

PROJETO DE DADOS

PROJETO ARQUITETURAL BÁSICO

Projeto de Programas – PPR0001

Atividades Envolvidas

Preliminar

- Realizar a organização dos dados considerando a tecnologia que será utilizada em módulos (exemplo: definir as classes);
- Elencar as operações do sistema.
- Definir encapsulamento

Refinamento

- Definir relação entre os módulos (organização hierárquica);
- Definir a estrutura global do software

Modela o Aspecto **Estático** (estrutural) do sistema

Tipos de Dados

- Com o projeto de dados pronto (DER), escolhe-se os tipos de dados que serão utilizados para representar cada dado/informação do sistema.
 - ☐ Primitivos
 - ☐ Composto Nativo
 - ☐ Composto Próprio
 - Não reinventar a roda quando não é preciso!

Estruturas de Dados

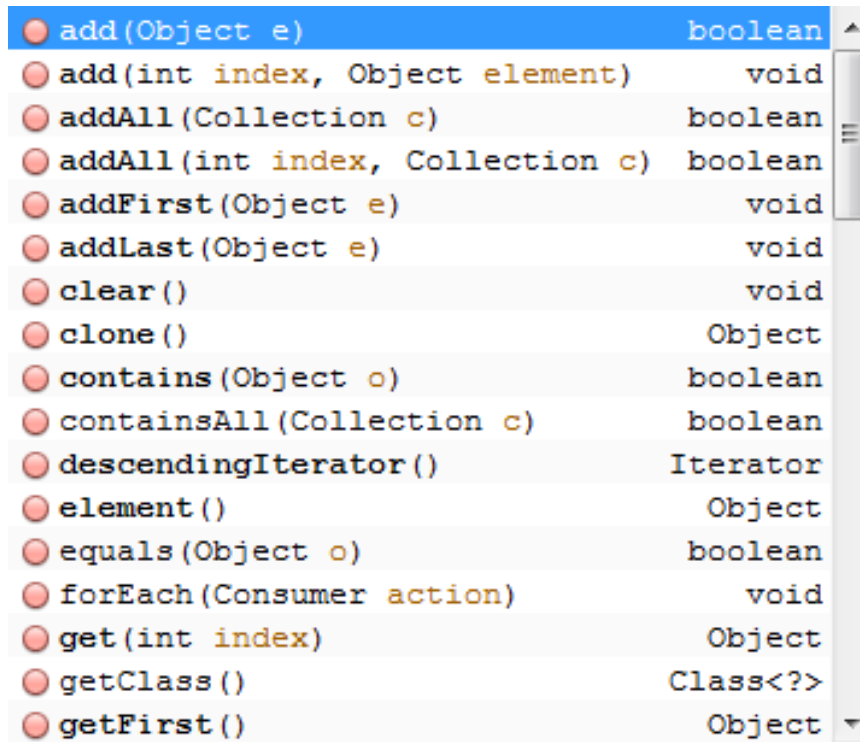
Lista Encadeada

```
LinkedList lista_1 = new LinkedList();  
LinkedList<Integer> lista_2 = new LinkedList<Integer>();  
lista_1.add( "teste" );  
lista_1.add( 23.56 );  
lista_2.add( 10 );
```

- Se nenhum tipo for especificado no “diamante” a lista poderá armazenar qualquer tipo de objeto, mas todo valor será considerado como *Object* (necessário *casting*)
- Se for especificado um tipo de dados então a lista só poderá possuir elementos daquele tipo

Estruturas de Dados

Lista Encadeada



add(Object e)	boolean
add(int index, Object element)	void
addAll(Collection c)	boolean
addAll(int index, Collection c)	boolean
addFirst(Object e)	void
addLast(Object e)	void
clear()	void
clone()	Object
contains(Object o)	boolean
containsAll(Collection c)	boolean
descendingIterator()	Iterator
element()	Object
equals(Object o)	boolean
forEach(Consumer action)	void
get(int index)	Object
getClass()	Class<?>
getFirst()	Object

Métodos Principais:

- add => adiciona elemento
- addFirst => adiciona no inicio
- addLast => adiciona no fim
- clear => limpa a lista
- contains => verifica existência
- get => pegar um elemento
- pop => remove elemento
- remove => remove elemento
- size => tamanho da lista
- sort => ordena lista

Estruturas de Dados

ArrayList

```
ArrayList lista_3 = new ArrayList();  
ArrayList<Double> lista_4 = new ArrayList<Double>();  
lista_3.add( lista_4 );
```

- ArrayLists possuem os mesmos métodos das listas encadeadas. O que se altera é o seu funcionamento interno. Entretanto, para esta disciplina não vamos nos importar com isso.

Tipos de Dados

Árvores / Conjunto - Set

```
Set<Integer> arvore = new TreeSet();  
arvore.add( 5 );  
boolean eh_elemento = arvore.contains( 2 );
```

- Árvores geralmente são úteis quando deseja-se maior desempenho em determinadas funções;
- Árvores são implementadas como conjuntos, a princípio; logo, não possuem elementos repetidos.

