Programação Orientada a Objetos

Prof. Paulo André Castro
pauloac@ita.br
www.comp.ita.br/~pauloac
ITA – Stefanini

Planejamento

- Aula 1
  - Introdução
  - Conceitos Básicos
    - Classe, Objeto, Método,Herança, interfaces, polimorfismo, Encapsulamento
  - Introdução a linguagem Java
- **Aula 2**
  - **Modelagem de Programas Orientada a Objetos**
  - **Introdução a Padrões de Projeto (Design Patterns)**
  - **Introdução a Ambientes Integrados de Desenvolvimento**
  -
- Aula 3
  - Introdução a Ambientes Integrados de Desenvolvimento
  - Desenvolvimento de Programas Básicos
  - Manipulação de E/S em Java
  - Tipos genéricos

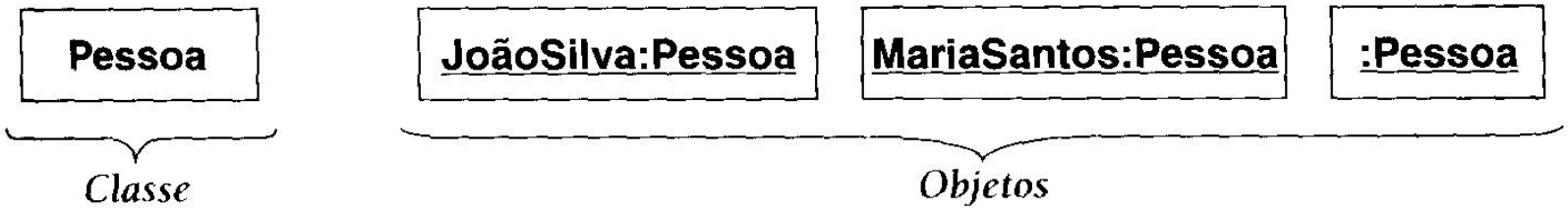
Sumário de Hoje

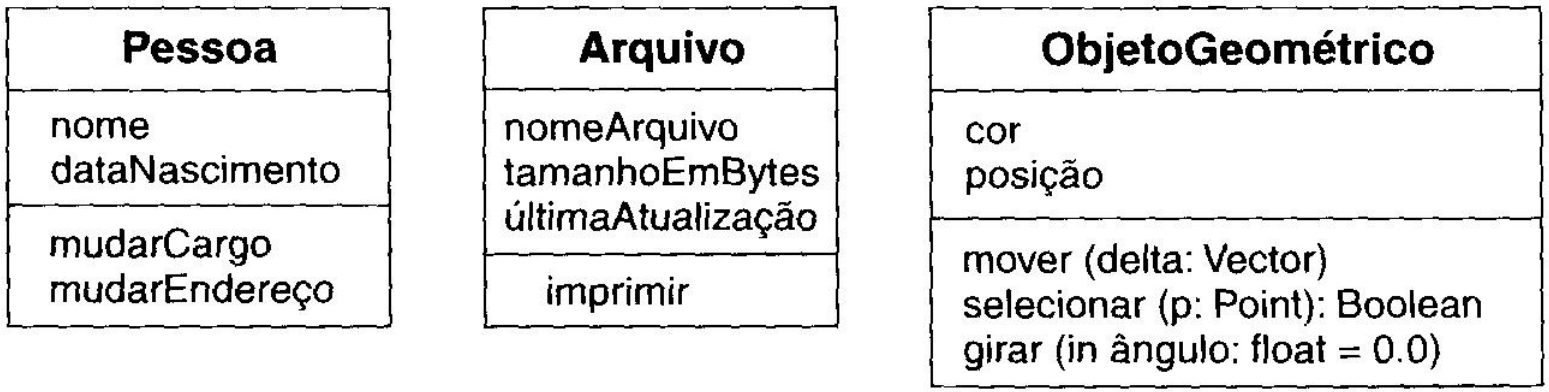
- **Modelagem de Programas Orientada a Objetos**
- Introdução a Padrões de Projeto (Design Patterns)
- Introdução a Ambientes Integrados de Desenvolvimento

Modelagem de Programas Orientada a Objetos

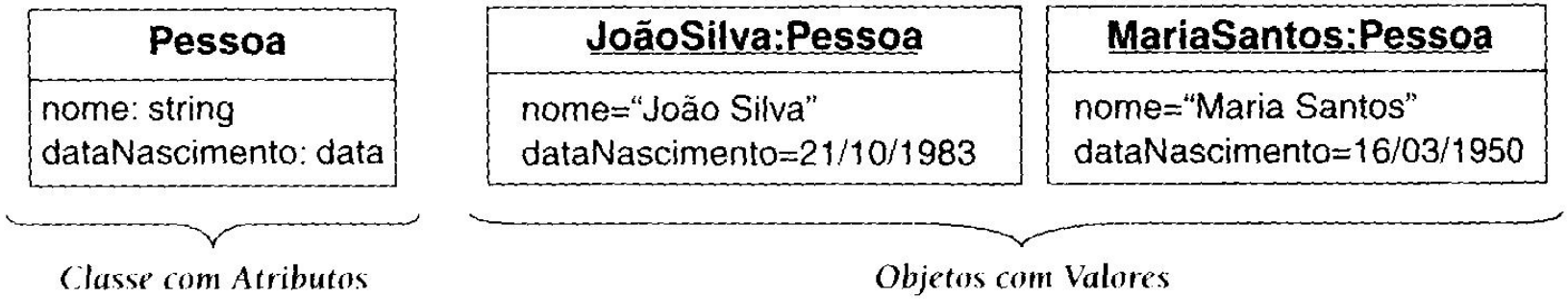
- Modelagem de Conceitos Básicos
- Ligação e Associação
- Generalização e Herança
- Um Exemplo de Modelo de Classes
- Navegação dos Modelos de Classes
- Conceitos Avançados
- Associações N-árias
- Agregação versus Composição
- Classes Abstratas
- Herança Múltipla
- Metadados
- Pacotes
- Dicas Práticas

Classes e Objetos

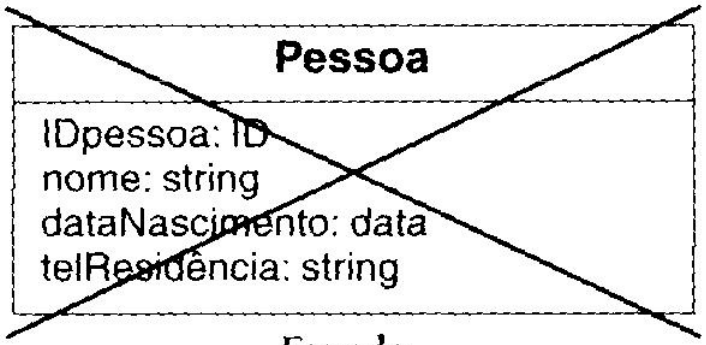




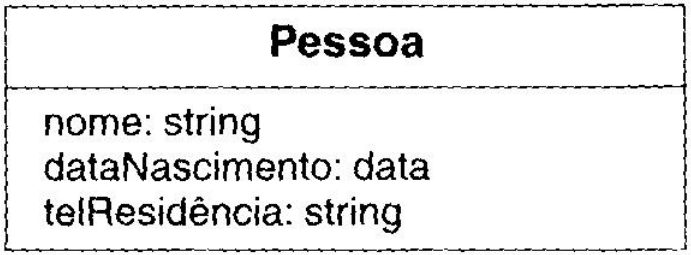
Classes com atributos e Objetos com Valores



Atributos implícitos: identificadores



Errado



Correto

Ligação e Associação

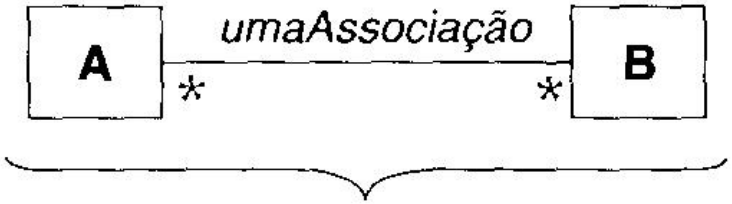


Diagrama de classes

Diagrama de objetos

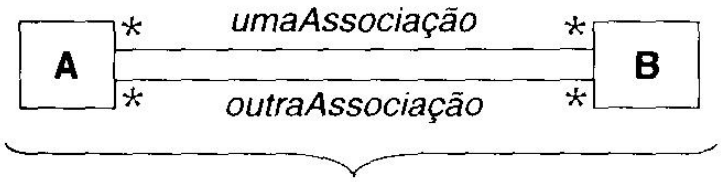


Diagrama de classes

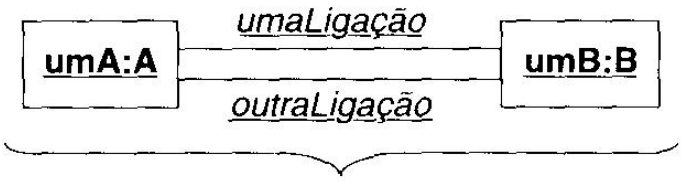
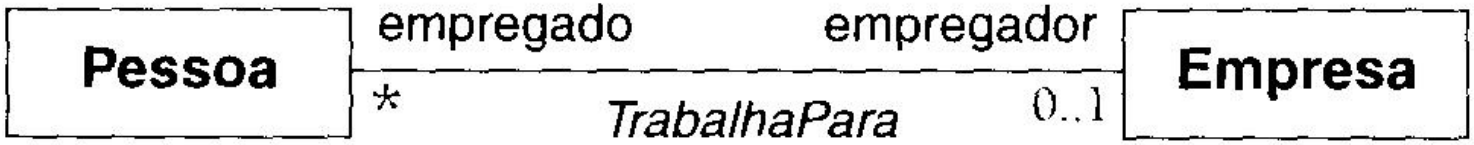
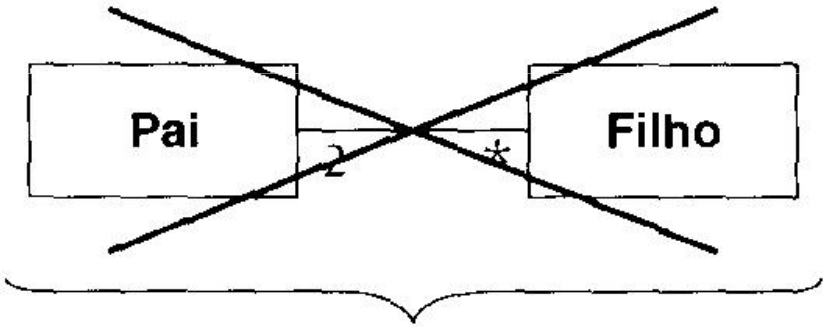


