POO2

Atribuição de Responsabilidades

Atribuição de Responsabilidades

- No projeto orientado a objetos, frequentemente temos que decidir que objetos são **responsáveis** por **realizar** determinada **tarefa** ou **conhecer** determinado **dado**

"Não tenho esse problema, porque pra mim uma ou duas classes bastam para codificar um sistema de grande porte"

– Angus MacGyver

Análise Textual de Abbot

- Técnica para identificação de classes (1983)
  - **Parte de documentos de requisitos e especificações**
  - **Destacar os nomes (substantivos, adjetivos)**
  - **Remover sinônimos (ex. Aluno e Estudante)**
  - **Cada termo remanescente se enquadra em uma situação a seguir:**

    1) Se torna uma classe

    2) Se torna um atributo

    3) Não tem relevância

Análise Textual de Abbot

- Para identificação de métodos e associações:
  - **Destacar verbos do texto**
  - **Verbos de ação (ou transitivos): candidatos a operações/métodos (calcular, cancelar, comprar, transferir, etc)**
  - **Verbos com sentido de "ter": candidatos a agregações ou composições**
  - **Verbos com sentido de "ser": candidatos a generalizações**
  - **Demais verbos: candidatos a outras associações**

Análise Textual de Abbot

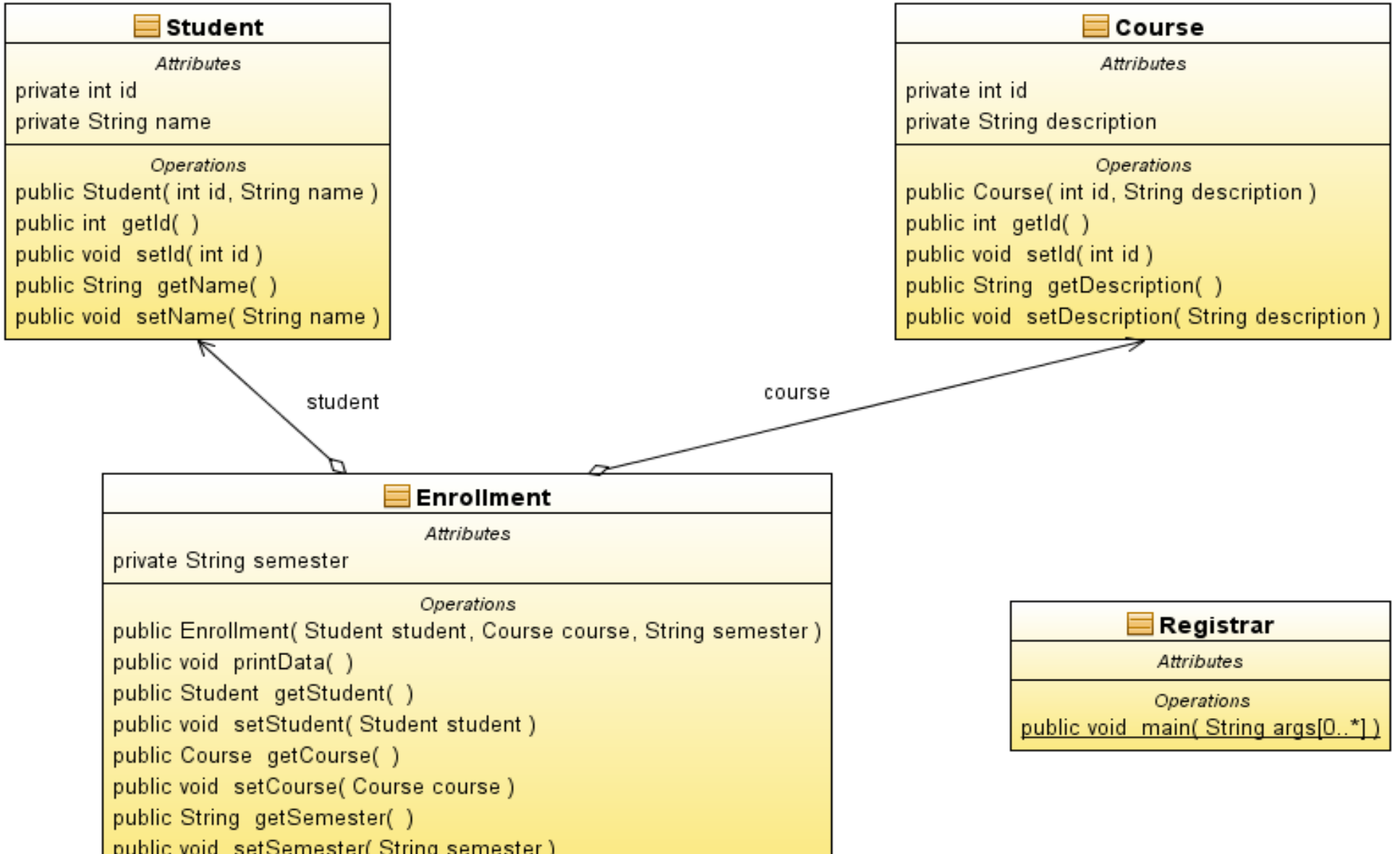
- Em qual Classe colocar determinado método?
  - **Sujeito da frase em que o verbo é utilizado (e.g. Aluno *realiza* Matrícula em Curso, Jogador *ataca* Inimigo)**

Análise Textual de Abbot

- Dilema da classe de Associação
  - **Quem conhece quem?**
    - Aluno conhece disciplinas?
    - Disciplina conhece alunos?

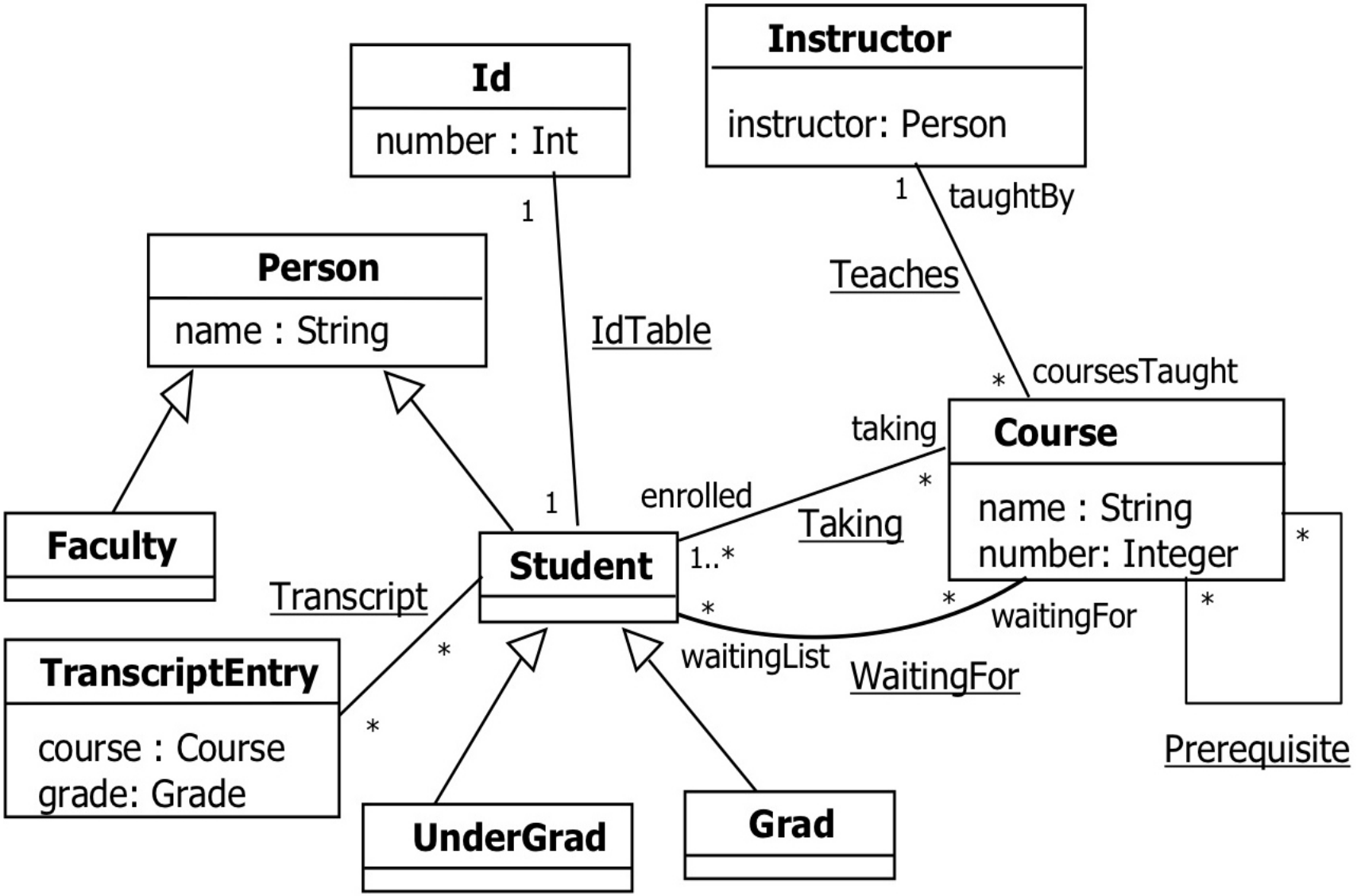
Análise Textual de Abbot

- Aluno *realiza* Matrícula em Curso



Análise Textual de Abbot

- Aluno *matricula-se* em Curso



Análise Textual de Abbot

- Exemplo:

  Se um cliente entra em uma loja com a intenção de comprar um brinquedo para uma criança, então informação/aconselhamento deve estar disponível em tempo razoável em relação ao brinquedo ser apropriado ou recomendado à criança. Isso vai depender da faixa etária, sexo da criança e do tipo do brinquedo. Se o brinquedo é perigoso, ele é inapropriado. Se o brinquedo não é pro mesmo sexo da criança, ele não é recomendado.

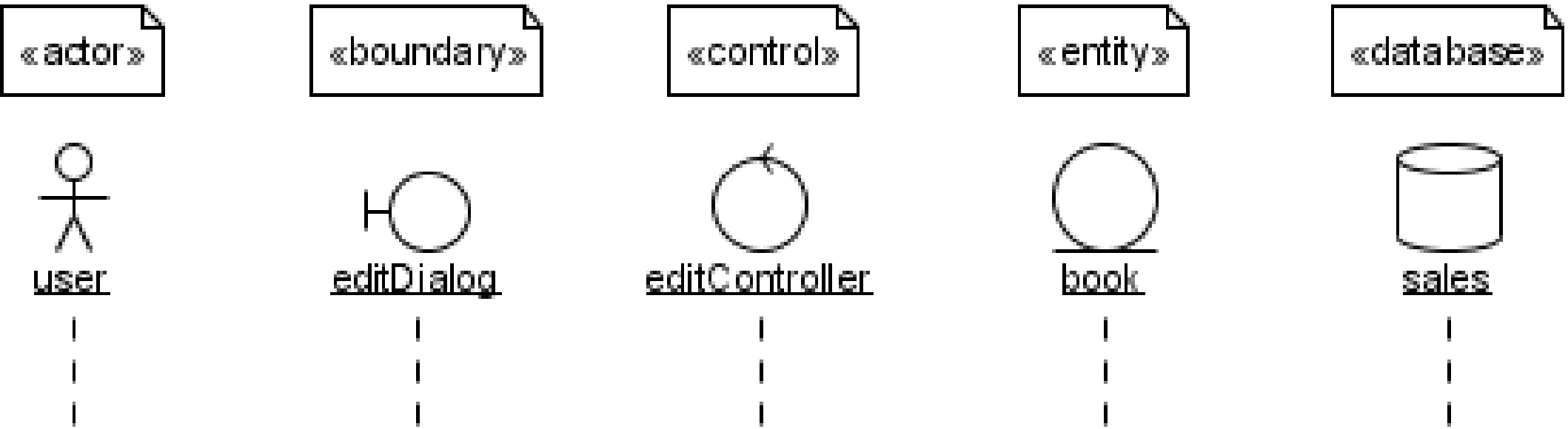
Análise Textual de Abbot

- Método **não** é formal, é apenas uma heurística informal
- Essa técnica pode ser automatizada (Saeki et al., 1989)
- Problema de linguagem: muitas classes candidatas não são relevantes

  Pior ainda, os requisitos tem que ser completos e um tanto técnicos para incorporar elementos importantes, que às vezes não estão explícitos (e.g. classe de controle)

Análise BCE

- Caso especial da Análise de Abbot
- Outra forma para identificação de classes é simplesmente a análise BCE
- Tentamos identificar, para um Caso de Uso, que classes podem fornecer aquele **comportamento**



Análise BCE

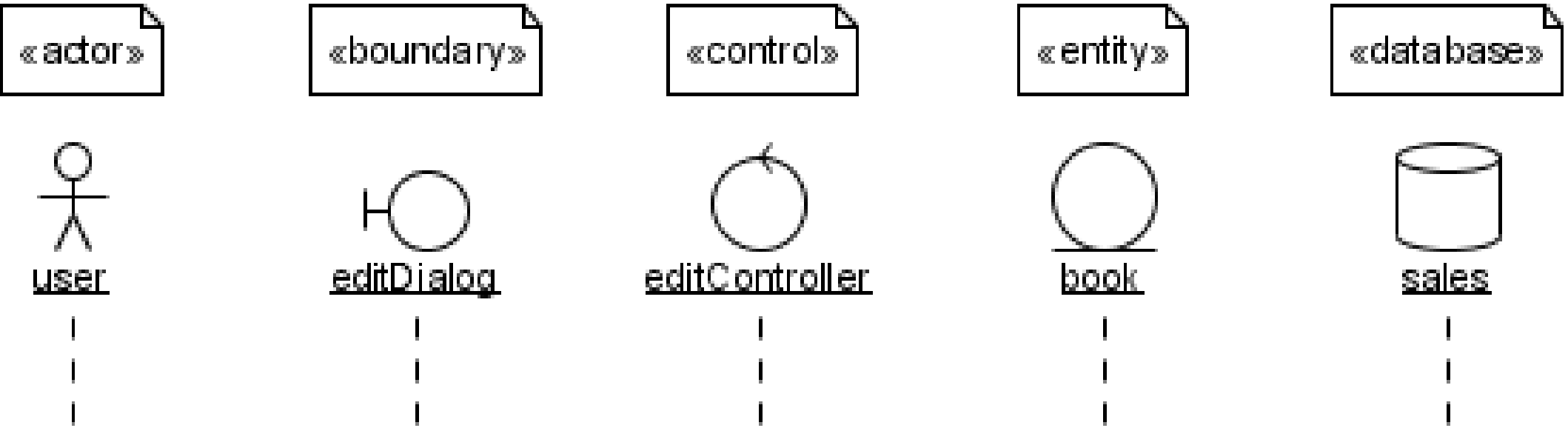
- Para cada Caso de Uso:
  - **Identifique classes a partir do comportamento**
  - **Distribua o comportamento pelas classes identificadas**
- Para cada classe resultante:
  - **Descreva suas responsabilidades**
  - **Descreva atributos e associações**
- Estabeleça a relação entre as classes

Análise BCE

- **Categorização BCE** ou **Análise de Robustez** (Jacobson et al., 1992),
- Usada no processo ICONIX (Rosenberg & Scott, 2001)
- Possui correspondência (mas não equivalência) como o modelo MVC (Model-View-Controller)
- De acordo com a técnica, cada objeto pode ser dividido em **Boundary**, **Control** ou **Entity**, como vimos (daí o BCE). Objetos de **Fronteira**, **Controle** ou **Entidade**

Análise BCE

- De acordo com a técnica, cada objeto pode ser dividido em **Boundary**, **Control** ou **Entity** (daí o BCE). Objetos de **Fronteira**, **Controle** ou **Entidade**



Análise BCE

- Objetos de Fronteira:
  - **“Limitantes” ou fronteiras para sub-sistemas. Comunicação do sistema com atores (humanos, sistemas externos ou dispositivos).**
- Responsabilidades: 1) Notificar demais objetos de eventos externos, gerados pelo ambiente, e 2) Notificar atores do resultado de interações
- Ex: **Candidato** realiza inscrição no concurso, inscrição é confirmada pelo **Sistema de Pagamento** mediante pegamento da taxa do boleto.