Estruturas de Dados

Árvores / Conjunto - Set

add(Integer e)	boolean
addAll(Collection<? extends Integer> c)	boolean
clear()	void
contains(Object o)	boolean
containsAll(Collection<?> c)	boolean
equals(Object o)	boolean
forEach(Consumer<? super Integer> action)	void
getClass()	Class<?>
hashCode()	int
isEmpty()	boolean
iterator()	Iterator<Integer>
notify()	void
notifyAll()	void
parallelStream()	Stream<Integer>
remove(Object o)	boolean
removeAll(Collection<?> c)	boolean
removeIf(Predicate<? super Integer> filter)	boolean

Métodos Principais:

- add => adiciona elemento
- clear => limpa a lista
- contains => verifica existência
- remove => remove elemento
- size => tamanho da lista

Estruturas de Dados

Como Iterar pelos elementos?

```
Iterator it = arvore.iterator();  
while( it.hasNext() ){  
    System.out.println( it.next() );  
}
```

Método 1 – Usando um *Iterator*

```
for( Integer a : arvore ){  
    System.out.println(a);  
}
```

Método 2 – utilizando um *for*

```
Consumer<Integer> cons = (Integer v) -> System.out.println(v);  
arvore.forEach( cons );
```

Método 3 – utilizando um “foreach” com *Consumer* (Java 8)

MODELAGEM DO PROJETO ARQUITETURAL

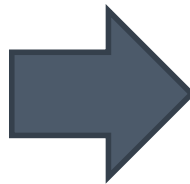
Classes, Atributos e Métodos

- Queremos modelar as casas abaixo para um sistema.
 - Quais atributos e métodos são interessantes? (fazem parte do escopo)



Classes, Atributos e Métodos

- Queremos modelar as casas abaixo para um sistema.
 - Quais atributos e métodos são interessantes? (fazem parte do escopo)



```
int numero;  
Color cor;  
boolean porta_fechada;  
static String construtora;  
  
boolean abrirPorta();  
boolean fecharPorta();  
void pintar( Color );
```

Classes, Atributos e Métodos

- Para representar uma classe utilizamos um **Diagrama de Classe**
 - A ferramenta **ASTAH Community** permite a geração destes diagramas;
- Os principais elementos de uma classe são:

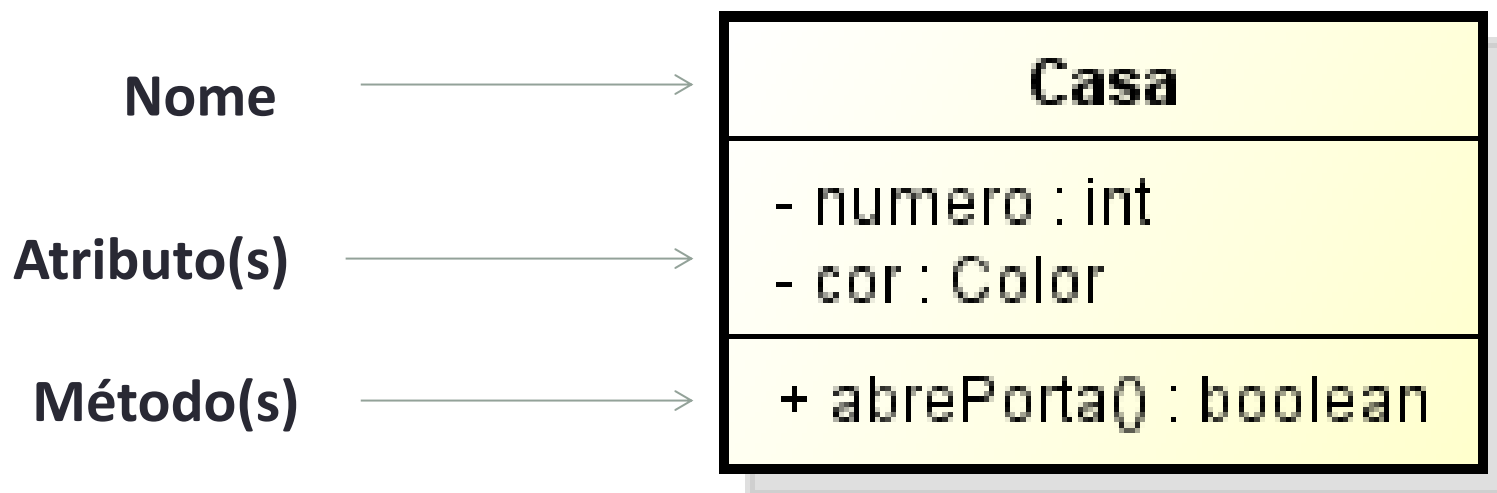


Diagrama de Classes

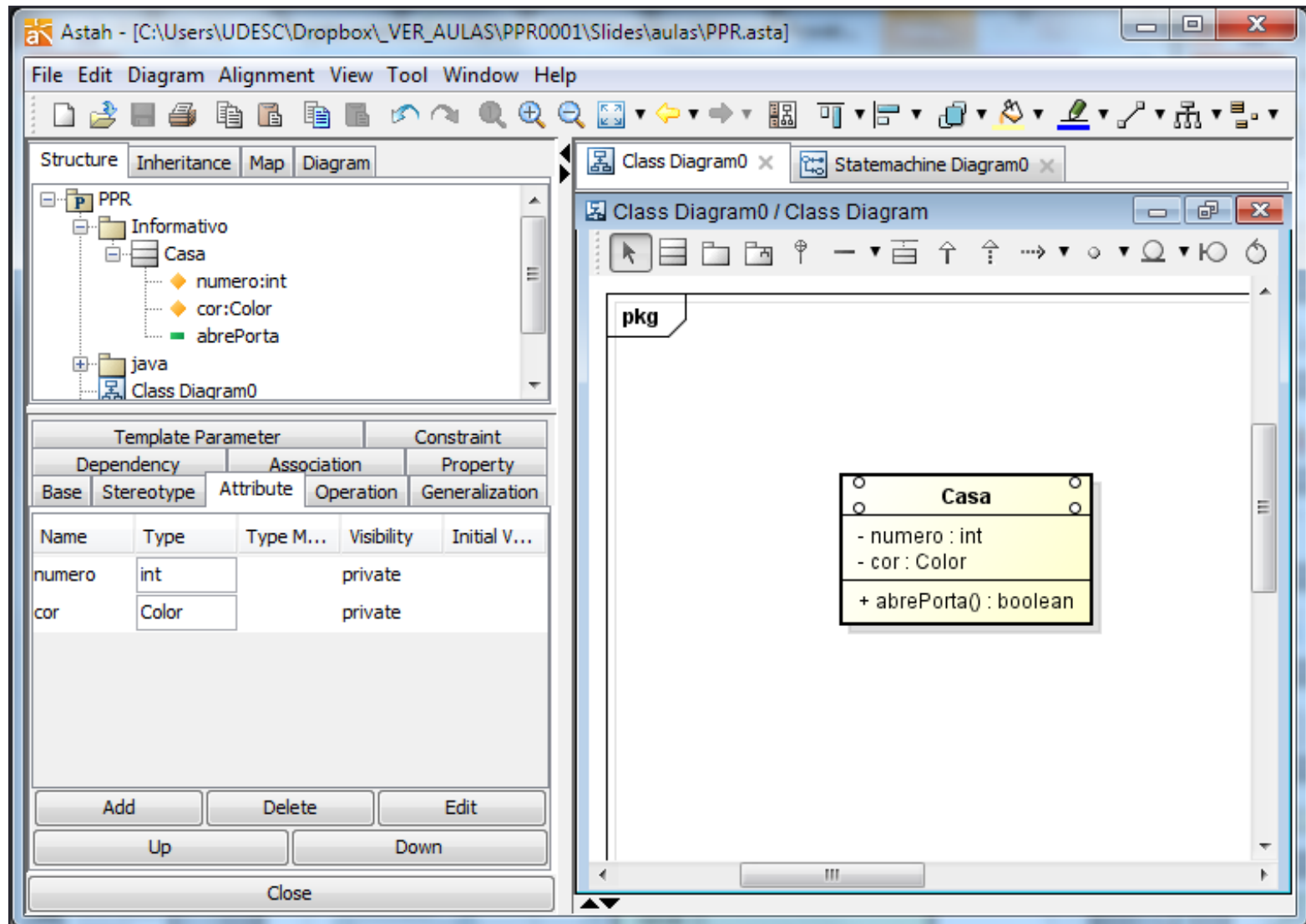


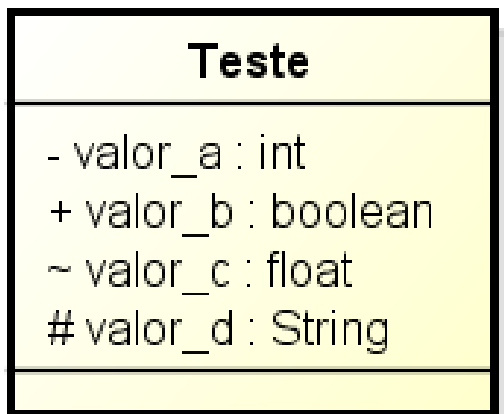
Diagrama de Classes

- Elementos importantes dos **Atributos**:

[Encapsulamento] [Nome] : [Tipo]

- Elementos importantes dos **Métodos**:

[Encapsulamento] [Nome] ({[tipos dos parâmetros]}) : [Tipo Retorno]