Diagrama de objetos

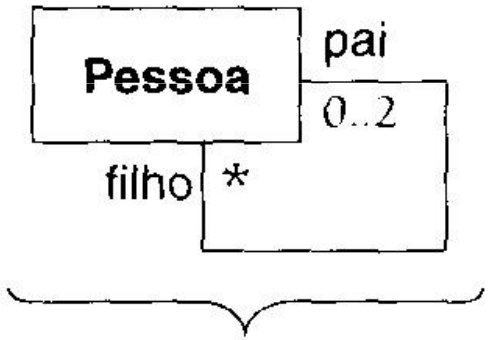
Associações com extremidades nomeadas



Associações entre Classes

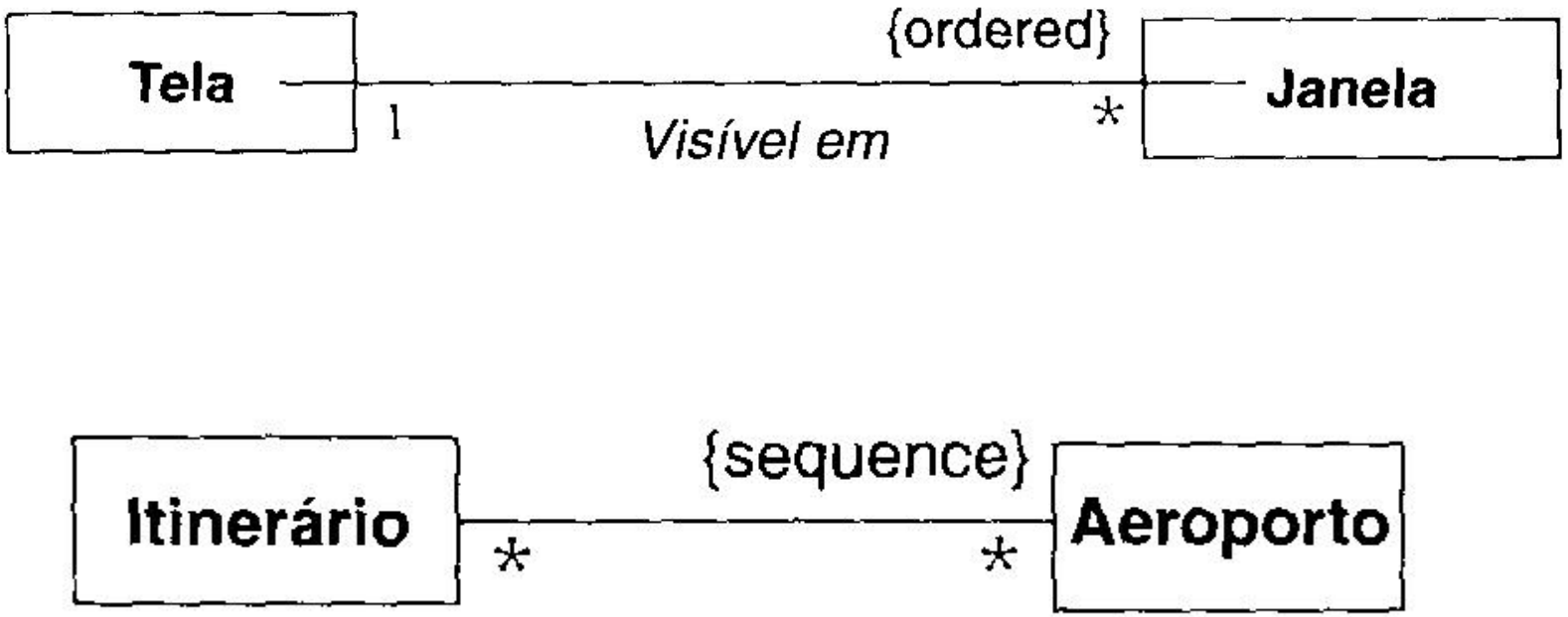


Modelo errado



Modelo correto

Seqüência, bag e ordenação

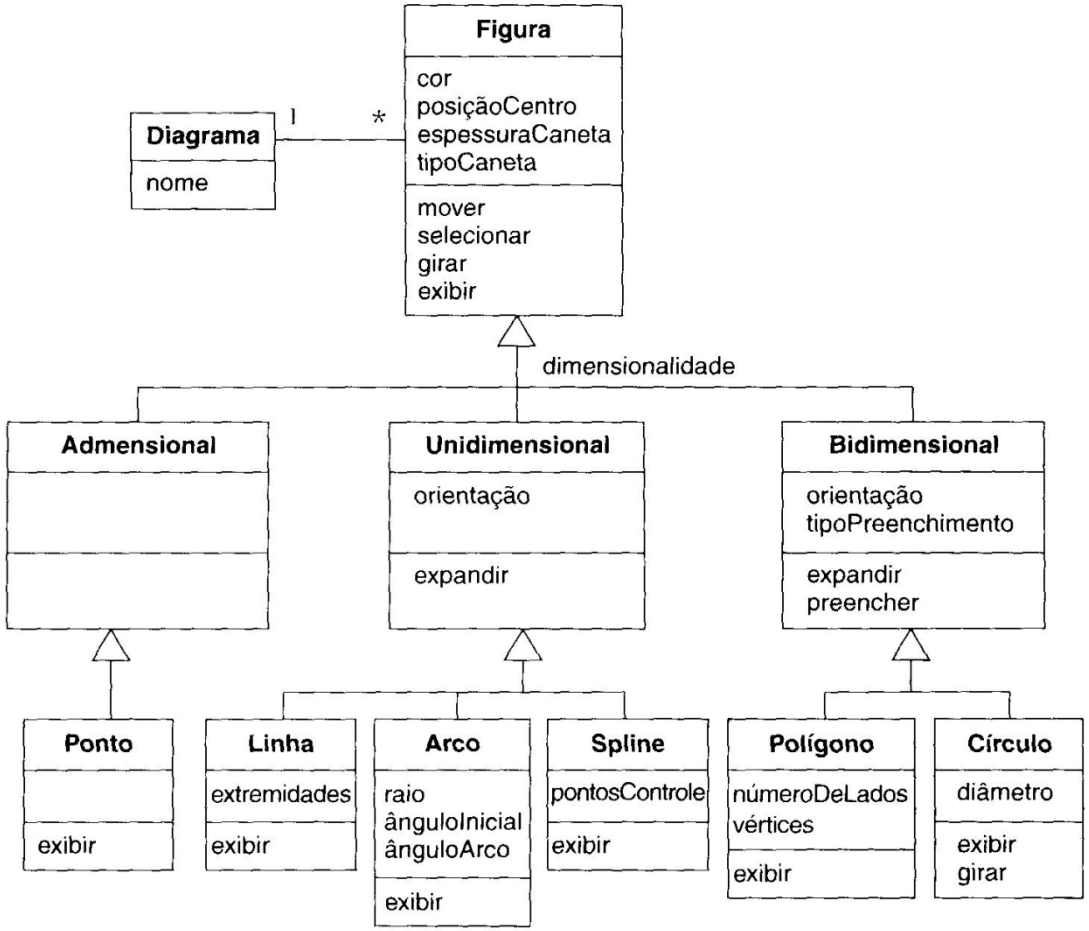


Ordenação = conjunto com ordenação e sem duplicatas

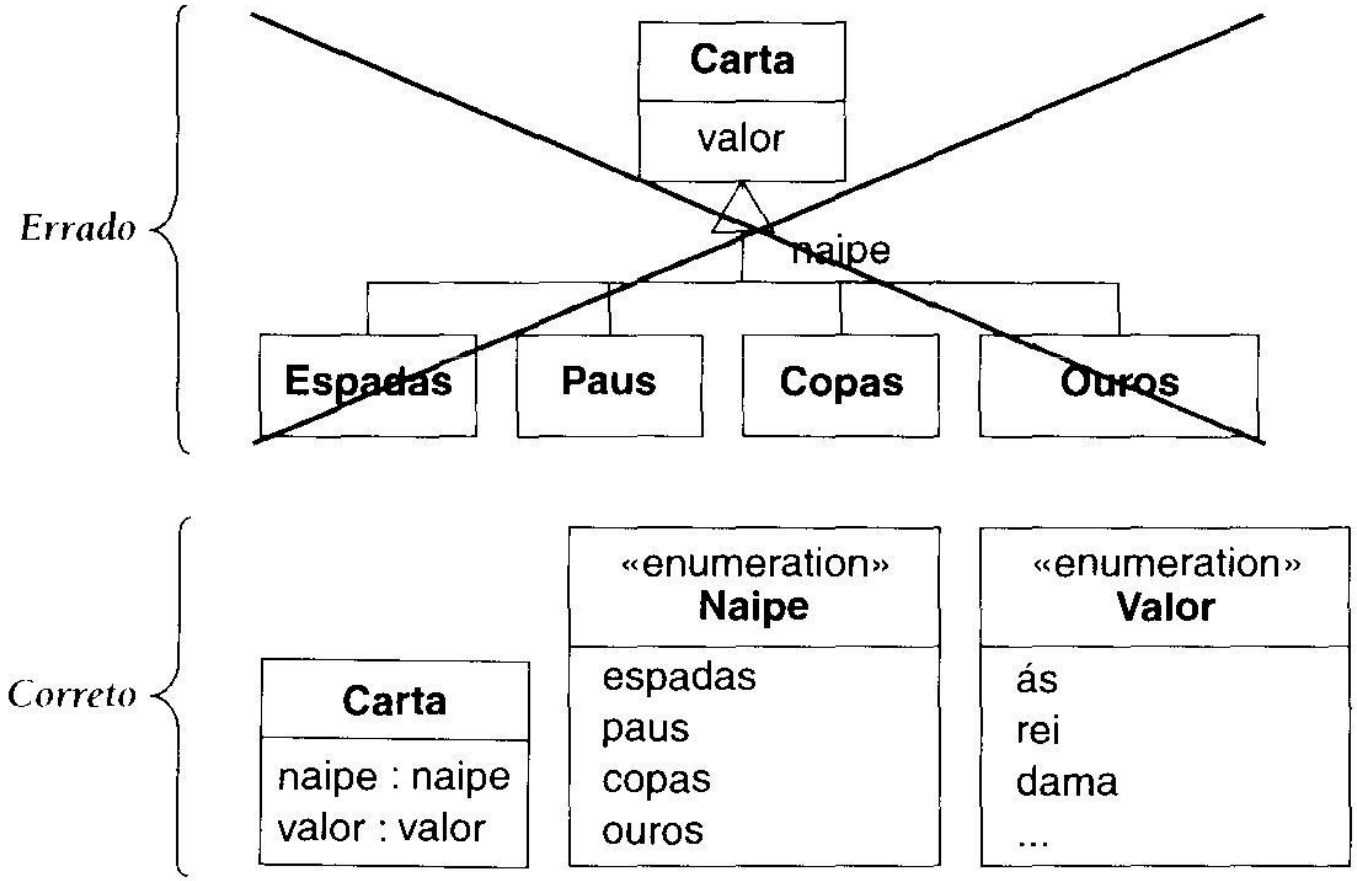
Seqüência= conjunto com ordenação e com duplicatas

Bag = conjunto sem ordenação com duplicatas

Generalização e Herança



Enumerações



Enumerações em Java

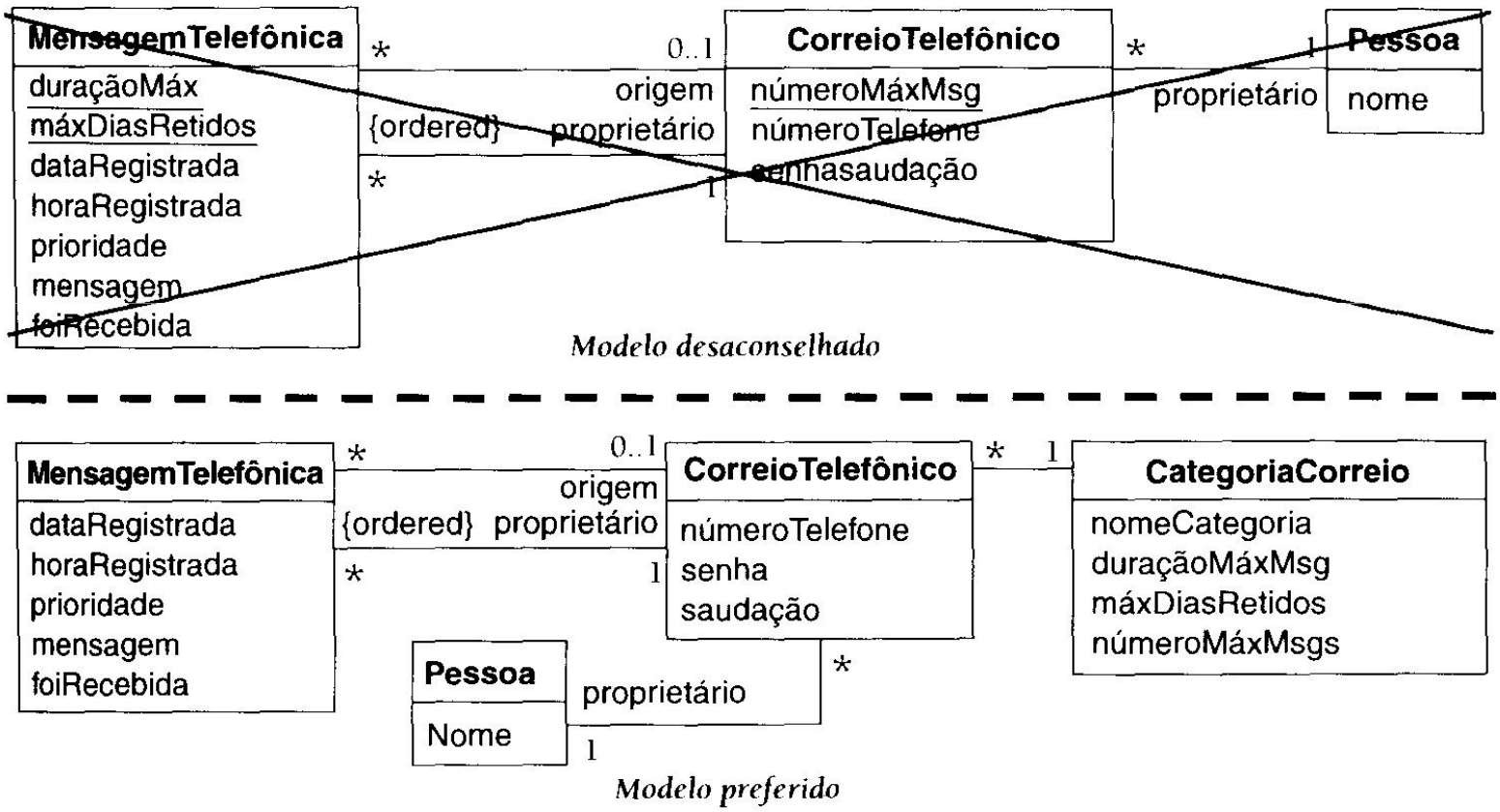
```
enum Naipe { ESPADAS, PAUS, COPAS, OURO}
enum Valor { AS, REI, DAMA, VALETE,
   DEZ,NOVE,OITO,SETE,SEIS,CINCO,QUATRO,TRES,DOIS }

class Carta {
public Naipe naipe;    public Valor valor;
public Carta(Naipe naipe,Valor valor) {
this.naipe=naipe;
this.valor=valor;
}
}
```

Enumerações em Java – cont.

```
public class EnumerationTest {
  public static void main(String[] args) {
          for (Naipe n : Naipe.values())   {
                  for( Valor v: Valor.values()) {
                          System.out.printf("%s de %s\n", v, n);
                  }
           }
           Carta carta=new Carta(Naipe.OURO,Valor.AS);
           System.out.printf("%s de %s", carta.valor,carta.naipe);
    }
}
```

Um Exemplo de Modelo de Classes de um serviço de mensagens telefônicas: grupos ou categorias