Análise BCE

- Objetos de Controle:
  - **São a "ponte de comunicação" entre objetos de fronteira e objetos de entidade.**
  - **Responsáveis por controlar a lógica de execução correspondente a um caso de uso.**
  - **Decidem o que o sistema deve fazer quando um evento de sistema ocorre.**
  - **Eles realizam o controle do processamento**
  - **Agem como gerentes (coordenadores, controladores) dos outros objetos para a realização de um caso de uso.**
  - **Traduzem eventos de sistema em operações que devem ser realizadas pelos demais objetos.**

Análise BCE

- Objetos de Controle:
  - **Exemplos: GerenciadorContas, ControladorInscricao, GerenciadorReservasDeCarros, MarcadorTempo, AnimadorDeSprites, ArmazenadorDeObjetos**
  - **Normalmente são únicos (Singleton?)**
- Lembremos: Controle quer dizer fluxo, e não regras.
- Relacionados a Aplicação

Análise BCE

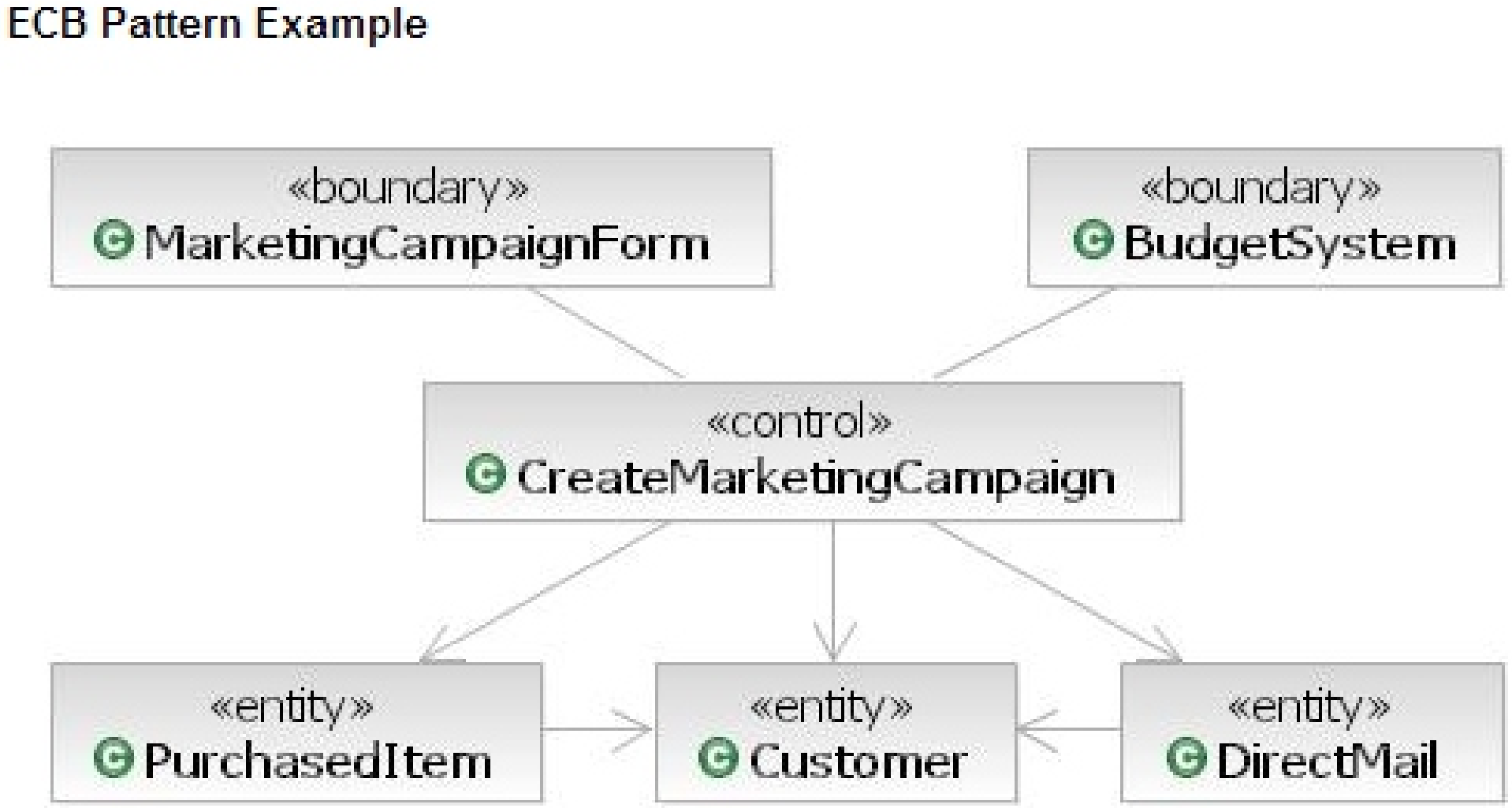
- Objeto Entidade:
  - **Repositório para informações e as regras de negócio manipuladas pelo sistema.**
  - **Representam conceitos do domínio do negócio.**
  - **Características**
    - Normalmente armazenam informações persistentes.
    - É comum **várias** instâncias da mesma entidade existindo no sistema (oposto do Controlador).
    - Participam de vários casos de uso e têm ciclo de vida longo.
    - Relacionados a Domínio

Análise BCE

- Assim, temos cada objeto fazendo uma de três tarefas:
  - **Comunicar-se com exterior (fronteira)**
  - **Manter e manipular informações (entidade)**
  - **Realizar uma tarefa (controle)**

Análise BCE

- Exemplo do site do Eclipse:



Análise BCE

- **Domínio vs Aplicação**
  - Partes da Análise do problema (que antecede o projeto no processo)
  - **Análise de Domínio**: elementos do mundo real processados na aplicação em desenvolvimento, levantamento dos objetos importantes para a área, com ajuda de especialista (descobrimos classes)
  - **Análise de Aplicação**: elementos não necessariamente parte comum do domínio, mas necessários pra realização da tarefa. Somente tem sentido num contexto de sistema de software, enquanto elementos de domínio tem sentido for a do contexto de software (inventamos classes)

Identificação Dirigida a Responsabilidades

- O que queremos é encapsular comportamento
- Abstraímos detalhes internos e nos preocupamos nas responsabilidades da classe que são úteis externamente
- A **responsabilidade** é algo que o objeto **conhece** ou **faz**
- Muitas vezes um objeto sozinho não consegue concluir uma responsabilidade, e precisa de colaborações
- Ex: inscrição de aluno em disciplina

Sumário: Aluno usa o sistema para realizar inscrição em disciplinas.

Ator Primário: Aluno

Realizar Inscrição (CSU01)

Atores Secundários: Sistema de Faturamento

Precondições: O Aluno está identificado pelo sistema.

Fluxo Principal

1. O Aluno solicita a realização de inscrição.
2. O sistema apresenta as disciplinas disponíveis para o semestre corrente e para as quais o aluno tem pré-requisitos.
3. O Aluno seleciona as disciplinas desejadas e as submete para inscrição.
4. Para cada disciplina selecionada, o sistema aloca o aluno em uma turma que apresente uma oferta para tal disciplina.
5. O sistema informa as turmas nas quais o Aluno foi alocado. Para cada alocação, o sistema informa o professor, os horários e os respectivos locais das aulas de cada disciplina.
6. O Aluno confere as informações fornecidas.
7. O sistema envia os dados sobre a inscrição do aluno para o Sistema de Faturamento e o caso de uso termina.

Fluxo Alternativo (4): Inclusão em lista de espera

a. Se não há oferta disponivel para alguma disciplina selecionada pelo aluno, o sistema reporta o fato e fornece a possibilidade de inserir o Aluno em uma lista de espera.

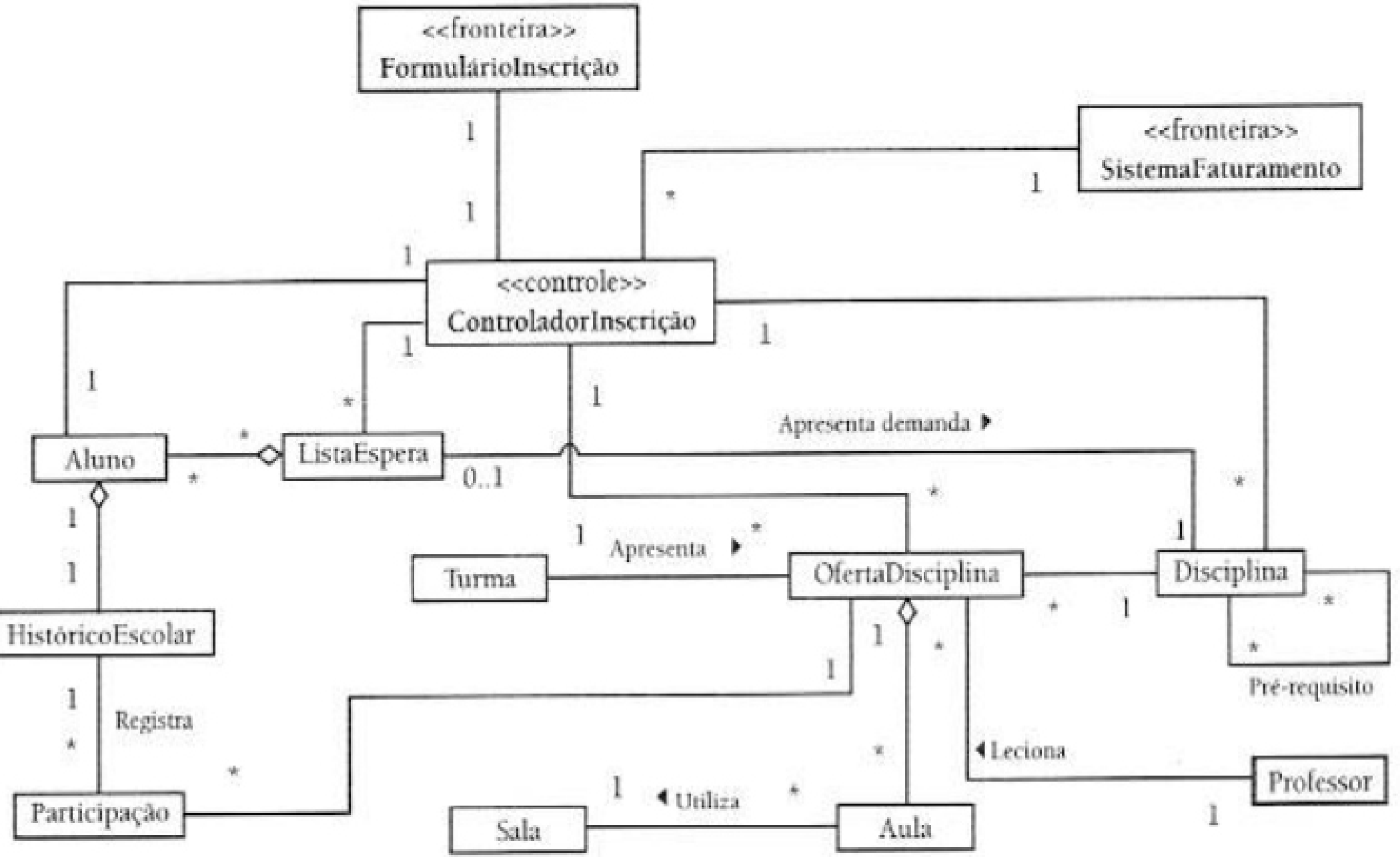
b. Se o Aluno aceitar, o sistema o insere na lista de espera e apresenta a posição na qual o aluno foi inserido na lista. O caso de uso retorna ao passo 4.

c. Se o Aluno não aceitar, o caso de uso prossegue a partir do passo 4.

Fluxo de Exceção (4): Violação de RN01

a. Se o Aluno atingiu a quantidade máxima de inscrições (RN01), o sistema informa ao aluno a quantidade de disciplinas que ele pode selecionar, e o caso de uso retorna ao passo 2.

Pós-condições: O aluno foi inscrito em uma das turmas de cada uma das disciplinas desejadas, ou foi adicionado a uma ou mais listas de espera.


<<fronteira>>
FormulárioInscrição
<<fronteira>>
SistemaFaturamento
<<controle>>
ControladorInscrição
Aluno
ListaEspera
Apresenta demanda
0..1
Turma
Apresenta
OfertaDisciplina
Disciplina
Pré-requisito
HistóricoEscolar
Registra
Participação
Sala
Utiliza
Aula
Leciona
Professor

Definição de Propriedades

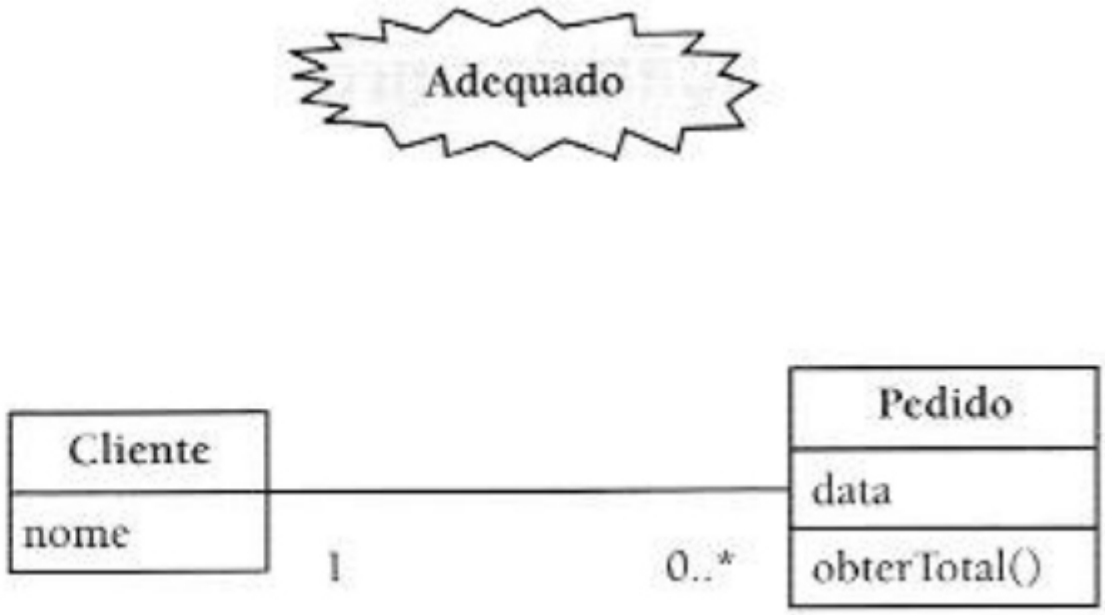
- Agora que temos as classes, vamos discutir como podemos atribuir **propriedades**
- Propriedade é o nome coletivo que damos para:
  - **Métodos**
  - **Atributos**
- Como mapear responsabilidades de uma classe para propriedades dela?