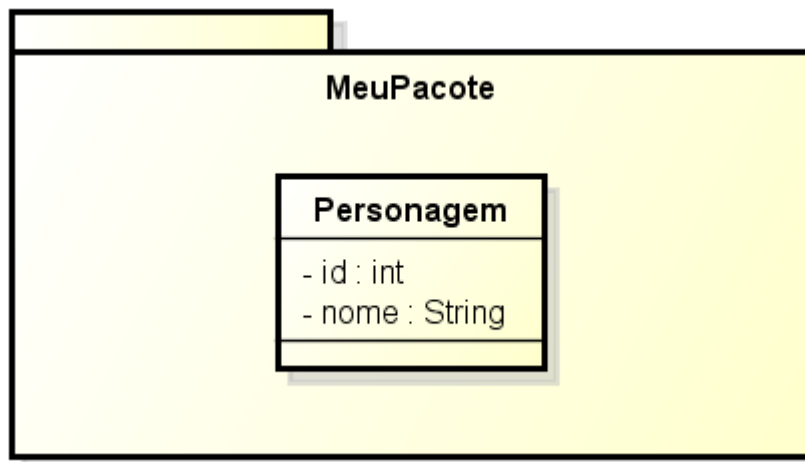
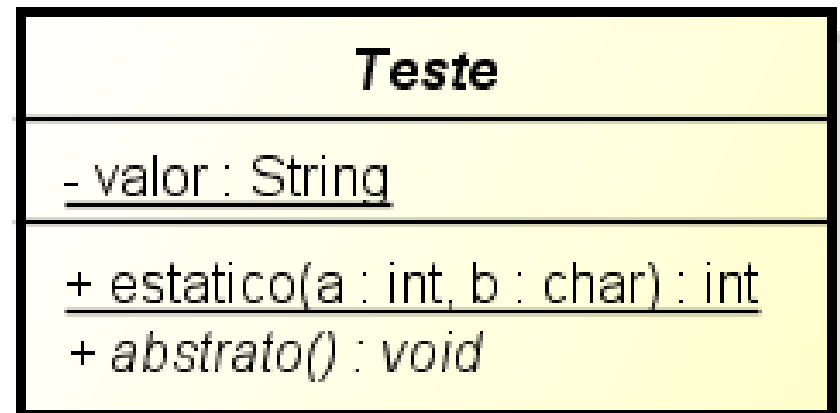
➤ Encapsulamento ou visibilidade:

- *private* : somente a classe tem acesso
- + *public* : todas as classes tem acesso
- ~ *package (default)*: classes do mesmo pacote tem acesso
- # *protected* : igual ao package + acesso por herança (classes que estendem)

Diagrama de Classes

➤ Outras propriedades:

- ❖ Classes Abstratas
- ❖ Atributo estático
- ❖ Método estáticos
- ❖ Métodos abstratos



➤ Agrupamentos:

- ❖ Pacotes

Na próxima aula vamos trabalhar mais a com a questão do agrupamento

Exemplo

Vamos escrever algumas classes para o sistema acadêmico:

❑ CURSO:

- Atributos: id, nome;
- Métodos: nenhum;

❑ DISCIPLINA:

- Atributos: id, curso, sigla, ch;
- Métodos: nenhum;

❑ ACADEMICO:

- Atributos: matricula, nome, curso, endereço, telefone, senha;
- Métodos: alterarEndereco(), verificarSenha();

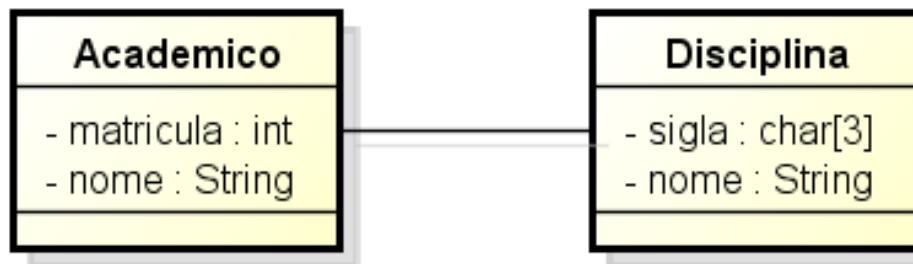
❑ SISTEMA:

- Atributos: nenhum;
- Métodos: matricular(), lancarMedia();

Relacionamentos entre Classes

Associação

- Representa uma relação de referência entre duas classes.
 - Uma classe A mantém uma referência para uma classe B



