Programação Orientada a Objetos

Modelagem Orientada a Objetos e Introdução a Padrões de Projetos

Sumário

- **Modelagem de Programas Orientada a Objetos**

- Introdução a Padrões de Projeto (Design Patterns)

Modelagem de Programas Orientados a Objetos

- Dada uma especificação de sistema a ser construído em OO é preciso definir as classes que serão necessárias à construção do sistema suas responsabilidades e seus relacionamentos. Esse processo chamado de modelagem ou mais comumente Projeto de Sistemas (Software)

- Projeto de Software será alvo de disciplinas exclusivas específicas como Fundamentos de Engenharia de Software e Engenharia de Software incluindo questões como ciclo de vida de software, metodologias de desenvolvimento, especificação de requisitos, etc.

- Aqui, abordaremos problemas mais simples e mais proximos da implementação, que poderíamos chamar de Projeto de Classes

Projeto de Classes

- Ao projetar classes de um sistema executamos as seguintes tarefas:

1. Descobrir as classes
2. Determinar as responsabilidades de cada classe
3. Descrever os relacionamentos entre as classes

Descobrindo classes

- Classes são coleções de objetos e os objetos não são ações – são entidades.
- Duas regras simples e úteis:
  - Nomes de Classes devem ser substantivos e “substantivos fortes” em uma especificação são bons candidatos a classes.
  - Nomes de métodos devem ser verbos

Boas Classes

- Uma classe deve representar um único conceito. Por exemplo: Point, Rectangle, Ellipse, Frame, etc.
- Algumas classes são abstrações de entidades da vida real:
  - BankAccount, Bag, etc.

- Uma categoria comum e útil de classes são os chamados **atores.** Objetos de classes atores fazem algum tipo de trabalho. Por exemplo:
  - StringTokenizer (quebra string segundo delimitadores…)
  - Random (gera números aleatórios….um melhor nome para esta classe seria RandomNumberGenerator)
  - Use terminações “-er” ou “-or” em inglês para classes atores. Use “or” em português.

Boas Classes

- As classes utilitárias são aquelas que não tem objetos, mas contém uma coleção de métodos estáticos e/ou constantes. A classe Math é um exemplo típico.
- Por vezes, são criadas classes que tem apenas um método main e cujo único propósito é iniciar um programa.
- Classes utilitárias e classes “main” são em geral pouco importantes durante a fase de Projeto de Classes.

Classes não tão boas….

- Digamos que seu trabalho seja escrever um programa que imprima cheques de pagamentos (pay checks).
- Que tal a classe PayCheckProgram?
  - Oque faria um objeto dessa classe?
    - Todo o trabalho?...Isso não simplifica as coisas…
  - Melhor seria utilizar PayCheck e seu programa manipularia vários objetos PayCheck

Classes não tão boas….2

- Outro erro comum é transformar uma ação em uma classe.
- No programa anterior, isso poderia ser criar uma classe ComputePayCheck.
- Uma melhor decisão seria criar um método compute (verbo) dentro de uma classe PayCheck (substantivo)

Características de Boas Classes

- Já dissemos que uma classe deve representar um único conceito.
- Além disso, os métodos e constantes públicas de uma classe devem se referir a esse único conceito.
- Quanto mais aderente ao conceito da classe for sua interface pública (métodos e constantes públicas) mais **coesa** é a classe.
- Coesão é uma característica importante para uma boa classe.

Exemplo: A classe abaixa é coesa? Porquê?

```
public class Purse {
   public Purse() {…..}
    public void addNickels(int count) {….}
   public void addDimes(int count) {….}
   public void addQuarters(int count) {….}
   public double getTotal() {…..}
   public static final double NICKEL_VALUE=0.05;
 public static final double DIME_VALUE=0.10;
 public static final double QUARTER_VALUE=0.25;
}
```

Porque não criar uma classe Coin, além da classe Purse?

Caracerísticas de Boas Classes 2

- Diz-se que uma classe A depende de uma classe B, se A utiliza qualquer método ou atributo de B.
- Em UML, representa-se com uma seta tracejada entre as duas classes. A seta parte da classe dependente (A) para B.