Definição de Propriedades

- Operações: *fazer*
  - **Contribuição para algum caso de uso. Os diagramas de Interação (Comunicação, Sequência e outros) vão nos ajudar aqui!**
- Atributos: *conhecer*
  - **É atributo se se aplica a todos os membros da classe e tem valor atômico. Valem estruturas de dados também.**
- Deixe identificadores unívocos e "chaves primárias" (como codigoCliente) para o final
- Cuidado com exceções: A classe **Pedido** deve *conhecer* seu **Cliente**, mas isso é associação!
- Um atributo tem relação forte de dependência com o objeto que o contém. Cuidado!

Definição de Propriedades

- Cuidado com exceções: A classe **Pedido** deve conhecer seu **Cliente**, mas isso é associação!
- Associações: propriedades de *conhecer* que não podemos confinar a apenas uma classe



Padrões para Atribuir Responsabilidades

- Vamos falar um pouco de mensagens e métodos

Modelagem de Interações

- **Mensagem** vs **Responsabilidade**
  - Uma mensagem implica a existência de um método no objeto receptor
  - Assim, quando há uma troca de mensagens (por exemplo, em um diagrama e sequência que veremos em Modelagem), estamos na verdade mostrando métodos que o objeto “alvo” deve ter
  - A mensagem é uma chamada de método

Padrões para Atribuir Responsabilidades

- Um sistema OO é composto de objetos que enviam mensagens uns para os outros
  - **Uma mensagem é um método executado no contexto de um objeto**
- Escolher como distribuir as responsabilidades entre objetos (ou classes) é crucial para um bom projeto
  - **Uma má distribuição leva a sistemas e componentes frágeis e difíceis de entender, manter, reusar e estender**

Padrões para Atribuir Responsabilidades

- Mostraremos alguns padrões de distribuição de responsabilidades
  - **Padrões GRASP (General Responsibility Assignment Software Patterns)**
  - **São chamados também de padrões de Larman**
- Lembrando que padrões = princípios de um bom projeto e desenvolvimento OO

Padrões para Atribuir Responsabilidades

- Responsabilidades são obrigações de um tipo ou de uma classe
  - **Obrigações de fazer algo**
  - **Fazer algo a si mesmo**
  - **Iniciar ações em outros objetos**
  - **Controlar ou coordenar atividades em outros objetos**
- Obrigações de conhecer algo
  - **Conhecer dados encapsulados**
  - **Conhecer objetos relacionados**
  - **Conhecer coisas que se pode calcular**

Padrões para Atribuir Responsabilidades

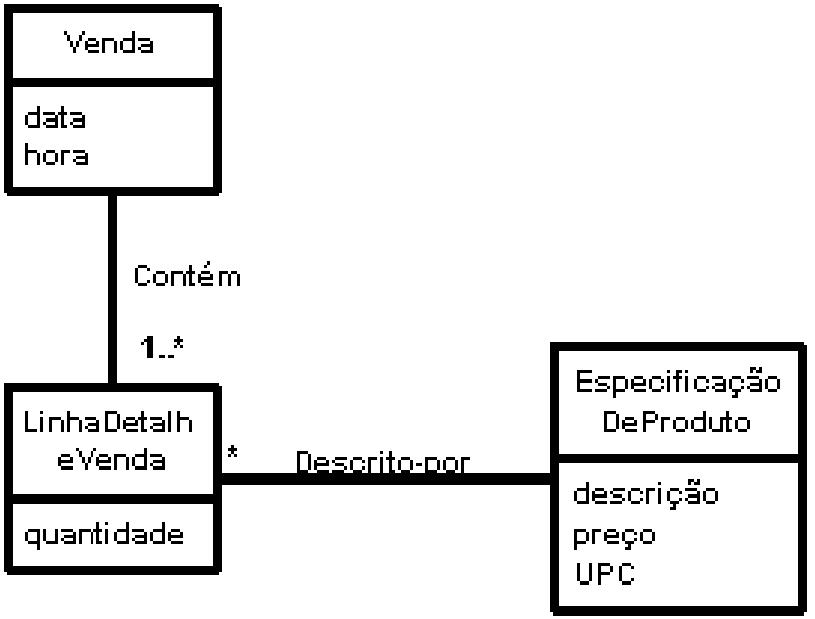
- Exemplos
  - **Um objeto Venda tem a responsabilidade de criar Lista de Pedidos (fazer algo)**
  - **Um objeto Venda tem a responsabilidade de saber sua data (conhecer algo)**
- Granularidade
  - **Uma responsabilidade pode envolver um único método (ou poucos)**
    - Exemplo: Detalhar uma Venda lendo seus atributos
  - **Uma responsabilidade pode envolver dezenas de classes e métodos**
    - Exemplo: Responsabilidade de fornecer acesso a um BD
  - **Uma responsabilidade não é igual a um método**
    - Mas métodos são usados para implementar responsabilidades

Expert

- Problema
  - **Quem realiza a tarefa necessária para cumprir um requisito?**
- Solução
  - **Atribuir uma responsabilidade ao expert de informação - a classe que possui a informação necessária para cumprir a responsabilidade**

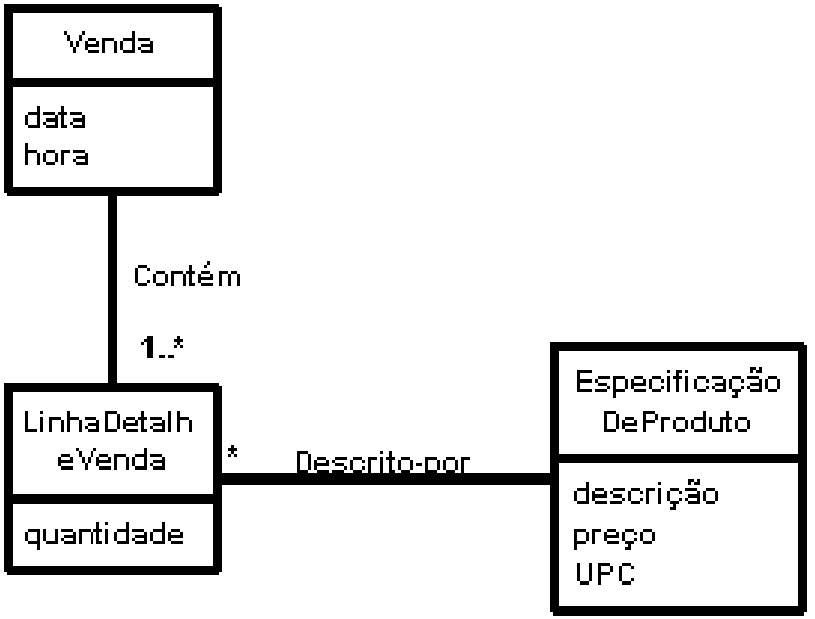
Expert

- Exemplo
  - **Num sistema de Vendas Online, alguma classe precisa saber o total de uma venda**
  - **Podemos dizer isso sobre a forma de uma responsabilidade:**
    - Quem deveria ser responsável pelo conhecimento do total de uma venda?
    - Pelo padrão Expert, escolhemos a classe que possui a informação necessária para determinar o total



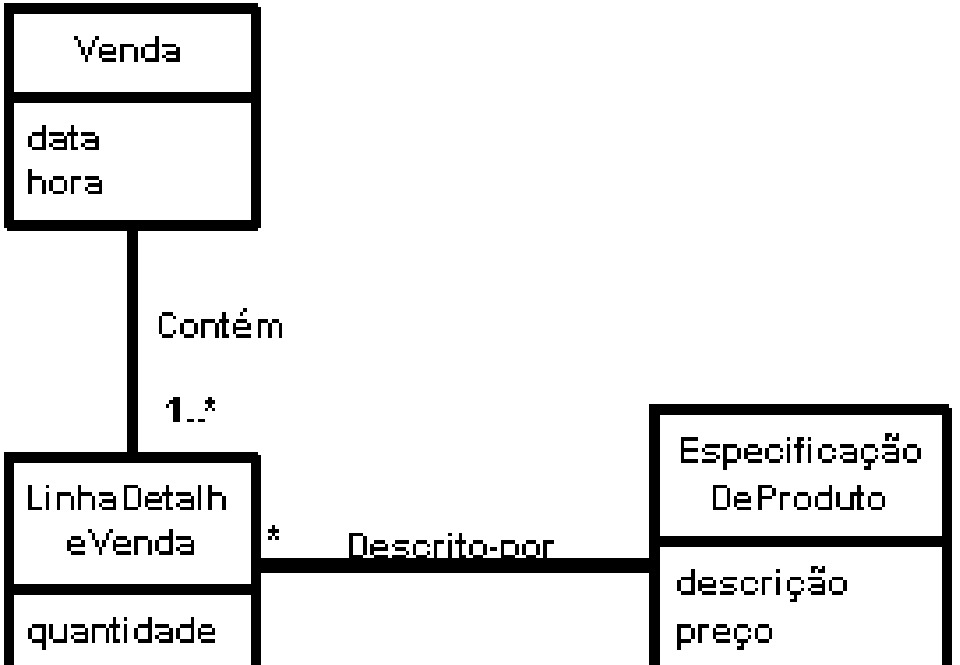
Expert

- Exemplo
  - **Qual é a informação necessária?**
    - Precisamos conhecer (ter acesso a) todos os LinhaDetalheVenda
  - **Qual é information expert?**
    - É a classe **Venda**
    - Podemos agora fazer parte do diagrama de colaboração e do diagrama de classes



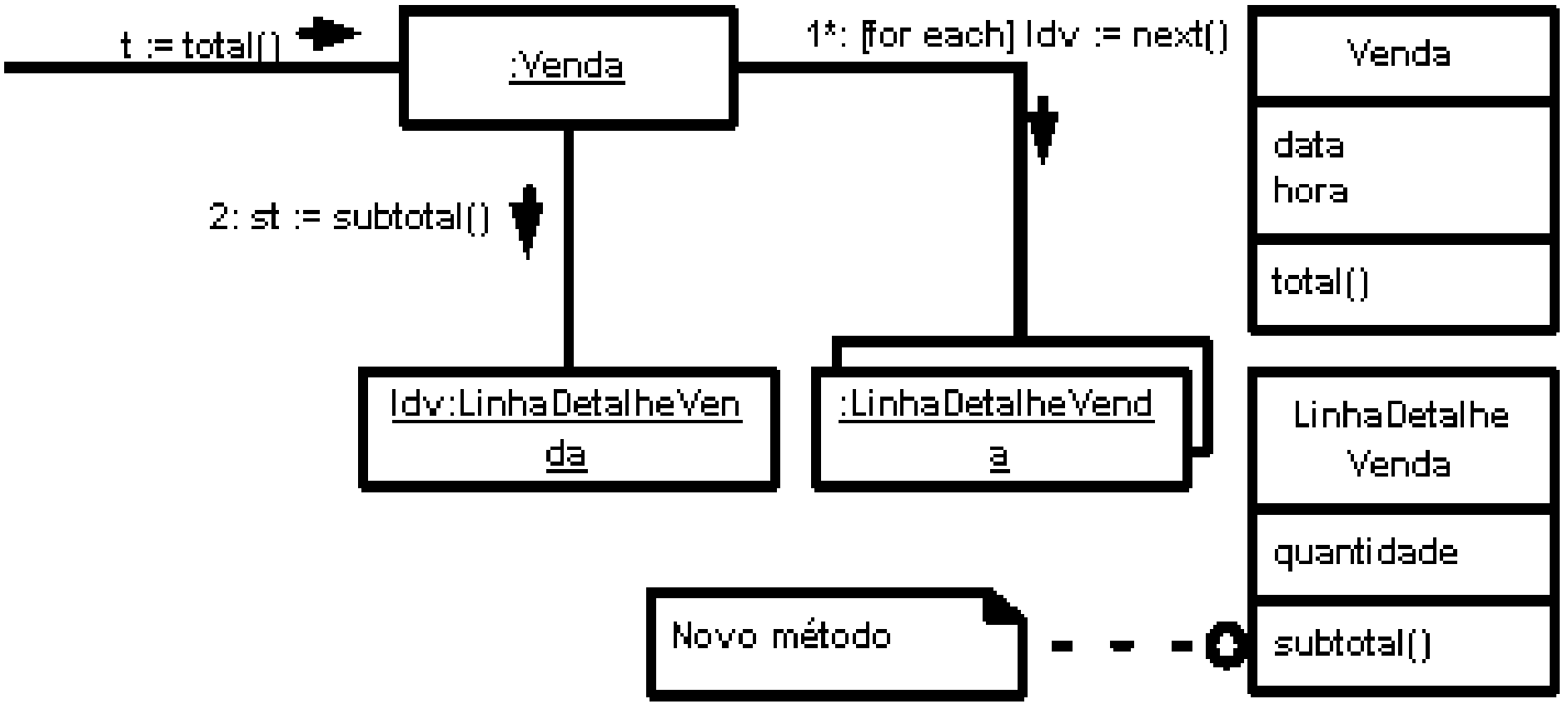
Expert

- Ainda não terminamos. Qual informação é necessária para determinar o subtotal para um item (uma linha de detalhe, ou o total de um item de pedido)?
  - **Precisamos de LinhaDeVenda.quantidade e de EspecificaçãoDeProduto.preço**
  - **Quem é o information expert?**
  - **É a classe LinhaDetalheVenda**