ex1 – um acadêmico foi aprovado em uma ou várias disciplinas

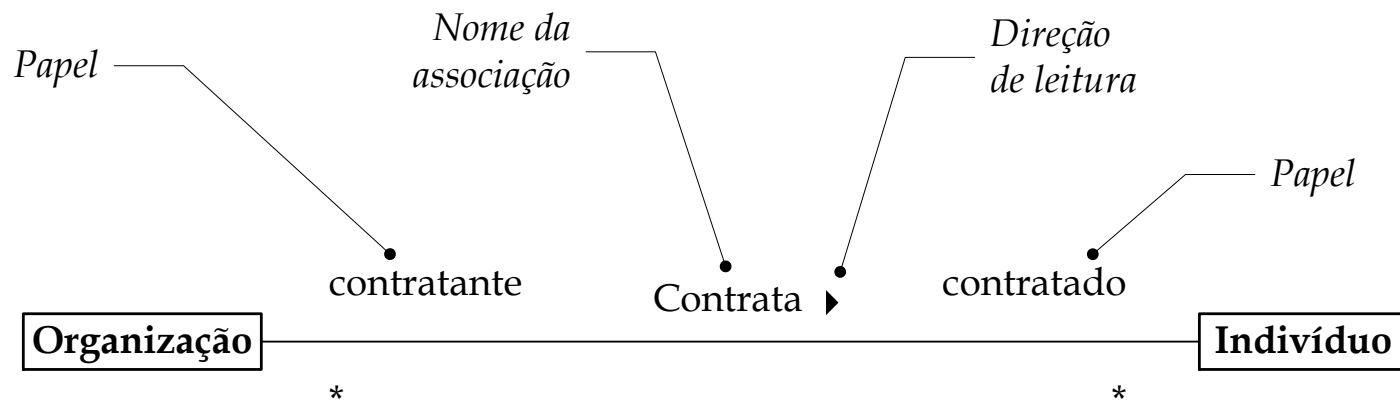
- Nome, direção e multiplicidade:



Relacionamentos entre Classes

Associação

- Nome, direção e multiplicidade:



- Multiplicidade:

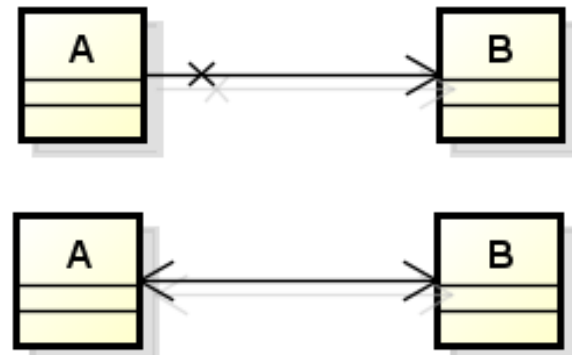
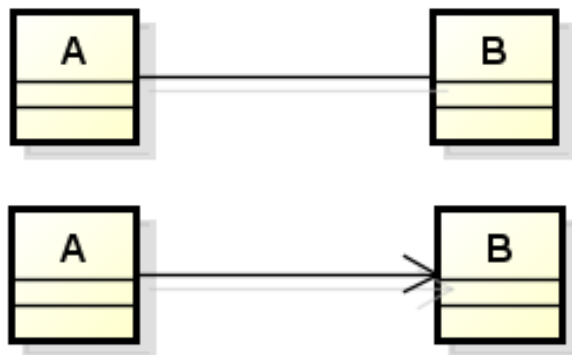
- **Obrigatória:** multiplicidade ≥ 1
- **Opcional:** multiplicidade pode ser 0

Nome	Simbologia
Apenas Um	1..1 (ou 1)
Zero ou Muitos	0..* (ou *)
Um ou Muitos	1..*
Zero ou Um	0..1
Intervalo Específico	$l_i..l_s$

Relacionamentos entre Classes

Associação

- Navegabilidade: indica para qual lado ocorre a referência
 - **Não especificada (sem seta)**: utilizada para representar relações não fortes e sem restrição de desempenho (ex: representar objeto pelo id);
 - **Associação navegável (seta unidirecional/bidirecional)**: representa que uma classe possui referência da outra classe e restringe acesso eficiente (ex: uso de referência)
 - **Associação não navegável (x)**: representa que um lado não é acessível através de outra classe.



Relacionamentos entre Classes

Associação

ex2 – um *Player* obrigatoriamente possui um *Asset*
 (OBS: um *Asset* pode ser comprado por apenas um player)

```
class Asset {...}
```

```
class Player{
    Asset asset;
    public Player (Asset purchased_asset) { ... }
}
```

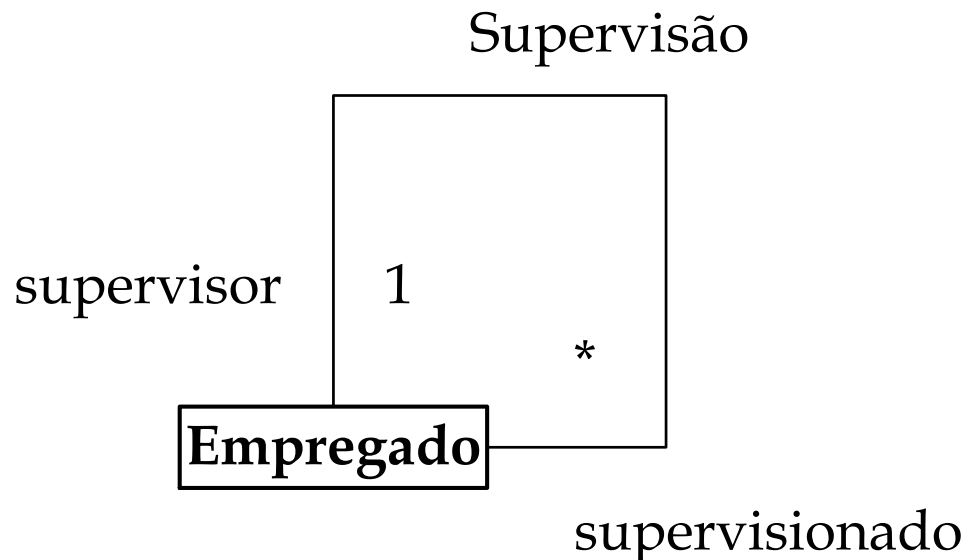


Q: Como ficaria a relação entre acadêmico e disciplina para cada caso?