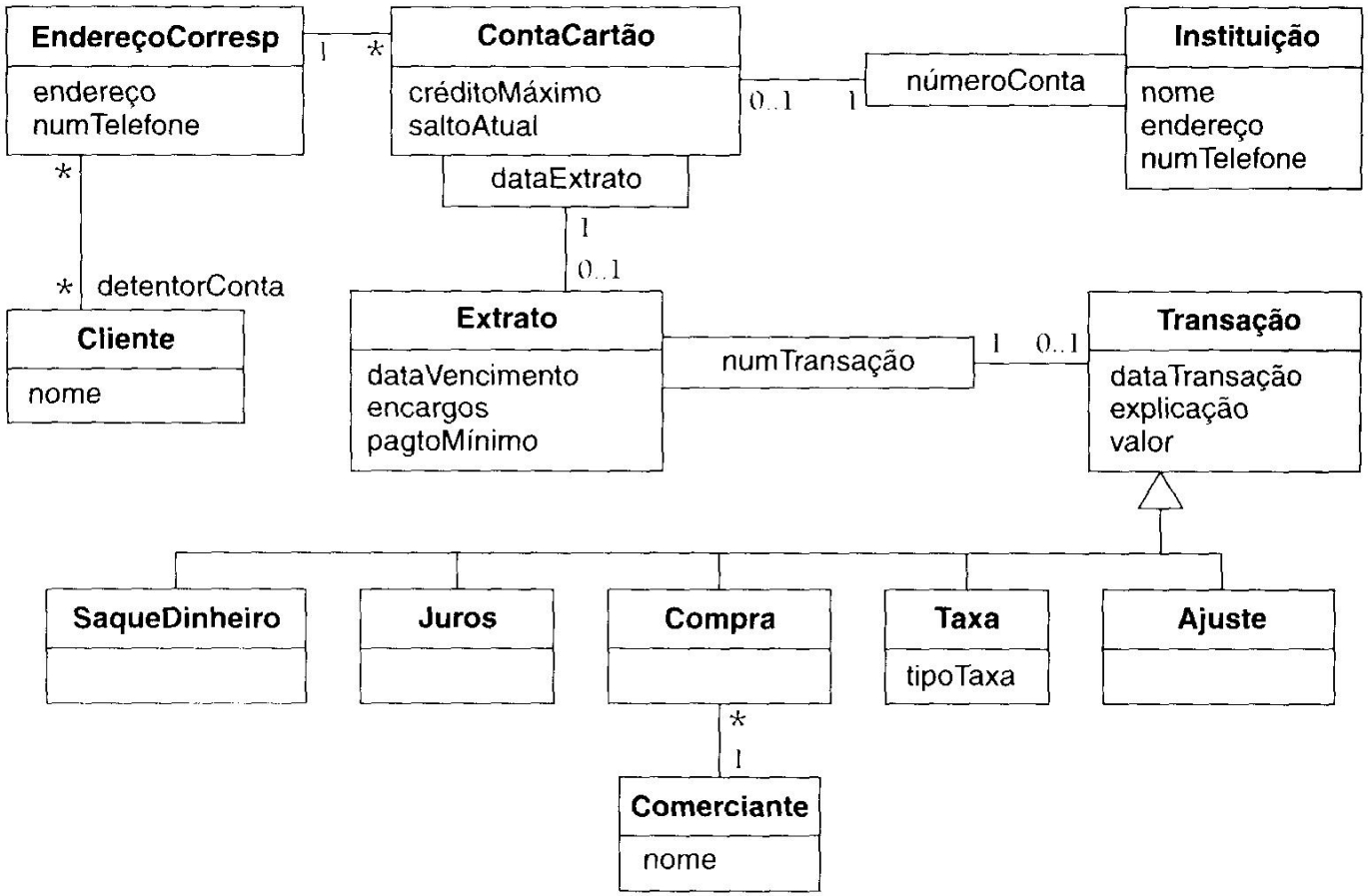


Exercício: sistema de cartão de crédito

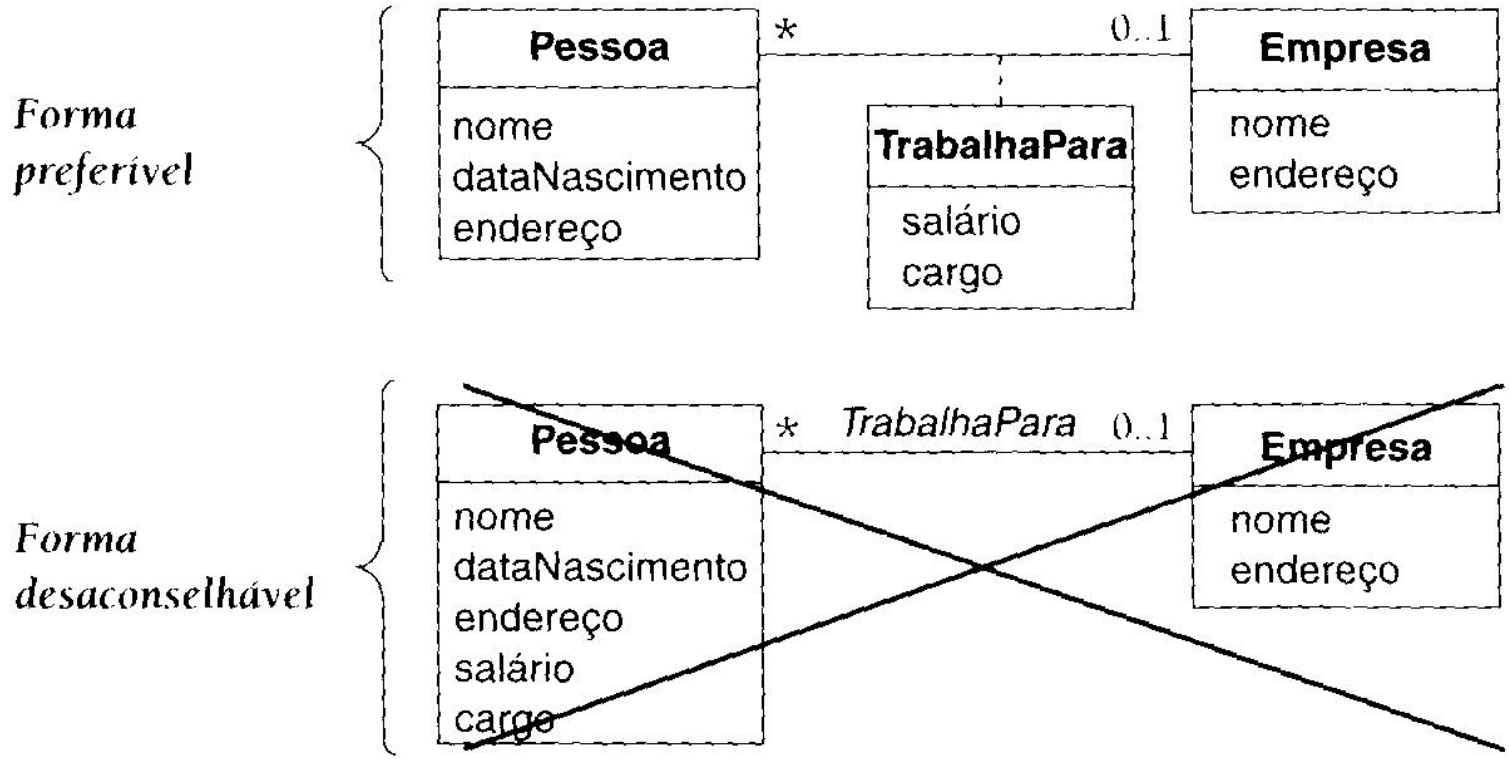
- Modele em uma diagrama de classes um subsistema de informações sobre operações de cartões de crédito contemplando cliente, comerciante, tipos de transação, conta bancária vinculada ao cartão e instituição financeira, extratos do cartão....

- Identifique as classes (conceitos mais relevantes), depois as relações....

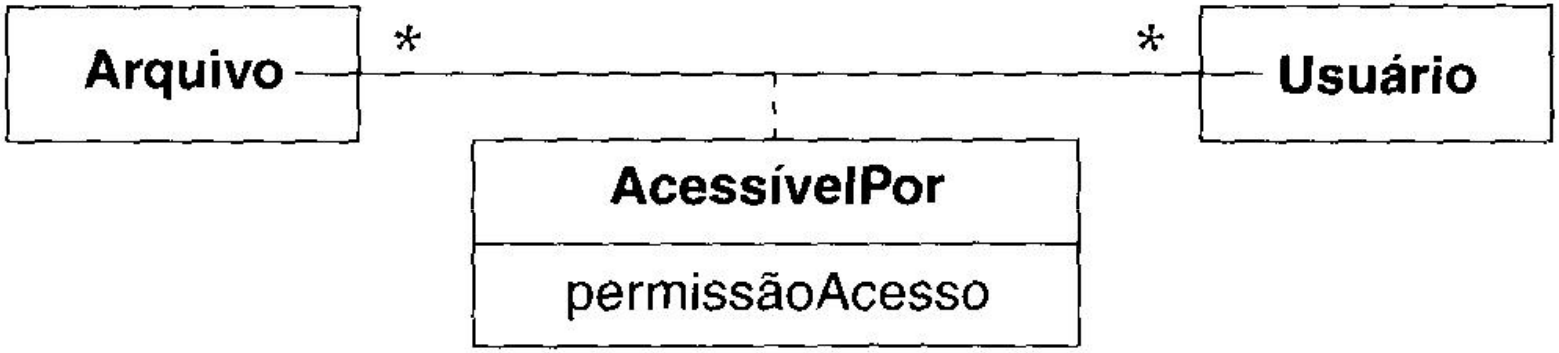
Exemplo: cartão de crédito



Classes de Associação com dados sobre associação

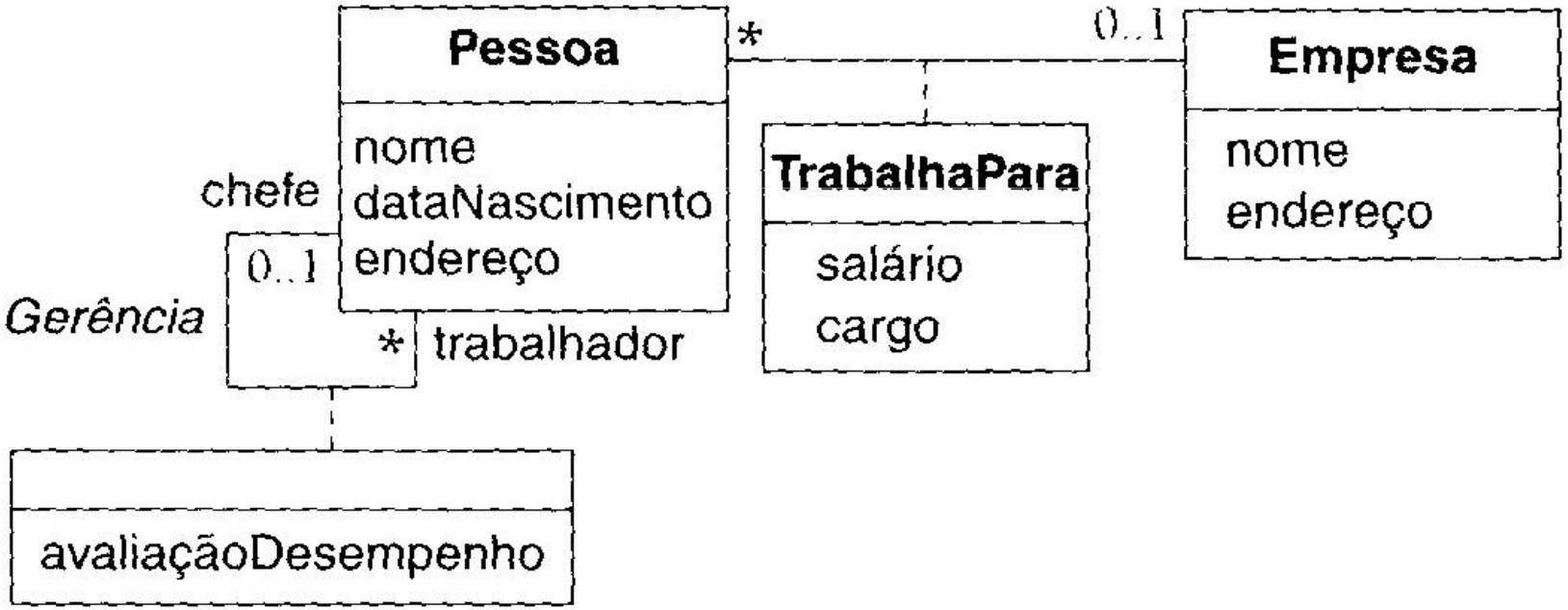


Classes de Associação

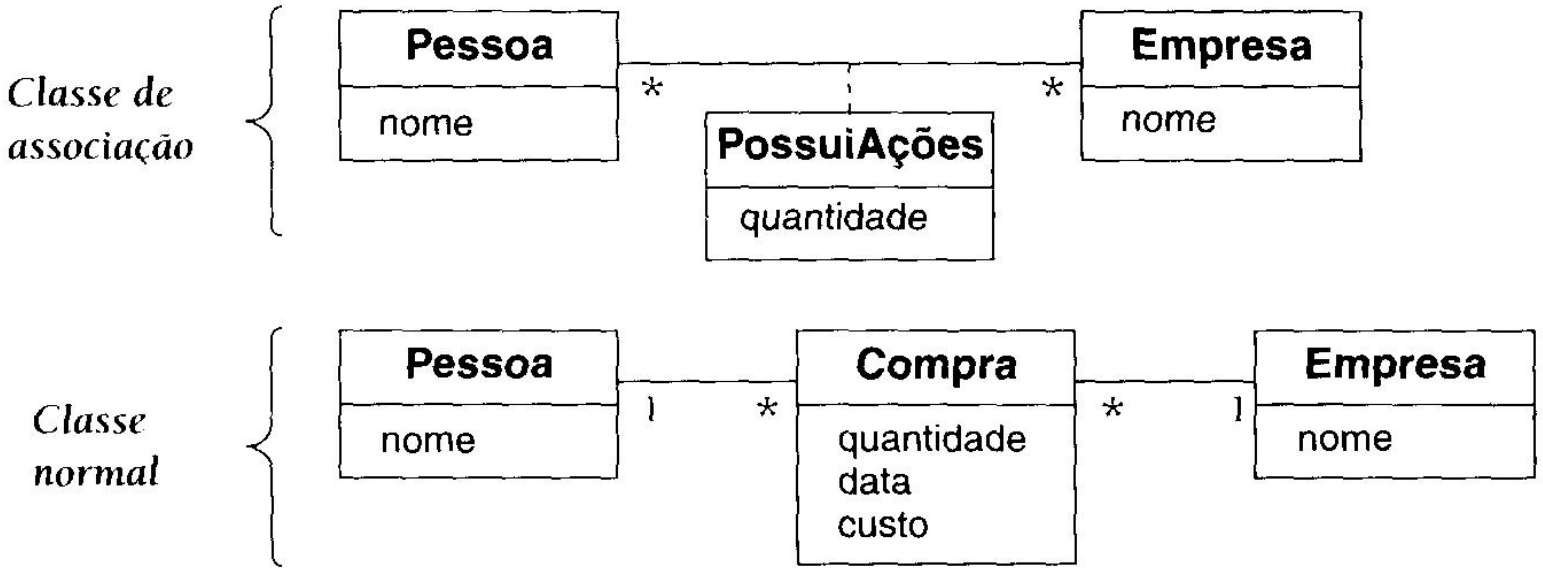


/etc/termcap	read	João da Silva
/etc/termcap	read-write	Maria Brito
/usr/silva/.login	read-write	João da Silva