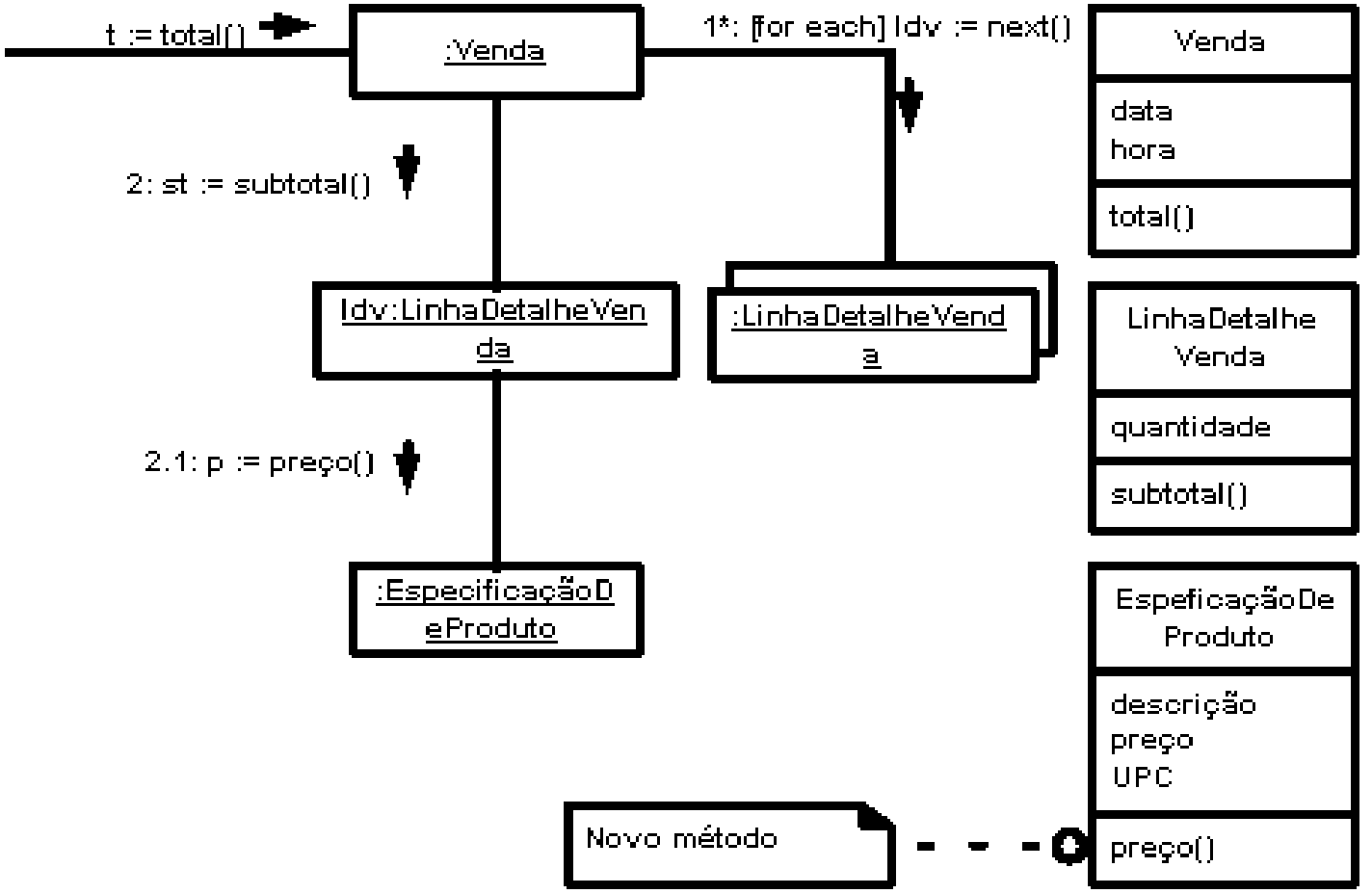
Expert

- Chegamos aos seguintes diagramas de comunicação



Expert

- Qual é o information expert para saber o preço de um produto?
  - **É EspecificaçãoDeProduto**
- Chegamos aos seguintes diagramas:



Expert

- É o padrão que corresponde aos métodos de atribuição responsabilidade que temos visto
- Padrão **mais usado**
- Característica: A informação necessária freqüentemente está espalhada em vários objetos
  - **Portanto, tem muitos "experts parciais"**
  - **Exemplo: determinar o total de uma venda no caso acima requer a colaboração de diversos objetos, em 3 classes diferentes**

Expert

- Mensagens são usadas para estabelecer as colaborações
- O resultado final é diferente do mundo real
  - **No mundo real, uma venda não calcula seu próprio total**
  - **Isso seria feito por uma pessoa (se não houvesse software)**
  - **Mas no mundo de software, não queremos atribuir essa responsabilidade ao Caixa ou ao Terminal Ponto-De-Venda!**
  - **No mundo de software, coisas inertes (ou até conceitos como uma Venda) podem ter responsabilidades: Tudo está vivo!**

Expert

- Consequências
  - **O encapsulamento é mantido, já que objetos usam sua própria informação para cumprir suas responsabilidades**
    - Leva a fraco acoplamento entre objetos e sistemas mais robustos e fáceis de manter
  - **Leva a alta coesão, já que os objetos fazem tudo que é relacionado à sua própria informação**

Modelagem de Interações

- Coesão e acoplamento
  - **Nosso objetivo final é decompor as responsabilidades e alocá-las em classes na forma de métodos**
  - **Para N responsabilidades, podemos**
    - Criar N classes, cada uma assumindo uma responsabilidade (acoplamento total!)
    - Criar uma classe que assuma as N responsabilidades (coesão zero!)
    - Qualquer combinação entre esses dois extremos... como escolher a melhor?

Modelagem de Interações

- **Coesão**: quão fortemente relacionadas são as responsabilidades de uma classe. É bom, pois encapsula a abstração, provendo clareza, reusabilidade, etc.
  - **Se temos atributos poucos relacionados e métodos pouco relacionados numa mesma classe, é indício de que poderíamos ter duas classes ou mais no lugar (risco de anti-padrão God Object)**
  - **Classes complexas demais: perde-se Orientação a Objeto, compreensão**

Modelagem de Interações

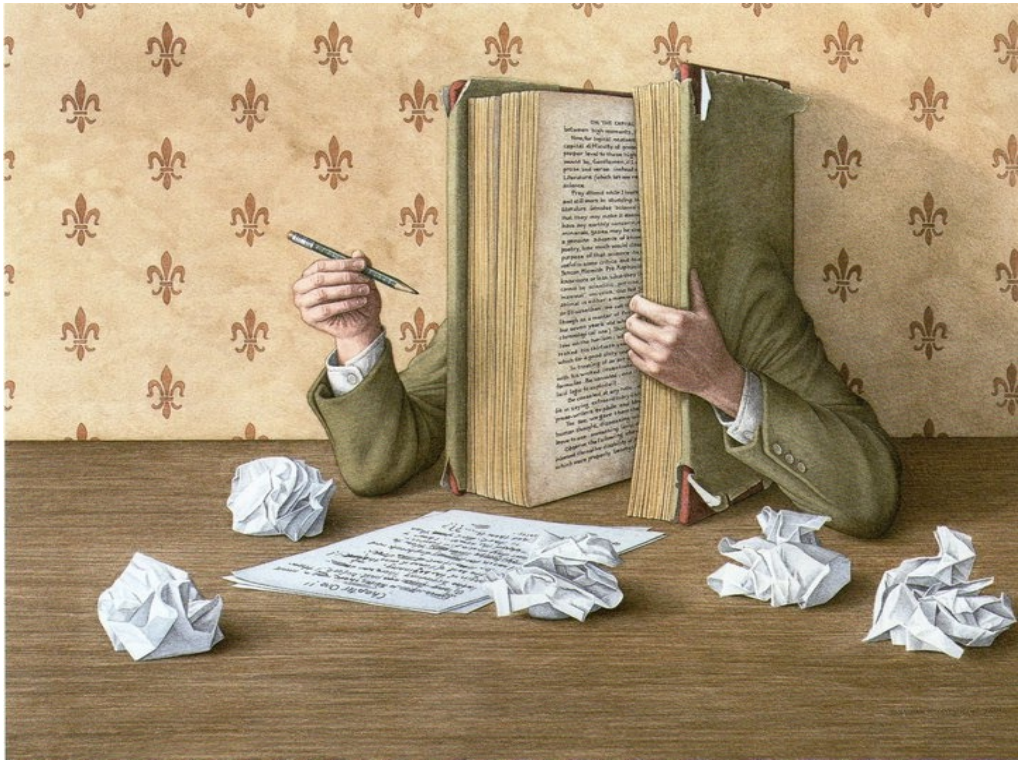
- **Acoplamento**: quão fortemente uma classe está relacionada a outras classes (**conhece** ou **depende** das mesmas).
  - **Alto acoplamento reduz reusabilidade, aumenta complexidade, deixa responsabilidades fragmentadas pelo código (menos legível), pouca modularidade e maior sensibilidade a mudanças**

Modelagem de Interações

- Quando criamos uma chamada de um objeto para outro, estamos criando uma dependência entre os dois objetos. (aumento do acoplamento)
- Portanto, cuidado! Quanto menos associações no diagrama de classes, melhor
- Padrões de Projeto como o **Façade** e **Mediator** nos ajudam aqui

Expert

- Também conhecido como
  - **"Colocar as responsabilidades com os dados"**
  - **"Quem sabe, faz"**
  - **"Animação"**
  - **"Eu mesmo faço"**
  - **"Colocar os serviços junto aos atributos que eles manipulam"**

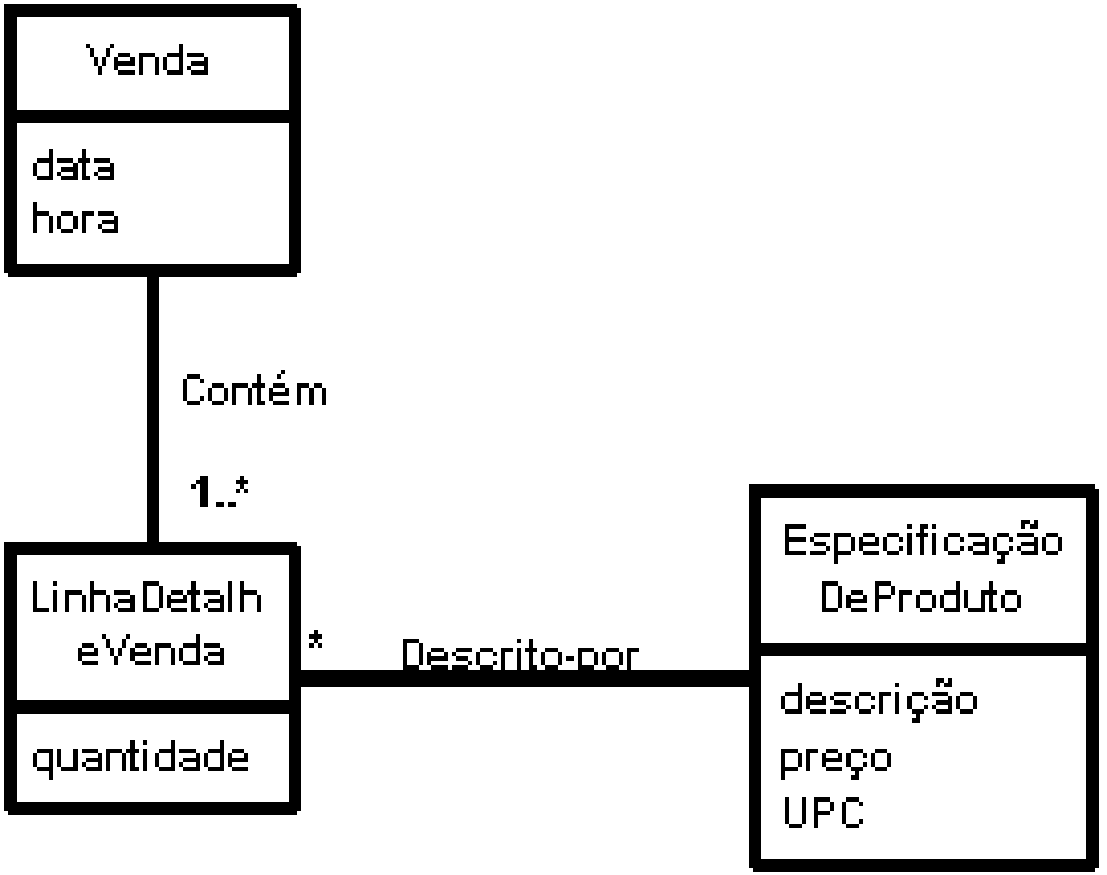


Creator

- Outro problema:
  - **Quem deve criar instâncias de uma classe?**
- Solução:
  - **Atribuir à classe B a responsabilidade de criar instância da classe A se uma das seguintes condições se aplicar:**
    - B agrega objetos da classe A
    - B contém objetos da classe A
    - B registra instâncias da classe A
    - B usa muitos objetos da classe A
    - B possui os dados usados para inicializar A
  - **B é um criador (*creator*) de objetos da classe A**
  - **Se mais de um candidato se aplica, escolha o B que agregue ou contenha objetos da classe A**

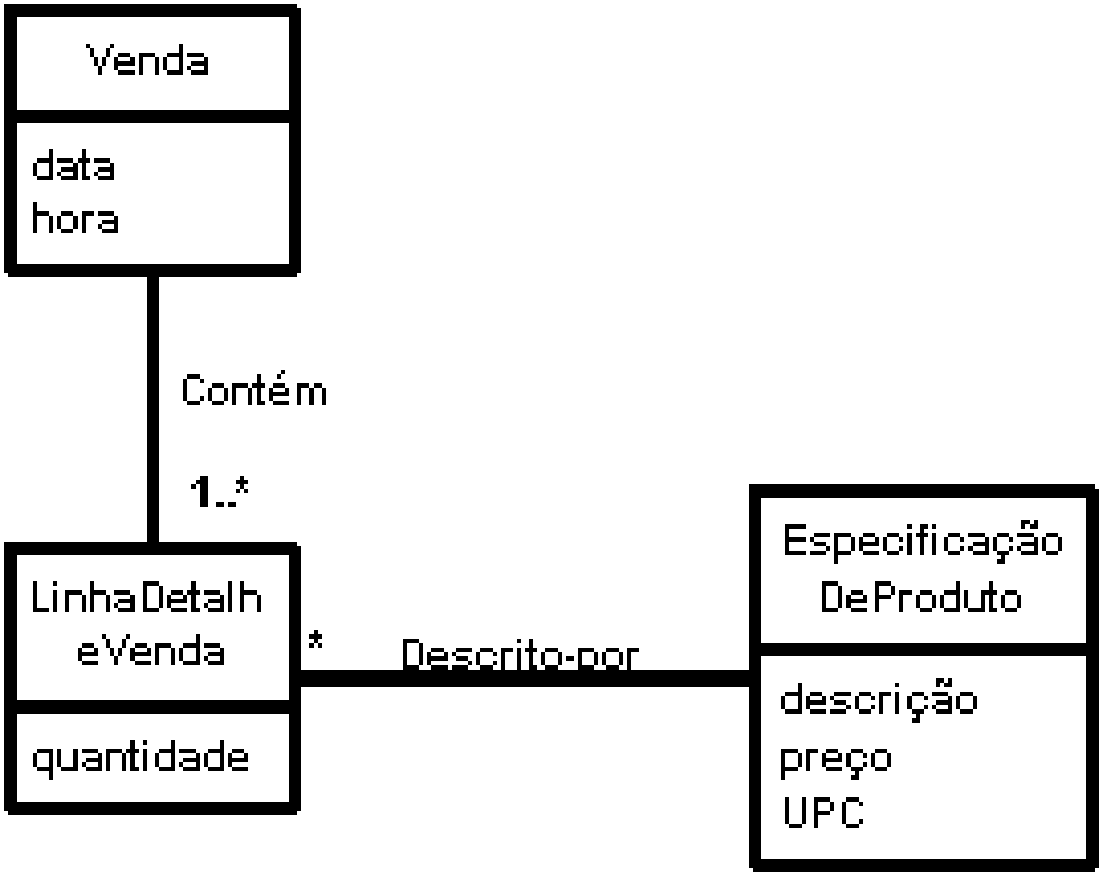
Creator

- Exemplo
  - **No exemplo anterior (Terminal de Venda), quem deveria criar uma instância de LinhaDetalheVenda?**
  - **Pelo padrão Creator, precisamos achar alguém que agrega, contém, ... instâncias de LinhaDetalheVenda**
  - **Considere o modelo conceitual parcial abaixo:**



