

# **Técnicas Avançadas em Orientação a Objetos**

## **aula 09**

Jobson Luiz Massollar  
[jobson.luiz@gmail.com](mailto:jobson.luiz@gmail.com)

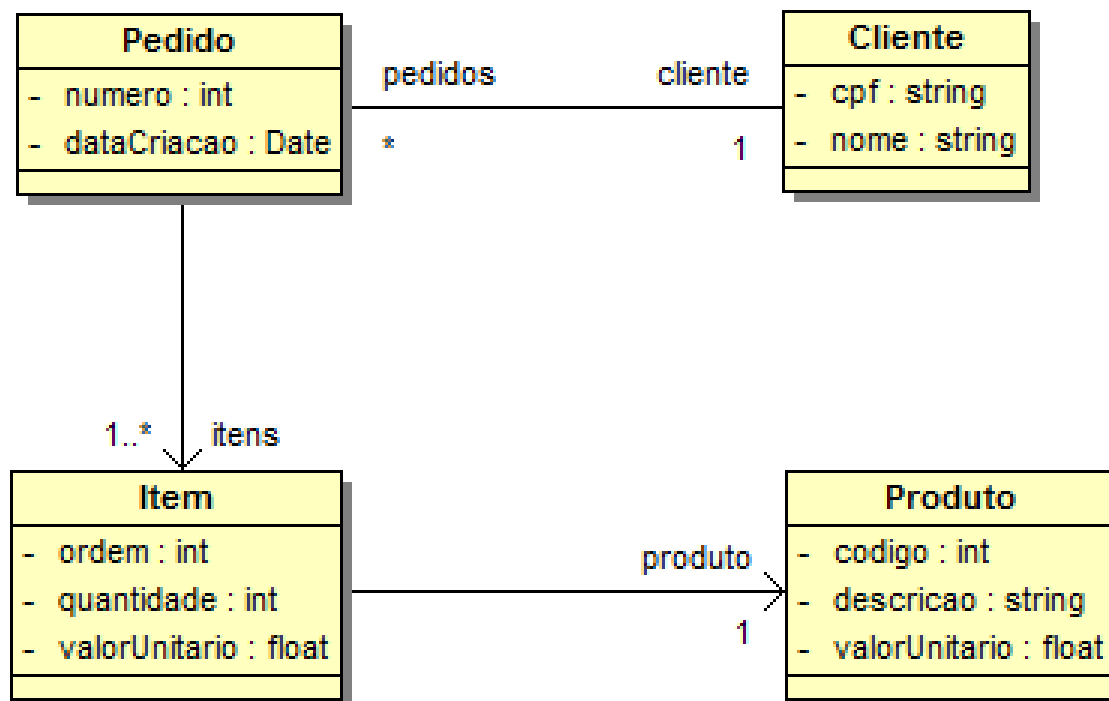
- Em projetos OO com alguma complexidade, é difícil enxergar as diversas classes que serão necessárias para criar uma solução computacional.
- Conforme visto anteriormente é preciso dividir o problema em várias classes com responsabilidades claras e objetivas, com o cuidado para não gerar uma classe "faz tudo".
- Essa divisão do problema leva, geralmente, à necessidade de criação de um grande número de classes.

- Alguns tipos de classes que podem ser identificadas em um problema:
  - **Classes de Domínio:** modelam objetos associados ao domínio do problema.
    - aluno, curso, turma, disciplina, docente, etc.
    - médico, paciente, exame, diagnóstico, etc.
    - advogado, juiz, processo, julgamento, etc.
  - **Classes de Fronteira:** modelam objetos que realizam a comunicação entre o sistema e a sua vizinhança, ou seja, os elementos externos.
    - formulário, janela, etc.
    - catraca, sensor de presença, etc.
    - sistema de cobrança, administradora de cartão, etc.