Relacionamentos entre Classes

Associação – Associações Reflexivas ou auto associações

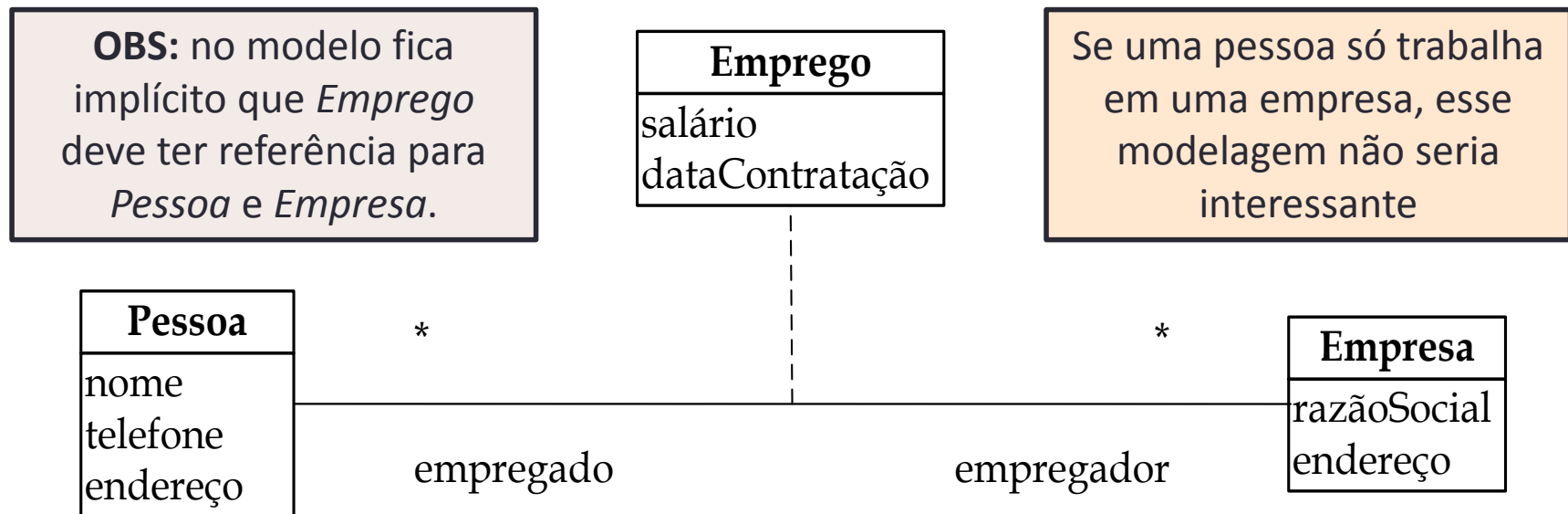
- Quando uma classe se relaciona consigo mesma.
- Indica que um objeto da classe pode/deve se relacionar com outro objeto da mesma classe com papéis distintos.



Relacionamentos entre Classes

Classe associativa

- É uma classe que está ligada a uma associação
- Normalmente necessário quando dados de uma associação entre duas classes precisam ser usados.



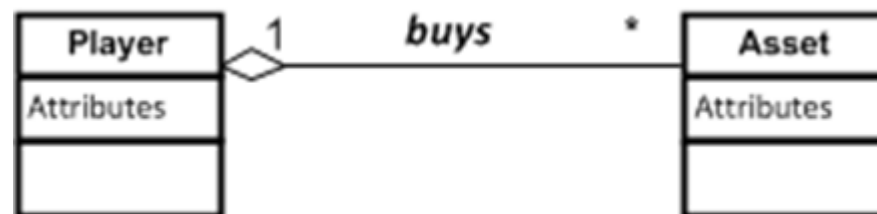
Relacionamentos entre Classes

Agregação

- Representa uma associação fraca de maior multiplicidade : *um-para-muitos, muitos-para-muitos, parte-todo*
- Semelhante a uma associação e considerado redundante
- Indica que um ou vários objetos associados podem ser agregados a qualquer momento

```
class Asset {...}
```