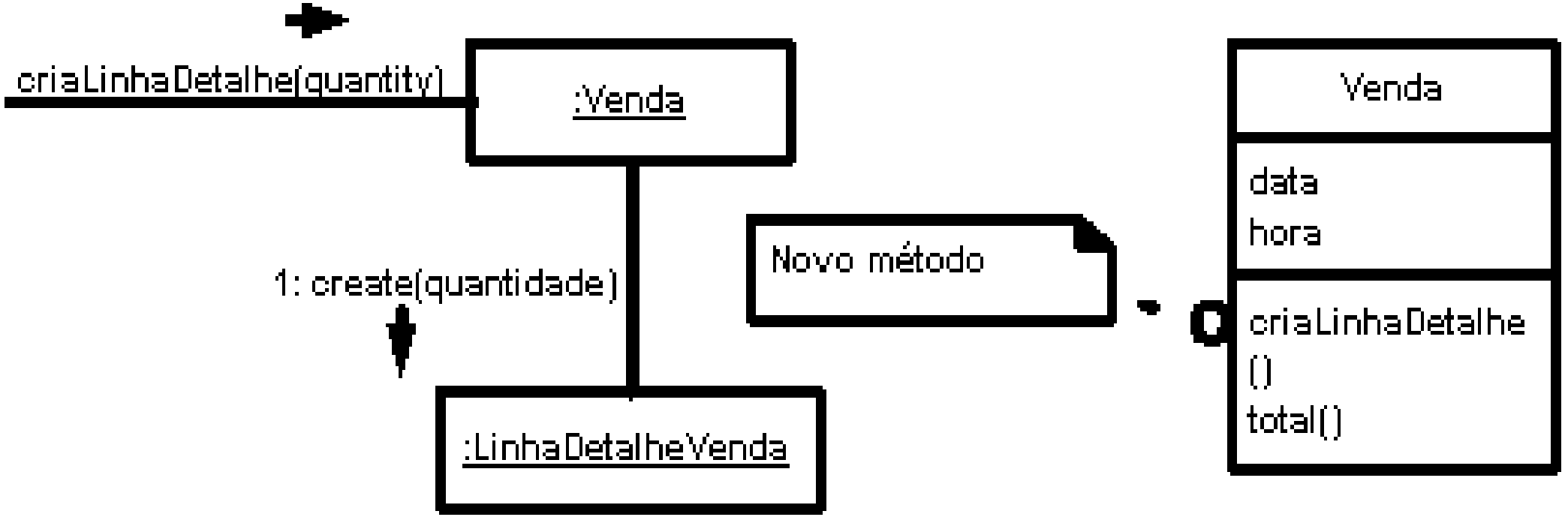
Creator

- Exemplo
  - **Venda agrega instâncias de LinhaDetalheVenda e é portanto um bom candidato para criar as instâncias**



Creator

- Exemplo
  - **Chegamos aos seguintes diagramas**



Creator

- Discussão
  - **Escolhemos um criador que já estará acoplado de qualquer forma ao objeto após sua criação**
  - **Não criamos novos acoplamentos**
  - **Exemplo de criador que possui os valores de inicialização**
    - Uma instância de **Pagamento** deve ser criada
    - A instância deve receber o total da venda
    - Quem tem essa informação? Venda
    - Venda é um bom candidato para criar objetos da classe Pagamento

Baixo Acoplamento

- Vejamos mais um padrão de atribuição de responsabilidades
  - **Minimizar dependências e maximizar reuso**
  - **Acoplamento alto:**
    - Mudanças em uma classe relacionada força mudanças locais à classe
    - A classe é mais difícil de entender isoladamente
    - A classe é mais difícil de ser reusada, já que depende da presença de outras classes
  - **Solução**
    - Atribuir responsabilidades de forma a minimizar o acoplamento

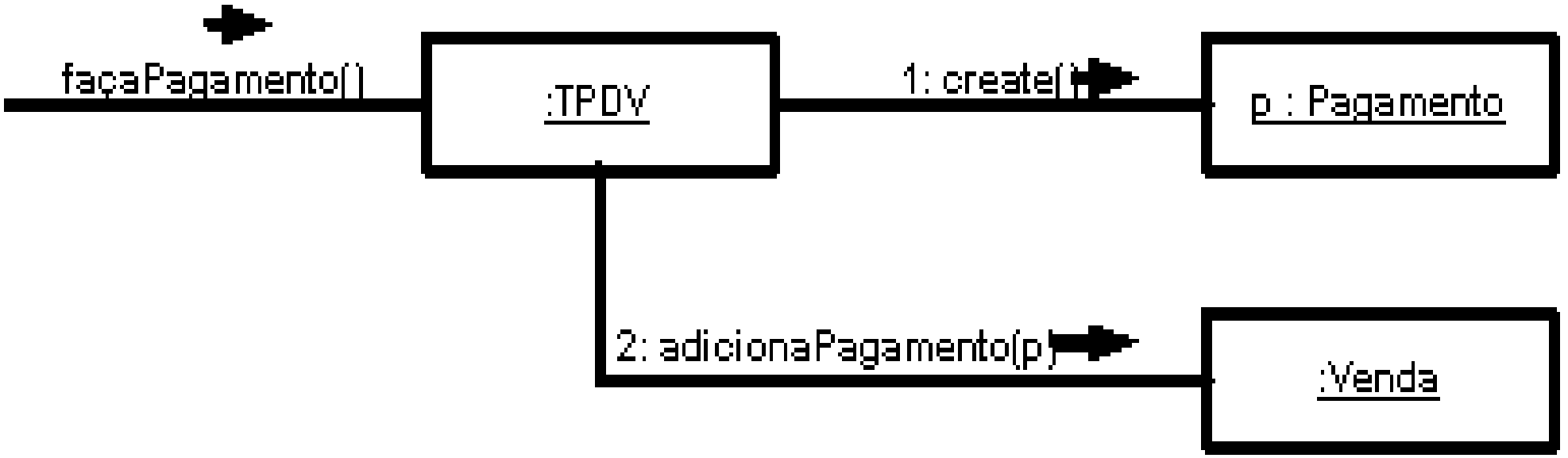
Baixo Acoplamento

- Exemplo

- Suponha que temos que criar um Pagamento e associá-lo a uma Venda
- Que classe deveria ter essa responsabilidade?
- Alternativa 1: No mundo real, um TPDV "registra" um pagamento e o padrão Creator sugere que TPDV poderia criar Pagamento
  - **TPDV deve então passar o pagamento para a Venda**
  - **Veja o resultado no próximo slide**

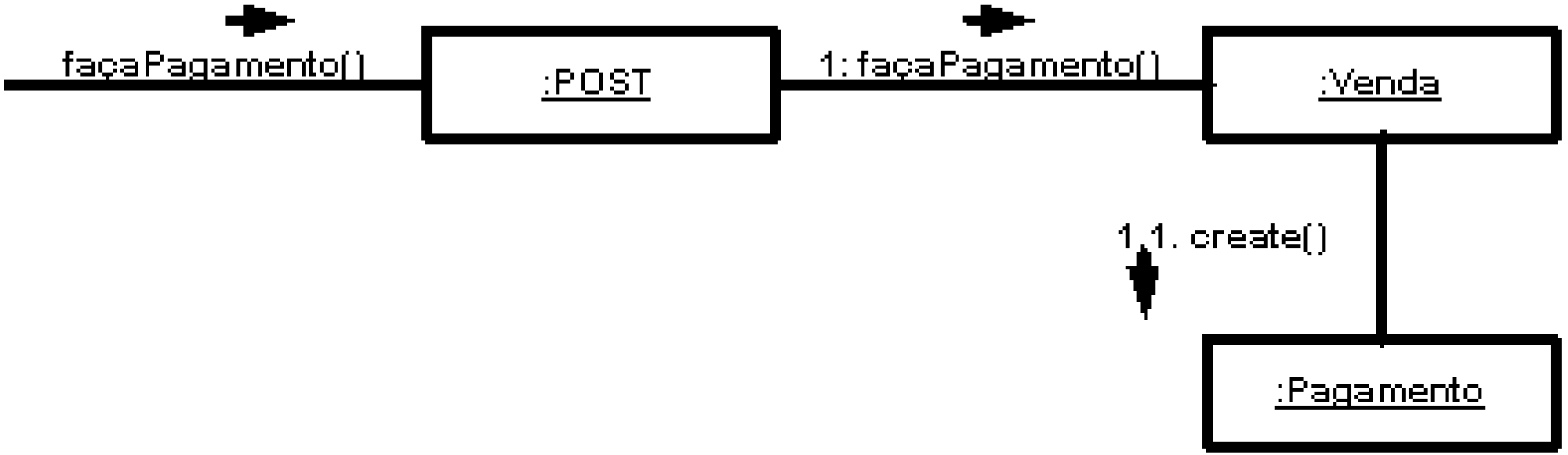
Baixo Acoplamento

- Alternativa 1: No mundo real, um TPDV "registra" um pagamento e o padrão Creator sugere que TPDV poderia criar Pagamento



Baixo Acoplamento

- Alternativa 2: Criar o Pagamento com Venda e associá-lo à Venda



Baixo Acoplamento

- Supondo que a Venda deva ter conhecimento do pagamento (depois da criação) de qualquer jeito, a alternativa 2 tem menos acoplamento (TPDV não está acoplado a Pagamento)
  - **Dois padrões (Creator e Low Coupling) sugeriram diferentes soluções**
  - **Nesse caso, Baixo Acoplamento "ganha"**

Baixo Acoplamento

- Discussão
  - **Acoplamento Abstrato é preferível (ou padrão Observer)**
  - **Não se deve minimizar acoplamento criando alguns poucos objetos monstruosos (God classes)**
  - **Tipos de acoplamentos (do menos ruim ao pior)**
    - Acoplamento de dados
    - Acoplamento de controle
    - Acoplamento de dados globais
    - Acoplamento de dados internos

Baixo Acoplamento

- Acoplamento de dados
  - **Situações**
    - Saída de um objeto é entrada de outro
    - Uso de parâmetros para passar itens entre métodos
  - **Ocorrência comum:**
    - Objeto a passa objeto x para objeto b
    - Objeto x e b estão acoplados
      - **Uma mudança na interface de x pode implicar em mudanças a b**

```
class Servidor {
  public void mensagem(MeuTipo x) {
    // código aqui
    x.façaAlgo(Object dados); // dados e x estão acoplados
                              // (se interface de dados mudar x terá que mudar)
    // mais código
  }
}
```

Baixo Acoplamento

- Acoplamento de dados
  - **Exemplo pior:**
    - Objeto a passa objeto x para objeto b
    - x é um objeto composto ou agregado (talvez Composite)
    - Objeto b deve extrair objeto y de dentro de x
    - Há acoplamento entre b, x, representação interna de x, y
  - **Exemplo: ordenação de registros de alunos por matrícula (próximo slide)**

Baixo Acoplamento

- Acoplamento de dados

```
class Aluno {
   String nome;
   long   matrícula;

   public String getNome() { return nome; }
   public long   getMatrícula() { return matrícula; }

   // etc.
}

ListaOrdenada listaDeAlunos = new ListaOrdenada();
listaDeAlunos.add(new Aluno(...));
//etc.
```

- Vamos ver como ocorre a ordenação

Baixo Acoplamento

- Acoplamento de dados

```
class ListaOrdenada {
  Object[] elementosOrdenados = new Object[tamanhoAdequado];

  public void add(Aluno x) {
    // há código não mostrado aqui
    long matrícula1 = x.getMatrícula();
    long matrícula2 = elementosOrdenados[k].getMatrícula();
    if(matrícula1 < matrícula2) {
      // faça algo
    } else {
      // faça outra coisa
    }
  }
```

- ListaOrdenada sabe muita coisa de Aluno!

Baixo Acoplamento

- Acoplamento de dados
- O que ListaOrdenada sabe sobre aluno?
  - **O fato de que a comparação de alunos é feita com a matrícula**
  - **O fato de que a matrícula é obtida com getMatrícula()**
  - **O fato de que matrículas são long (representação de dados)**
  - **Como comparar matrículas (com <)**
- O que ocorre se mudarmos qualquer uma dessas coisas?
- Solução possível: mande uma mensagem pro próprio objeto se comparar com outro

Baixo Acoplamento

- Acoplamento de dados

```
class ListaOrdenada {
  Object[] elementosOrdenados = new Object[tamanhoAdequado];

  public void add(Aluno x) {
    // código não mostrado
    if(x.compareTo(elementosOrdenados[K]) < 0) {
      // faça algo
    } else {
      // faça outra coisa
    }
  }
```

Baixo Acoplamento

- Acoplamento de dados
  - **Reduzimos o acoplamento escondendo informação atrás de um método**
  - **Problema: ListaOrdenada só funciona com Aluno**
  - **Aprimoramento da solução: use *interfaces* para desacoplar mais ainda**

Baixo Acoplamento

```
interface Comparable {
  public int compareTo(Object outro);
}

class Aluno implements Comparable {
  public int compareTo(Object outro) {
    // compare registro de aluno com outro
    // retorna valor < 0, 0, ou > 0 dependendo da comparação
  }
}

class ListaOrdenada {
  Object[] elementosOrdenados = new Object[tamanhoAdequado];

  public void add(Comparable x) {
    // código não mostrado
    if(x.compareTo(elementosOrdenados[K]) < 0) {
      // faça algo
    } else {
      // faça outra coisa
    }
  }
```

Baixo Acoplamento

- Acoplamento de controle
  - **Passar "flags" de controle entre objetos de forma que um objeto controle as etapas de processamento de outro objeto**
  - **Ocorrência comum:**
    - Objeto a manda uma mensagem para objeto b
    - b usa um parâmetro da mensagem para decidir o que fazer

Baixo Acoplamento

- Acoplamento de controle

```
class Lampada {
  public final static int ON = 0;

  public void setLampada(int valor) {
    if(valor == ON) {
      // liga lampada
    } else if(valor == 1) {
      // desliga lampada
    } else if(valor == 2) {
      // pisca
    }
  }
}

Lampada lampapa = new Lampada();
lampada.setLampada(Lampada.ON);
lampada.setLampada(2);
```

Baixo Acoplamento

- Acoplamento de controle
  - **Solução: decompor a operação em múltiplas operações primitivas**

```
class Lampada {
  public void on() { // liga lampada }
  public void off() { // desliga lampada }
  public void pisca() { // pisca }
}

Lampada lampada = new Lampada();
lampada.on();
lampada.pisca();
```

Baixo Acoplamento

- Acoplamento de controle
  - **Ocorrência comum:**
    - Objeto a manda mensagem para objeto b
    - b retorna informação de controle para a
    - Exemplo: retorno de código de erro

```
class Teste {
  public int printFile(File aImprimir) {
    if(aImprimir está corrompido ) {
      return CORRUPTFLAG;
    }
    // etc. etc.
  }
}

Teste umTeste = new Teste();
int resultado =  umTese.printFile(miniTeste);
if(resultado == CORRUPTFLAG) {
  // oh! oh!
} else if(resultado == -243) {
  // etc. etc.
```