- Alguns tipos de classes que podem ser identificadas em um problema:
  - **Classes de Controle**: modelam objetos que exercem controle sobre outros objetos.
    - criação de objetos
    - controle de concorrência
  - **Classes Utilitárias**: modelam comportamentos que representam algoritmos de uso comum.
    - verificação de CPF e CNPJ
    - ordenação de listas
    - formatação de dados para geração de relatórios, etc.
  - **Classes de Exceção**: modelam objetos que representam uma exceção ou falha em um determinado domínio.

- Como devemos, então, atacar o problema ? Por onde começamos ?
- Em projetos OO um pouco mais complexos devemos lançar mão de modelos que nos permitam desenhar a solução, ou boa parte dela, antes de iniciarmos a programação propriamente dita.
- Essa modelagem é iniciada por modelos que chamamos de **modelos conceituais** ou **modelos de domínio**.
- Conforme o projeto avança da fase de análise para a fase de design, precisamos transformar o nosso modelo conceitual em modelo de design.
- O objetivo dessa transformação é acrescentar aos nossos modelos características que vão nos permitir chegar a uma solução computacional.

➤ Observe o modelo conceitual abaixo:



- A partir desse modelo podemos dizer que:
  1. A partir de um cliente podemos obter todos os seus pedidos (pode ser zero ou mais)
  2. A partir de um pedido podemos saber qual cliente o realizou (obrigatoriamente 1)
  3. A partir de um pedido podemos saber que itens fazem parte do pedido (pelo menos 1 item)
  4. A partir do item podemos saber que produto está sendo comercializado (obrigatoriamente 1)
  5. A partir de um produto, NÃO conseguimos saber em que itens de quais pedidos eles foram comercializados
  6. A partir de um item NÃO conseguimos saber a qual pedido ele está associado
  
- Essas são as regras de **navegação** estabelecidas no modelo conceitual ou modelo de domínio.

- Uma das primeiras decisões que tomamos na passagem de modelos conceituais para modelos de design está relacionada ao tratamento das relações **1..\*** ou **0..\***.
- Em projetos OO, relações **1..\*** e **0..\*** são tratadas usando listas ou outro tipo adequado de coleção:

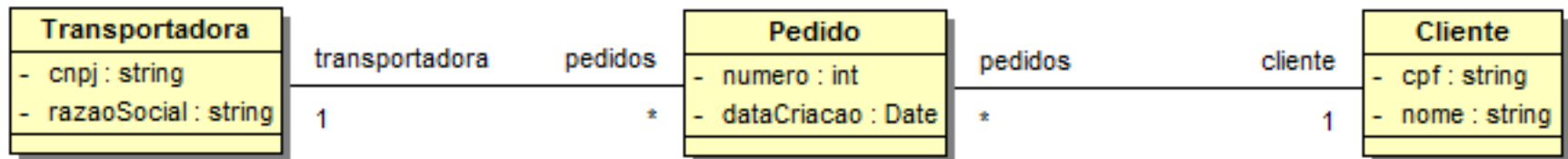
```
public class Pedido {  
    private ArrayList<Item> itens;  
}
```

Mas, por que isso não funciona (ou pelo menos, não é o mais adequado) ?

Vamos ver outro exemplo...



- Acrescentando a **Transportadora** no modelo anterior, imagine que precisamos obter o somatório dos pedidos transportados por uma transportadora e o somatório dos pedidos feitos por um cliente:



- Então devemos:
  1. Criar as listas de Pedidos em Cliente e Transportadora
  2. Implementar métodos para realizar o somatório


- Com um projeto que trata a lista de Pedidos internamente em cada classe que a referencia teremos:

```
public class Transportadora {
    private ArrayList<Pedido> pedidos;

    public double getValorTotalPedidos() {...}
}

public class Cliente {
    private ArrayList<Pedido> pedidos;

    public double getValorTotalPedidos() {...}
}
```



E agora, por que não funciona (ou pelo menos, não é o mais adequado) ? Onde está o problema ?