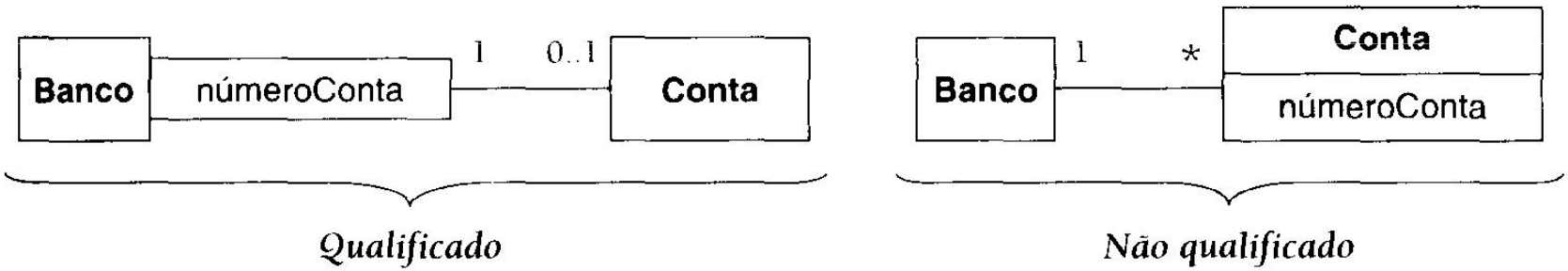
Classes de Associação - 2



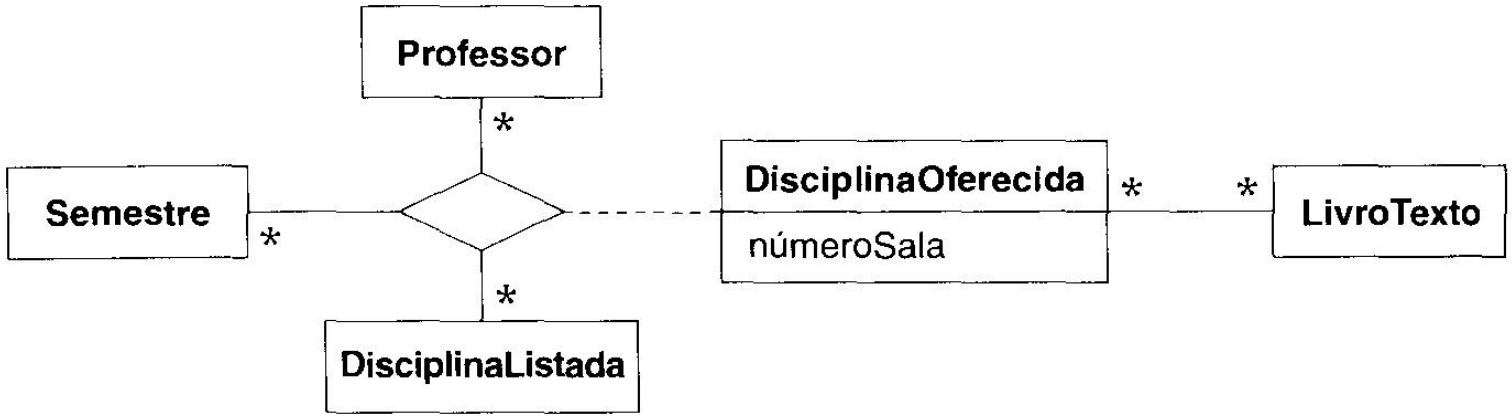
Classes de Associação e Classes Normais



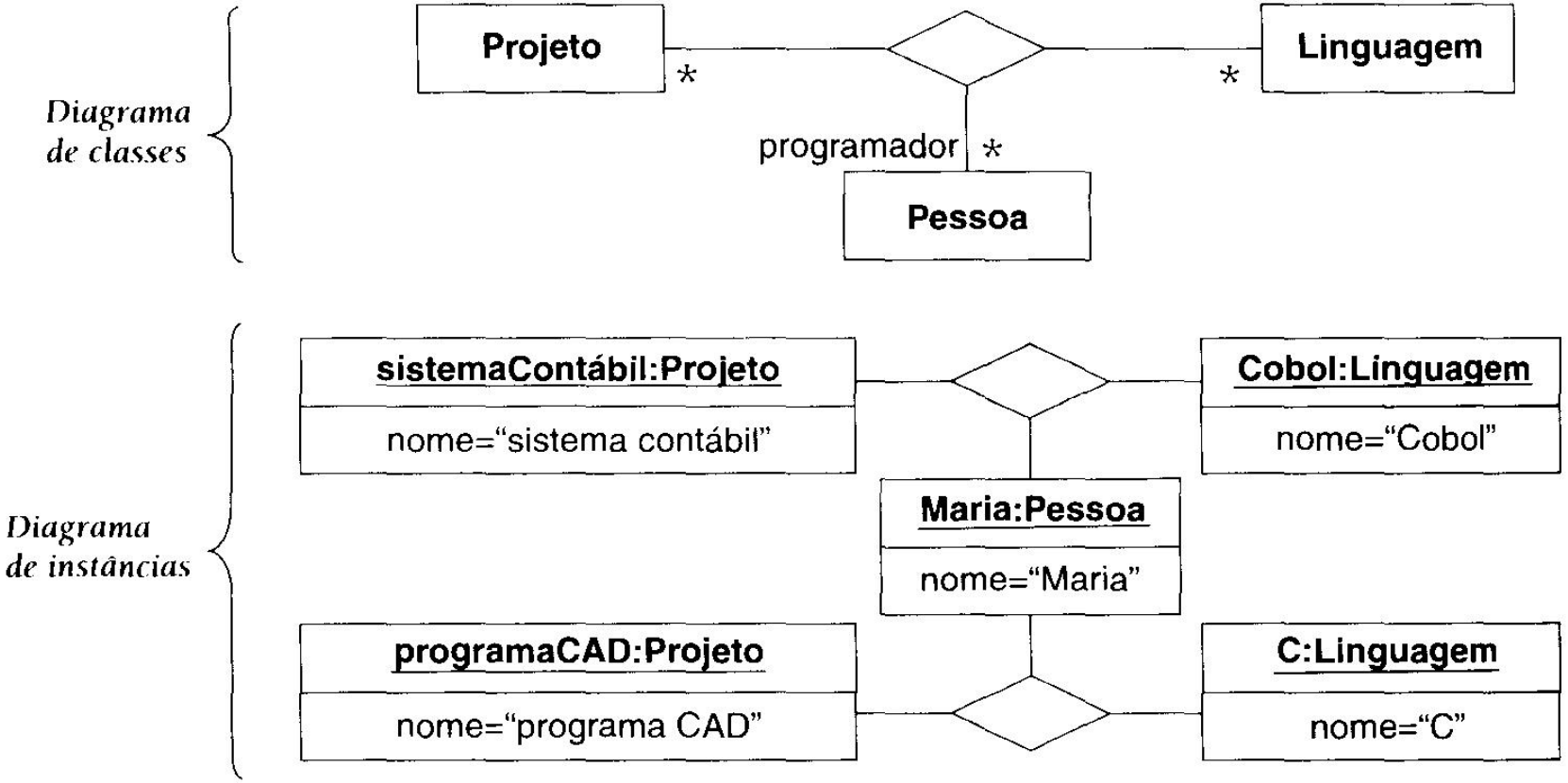
Associações Qualificadas



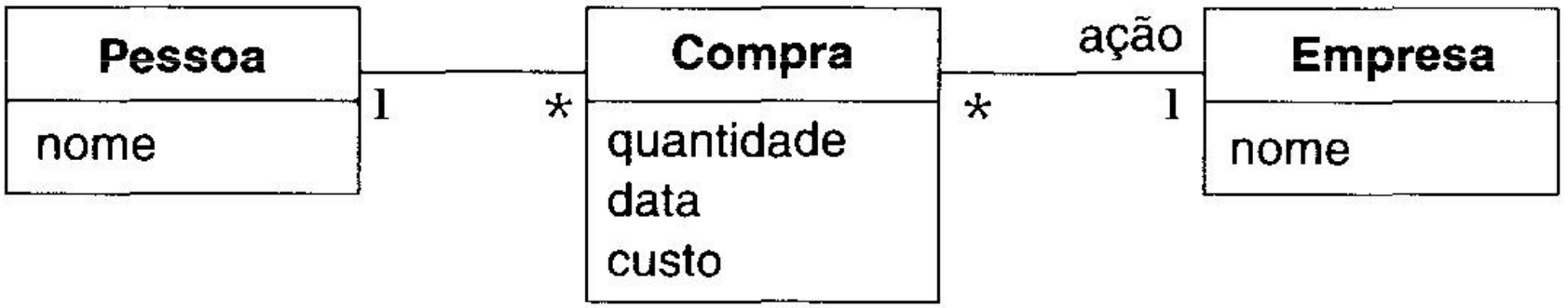
Associações N-árias Verdadeiras



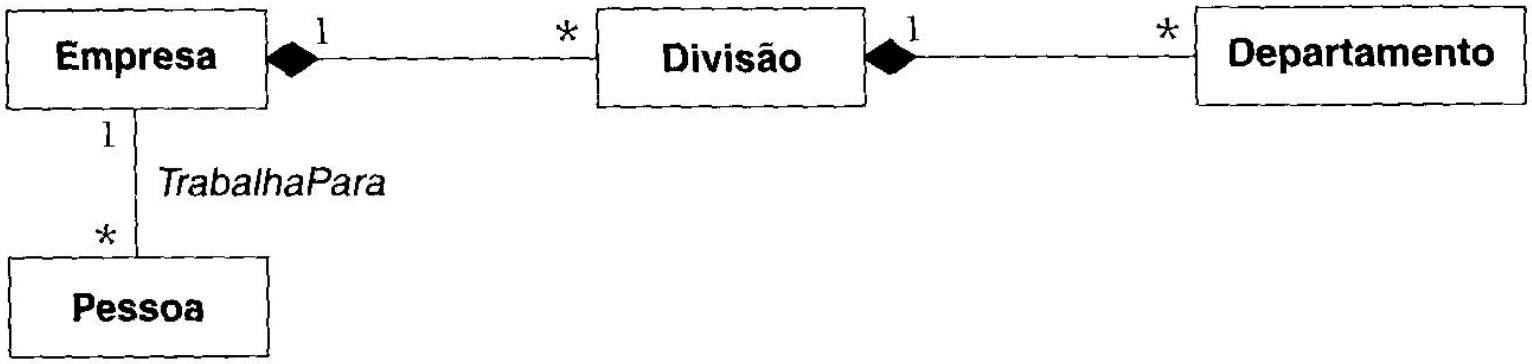
Associações n-árias



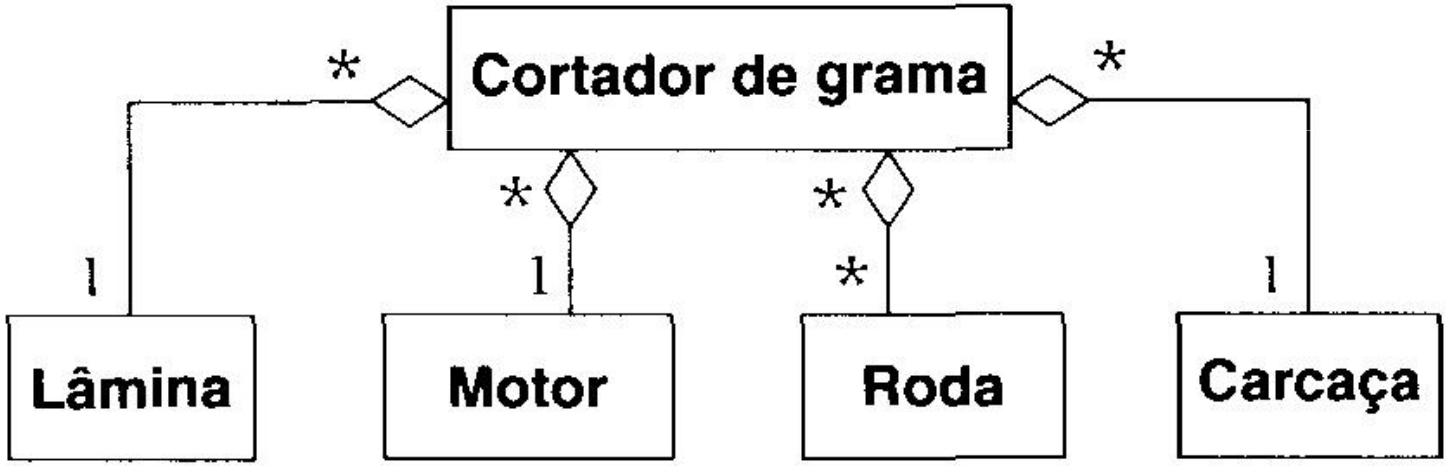
De Associação n-ária para binárias



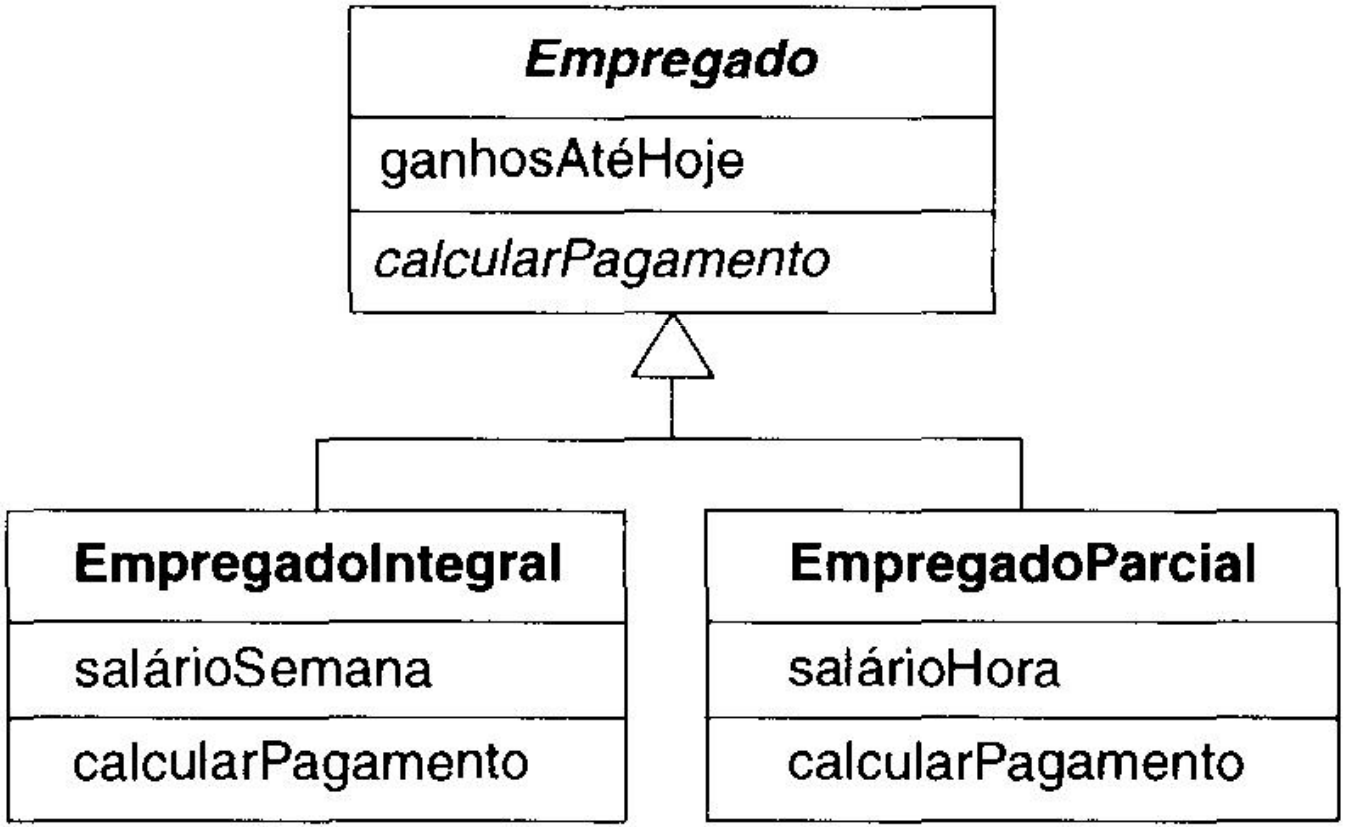
Agregação versus Composição: Composição



Agregação versus Composição: Agregação



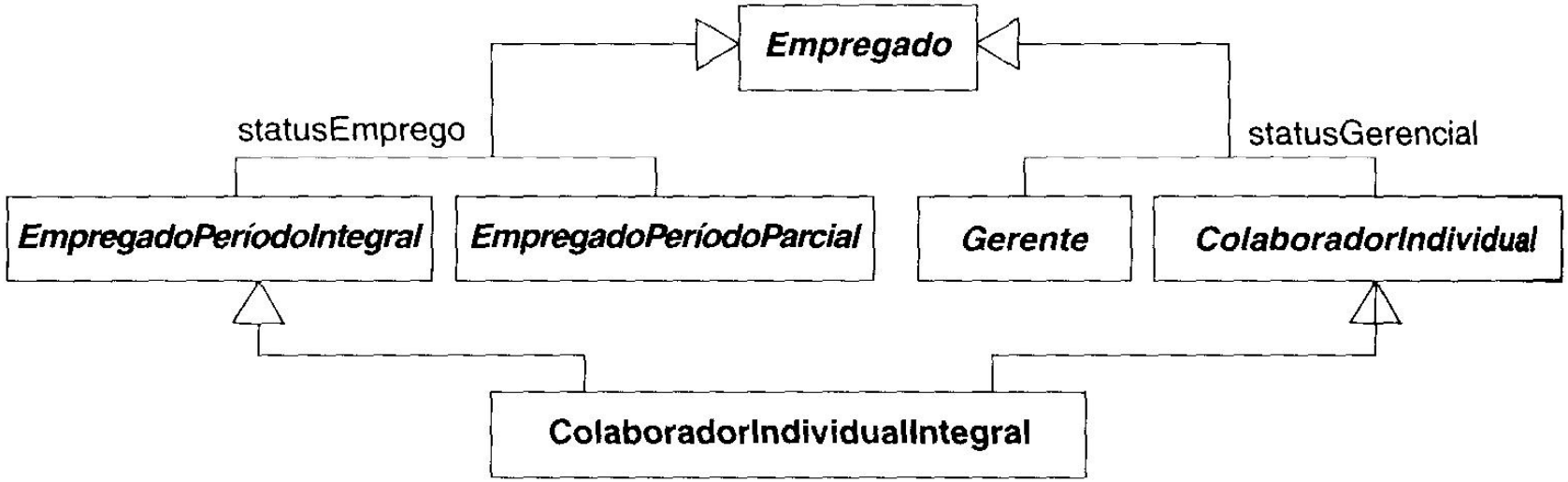
Classes Abstratas



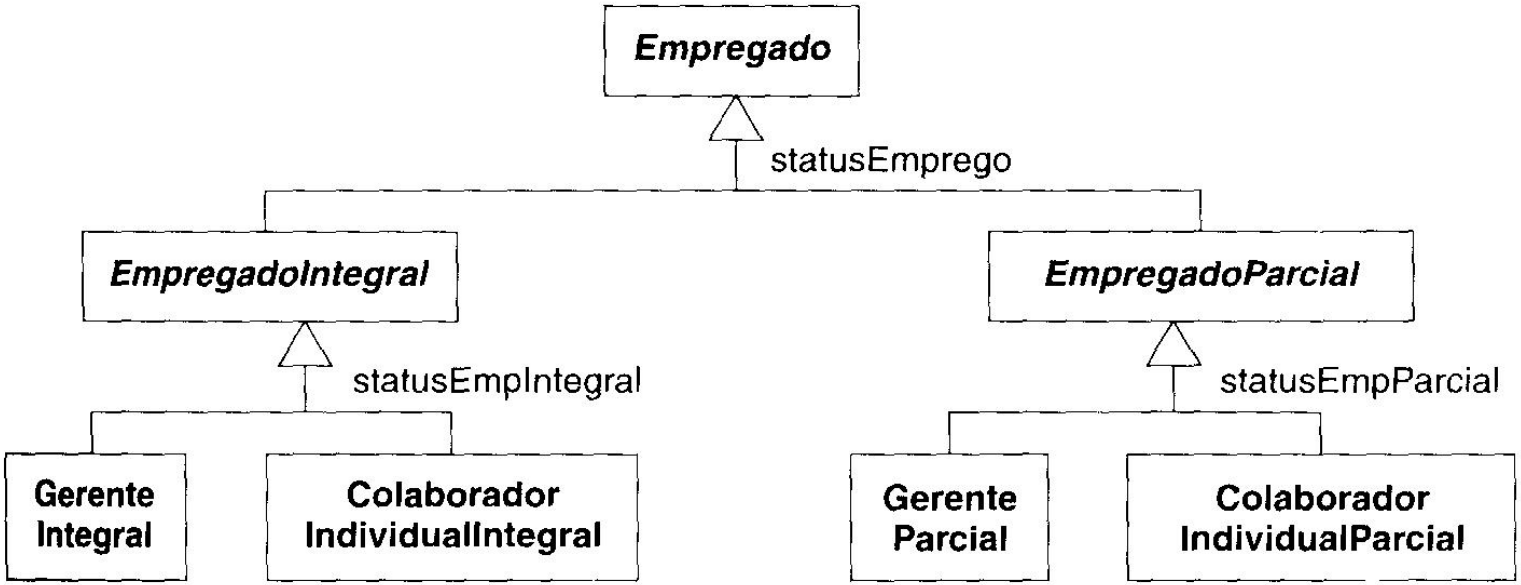
Classe abstrata em java

```
public class Empregado {
  private double ganhosAteHoje;
  public abstract calculaPagamento();
}
```

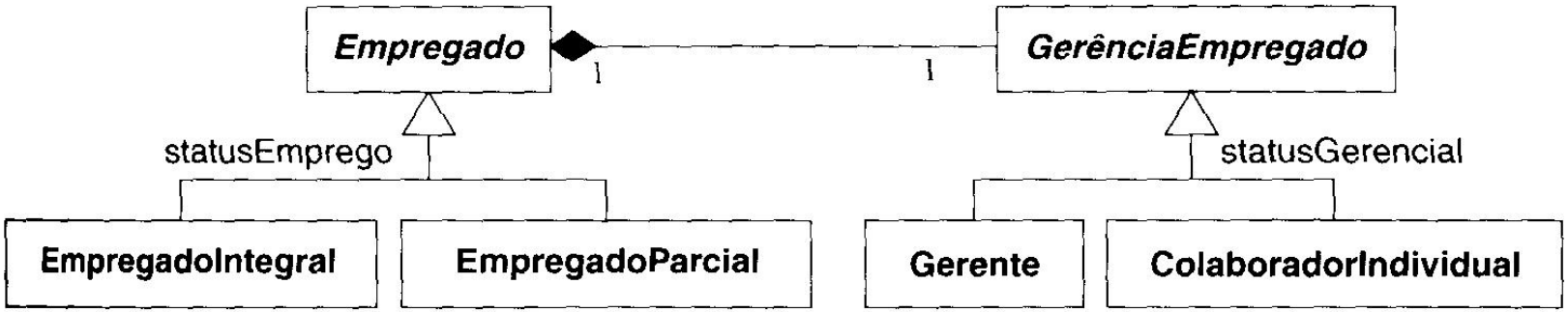
Herança Múltipla



Evitando Herança múltipla por Herança Aninhada



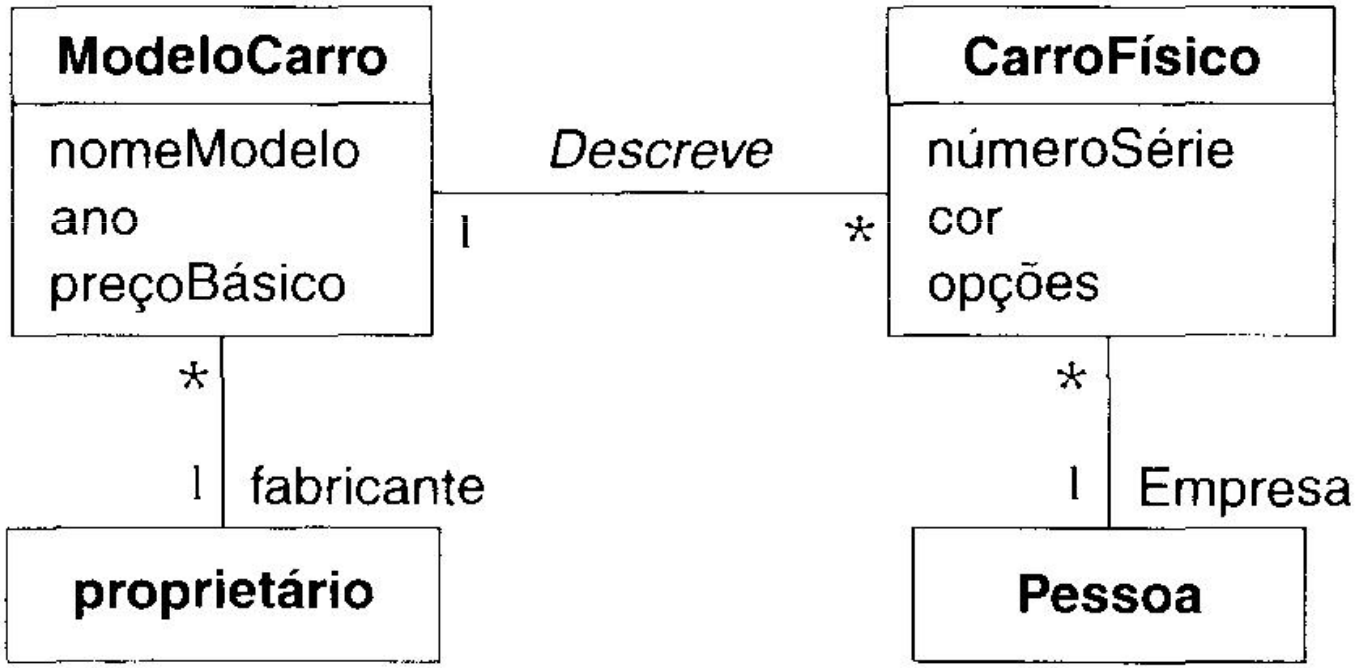
Evitando Herança múltipla com delegação



Metadados

- Classes que descrevem informações sobre outras classes: Ex. modelo de carro, tipo de aeronave, etc.

Metadados


ModeloCarro
nomeModelo
ano
preçoBásico
Descreve
1
*
CarroFísico
númeroSérie
cor
opções
*
1
fabricante
proprietário
*
1
Empresa
Pessoa

Pacotes

- Pacotes podem ser utilizados para organizar classes relacionadas.

Dicas Práticas

- Enumerações. Ao construir um modelo, você deve declarar enumerações, pois elas normalmente ocorrem e são importantes para os usuários. Só especialize uma classe quando as subclasses tiverem atributos, operações ou associações diferentes.

- Atributos com escopo de classe (estáticos). É aceitável usar um atributo com escopo de classe para manter a abrangência de uma classe. Caso contrário, você deve evitar atributos com escopo de classe, pois eles podem levar a um modelo de qualidade inferior. Você pode melhorar um modelo modelando explicitamente grupos e definindo atributos para eles.

Dicas Práticas

- Associações n-árias. Tente evitar associações n-árias. A maioria delas pode ser decomposta em associações binárias.
- Herança múltipla. Limite o uso de herança múltipla. Prefira utilizar delegações ou heranças aninhadas.
- Modelos grandes. Use pacotes para organizar modelos grandes, de modo que o leitor possa entender uma parte do modelo de cada vez, em vez de ter que lidar com todo o modelo ao mesmo tempo.

Sumário de Hoje

- Modelagem de Programas Orientada a Objetos
- **Introdução a Padrões de Projeto (Design Patterns)**
- Introdução a Ambientes Integrados de Desenvolvimento

Projetando Programas Orientados a Objeto

- Projetar um sistema OO consiste em descrever em termos de classes, objetos e seus respectivos relacionamentos e comportamentos o sistema desejado de modo a facilitar :
  - a reutilização de código;
  - a portabilidade do sistema;
  - a produtividade dos desenvolvedores;
  - a extensibilidade do sistema e
  - o entedimento da construção do sistema por parte de terceiros
- Projetar eficientemente é crucial para grandes programas
- O projeto de um sistema pode ser facilitado através do uso de padrões

Porque Padrões?

- ***"Designing object-oriented software is hard and designing reusable object-oriented software is even harder." - Erich Gamma***