Baixo Acoplamento

- Acoplamento de controle
  - **Solução: Exceções**

```
class Teste {
  public int printFile(File aImprimir) throws PrintExeception {
    if(aImprimir está corrompido ) {
      throw new PrintExeception();
    }
    // etc. etc.
  }
}

try {
  Teste umTeste = new Teste();
  umTeste.printFile(miniTeste);
} catch(PrintException printError) {
  // mail para a turma: não tem miniteste amanhã!
}
```

Baixo Acoplamento

- Acoplamento de dados globais
  - **Dois ou mais objetos compartilham estruturas de dados globais**
  - **É um acoplamento muito ruim pois está "escondido"**
    - Uma chamada de método pode mudar um valor global e o código não deixa isso aparente
  - **Por essas e outras evitamos variáveis globais**

Baixo Acoplamento

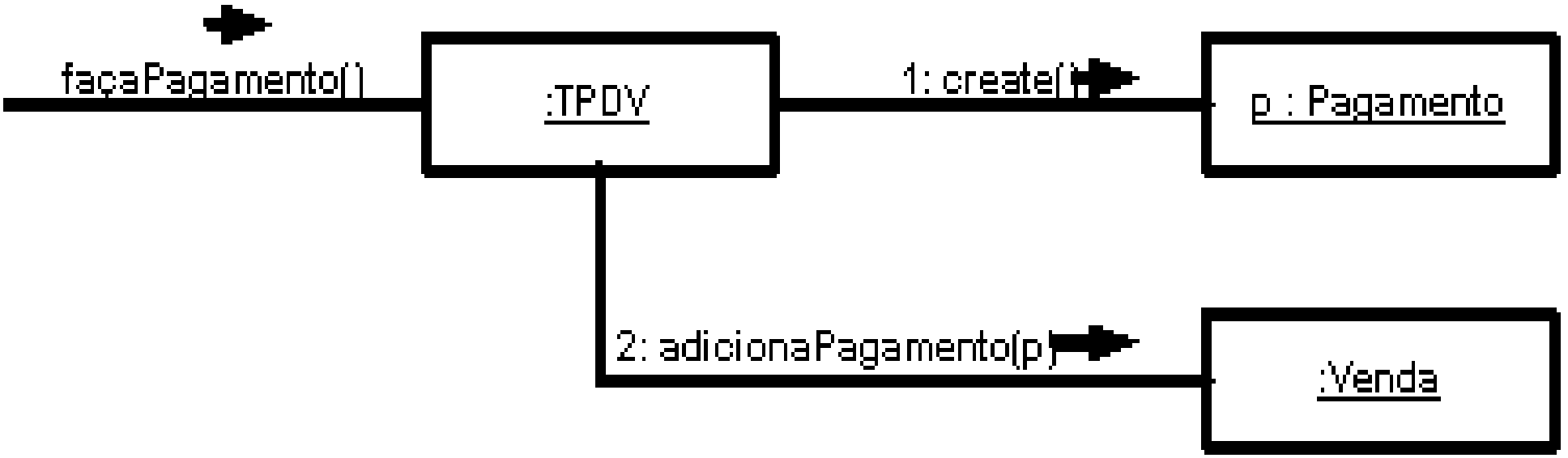
- Acoplamento de dados internos
  - **Um objeto altera os dados locais de um outro objeto**
  - **Ocorrência comum:**
    - Friend Class em C++
    - Dados públicos, package visibility ou mesmo protected em Java
    - Falta de *getters* and *setters*
  - **Use com MUITO cuidado!**
  - **Interação com atributos de um objeto deve ser sempre controlada e baseada numa *interface* bem definida**

Alta Coesão

- Problema: Como gerenciar a complexidade?
- A coesão mede quão relacionadas ou focadas estão as responsabilidades da classe
  - **Também chamada de "coesão funcional" (ver à frente)**
- Uma classe com baixa coesão faz muitas coisas não relacionadas e leva aos seguintes problemas:
  - **Difícil de entender**
  - **Difícil de reusar**
  - **Difícil de manter**
  - **"Delicada": constantemente sendo afetada por outras mudanças**
- Uma classe com baixa coesão assumiu responsabilidades que pertencem a outras classes

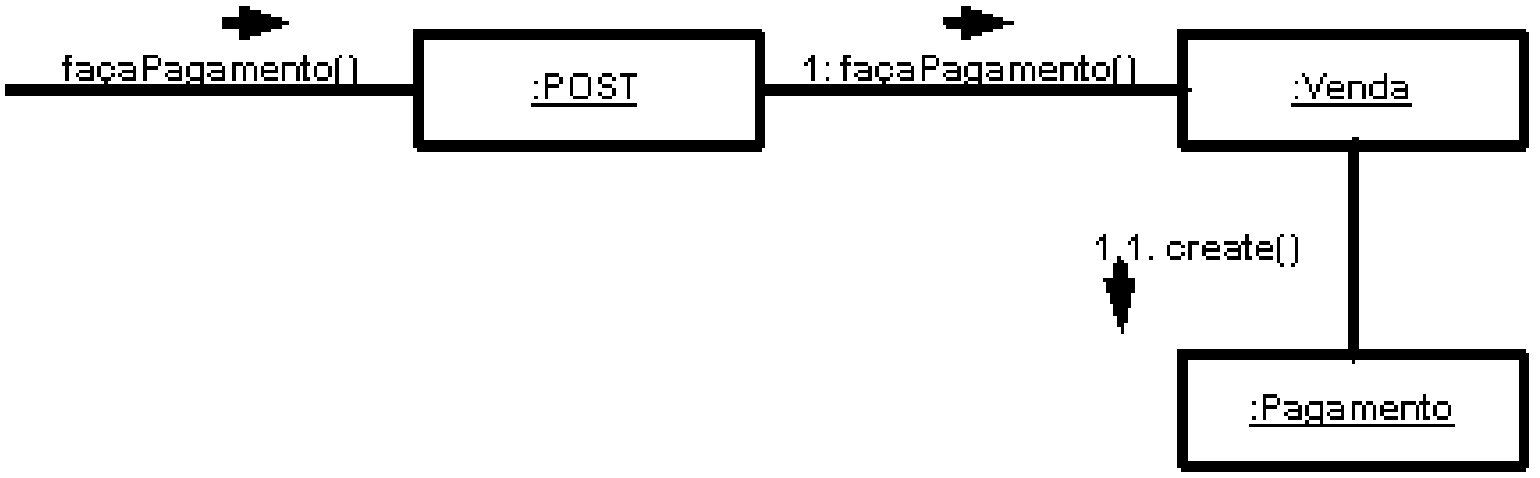
Alta Coesão

- Solução: Atribuir responsabilidades que mantenham alta coesão
- Exemplo: Voltemos ao exemplo do acoplamento
  - **Na primeira alternativa, TPDV assumiu uma responsabilidade de efetuar um pagamento (método façaPagamento())**



Alta Coesão

- Até agora, não há problema, mas suponha que o mesmo ocorra com várias outras operações de sistema
  - **TPDV vai acumular um monte de métodos não muito focados**
  - **Resultado: baixa coesão**
- A segunda alternativa delega façaPagamento() para a classe Venda
  - **Mantém maior coesão em TPDV**



Alta Coesão

- Discussão
  - **Alta coesão é outro princípio de ouro que deve ser sempre mantido em mente durante o projeto**
  - **Tipos de coesão entre módulos**
    - Coincidente (pior)
    - Lógico
    - Temporal
    - Procedural
    - De comunicação
    - Sequencial
    - Funcional (melhor)

Alta Coesão

- Coesão Coincidente
  - **Há nenhuma (ou pouca) relação construtiva entre os elementos de um módulo**
  - **No linguajar OO: Um objeto não representa nenhum conceito OO**
  - **Uma coleção de código comumente usado e herdado através de herança (provavelmente múltipla)**

```
class Angu {
  public static int acharPadrão(String texto, String padrão) {
    // ...
  }
  public static int média(Vector números) {
    // ...
  }
  public static outputStream abreArquivo(string nomeArquivo) {
    // ...
  }
}

class Xpto extends Angu { // quer aproveitar código de Angu
  ...
}
```

Alta Coesão

- Coesão Lógica
  - **Um módulo faz um conjunto de funções relacionadas, uma das quais é escolhida através de um parâmetro ao chamar o módulo**
  - **Semelhante a acoplamento de controle**
  - **Cura: quebrar em métodos diferentes**

```
public void faça(int flag) {
  switch(flag) {
  case ON:
    // coisas para tratar de ON
    break;
  case OFF:
    // coisas para tratar de OFF
    break;
  case FECHAR:
    // coisas para tratar de FECHAR
    break;
  case COR:
    // coisas para tratar de COR
    break;
  }
}
```

Alta Coesão

- Coesão Temporal
  - **Elementos estão agrupados no mesmo módulo porque são processados no mesmo intervalo de tempo**
  - **Exemplos comuns:**
  - **Método de inicialização que provê valores *default* para um monte de coisas diferentes**
  - **Método de finalização que limpa as coisas antes de terminar**

```
procedure inicializaDados() {
   font = "times";
   windowSize = "200,400";
   xpto.nome = "desligado";
   xpto.tamanho = 12;
   xpto.localização = "/usr/local/lib/java";
}
```

Alta Coesão

- Coesão Temporal
  - **Cura, nesses casos dos exemplos: construtores e destrutores**

```
class Xpto {
   public Xpto() {
      this.nome = "desligado";
      this.tamanho = 12;
      this.localização = "/usr/local/lib/java";
   }
}
```

Alta Coesão

Coesão Procedural

- **Associa elementos de acordo com seus relacionamentos procedurais ou algorítmicos**
- **Um módulo procedural depende muito da aplicação sendo tratada**
  - Junto com a aplicação, o módulo parece razoável
  - Sem este contexto, o módulo parece estranho e muito difícil de entender
    - **"O que está acontecendo aqui!?"**
- **Não pode entender o módulo sem entender o programa e as condições que existem quando o módulo é chamado**
- **Cura: reprojete/refatore o sistema**

Alta Coesão

- Coesão de Comunicação
  - **Todas as operações de um módulo operam no mesmo conjunto de dados e/ou produzem o mesmo tipo de dado de saída**
  - **Cura: isole cada elemento num módulo separado**
  - **"Não deveria" ocorrer em sistemas OO usando polimorfismo (classes diferentes para fazer tratamentos diferentes nos dados)**

Alta Coesão

- Coesão Sequencial
  - **A saída de um elemento de um módulo serve de entrada para o próximo elemento**
  - **Cura: decompor em módulos menores (ou aglutinar módulos, se houver coesão funcional)**

Alta Coesão

- Coesão Funcional (a melhor)
  - **Um módulo tem coesão funcional se as operações do módulo puderem ser descritas numa única frase de forma coerente**
  - **Num sistema OO:**
    - Cada operação na interface pública do objeto deve ser funcionalmente coesa
    - Cada objeto deve representar um único conceito coeso
  - **Exemplo: um objeto que esconde algum conceito ou estrutura de dados ou recurso e onde todos os métodos são relacionados por um conceito ou estrutura de dados ou recurso**
    - Meyer chama isso de "information-strength module"

“Padrão Básico“: Interface e Polimorfismo

- Herança de implementação versus herança de interface
  - **Há uma diferença grande entre uma classe e seu tipo**
    - A classe define ambos uma interface e uma implementação dessa interface
    - A interface define apenas a interface oferecida para acessar objetos da classe
    - Um objeto pode ter muitas interfaces, como sabemos
    - Classes diferentes podem implementar a mesma interface

“Padrão Básico“: Interface e Polimorfismo

- Herança de implementação versus herança de interface
  - **Herança de classe ("extends") significa herança de implementação**
    - A sub-classe herda a implementação da super-classe
    - É um mecanismo para compartilhar código e representação
  - **Herança de interface (ou subtipos) descreve quando um objeto pode ser usado em vez de outro**
  - **Ao fazer herança de classe, automaticamente faz também herança de interface**
  - **Algumas linguagens não permitem definir tipos separadamente de classes**
    - Mas, neste caso, a classe puramente abstrata serve

“Padrão Básico“: Interface e Polimorfismo

- "Program to an interface, not an implementation"
  - **O fato de que a herança de implementação permite facilmente reusar a funcionalidade de uma classe é interessante mas não é o aspecto mais importante a ser considerado**
  - **Herança oferece a habilidade de definir famílias de objetos com interfaces idênticas**
  - **Isso é extremamente importante pois permite desacoplar um objeto de seus clientes através do polimorfismo**
  - **A herança de interface corretamente usada (sem eliminar partes da interface nas sub-classes) acaba criando subtipos, permitindo o polimorfismo**

“Padrão Básico“: Interface e Polimorfismo

- "Program to an interface, not an implementation"
  - **Programar em função de uma interface e não em função de uma implementação (uma classe particular) permite que:**
    - Clientes permanecem sem conhecimento do tipo de objetos que eles usam, desde que os objetos obedeçam a interface
    - Clientes permanecem sem conhecimento das classes
  - **A interface é o que há de comum entre as várias situações**
    - A flexibilidade vem da possibilidade de mudar a implementação da interface, até em tempo de execução (*late binding*)
  - **Em Java, uma classe pode implementar várias interfaces**
    - Isso permite ter mais polimorfismo mesmo sem que as classes pertençam a uma mesma hierarquia

“Padrão Básico“: Interface e Polimorfismo

- Exemplos
  - **Temos vários tipos de composites (coleções) que não pertencem a uma mesma hierarquia**
    - ColeçãoDeAlunos
    - ColeçãoDeProfessores
    - ColeçãoDeDisciplinas
  - **Temos um cliente comum dessas coleções**
    - Digamos um selecionador de objetos usado numa interface gráfica para abrir uma **list box** para selecionar objetos com um determinado nome

“Padrão Básico“: Interface e Polimorfismo

- Exemplos
  - **Exemplo:**
    - Quero listar todos os alunos com nome "João" e exibí-los numa *list box* para escolha pelo usuário
    - Idem para listar professores com nome "Leandro"
    - Idem para listar disciplinas com nome "Programação"
  - **Queremos fazer um único cliente para qualquer uma das coleções**
  - **O exemplo a seguir tem polimorfismo em dois lugares**

“Padrão Básico“: Interface e Polimorfismo

- Exemplos

```
ComponenteDeSelecao cds =
      new ComponenteDeSelecao(umaColeçãoDeAlunos, "João");
response.out.println(cds.geraListBox());

cds = new ComponenteDeSelecao(umaColeçãoDeDisciplinas, "Programação");
response.out.println(cds.geraListBox());
```

“Padrão Básico“: Interface e Polimorfismo

- Exemplos

```
interface SelecionavelPorNome {
  Iterator getIteradorPorNome(String nome);
}

interface Nomeavel {
  String getNome();
}

class ColecaoDeAlunos implements SelecionavelPorNome {
  // ...
  Iterator getIteradorPorNome(String nome) {
    // ...
  }
}

class Aluno implements Nomeavel {
  // ...
  String getNome() { ... }
}
```