Dependência e Acoplamento

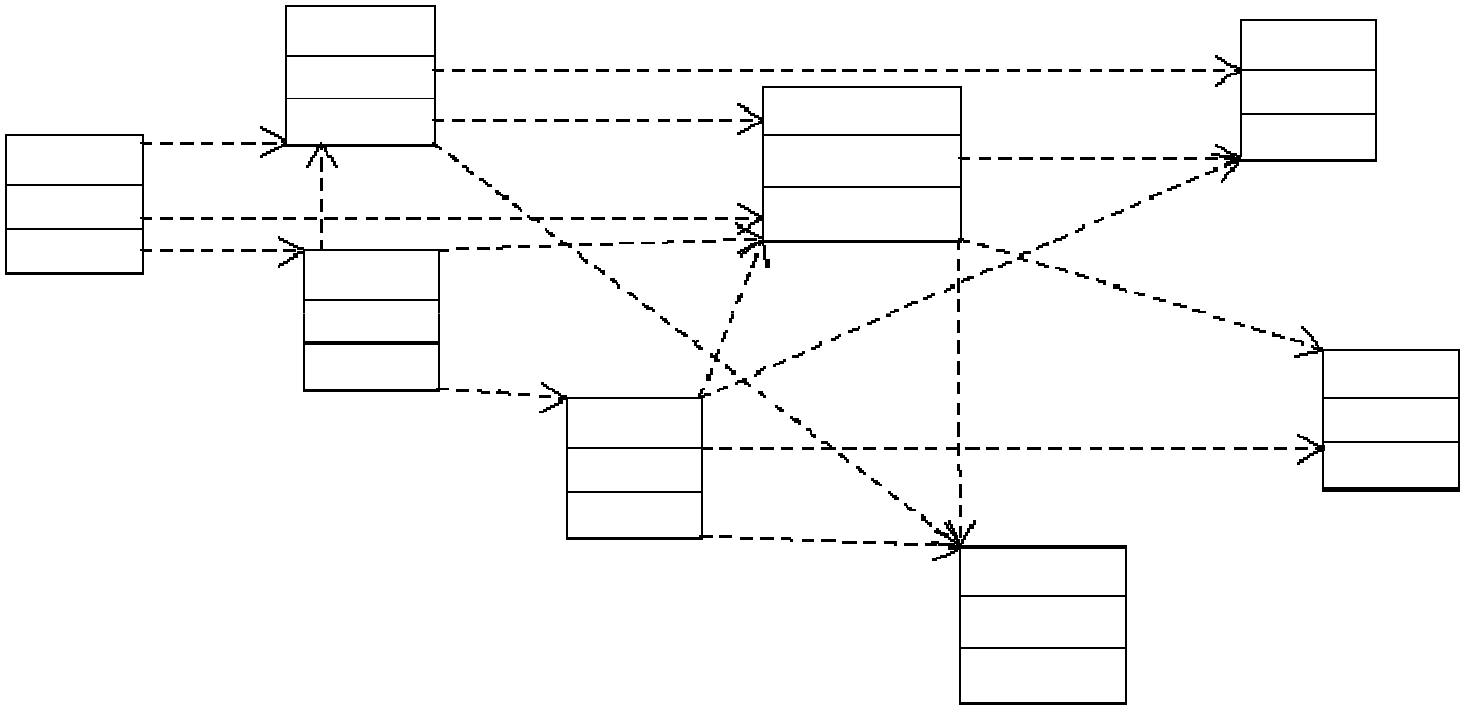
- Se muitas classes de um programa dependem uma das outras, então dizemos que o **acoplamento** entre classes é **alto**

- Se há poucas dependências entre classes, diz-se que o **acoplamento** é **baixo**.