- Projetistas experientes reusam soluções com as quais trabalharam no passado
- Sistemas OO bem projetados tem padrões recorrentes de classes e objetos

- Conhecimento de padrões que funcionaram bem no passado permite que o projetista seja mais produtivo e o design resultante seja mais flexível e reutilizável

Introdução a Padrões de Projeto

- *"Design Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context."*
  - Erich Gamma, Richard Helm, Ralph, Johnson, John Vlissides,
  - *Design Patterns: Elements of Reusable Object-Oriented Software*
- Exemplos Simples de Design Patterns:
  - Transfer Object (Value Object)
  - Strategy

Referências sobre Design Patterns

- *Design Patterns: Elements of Reusable Object-Oriented Software,* Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995
- *Design Patterns for Object-Oriented Software Development*, Wolfgang Pree, Addison-Wesley/ACM Press, 1995
- *Patterns of Software: Tales From The Software Community*, Richard P. Gabriel, Oxford University Press, 1996
- *Patterns in Java Volume 1*, Mark Grand, Wiley, 2nd Ed., 2002
- *Java Enterprise Design Patterns: Patterns in Java Volume 3*,Mark Grand, Wiley, 2001
- *Java Design Patterns - A Tutorial*, James W. Cooper, Addison-Wesley, 2000

Referências Adicionais sobre Design Patterns

- *C# Design Patterns - A Tutorial*, J. W. Cooper, Addison-Wesley, 2002
- *Design Patterns In C#*, Steven John Metsker, Addison-Wesley, 2004
- *Head First Design Patterns*, Freeman and Freeman, O'Reilly, 2004
- *Core Security Patterns - Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management*, Christopher Steel, Ramesh Nagappan and Ray Lai, Prentice Hall, 2005
- *Refactoring To Patterns*, Joshua Kerievsky, Addison-Wesley, 2005

Benefícios no uso de Design Patterns

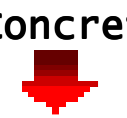
- Arquiteturas Provadas para desenvolver object-oriented software
  - Arquiteturas criadas a partir de experiência acumulada na indústria
- Reduzem a complexidade do processo de design
- Promovem reuso de design em sistemas futuros
- Ajudam a identificar erros e armadilhas comuns em design
- Ajudam a projetar independentemente de linguagem de programação
- Estabelecem um vocabulário comum para os projetistas
- Abreviam a fase de design em um processo de desenvolvimento de software
- Melhoram a qualidade da documentação do projeto

Usuários de Padrões de Projeto (cont.)

- Projetistas
  - Usuários diretos
  - Similar a elementos de arquitetura
    - arcos e colunas
- Programadores
  - Necessária familiaridade com patterns para entender como utilizá-los

Níveis de Abstração em Design Pattern

- **Projeto complexo para um aplicação inteira ou subsistema**
- **Solução para um problema geral de projeto em um contexto particular**
- **Projeto de classe simples e reutilizável tal como: lista ligada, tabela hash, etc.**



GoF Design Patterns

- GoF (Gangue of Four): Gamma, Helm, Johnson and Vlissides
- Os DP GoF estão no meio deste níveis de abstração pois
- Os DP GoF são descrições de objetos e classes que se comunicam e são customizadas para resolver um problema geral de design em um contexto particular.