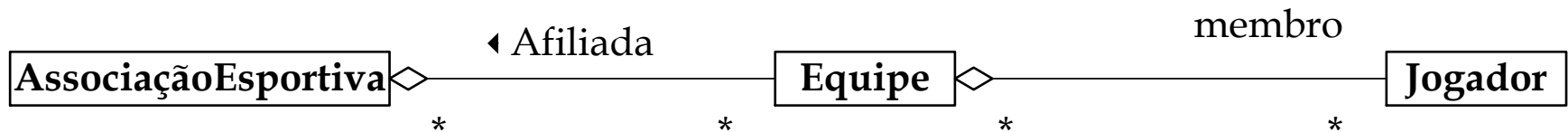
```
class Player {  
    List assets;  
    public void addAsset( Asset new_asset ) {  
        assets.add(new_asset);  
    }  
}
```



Relacionamentos entre Classes

Agregação

(exemplo adicional)



(instância de jogador pode existir mesmo sem uma equipe)

Relacionamentos entre Classes

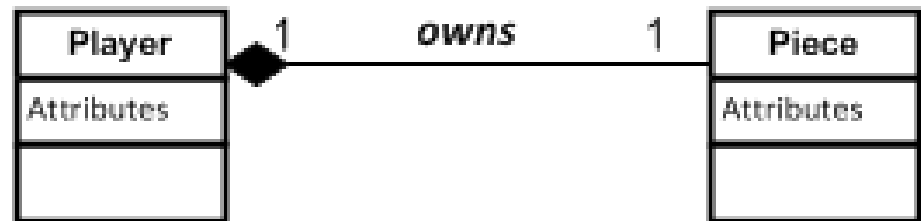
Composição

- Representa uma associação forte com responsabilidade de instanciação de criação:

classe A, que é composta da classe B, é responsável pelo controle do “tempo de vida” do(s) seu(s) objeto(s) da classe B

```
public class Piece {...}
```

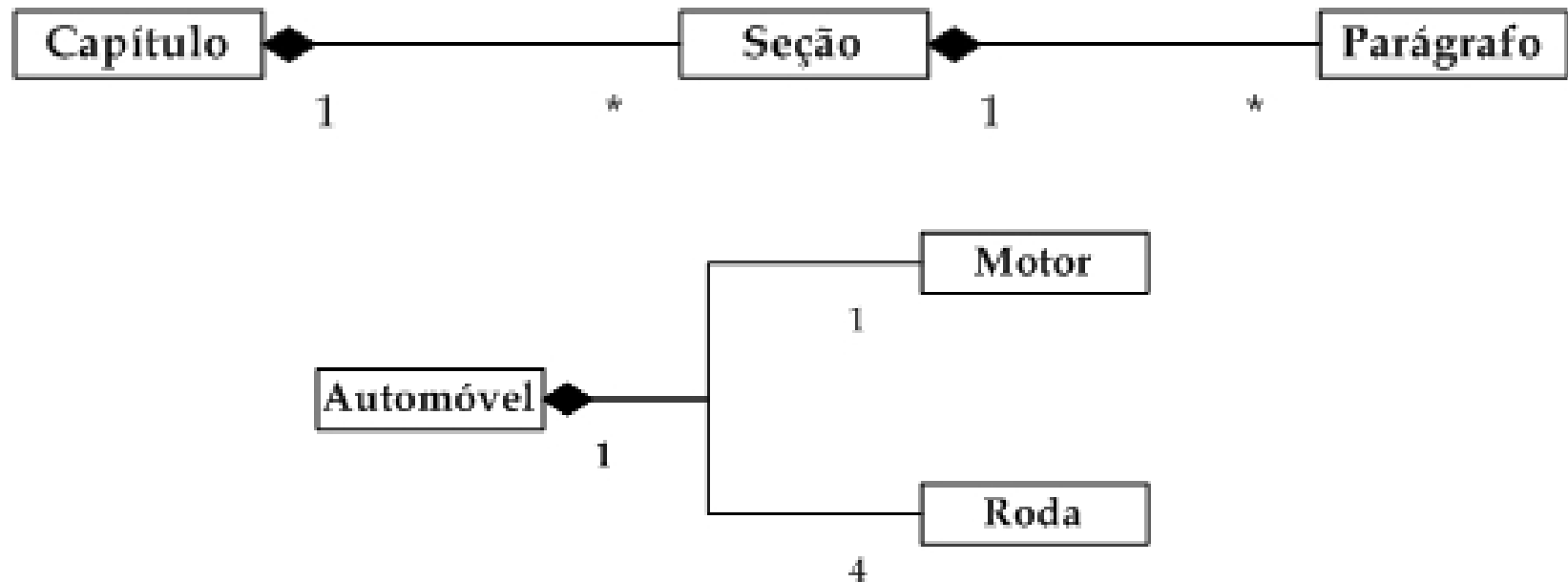
```
public class Player {  
    Piece piece;  
    public Player() {  
        piece = new Piece();  
    }  
}
```



Relacionamentos entre Classes

Composição

(exemplos adicionais)