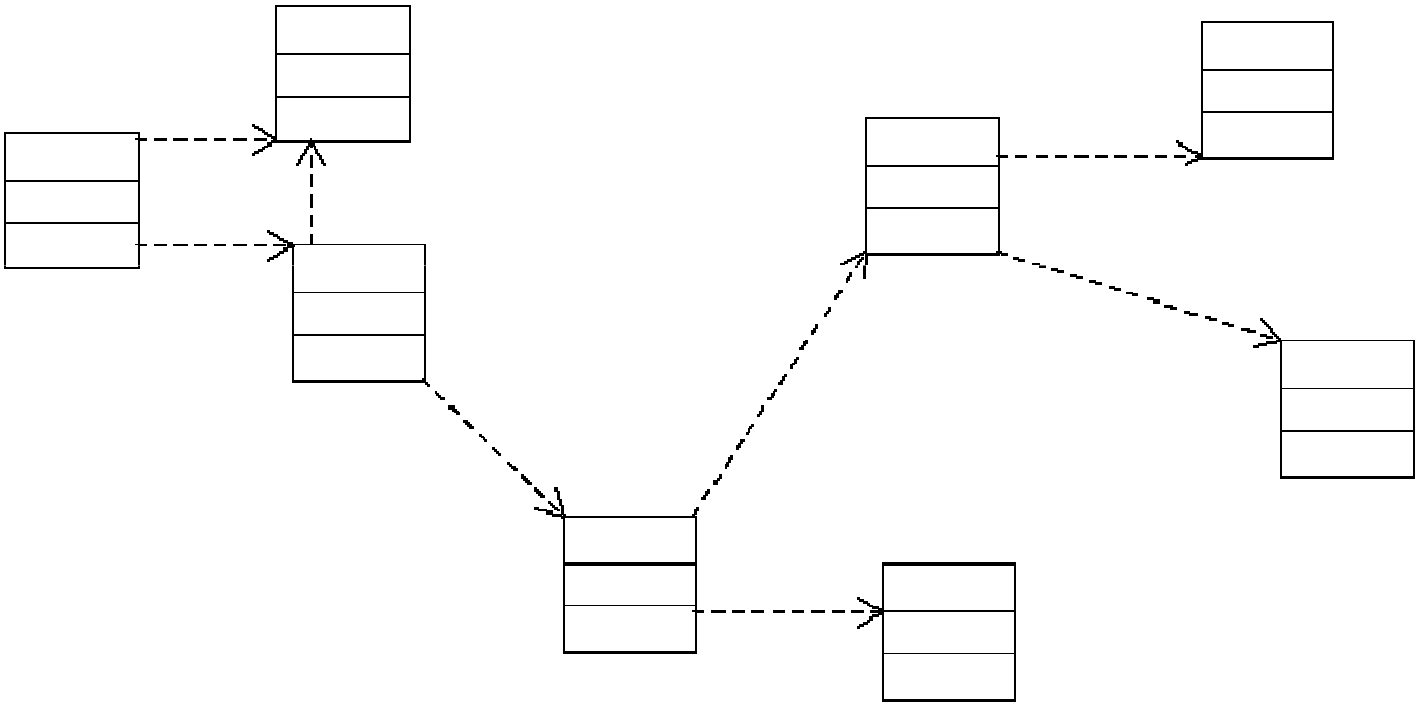
- Nesse caso, é melhor ser baixo ou alto? Porquê?

- Se uma classe muda, todas as classes acopladas a ela também podem ser afetadas

Classes com Alto Acoplamento



Classes com Baixo Acoplamento



Acoplamento e Interfaces

- De modo geral, queremos remover qualquer acoplamento desnecessário entre classes
- Um projeto com classes coesas naturalmente ajuda a ter também um projeto com baixo acoplamento
- Uma maneira de reduzir acoplamento é usando interfaces. Se a classe A depende de B,C e D mas todas implementam I. Posso fazer com que A dependa apenas de I.
  - O padrão de projeto abstract factory usa essa característica