Classificação dos Design Patterns

- Próposito – o que faz um padrão
  - Padrões Criacionais (Creational Patterns)
    - Tratam do processo de criação de objetos
  - Padrões Estruturais (Structural Patterns)
    - Lidam com as relações estruturais entre classes e objetos
  - Padrões Comportamentais (Behavioral Patterns)
    - Lidam com as interações entre classes e objetos

Classificação dos Design Patterns

- Escopo – aplicação do DP
- Padrões de Classes
  - Focam nas relações entre classes e suas subclasses
  - Geralmente, envolvem reuso via herança
- Padrões de Objeto
  - Focam nas relações entre objetos
  - Geralmente, envolvem reuso via composição

Componentes Essenciais de um Design Pattern

- Nome do Padrão
  - Um nome conciso e significativo que ajude a lembrar do conceito e facilite a comunicação entre desenvolvedores
- Problema
  - Qual o problema e contexto onde será usado o padrão ?
  - Quais as condições nas quais é aconselhável usar o padrão ?
- Solução
  - Descrição dos elementos que compõe o padrão de projeto
  - Enfatiza suas relações, responsabilidades e colaborações
  - Não é uma implementação ou projeto concreto, mas uma abstração que pode ser usada em vários projetos
- Consequências
  - Benefícios e desvantagens de utilizar o padrão
  - Inclui os impactos em reusabilidade, portabilidade e extensibilidade

Um Design Pattern Arquitetural (Model-View-Controller)

- Model-View-Controller: Padrão arquitetural para construir sistemas
  - Divide as responsabilidades do sistema em três partes
    - Modelo
      - Mantem a lógica e dados do programa
    - View
      - Apresenta representações visuais do modelo
    - Controller
      - Processam entradas de usuário e modificam o modelo
  - Step by step
    - Usuário invoca o **controller** para alterar dados no Modelo
    - **Modelo** informa a View a mudança
    - **View** muda a representação para refletir a mudança

Exemplos de Design Patterns usados nos packages **java.awt** e **javax.swing**

- Design patterns usados nos componentes GUI Java
  - Criacionais (Creational)
  - Estruturais (Structural)
  - Comportamentais (Behavioral)
  - Os componentes GUI Swing utilizam design patterns dos três tipos

5.1.1 Padrões de Projeto Criacionais

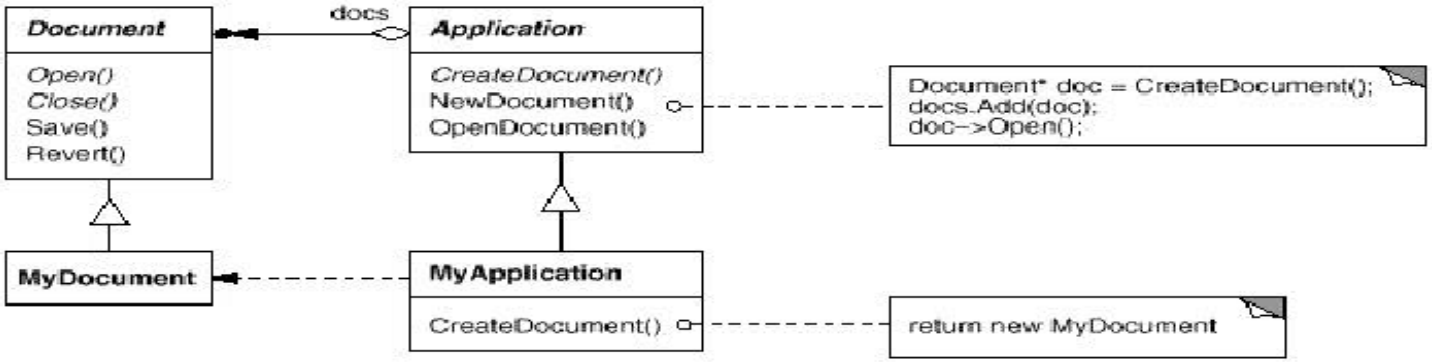
- Padrão de Projeto: método de fábrica
  - Suponha que você quer criar uma forma de abrir arquivos de imagens
    - Existem vários formatos de imagem (ex., GIF, JPEG, etc.)
      - Cada formato tem uma estrutura distinta
    - O método createImage da classe Component cria objetos Image
    - Dois objetos Image (um para GIF, outro para imagem JPEG)
    - Método createImage usa o parametro para determina a subclasse de Image apropriada para cada caso

      createImage("image.gif");
      - Retorna um objeto de uma subclasse de Image que trata GIFs

      createImage("image.jpg");
      - Retorna um objeto de uma subclasse que trata JPEGs
    - O método createImage é chamado de método fabrica *(factory method)* e determina a subclasse em tempo de execução

Padrão de Projeto Fábrica Abstrata (Abstract Factory Design Pattern)

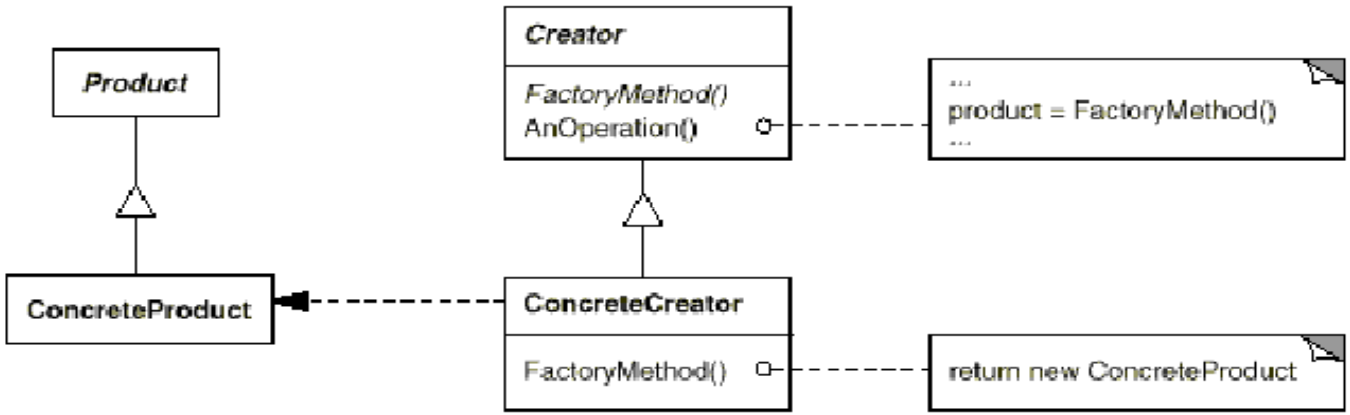
- Intent
  - ⇨ Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Motivation
  - ⇨ Consider the following framework:



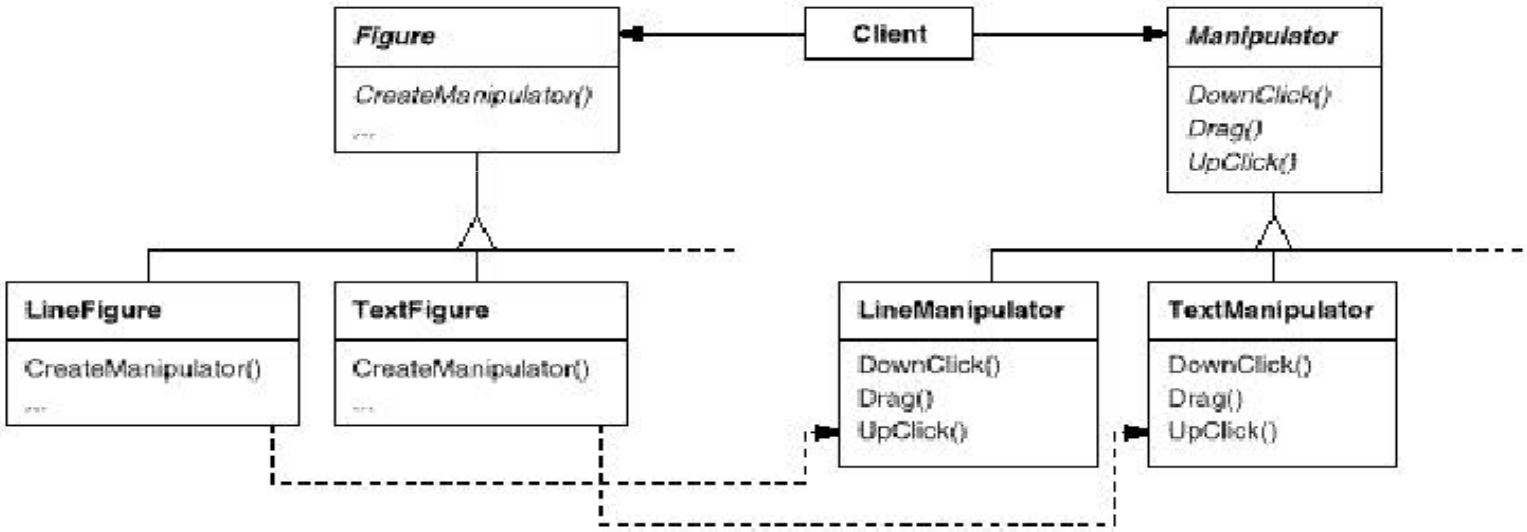
Padrão de Projeto Fábrica Abstrata (Abstract Factory Design Pattern)

- Objetivo
  - Criar uma interface para instanciar classes mas deixar as sub-classes concretas definirem qual classe concreta deve ser instanciada
- Motivação:
  - Trabalhar em um nível de abstração mais elevado e tratar várias implementações distintas de modo uniforme

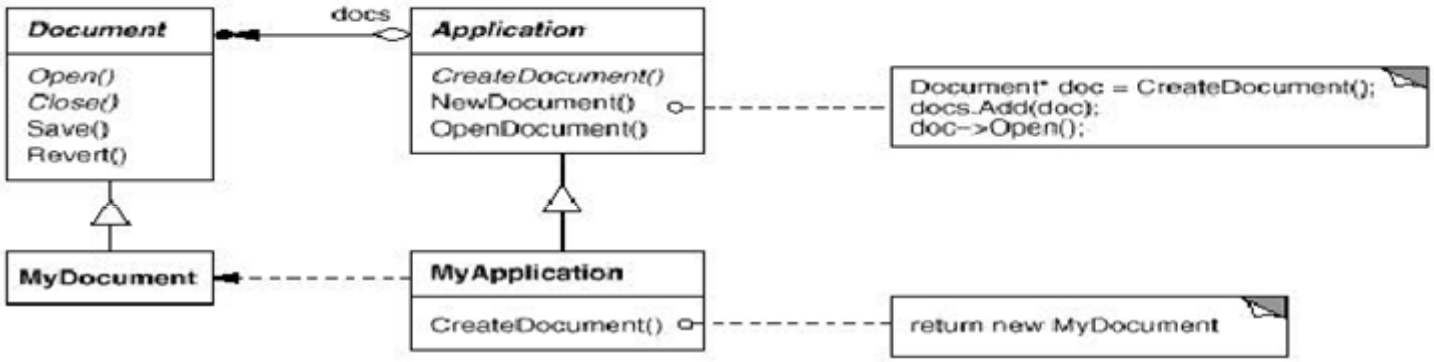
Estrutura


Product
Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

Exemplo – Abstract Factory



Exemplo 2 - Abstract Factory



Conseqüências

- Benefícios
  - O código é mais flexível e reutilizável pela eliminação da instanciação de classes específicas de implementação
  - O código cliente lida apenas com as interfaces da classe Product e assim pode trabalhar com qualquer implementação de Product
- Desvantagens
  - Clientes tem que criar uma nova subclasse da classe Creator apenas para instanciar um novo ConcreteProduct

5.1.2 Structural Design Patterns

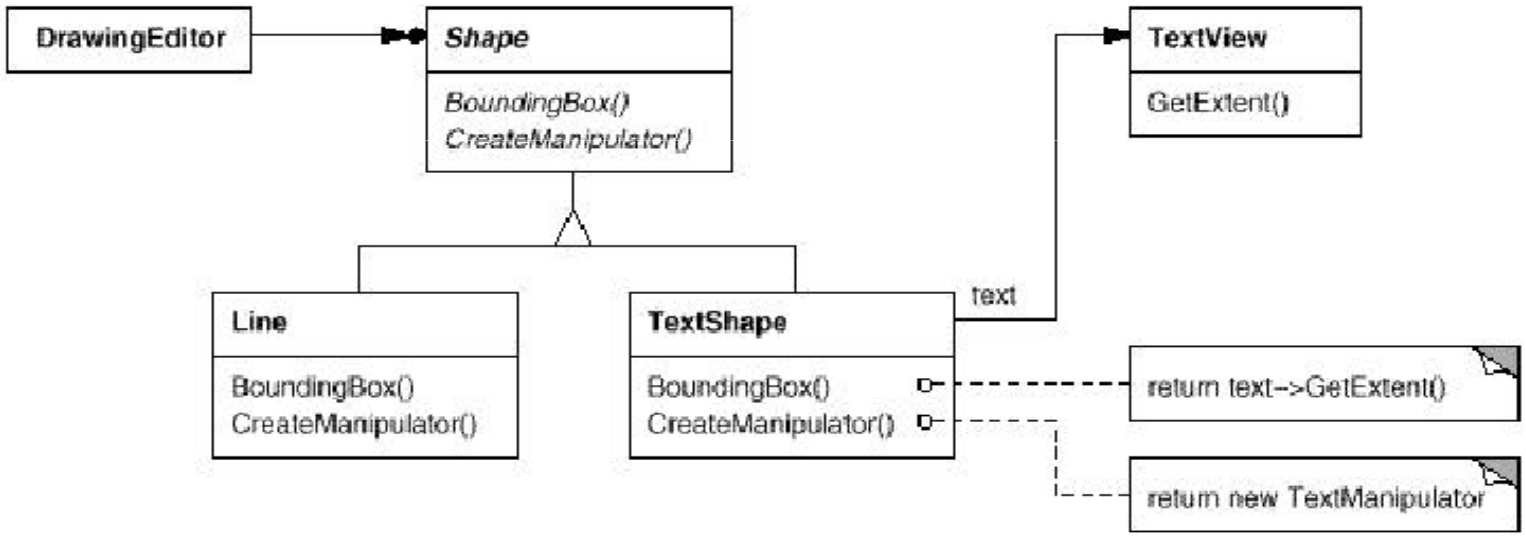
- Adapter design pattern
  - Used with objects with incompatible interfaces
    - Allows these objects to collaborate with each other
    - Object's interface *adapts* to another object's interface
  - Similar to adapter for plug on electrical device
    - European electrical sockets differ from those in United States
    - American plug will not work with European socket
    - Use *adapter* for plug
  - Class MouseAdapter
    - Objects that generate MouseEvents adapts to objects that handle MouseEvents

Adapter Design Pattern

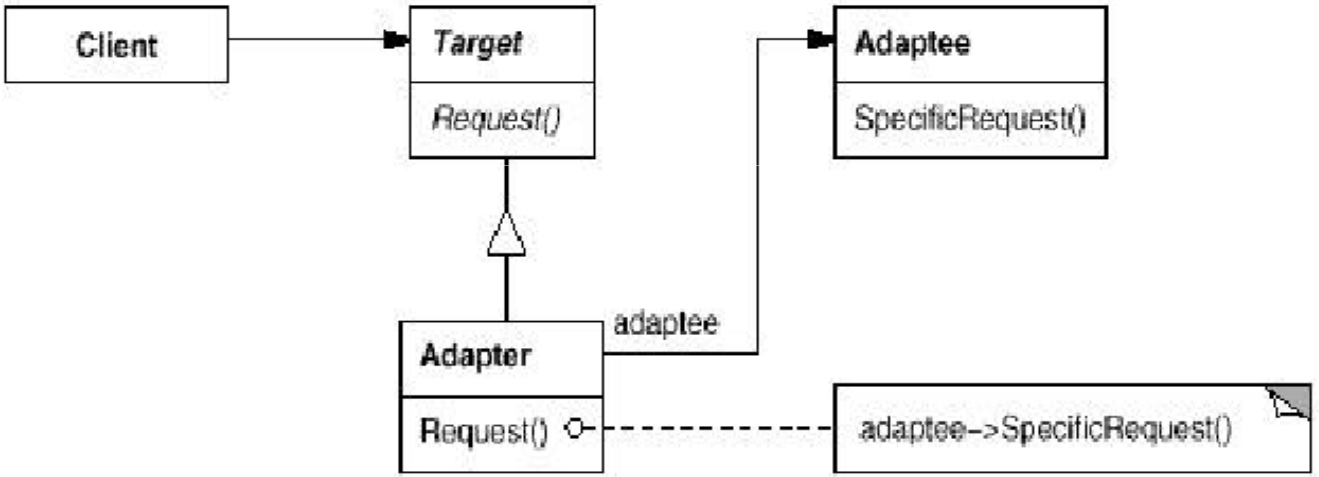
- Intent
  - ⇨ Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also Known As
  - ⇨ Wrapper
- Motivation
  - ⇨ Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
  - ⇨ We can not change the library interface, since we may not have its source code
  - ⇨ Even if we did have the source code, we probably should not change the library for each domain-specific application

Exemplo - Motivação

⇨ Example:



Estrutura

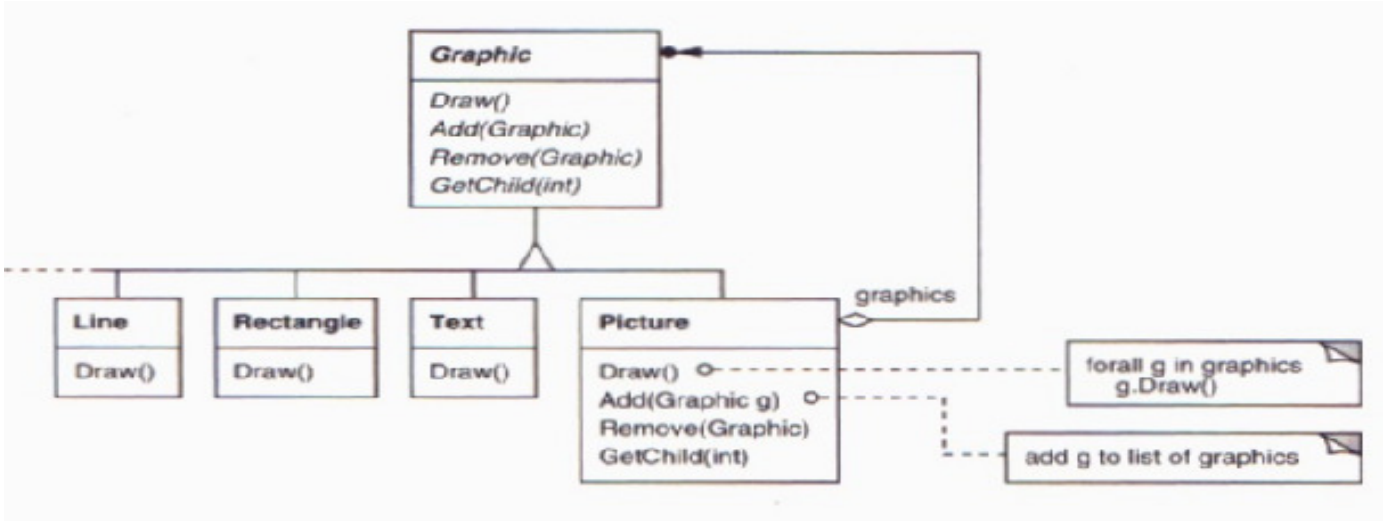

Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()

5.1.2 Structural Design Patterns

- Composite design pattern
  - Organiza componentes em uma estrutura de árvore hierarquica
    - Cada nó representa um componente
    - Todos os nós tem a mesma interface
      - Polimorfismo garante que os clientes podem tratar todos os nós de forma uniforme
- Usado por componentes Swing
    - JPanel é subclasse de JContainer
    - Um objeto JPanel pode conter GUI componentes
    - JPanel não precisar saber o tipo específico do componente GUI

Composite Design Pattern

- Objetivo
  - Compor objetos em estruturas de árvores em estruturas de árvores que representam hierarquias todo-parte. Composição permite aos clientes tratar objetos e composições de forma uniforme



Estrutura

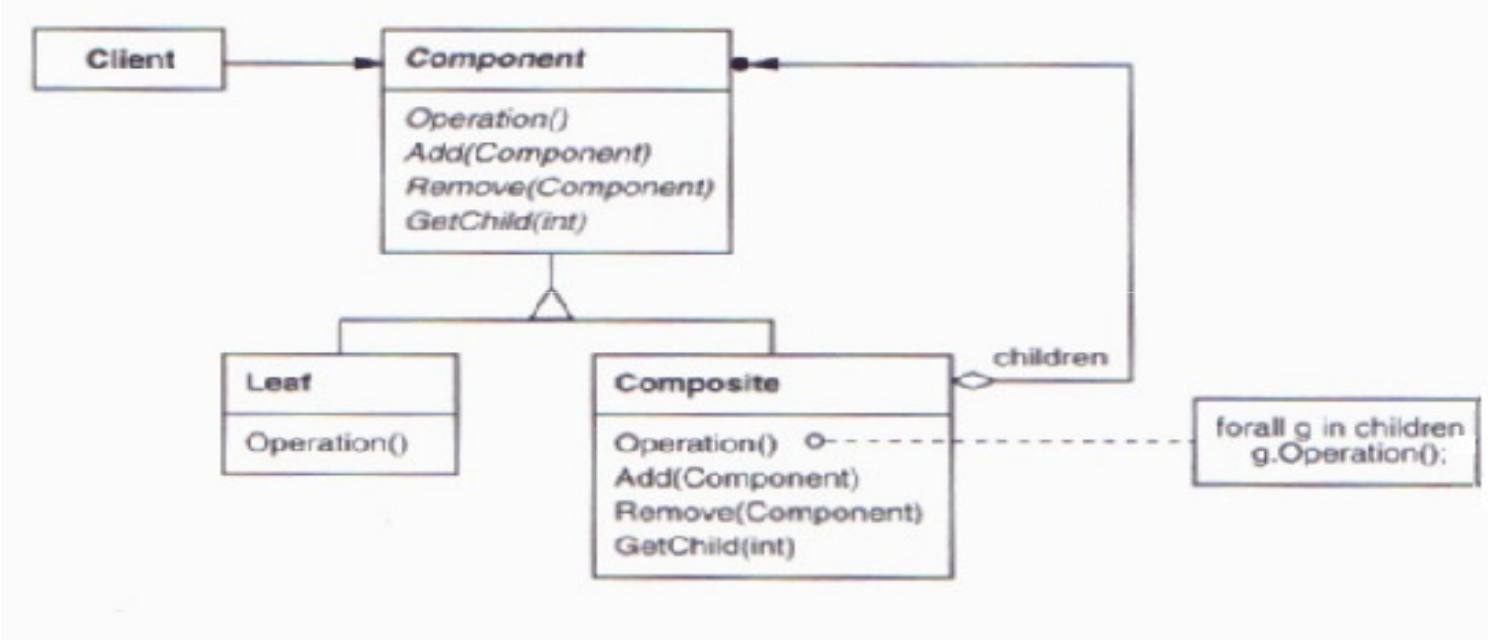
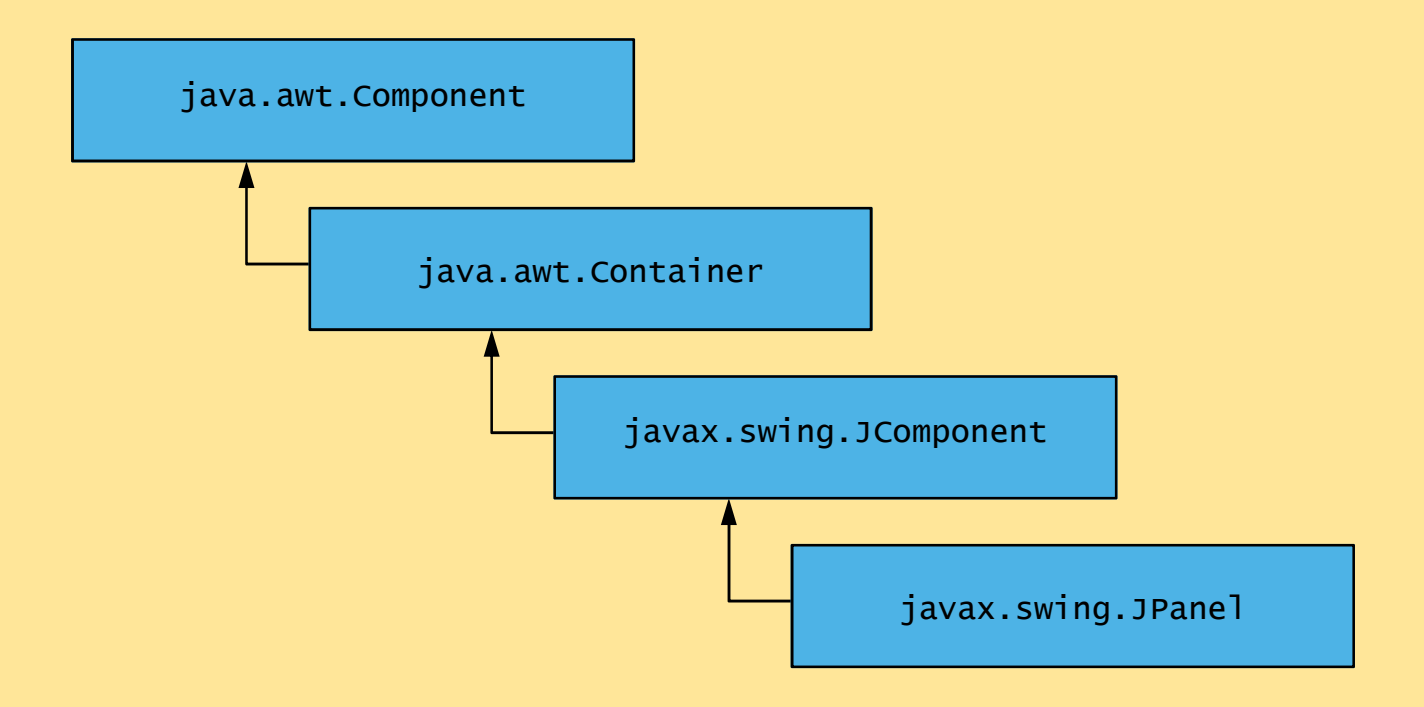

Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

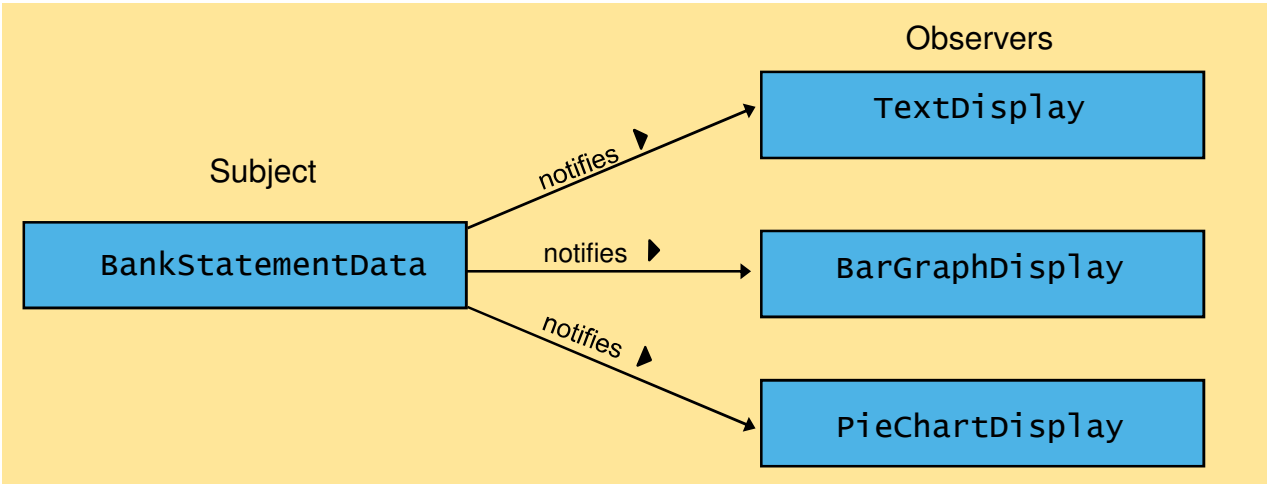
Fig. 4.27 Inheritance hierarchy for class `JPanel`



5.1.3 Behavioral Design Patterns

- Observer design pattern
  - Design program for viewing bank-account information
    - Class BankStatementData store bank-statement data
    - Class TextDisplay displays data in text format
    - Class BarGraphDisplay displays data in bar-graph format
    - Class PieChartDisplay displays data in pie-chart format
    - BankStatementData (subject) notifies Display classes (observers) to display data when it changes
  - Subject notifies observers when subject changes state
    - Observers act in response to notification
    - Promotes *loose coupling*
  - Used by
    - class java.util.Observable
    - class java.util.Observer

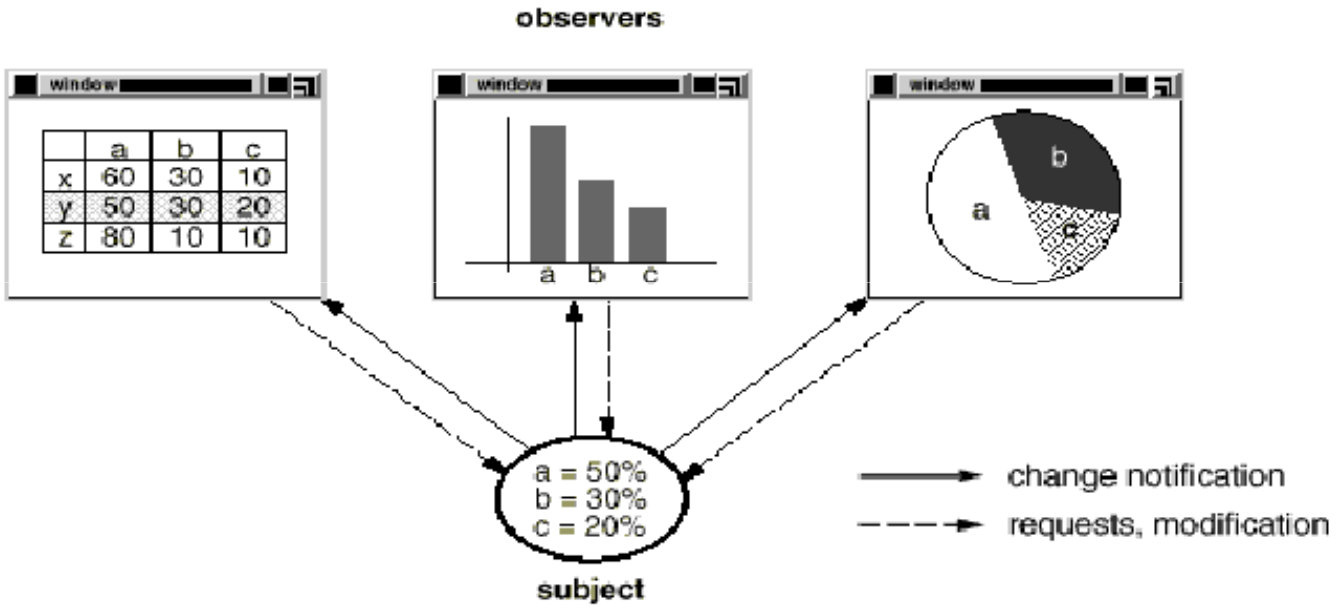
Fig. 4.28 Basis for the Observer design pattern



