), o que é importante é se ele é nada (swim) e anda (walk);
        - Isso facilita o processo de criação de novos tipos de pássaros sem se preocupar com a hierarquia da herança;
        - Além disso, isso permite que comportametos não planejados polo designer da classe;

Herança Múltipla

- Quando pensamos em herança, podemos pensar na nossa família;
- OOD também permite essa opção de herança múltipla, que permite uma classe herdar de múltiplas classes;
    - Na prática, essa opção pode se tornar bem complicada e algumas linguagem de programação a proibem (ex.: Java);
    - Mas, em alguns casos, a herança múltipla pode ser útil; Ex.: um Professor pode herdar de Pessoa e Funcionário;
- Um dos problemas dessa abordagem é haver métodos ou atributos iguais nas superclasses. Nesse caso, cada linguagem de programação pode tratar o problema de uma forma diferente.
    - Na prática, sugere-se fortemente evitar que isso aconteça.

Associação vs Herança

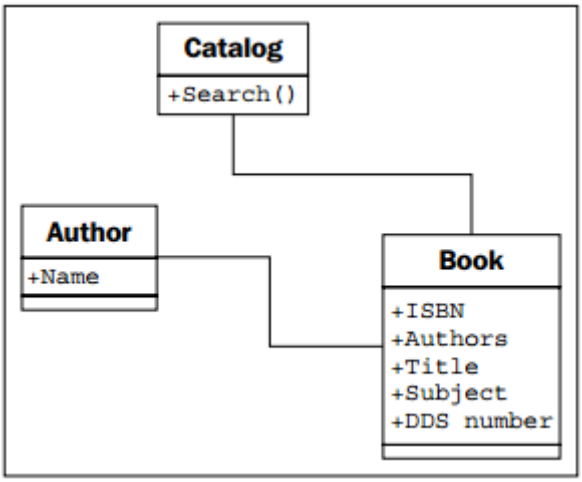
- Uma maneira simples de indentificar associação é quando usamo o termo "tem um"
- Já na herança usamos o termo "é um";
- Exemplo:
    - um carro TEM UM motor -> associação;
    - um corolla É UM carro -> herança;

Estudo de Caso

- Agora vamos modelar um caso mais prático: um sistema de catálogo de livrarias;
- Antes de falarmos com o cliente, o que sabemos sobre catálogos de livrarias:
  - Catálogos contém listas de livros;
  - Pessoas procurarm por livros de certos assuntos, títulos específicos ou por autores;
  - Livros são unicamente identificados por um *International Standard Book Number* (ISBN);
  - Cada livro tem um número de *Dewey Decimal System* (DDS) atribuído para ajudar a encontrá-lo em uma prateleira particular.