- A resposta é simples:
- Listas devem ser tratadas como **elementos de primeira categoria** no projeto porque elas também podem possuir **atributos** e **comportamentos**.
- Ou seja, podem existir atributos ou comportamentos que são inerentes a uma **coleção de elementos** e não a um elemento individualmente.

➤ Exemplos:

1. Qual o volume médio transportado por uma transportadora?
2. Qual o volume total transportado por uma transportadora em determinado período?
3. Qual o valor total de compras de um cliente por mês?
4. Qual a nota média dos alunos de um turma?
5. Qual a nota média dos alunos de uma disciplina?

➤ Assim, as nossas coleções também devem ser implementadas como **classes** !

Como ?

➤ Resposta: usando herança !

```
public class PedidoList extends ArrayList<Pedido> {  
    public double getValorTotal() {...}  
}
```

```
public class Transportadora {  
    private PedidoList pedidos;  
  
}
```

```
public class Cliente {  
    private PedidoList pedidos;  
  
}
```

Os atributos e métodos que dizem respeito à lista de Pedidos ficam implementados aqui !

- Essa mesma regra pode ser observada em todas as relações  $1..*$  e  $0..*$  existentes no nosso modelo conceitual.

- Assim, podemos definir a seguintes regra:

*R1 - sempre que houver uma relação  $0..*$  ou  $1..*$ , devemos transformá-la em uma relação 1 com uma lista do mesmo tipo do elemento de domínio destino da relação.*

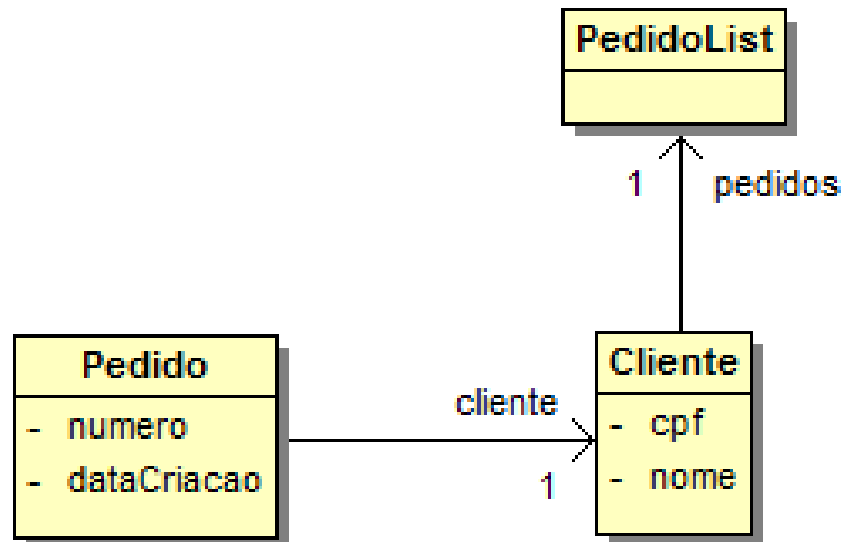
- Outro ponto importante: escolha um padrão de nomenclatura para as suas classes de lista:
  - ListaPedidos, ListaItem, etc.
  - PedidoList, ItemList, etc.
  - PedidoCollection, ItemCollection, etc.
  - ...

- Aplicando a regra R1 no nosso modelo, teremos:

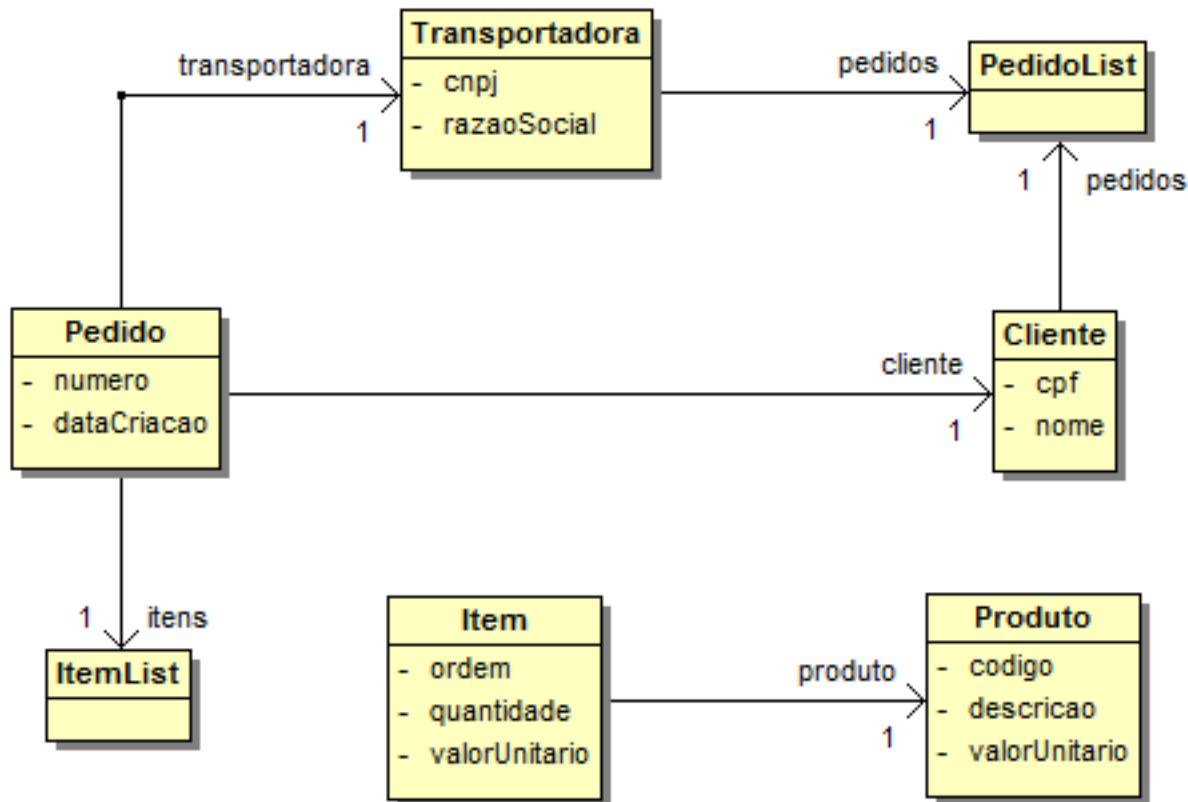
Repare que a relação de Pedido com Cliente foi quebrada em duas relações direcionadas.

A regra R1 afeta somente a navegação Cliente→Pedido

Foi preservada a navegação Pedido→Cliente do modelo original, pois não se enquadra na R1.

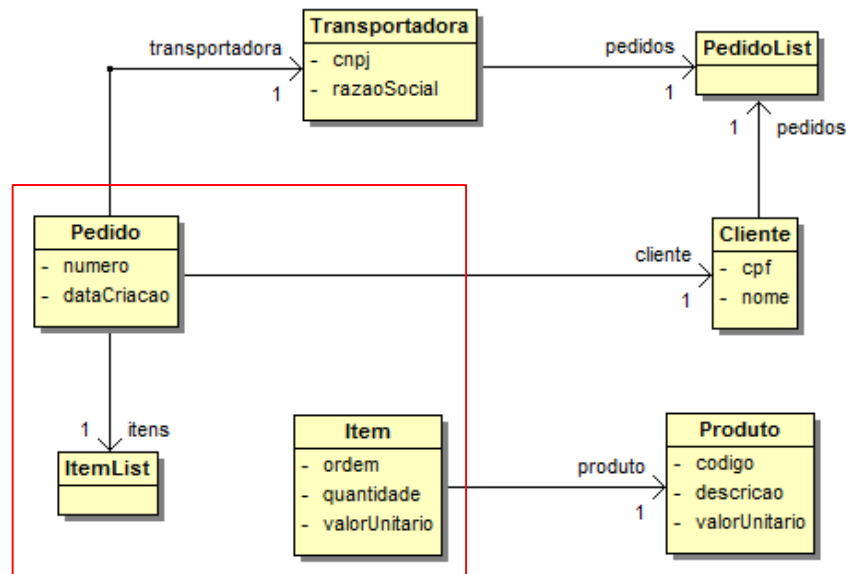


- Aplicando a regra R1 sobre todos os modelos obtemos o modelo abaixo. Repare que todas as navegações são **direcionadas** e tem multiplicidade **1**.