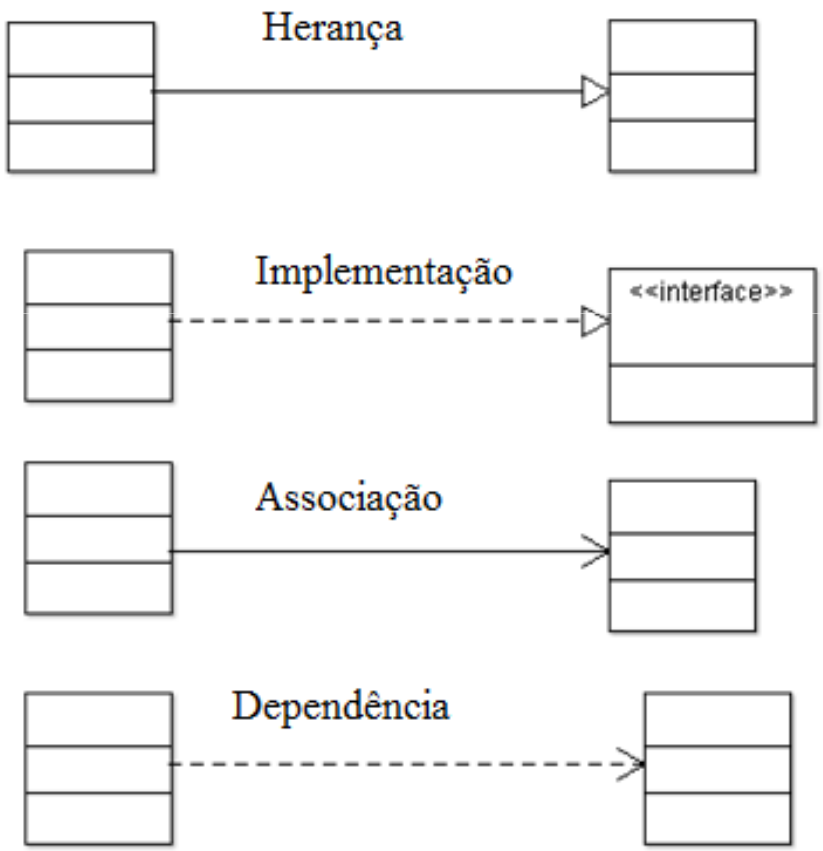
Projeto de Classes 2

- Uma vez encontrada as classes precisamos **Determinar as responsabilidades de cada classe**
  - Uma forma simples de fazer isso é procurar os verbos “fortes” na descrição da tarefa.
  - Depois disso é preciso atribuir a responsabilidade a uma das classes existentes ou criar um nova para tratar o problema

Projeto de Classes 3

- Com as classes e suas responsabilidades que serão implementados como métodos. Devemos identificar os relacionamentos entre as classes. Tal como discutido anteriormente.
  - Herança,
  - Composição,
  - Associação e
  - Dependência.

Relações entre classes



•Observe que toda associação é uma dependência, mas o inverso não é verdadeiro.

•Por exemplo: JPanel depende de Graphics mas não está associado a ele. JPanel precisa esperar uma chamada ao método paint para receber uma referência a Graphics.