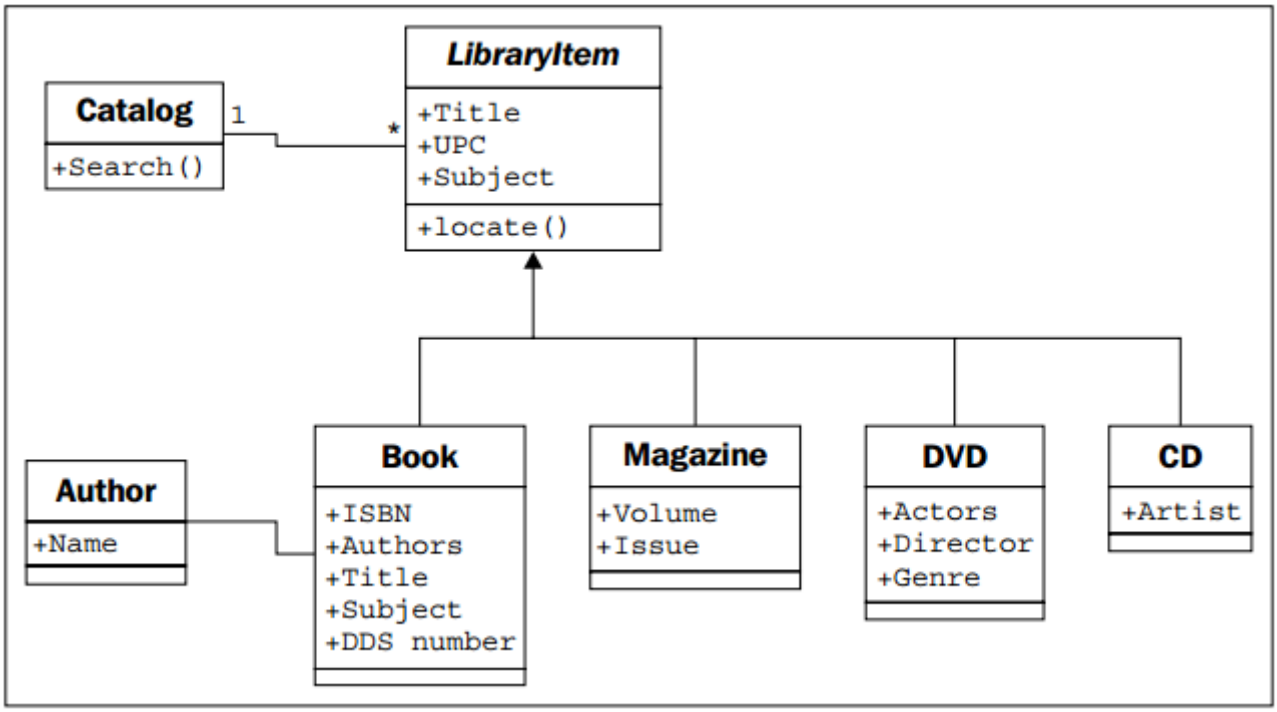
- A partir disso, podemos notar que Book será nosso principal objeto com muitos atributos como autor, título, assunto, ISBN, número DDS e catálogo;
- Observe que Autor será também um objeto, uma vez que um autor possui atributos como nome, filiação e etc.
  - A relação de livro e autor é uma associação, já que um livro TEM UM autor(es).
- Precisamos de uma classe *User*?
  - Apriori, não! Só precisamos de um catálogo e não precisamos rastrear os dados de usuários;

- Em relação ao catálogo?
  - Precisaremos de um método de busca;
    - A busca poderá ser realizada por autor, título e assunto;
  - Bem, já podemos criar um esquema para mostrá-lo ao bibliotecário:

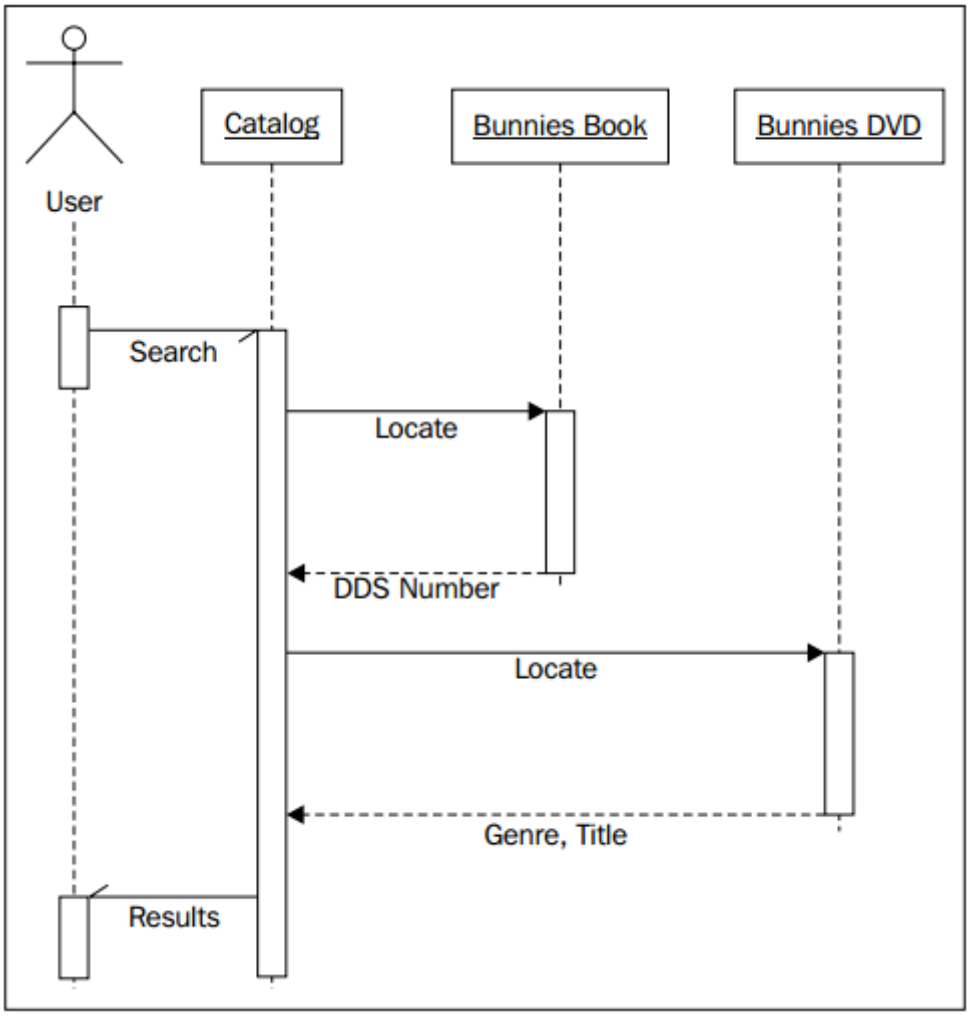


- Ao apresentar o simples esquema ao bibliotecário, este informou que livrarias não contém somentes livros, mas DVDs, revistas e CDs.
  - Nenhum desses possuem ISBN ou número DDS.
  - Como só existem poucos CDs, eles são organizados pelo último nome do autor;
  - Os DVDs são organizados por gênero e título;
  - Revistas são organizadas por título, volume e issue number;
  - Como dito anteriormente, os livros são organizados pelo DDS.

- Como todos os itens da livraria tem características em comum, podemos usar herança e definir nas subclasses as especificidades:



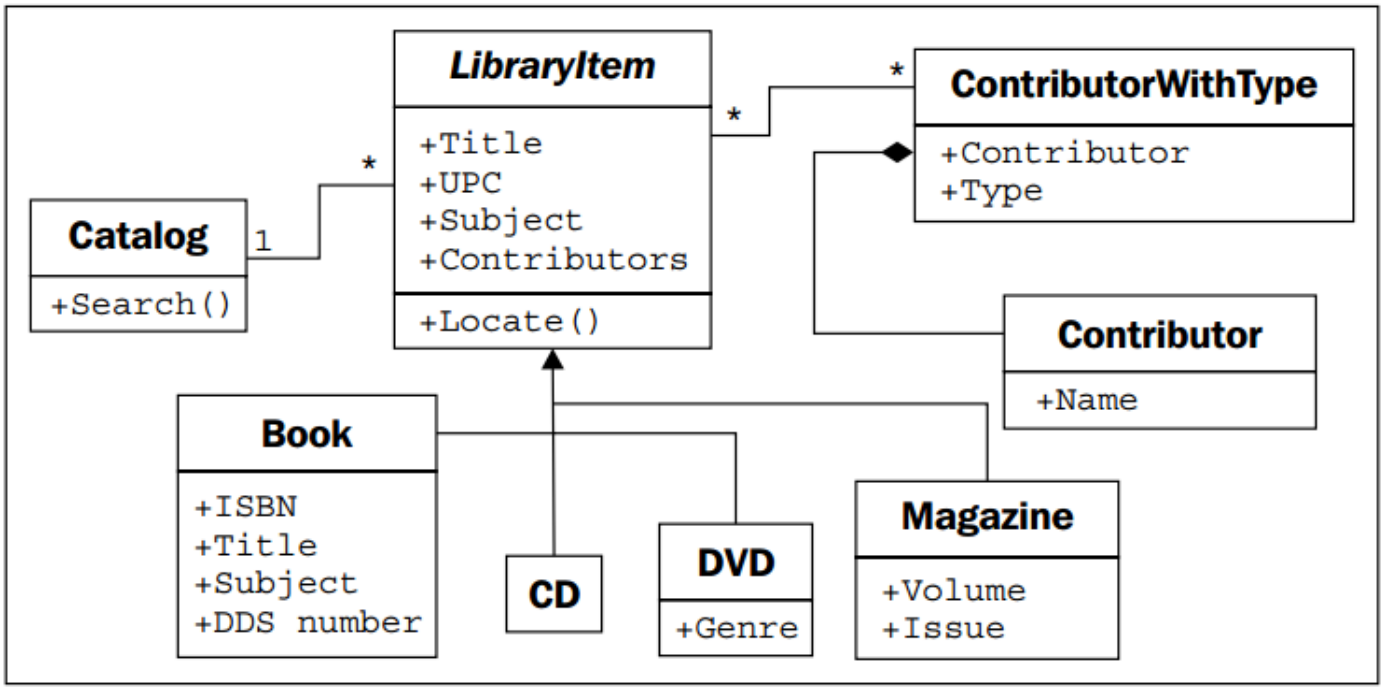
- O funcionário da livraria entendeu nosso sistema, mas ficou confuso em relação à funcionalidade do **locate**.
- Vamos explicar usando um diagrama de sequências onde um usuário procura pela palavra "bunnies". Veja abaixo como os objetos se comunicam:



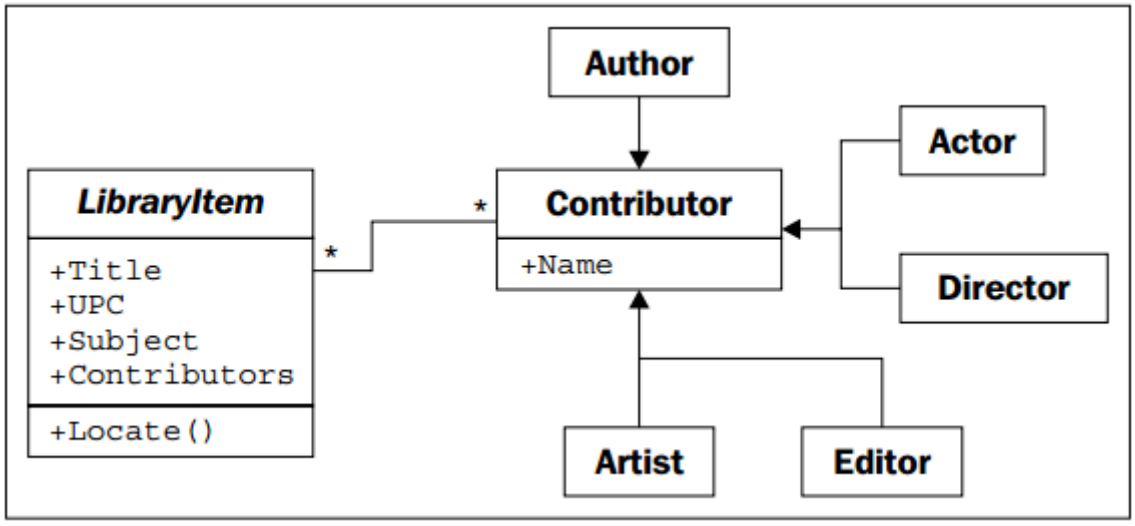
- O usuário faz uma requisição de busca ao catálogo. O catálogo procura na usa lista interna de itens e encontra um livro e um DVD onde aparece no título "bunnies". São retornodos para o usuário a localização

de ambos os itens.

- Várias pessoas podem contribuir com um item da livraria:
  - Para CDs, tem-se artistas, enquanto para livros temos os escritores responsáveis por contribuição.



Mais especificamente, ContributorWithType podem ser as várias classes de contribuidores:



Exercício

- Faça uma modelagem orientada a objetos de um sistema bancário. Idenfifique os objetos, desenhe alguns diagramas de classe ou de sequência (tente pesquisar na internet por "diagramas UML" - existem vários tutoriais).
- O trabalho pode ser feito em grupo de até 3 alunos.
- As equipes devem apresentar sua proposta de solução na próxima aula.