“Padrão Básico“: Interface e Polimorfismo

- Exemplos

```
class ComponenteDeSelecao {
  Iterator it;
  // observe o tipo do parâmetro (uma interface)
  public ComponenteDeSelecao(SelecionavelPorNome coleção, String nome) {
    it = coleção.getIteradorPorNome(nome); // chamada polimórfica
  }
  // ...
  String geraListBox() {
    StringBuffer resposta = new StringBuffer();
    resposta.append("<select name=\"nome\" size=\"1\">");
    while(it.hasNext()) {
      int i = 1;
      // observe o tipo do objeto
      Nomeavel obj = (Nomeavel)it.next();
      resposta.append("<option value=\"escolha" + i + "\">" +
                      obj.getNome() + // chamada polimórfica
                      "</option>");
    }
    resposta.append("</select>");
    return resposta.toString();
  }
}
```

“Padrão Básico“: Interface e Polimorfismo

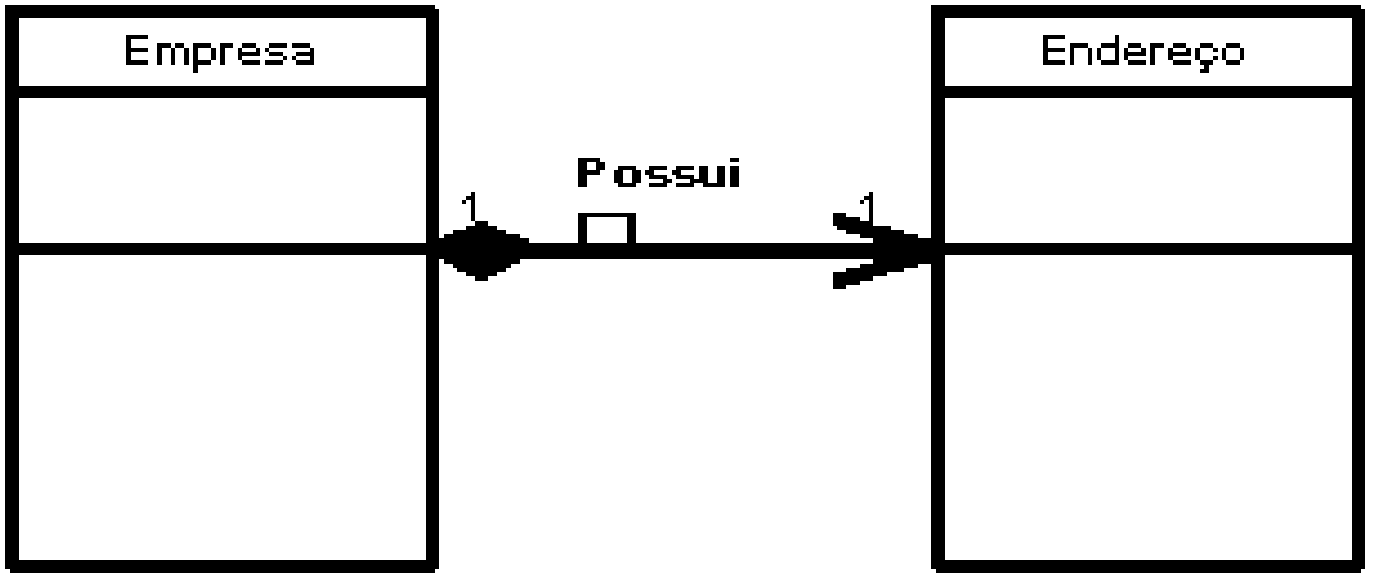
- Como achar interfaces
  - **Procure assinaturas repetidas**
    - Exemplo: várias classes que representam coisas que podem ser vendidas indicam o uso de uma interface VendávelIF
  - **Onde há delegação, um objeto se "esconde" atrás de outro: deve haver uma interface comum**
  - **Procure métodos que poderiam ser usados em aplicações semelhantes e crie interfaces para que a reusabilidade das classes clientes seja maior**
    - Exemplo: muitas coisas poderiam ser reserváveis, não só passagens de avião
    - Exemplo: muitas coisas pode ser alugadas, não só fitas de vídeo
    - Exemplo: muitos objetos são clonáveis
  - **Considere mudanças futuras para pensar em interfaces (novos objetos que poderiam aparecer)**
    - Mas cuidado com *overthinking*, já que tudo pode mudar

“Padrão Básico“: Herança vs Composição

- Composição e Herança
  - **Composição e herança são dois mecanismos para reutilizar funcionalidade**
  - **Alguns anos atrás (e na cabeça de alguns programadores ainda!), a herança era considerada a ferramenta básica de extensão e reuso de funcionalidade**
  - **A composição estende uma classe pela delegação de trabalho para outro objeto**
  - **a herança estende atributos e métodos de uma classe**
  - **Alguns consideram que a composição é muito superior à herança na maioria dos casos**
    - Emula um pouco os padrões Strategy e Decorator
  - **A herança deve ser utilizada em alguns contextos bem definidos, mais restritos ainda que *interfaces***

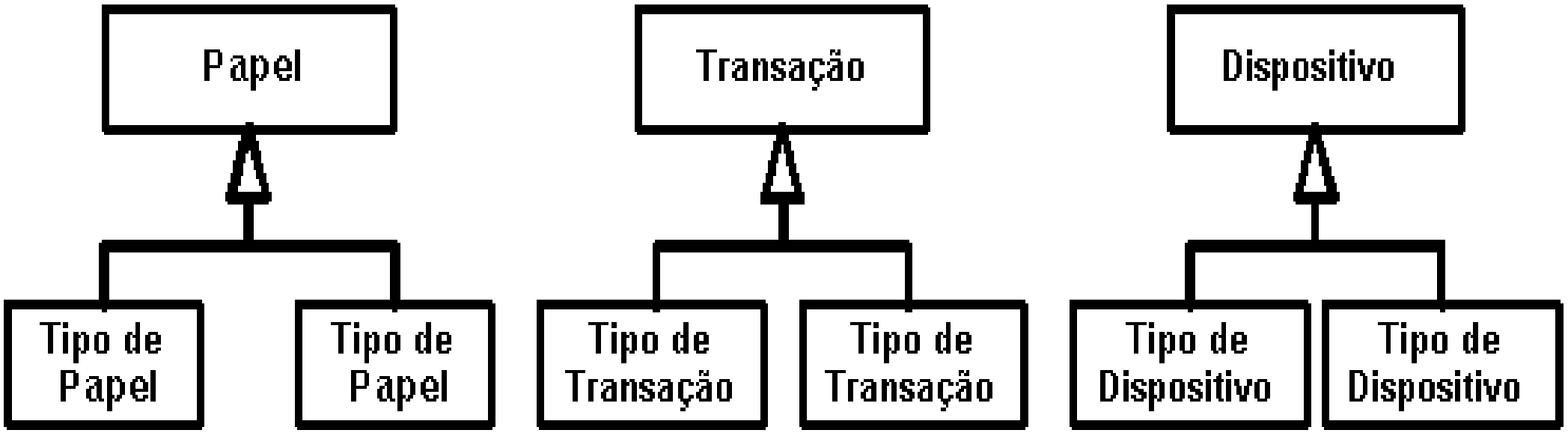
“Padrão Básico“: Herança vs Composição

- Um exemplo de composição
  - **Use composição para estender as responsabilidades pela delegação de trabalho a outros objetos**
  - **Um exemplo no domínio de endereços**
    - Uma empresa tem um endereço (digamos só um)
    - Uma empresa "tem" um endereço
    - Podemos deixar o objeto empresa responsável pelo objeto endereço e temos agregação composta (composição)



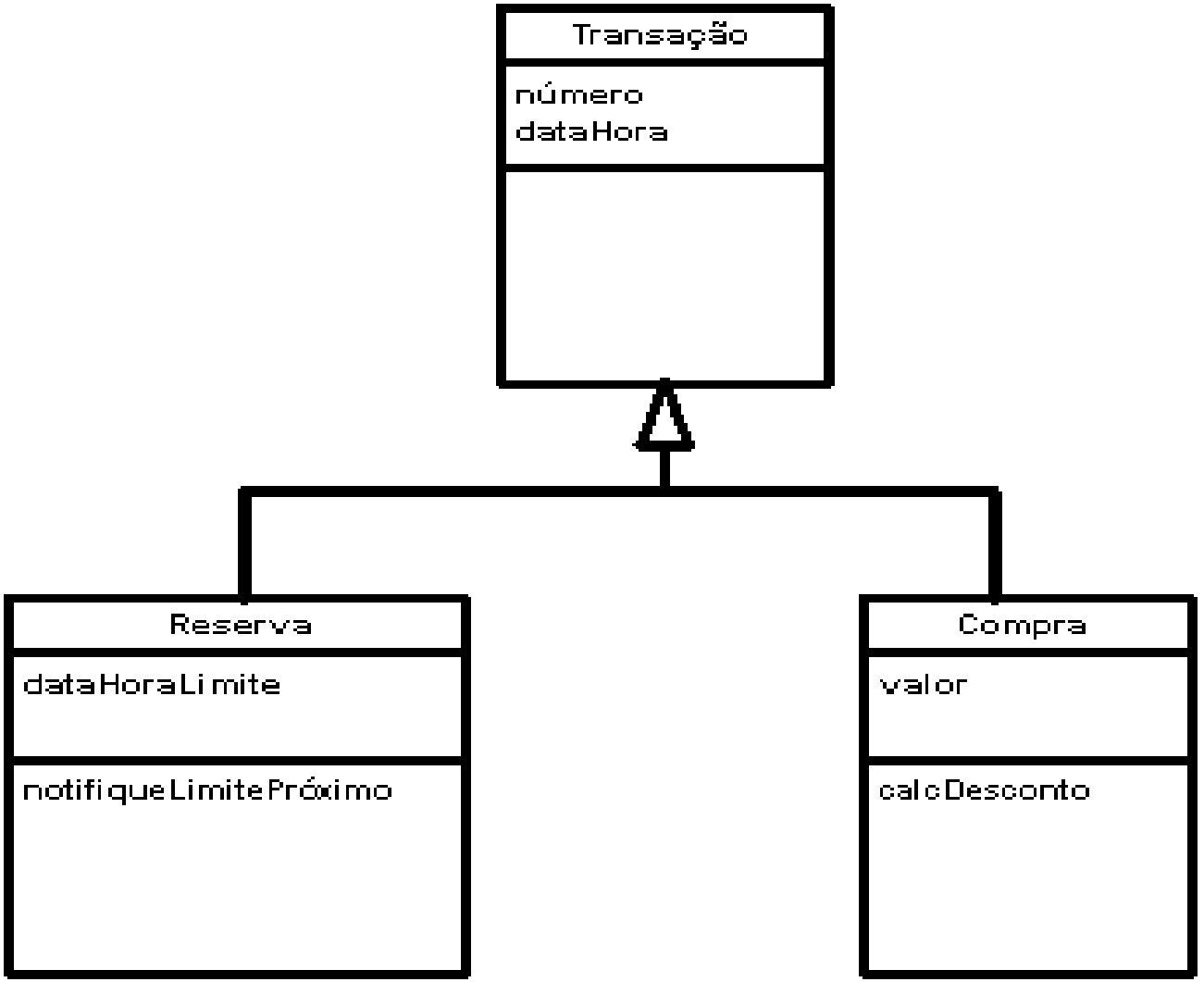
“Padrão Básico“: Herança vs Composição

- Um exemplo de herança
  - **Atributos, conexões a objetos e métodos comuns vão na superclasse (classe de generalização)**
  - **Adicionamos mais dessas coisas nas subclasses (classes de especialização)**
  - **Três situações comuns para a herança (figura abaixo)**
    - Uma transação é um momento notável ou intervalo de tempo



“Padrão Básico“: Herança vs Composição

- Um exemplo de herança
  - **Exemplo no domínio de reserva e compra de passagens de avião**



“Padrão Básico“: Herança vs Composição

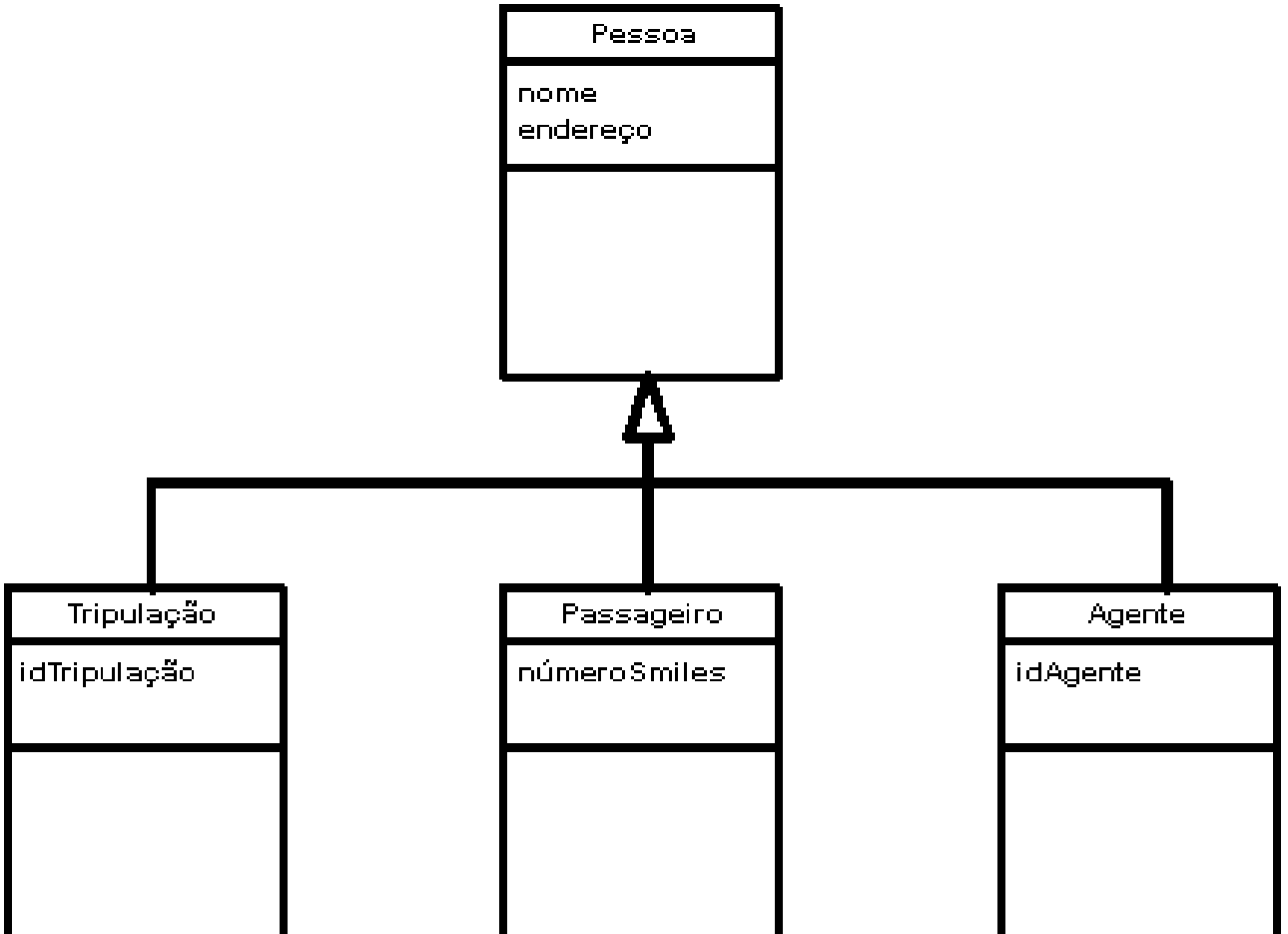
- Vantagens da Herança
  - **Captura o que é comum e o isola daquilo que é diferente**
  - **A herança é mais visível diretamente no código**