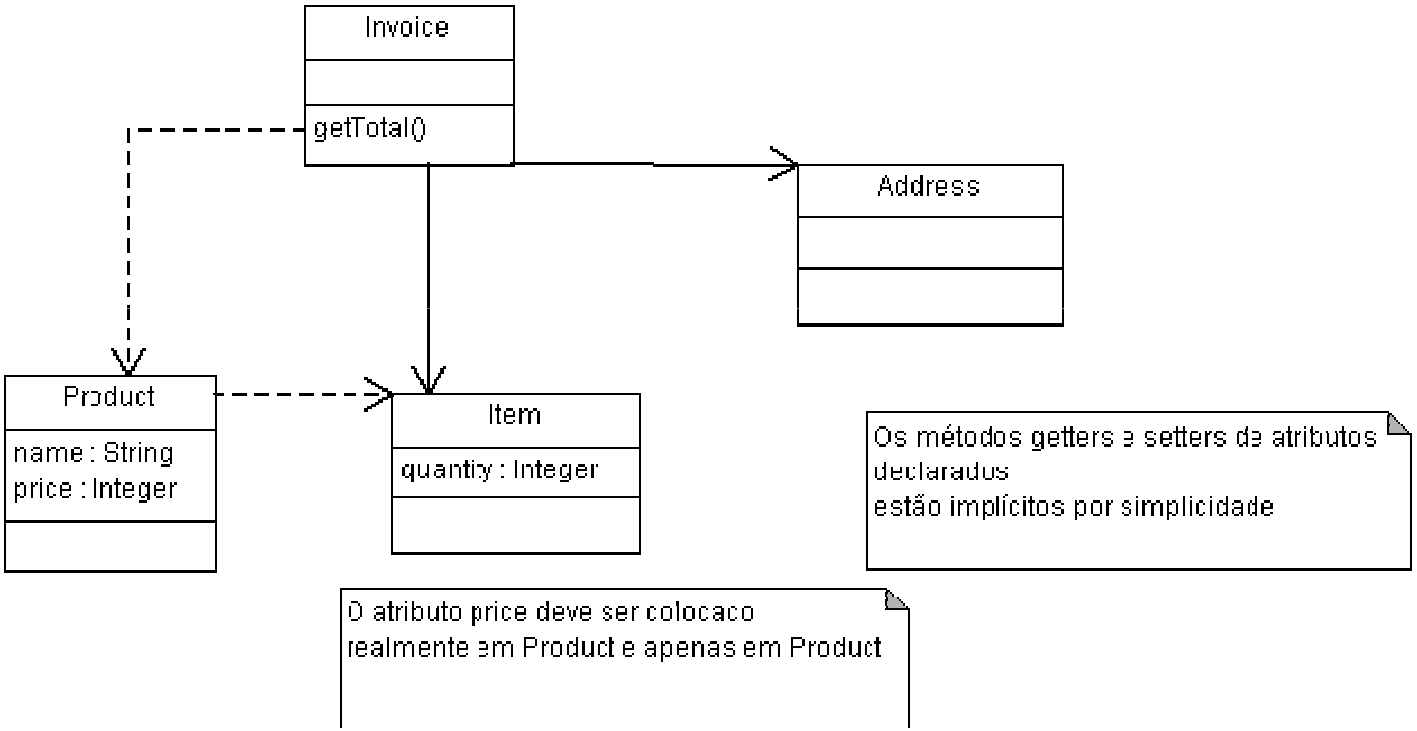
Representação em UML

- Classes e Objetos
- Associação
  - Agregação
  - Composição
- Mais problemas comuns em Projeto de Classes