Relacionamentos entre Classes

[Agregação] vs Composição

```
final class Car {  
    private Engine motor;  
  
    void setEngine(Engine motor) {  
        this.motor = motor;  
    }  
  
    void move() {  
        if (motor != null) {  
            motor.work();  
        }  
    }  
}
```

Relacionamentos entre Classes

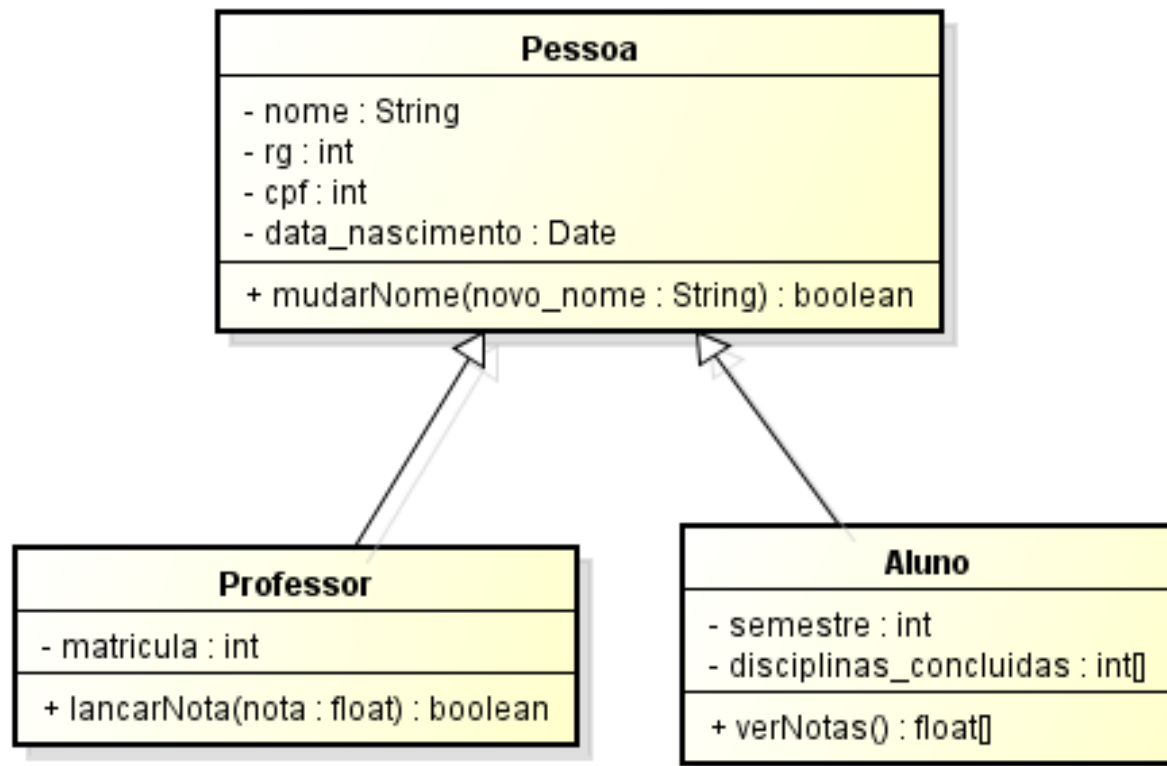
Agregação vs [Composição]

```
final class Car {  
    private final Engine motor;  
    Car(EngineSpecs specs) {  
        motor = new Engine(specs);  
    }  
  
    void move() {  
        motor.work();  
    }  
}
```

Relacionamentos entre Classes

Herança ou Generalização

- Representa que uma classe herda todos os atributos e métodos de outra classe (extends)



Relacionamentos entre Classes

Dependência

- Indica que uma classe pode vir a utilizar um método ou variável de outra classe, da qual se torna dependente
- Deixa evidente que uma modificação na classe de dependência pode gerar problemas na classe dependente

```
class Die{  
    public void rool() { ... }  
}
```

```
class Player{  
    public void takeTurn(Die die) {  
        die.roll();  
    }  
}
```



Atividade

Agora é a sua vez!

Realize a identificação das classes e das suas associações para o sistema descrito no documento que está disponível na página da disciplina.

Bibliografia

- **Básica:**

BEZERRA, E. Princípios de Análise e Projetos de Sistemas com UML. Rio de Janeiro: Campus, 2003.

PRESSMAN, R.S. Engenharia de Software. São Paulo: Makron Books, 2002.

SOMMERVILLE, I. Engenharia de Software. São Paulo: Addison Wesley, 2003.

- **Complementar:**

WARNIER, J. Lógica de Construção de Programas. Rio de Janeiro: Campus, 1984.

JACKSON, M. Princípios de Projeto de Programas. Rio de Janeiro: Campus, 1988.

PAGE-JONES, M. Projeto Estruturado de Sistemas. São Paulo: McGraw-Hill, 1988.

Association vs. Dependency vs. Aggregation vs. Composition. Disponível em:

<https://nirajrules.wordpress.com/2011/07/15/association-vs-dependency-vs-aggregation-vs-composition/>