“Padrão Básico“: Herança vs Composição

- Problemas da Herança
  - **O encapsulamento entre classes e subclasses é fraco (o acoplamento é forte)**
    - Mudar uma superclasse pode afetar todas as subclasses
      - **The weak base-class problem**
    - Isso viola um dos princípios básicos de OO (manter fraco acoplamento)
  - **Às vezes um objeto precisa ser de uma classe diferente em momentos diferentes**
    - Com herança, a estrutura está parafusada no código e não pode sofrer alterações facilmente em tempo de execução
    - A herança é um relacionamento estático que não muda com tempo
    - Cenário: pessoas envolvidas na aviação (figura a seguir)

“Padrão Básico“: Herança vs Composição

- Problemas da Herança
  - **Problema: uma pessoa pode mudar de papel a assumir combinações de papéis**
  - **Fazer papeis múltiplos requer 7 combinações (subclasses)**

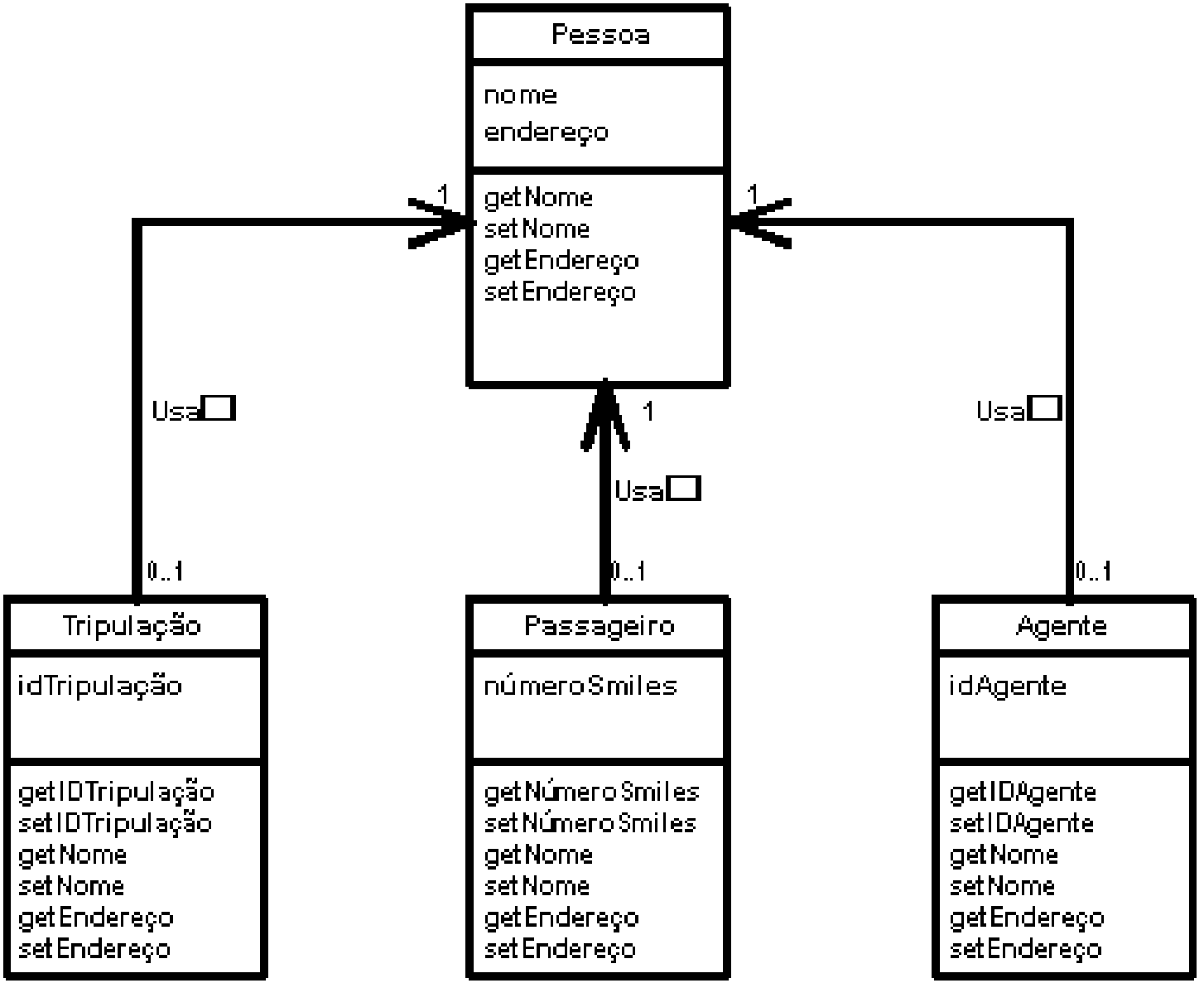


“Padrão Básico“: Herança vs Composição

- Problemas da Herança
  - **Já vimos como solucionar esse problema com o padrão Decorator**

“Padrão Básico“: Herança vs Composição

- Solucionando com Composição



“Padrão Básico“: Herança vs Composição

- Solucionando com Composição
  - **Estamos estendendo a funcionalidade de Pessoa de várias formas, mas sem usar herança**
  - **Observe que também podemos inverter a composição (uma pessoa tem um ou mais papeis)**
    - Pense na implicação para a interface de "pessoa"

“Padrão Básico“: Herança vs Composição

- Solucionando com Composição
  - **Aqui, estamos usando delegação: dois objetos estão envolvidos em atender um pedido (digamos setNome)**
    - O objeto tripulação (digamos) delega setNome para o objeto pessoa que ele **tem** por composição
    - Técnica também chamada de *forwarding*
      - **Semelhante a uma subclasse delegar uma operação para a superclasse (herdando a operação)**
        - Delegação sempre pode ser usada para substituir a herança
    - Se usássemos herança, o objeto tripulação poderia referenciar a pessoa com **this**
    - Com o uso de delegação, tripulação pode passar **this** para Pessoa e o objeto Pessoa pode referenciar o objeto original se quiser
    - Em vez de tripulação **ser** uma pessoa, ele **tem** uma pessoa

“Padrão Básico“: Herança vs Composição

- Solucionando com Composição
  - **A grande vantagem da delegação é que o comportamento pode ser escolhido em tempo de execução e vez de estar amarrado em tempo de compilação**
    - Evitamos também sub-classes "forçadas"
    - Quando queremos aproveitar apenas parte de uma super-classe
    - Queremos menos acoplamento
  - **A grande desvantagem é que um software muito dinâmico e parametrizado pode ser mais difícil de entender do que software mais estático**

“Padrão Básico“: Herança vs Composição

- O Resultado de usar Composição
  - **Em vez de codificar um comportamento estaticamente, definimos pequenos comportamentos padrão e usamos composição para definir comportamentos mais complexos**
  - **De forma geral, a composição é melhor do que herança normalmente, pois:**
    - Permite mudar a associação entre classes em tempo de execução;
    - Permite que um objeto assuma mais de um comportamento (ex. papel);
    - Herança acopla as classes demais e engessa o programa

“Padrão Básico“: Herança vs Composição

- 5 regras para o uso de herança (Coad's rules)
  - **O objeto "é um tipo especial de" e não "um papel assumido por"**
  - **O objeto nunca tem que mudar para outra classe**
  - **A subclasse estende a superclasse mas não faz override ou anulação de muitas variáveis e/ou métodos**
  - **Não é uma subclasse de uma classe "utilitária"**
  - **Para classes do domínio do problema, a subclasse expressa tipos especiais de papéis, transações ou dispositivos**

  - **Exemplo: Considere Agente, Tripulação e Passageiro como subclasses de Pessoa. Viola quais regras?**

“Padrão Básico“: Herança vs Composição

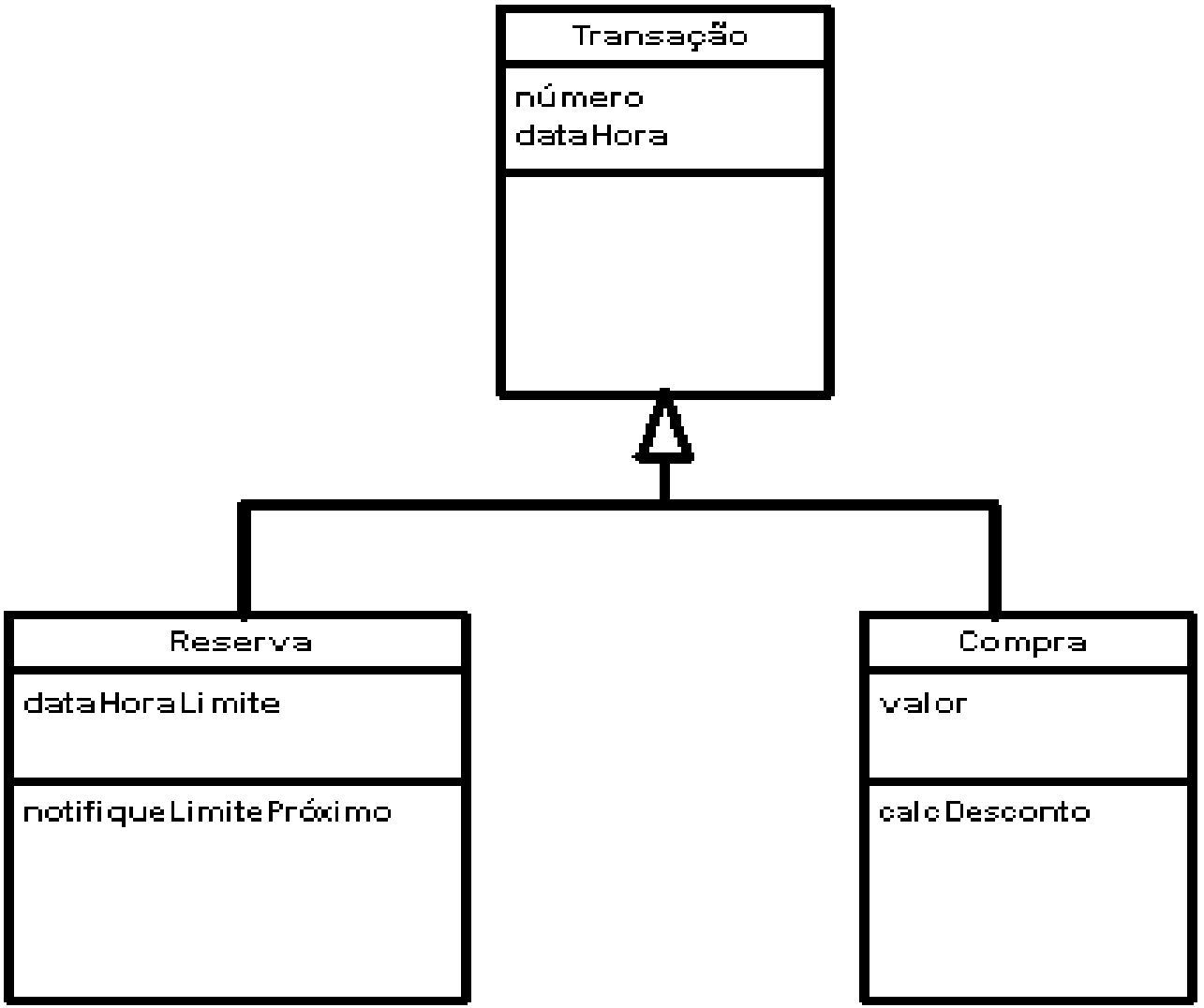
- 5 regras para o uso de herança (Coad's rules)
  - **Não é uma subclasse de uma classe "utilitária"**
    - Não é uma boa idéia fazer isso porque herdar de, digamos, HashMap deixa a classe vulnerável a mudanças futuras à classe HashMap
    - O objeto original não "é" uma HashMap (mas pode usá-la)
      - **Não é uma boa idéia porque enfraquece a encapsulação**
      - **Clientes poderão supor que a classe é uma subclasse da classe utilitária e não funcionarão se a classe eventualmente mudar sua superclasse**
    - Exemplo: x usa y que é subclasse de vector
      - **x usa y sabendo que é um Vector**
      - **Amanhã, y acaba sendo mudada para ser subclasse de HashMap**
      - **x se lasca!**

“Padrão Básico“: Herança vs Composição

- Exemplos de aplicações das regras
  - **Considere Agente, Tripulação e Passageiro como subclasses de Pessoa**
    - Regra 1 (tipo especial): não passa. Um Passageiro não é um tipo especial de Pessoa: é um papel assumido por uma Pessoa
    - Regra 2 (mutação): não passa. Um Agente pode se transformar em Passageiro com tempo
    - Regra 3 (só estende): ok.
    - Regra 4 (não é utility): ok.
    - Regra 5: não passa. Passageiro está sendo modelado como tipo especial de Pessoa e não como tipo especial de papel

“Padrão Básico“: Herança vs Composição

- Outro Exemplo: Transações
  - **Reserva e Compra podem herdar de Transação?**



“Padrão Básico“: Herança vs Composição

- Outro Exemplo: Transações
  - **Reserva e Compra podem herdar de Transação?**
    - Regra 1 (tipo especial): ok. Uma Reserva é um tipo especial de Transação e não um papel assumido por uma Transação
    - Regra 2 (mutação): ok. Uma reserva sempre será uma Reserva, e nunca se transforma em Compra (se houver uma compra da passagem, será outra transação). Idem para Compra: sempre será uma Compra
    - Regra 3 (só estende): ok. Ambas as subclasses estendem Transação com novas variáveis e métodos e não fazem override ou anulam coisas de Transação
    - Regra 4 (não estende classe utilitária): ok.
    - Regra 5 (tipo especial de papel/transação/dispositivo): ok. São tipos especiais de Transação

“Padrão Básico“: Herança vs Composição

- Leitura recomendada

- Maximize your code reusability

  http://www.javaworld.com/javaworld/javatips/jw-javatip107.html

Padrões GRASP

- Resumindo os padrões GRASP:
  - **Controller**
  - **Creator**
  - **Indirection (caso da Classe de Associação)**
  - **Information Expert**
  - **High Cohesion**
  - **Low Coupling**
  - **Polymorphism**
  - **Protected Variations (mudanças com mínimo "estrago")**
  - **Pure Fabrication (classes não relacionadas a Domínio)**

Responsabilidades e Padrões

- Exercício
  - Escreva assinaturas para os métodos do Caso de Uso do slide 23, usando o diagrama de classes do slide 25.
  - Considere os métodos GRASP
    - Onde considerar apropriado, comente suas decisões e raciocínio no código