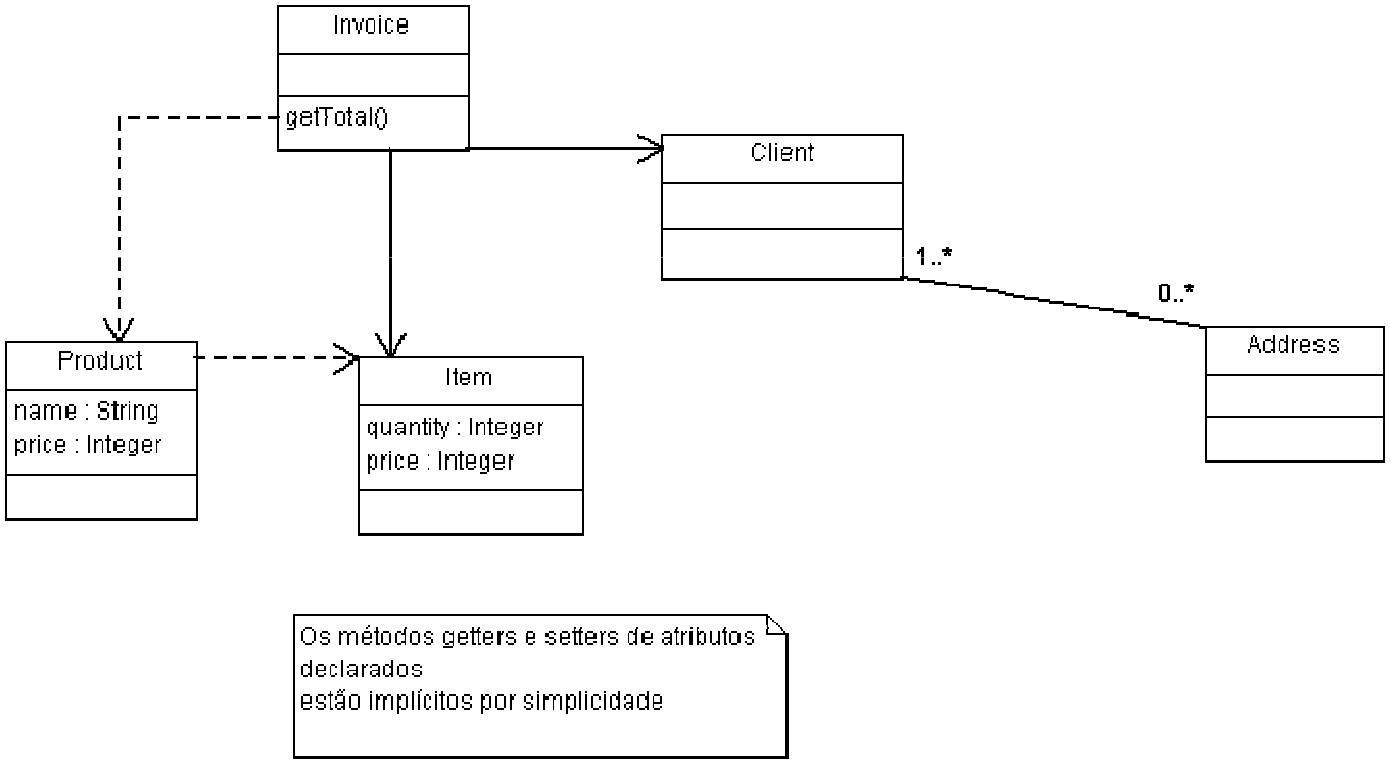
Exercício

- Projete as classes de um Sistema para representar os dados de uma fatura ou nota fiscal (Invoice), inclusive nome e endereço do clientes além dos itens, suas quantidades, preços e totais.
  - Identifique as classes e suas relações
  - As responsabilidades e métodos ficam como homework

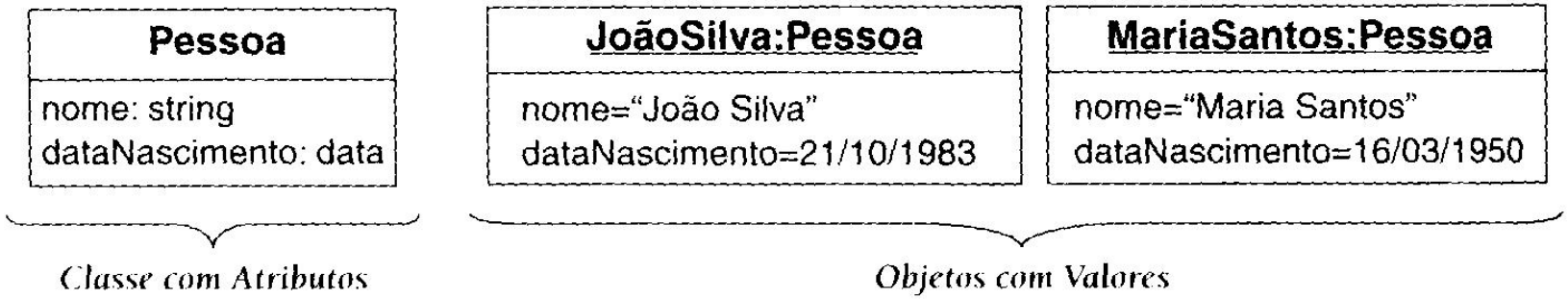
Possível Solução



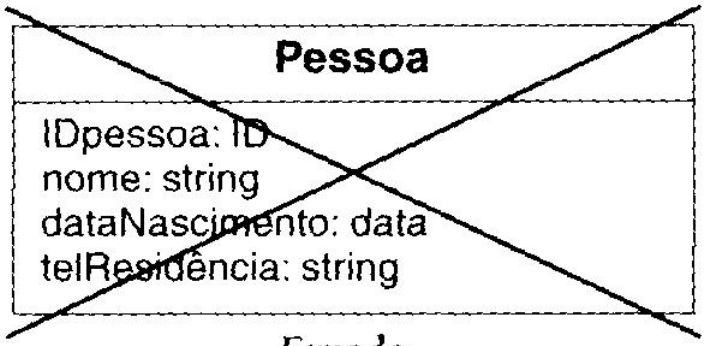
Outra Possível Solução



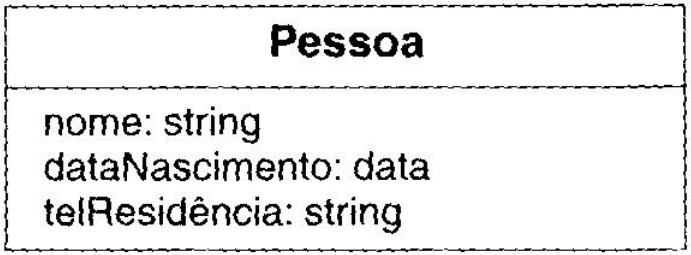
Representando Objetos com Valores e Classes com atributos