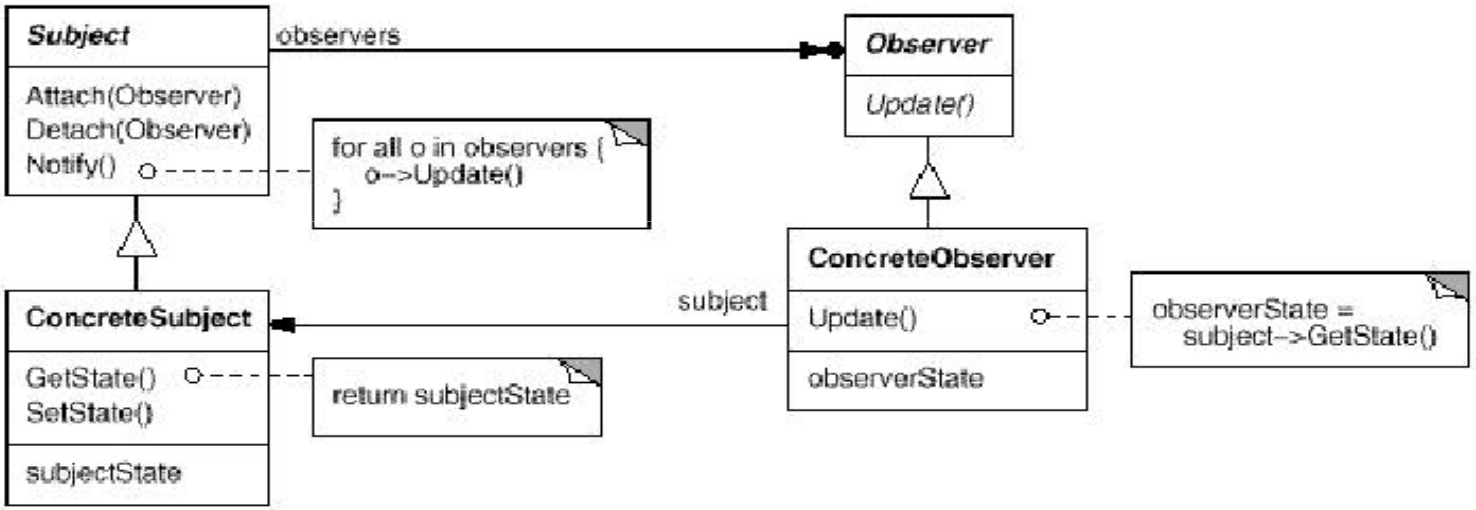
The Observer Pattern

- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Also Known As
  - Dependents, Publish-Subscribe, Model-View
- Motivation
  - The need to maintain consistency between related objects without making classes tightly coupled

Motivação


observers
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
a
b
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modification

Estrutura


Subject
observers
Attach(Observer)
Detach(Observer)
Notify()
for all o in observers {
o->Update()
}
Observer
Update()
ConcreteObserver
Update()
observerState
observerState =
subject->GetState()
subject
ConcreteSubject
GetState()
SetState()
subjectState
return subjectState

Participantes

- Subject
  - Keeps track of its observers
  - Provides an interface for attaching and detaching Observer objects
- Observer
  - Defines an interface for update notification
- ConcreteSubject
  - The object being observed
  - Stores state of interest to ConcreteObserver objects
  - Sends a notification to its observers when its state changes
- ConcreteObserver
  - The observing object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer update interface to keep its state consistent with the subject's

Conseqüências

- Benefits
  - Minimal coupling between the Subject and the Observer
  - Can reuse subjects without reusing their observers and vice versa
  - All subject knows is its list of observers
  - Observers can be added without modifying the subject
  - Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
  - Subject and observer can belong to different abstraction layers
- Support for event broadcasting
  - Subject sends notification to all subscribed observers
  - Observers can be added/removed at any time

Conseqüências

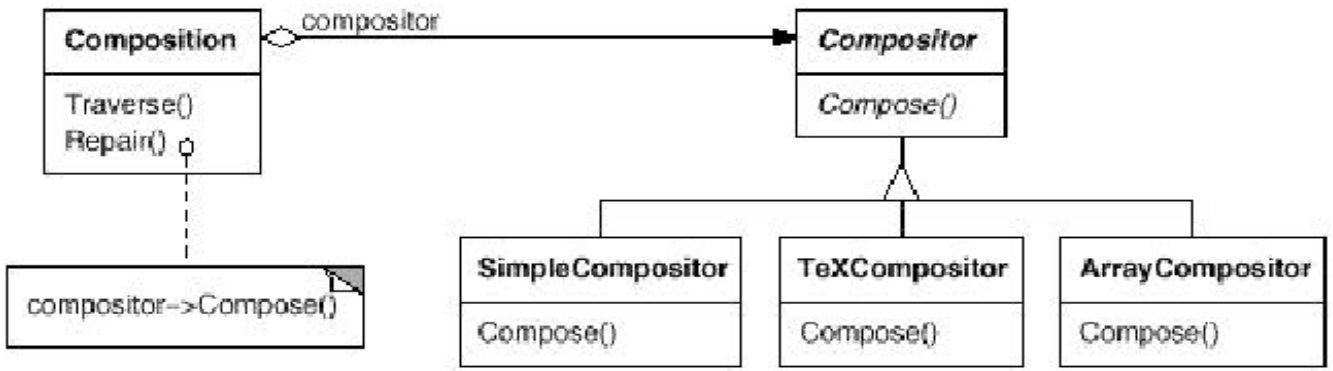
- Liabilities
  - Possible cascading of notifications
  - Observers are not necessarily aware of each other and must be careful about triggering updates
  - Simple update interface requires observers to deduce changed item

5.1.3 Behavioral Design Patterns

- Strategy design pattern
  - Encapsulates algorithm
  - LayoutManagers are strategy objects
    - Classes FlowLayout, BorderLayout, GridLayout, etc.
      - Implement interface LayoutManager
    - Each class uses method addLayoutComponent
      - Each method implementation uses different algorithm
        - FlowLayout adds components left-to-right
        - BorderLayout adds components in five regions
        - GridLayout adds components in specified grid
    - Class Container has LayoutManager reference
      - Use method setLayout
        - Select different layout manager at run time

Strategy Design Pattern

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Motivation:

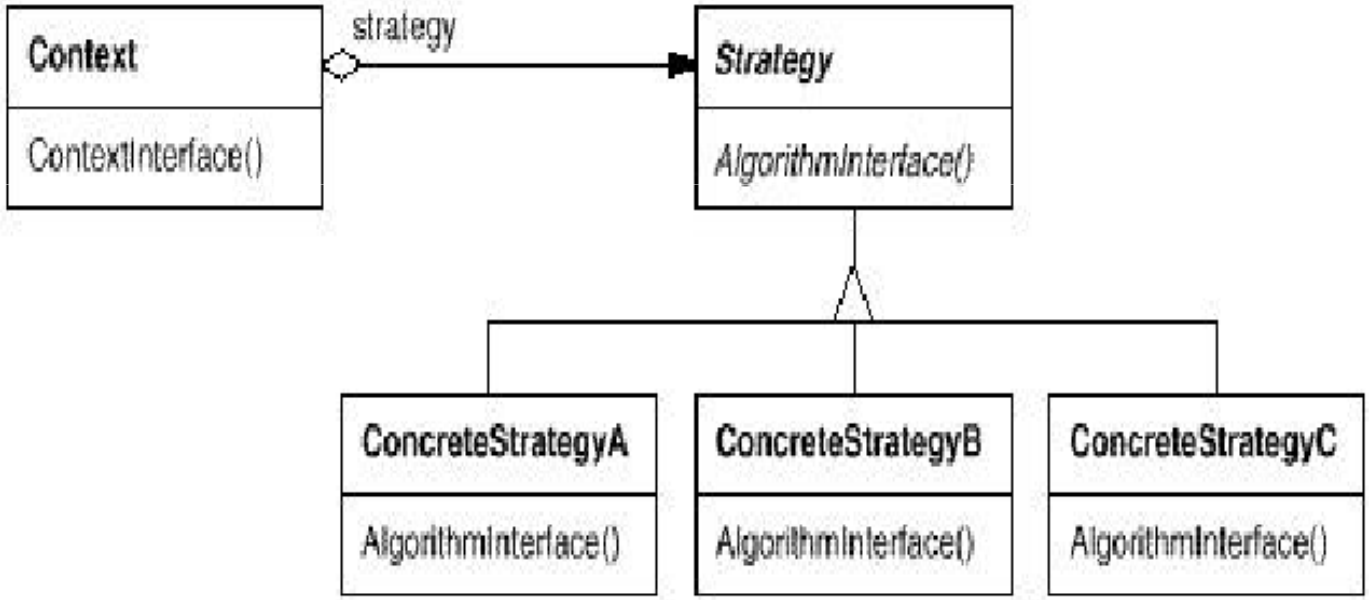


Aplicabilidade

- Use the Strategy pattern whenever:
  - Many related classes differ only in their behavior
  - You need different variants of an algorithm
  - An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Estrutura

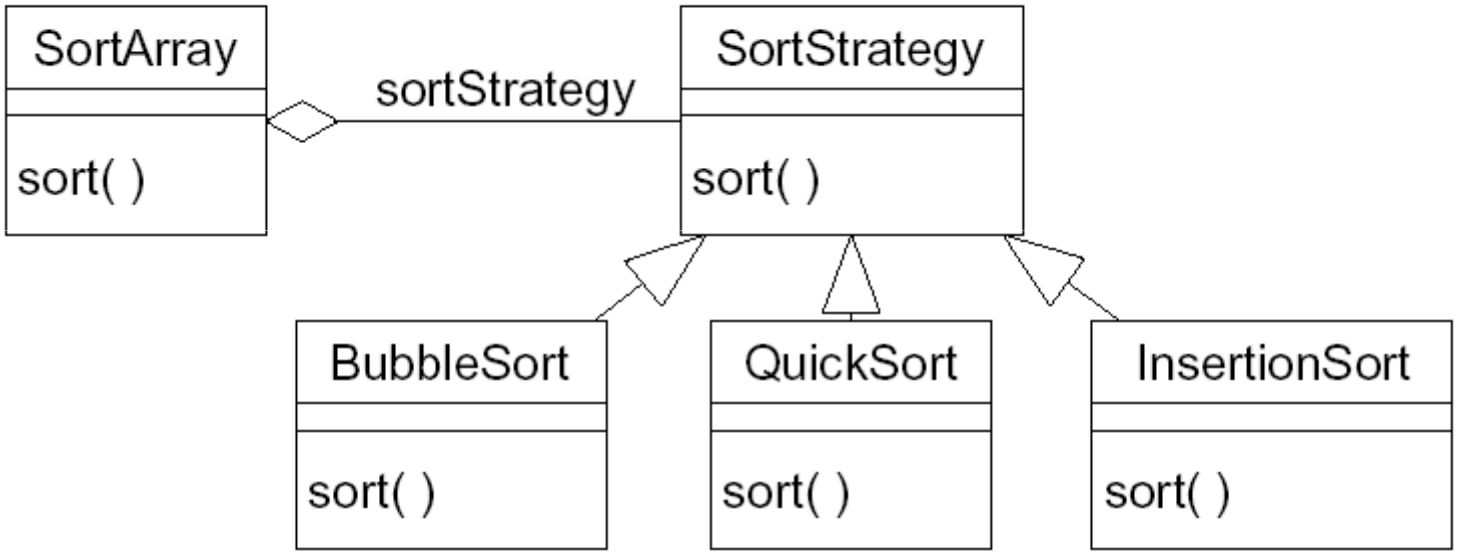
- Estrutura



Consequências

- Benefits
  - Provides an alternative to subclassing the Context class to get a variety of algorithms or behaviors
  - Eliminates large conditional statements
  - Provides a choice of implementations for the same behavior
- Liabilities
  - Increases the number of objects
  - All algorithms must use the same Strategy interface

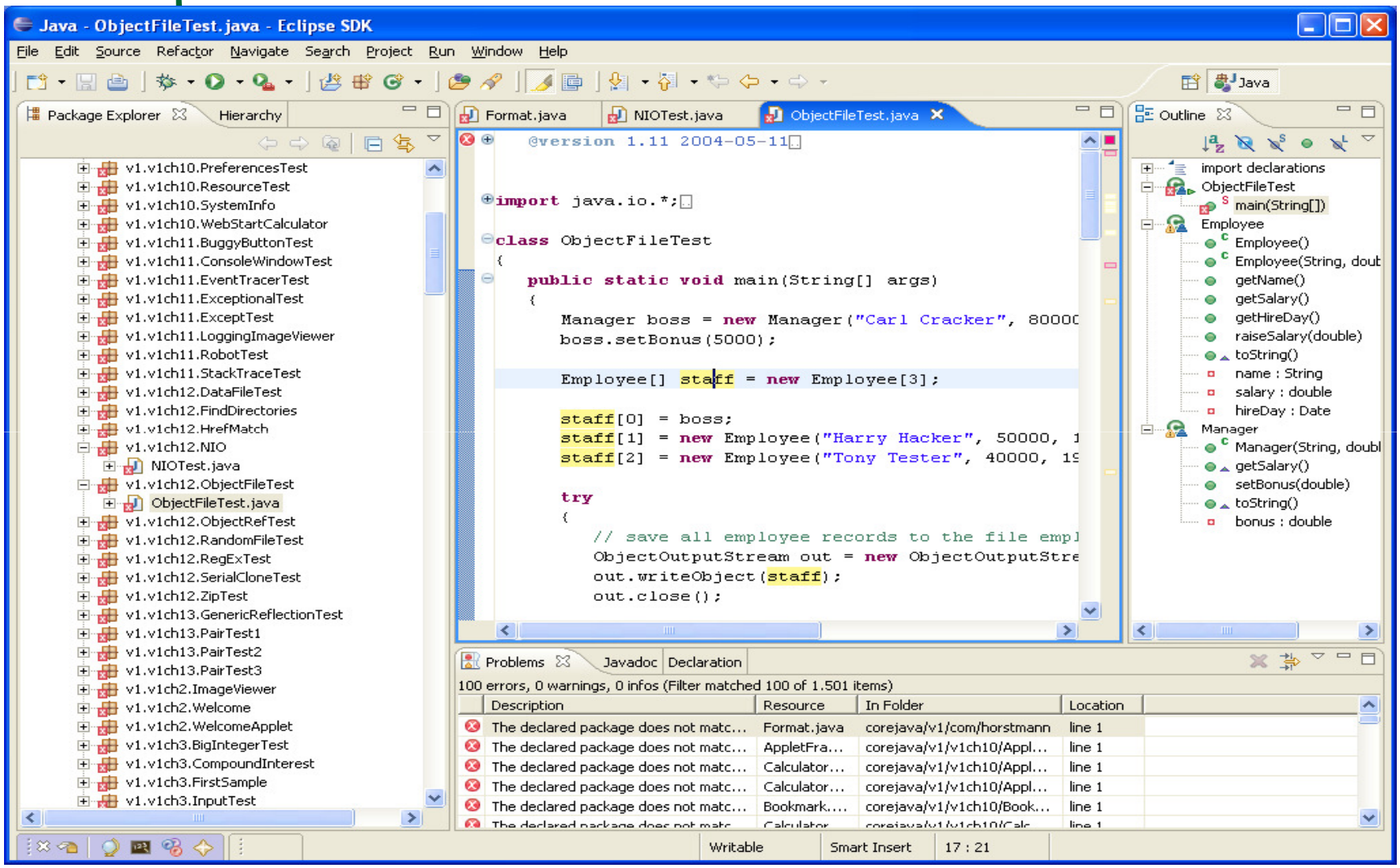
Exemplo - Strategy



Sumário de Hoje

- Modelagem de Programas Orientada a Objetos
- Introdução a Padrões de Projeto (Design Patterns)
- **Introdução a Ambientes Integrados de Desenvolvimento**

Eclipse IDE


Java - ObjectFileTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
Hierarchy
Format.java
NIOTest.java
ObjectFileTest.java
Outline
Problems
Javadoc
Declaration
100 errors, 0 warnings, 0 infos (Filter matched 100 of 1.501 items)
Description
Resource
In Folder
Location
Writable
Smart Insert
17 : 21