- O modelo de projeto OO obtido pode parecer estranho em um primeiro momento, pois o relacionamento entre Pedido e Item simplesmente desapareceu.
- Entretanto, não podemos esquecer que ItemList é uma coleção de elementos do tipo Item ou seja, em termos de projeto o Pedido referencia **UMA** coleção de Itens, que é o relacionamento 1..\* original.



- É importante ressaltar que nas classes que possuem coleções devem possuir métodos para adicionar, remover ou alterar elementos dessas coleções, conforme a necessidade.

## ➤ Exemplo:

```

Cliente c      = new Cliente("876.564.567-45");
Pedido  p      = new Pedido();
Produto prod1  = new Produto(2453);
Produto prod2  = new Produto(5487);
Item     item1 = new Item(1, prod1);
Item     item2 = new Item(1, prod2);
    
```

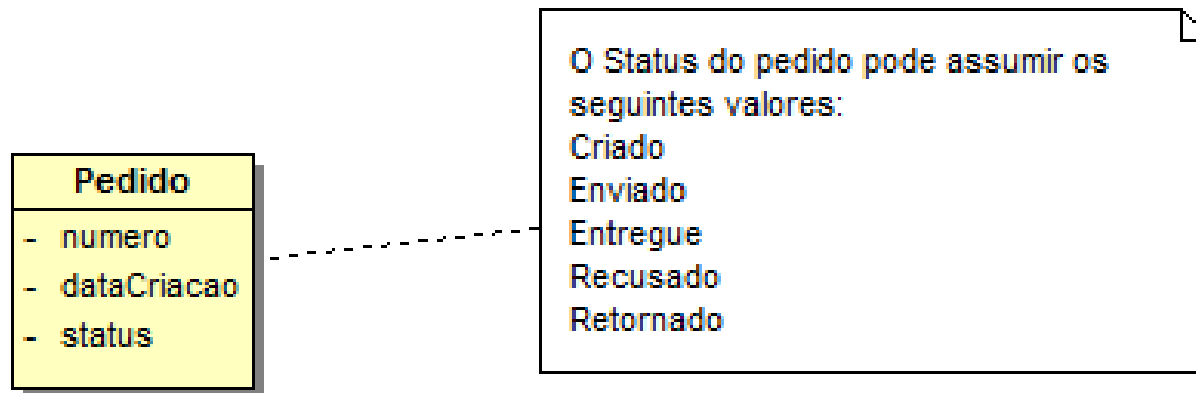
```

p.adicionar(item1);
p.adicionar(item2);
c.adicionar(p);
    
```

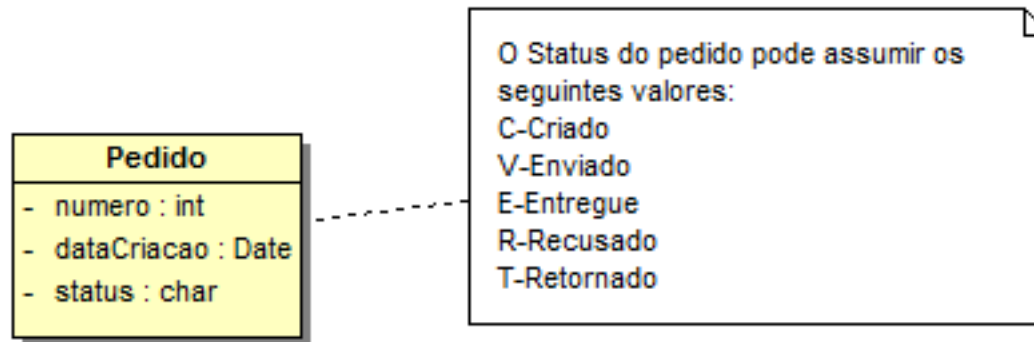
A manipulação das listas é feita dentro da classe que contem as listas !

- Exercício 22: crie a classe Catalogo que possui uma lista de Produtos com código, descrição e preço. Crie métodos para adicionar produtos, remover produtos e listar o catálogo por ordem de código ou preço. Ao final, imprima o valor total e a quantidade de produtos. Implemente, também, duas formas diferentes de imprimir os dados do Produto.

- Outra situação bastante comum é a existência de valores discretos associados a um atributo de uma classe.
- No modelo que estamos usando poderíamos ter:



- Em um primeiro momento, a tendência é criar um atributo do tipos **char** na classe e armazenar um código para o status do pedido:



- Porém, é comum necessitarmos verificar o status de um pedido para tomar uma ação qualquer ou obter uma descrição do status para apresentá-la ao usuário.

```

if (p.getStatus() == 'C')
    System.out.println("Criado");
    
```

- Manter esses códigos e descrições soltos pelo código é um pesadelo para a manutenção.
- Normalmente, encapsulamos essas informações na própria classe Pedido, que ficaria assim:

```
public class Pedido {
    public static final char CRIADO      = 'C';
    public static final char ENVIADO     = 'V';
    public static final char ENTREGUE    = 'E';
    public static final char RECUSADO    = 'R';
    public static final char RETORNADO   = 'T';

    private char status;

    public char getStatus() { return status; }

    public String getDescricaoStatus() {
        if (status == CRIADO) return "Criado";
        if (status == ENVIADO) return "Enviado";
        ...
    }
}
```

- Manter esses códigos e descrições soltos pelo código é um pesadelo para a manutenção.
- Normalmente, encapsulamos essas informações na própria classe Pedido, que ficaria assim:

```
public class Pedido {
    public static final char CRIADO      = 'C';
    public static final char ENVIADO     = 'V';
    public static final char ENTREGUE    = 'E';
    public static final char RECUSADO    = 'R';
    public static final char RETORNADO   = 'T';

    private char status;

    public char getStatus() { return status; }

    public String getDescricaoStatus() {
        if (status == CRIADO) return "Criado";
        if (status == ENVIADO) return "Enviado";
        ...
    }
}
```

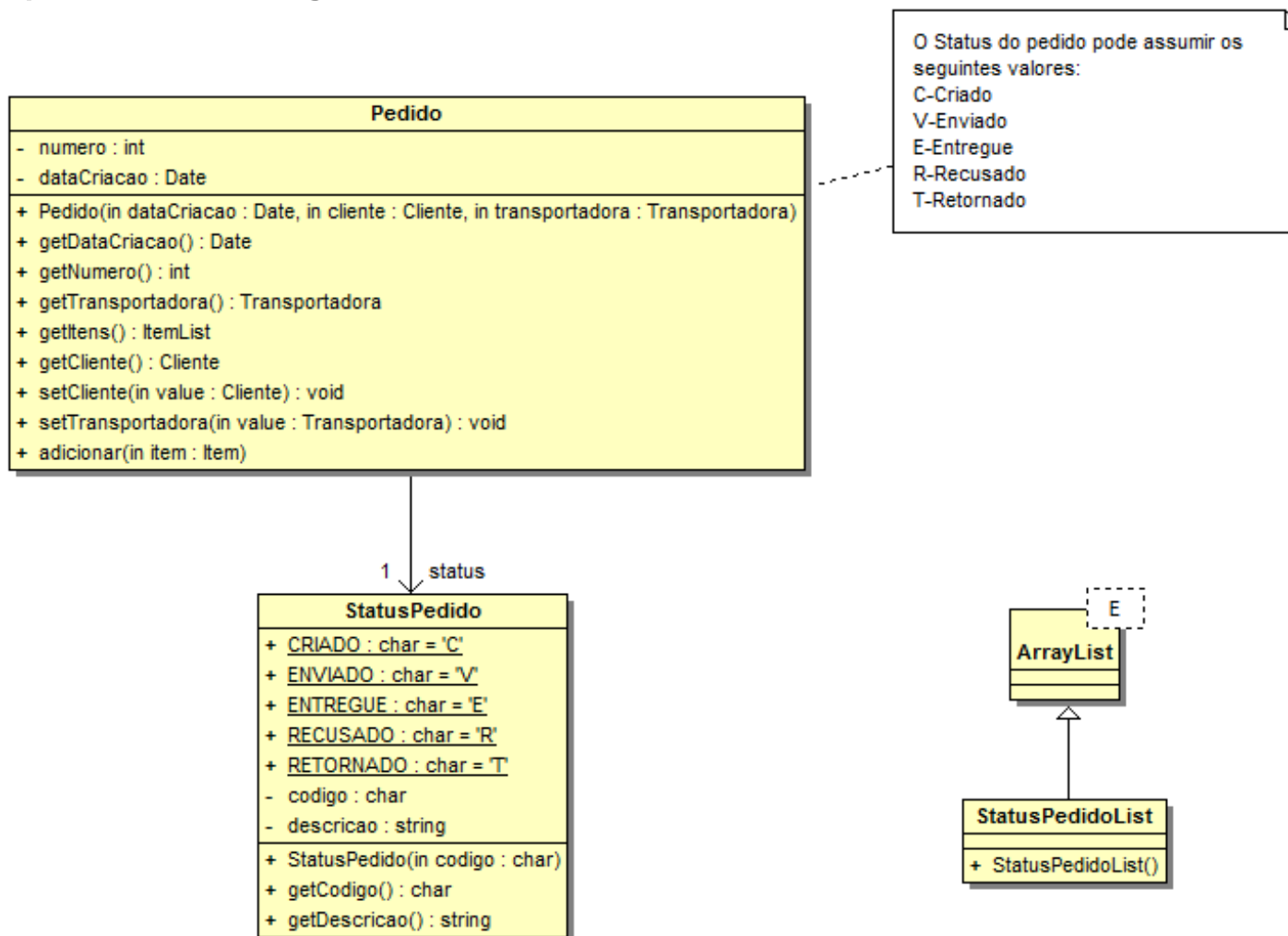
Apesar de ser um avanço em relação à versão anterior essa implementação ainda tem problemas:  
As constantes CRIADO, ENTREGUE, etc., assim como as descrições "Criado", "Entregue", etc., não dizem respeito ao Pedido, mas sim ao **Status do Pedido**.

- Esse raciocínio é reforçado pela ideia de que o Pedido possui um Status, ou seja, está associado a um Status.
- Além disso, podemos precisar, por exemplo, de gerar uma lista dos status possíveis para ser apresentada ao usuário. Como iremos modelar o tratamento dessa lista? Na classe Pedido?
- Essas questões nos levam a mais uma regra geral:

*R2 – Os atributos que possuem valores discretos devem referenciar classes ou tipos enumerados que encapsulam esses valores.*



- Aplicando a regra R2 ao nosso modelo teríamos:



➤ Aplicando a regra R2 ao nosso modelo teríamos:

```
public class StatusPedido {  
    public static final char CRIADO      = 'C';  
    public static final char ENVIADO    = 'V';  
    public static final char ENTREGUE   = 'E';  
    public static final char RECUSADO   = 'R';  
    public static final char RETORNADO  = 'T';  
  
    private char codigo;  
    private String descricao;  
  
    public StatusPedido(char cod) {  
        codigo = cod;  
        if (codigo == CRIADO) descricao = "Criado";  
        ...  
    }  
  
    public char getCodigo() { return codigo; }  
    public String getDescricao() { return descricao; }  
}
```