Atributos implícitos: identificadores



Errado



Correto

Ligação e Associação

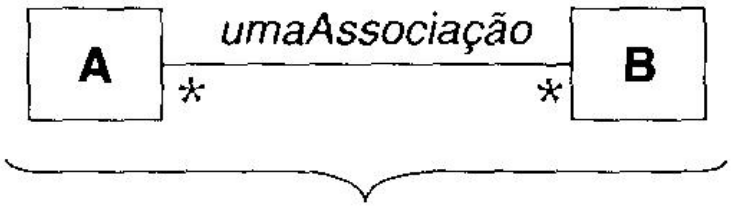


Diagrama de classes

Diagrama de objetos

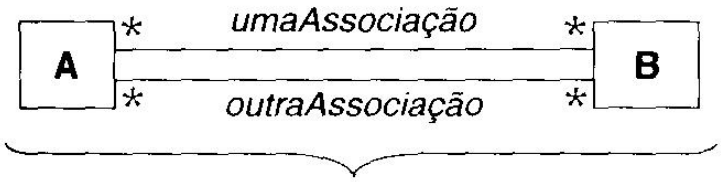


Diagrama de classes

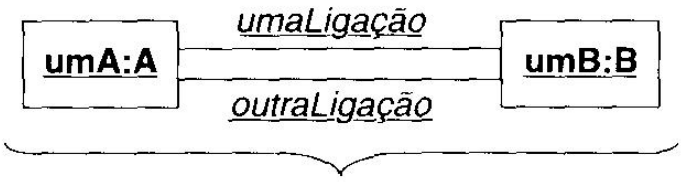
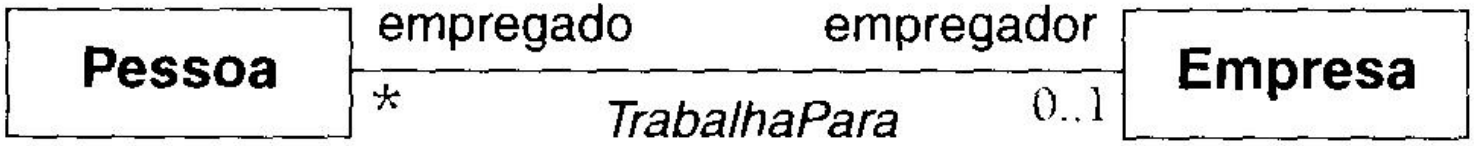
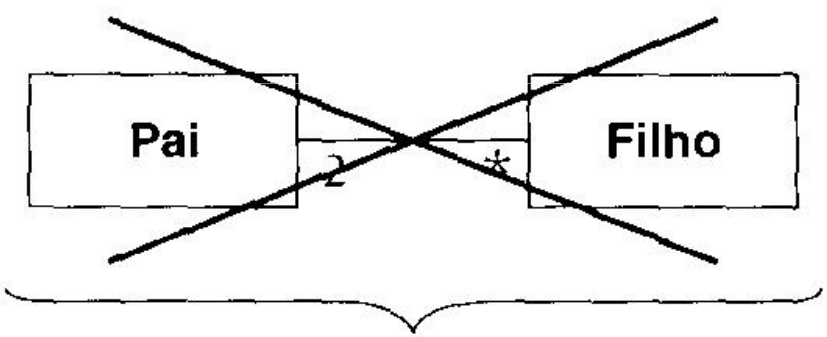


Diagrama de objetos

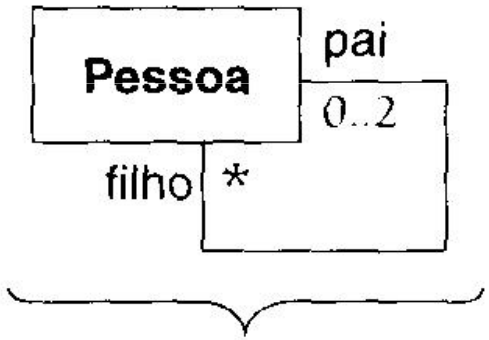
Associações com extremidades nomeadas



Associações entre Classes

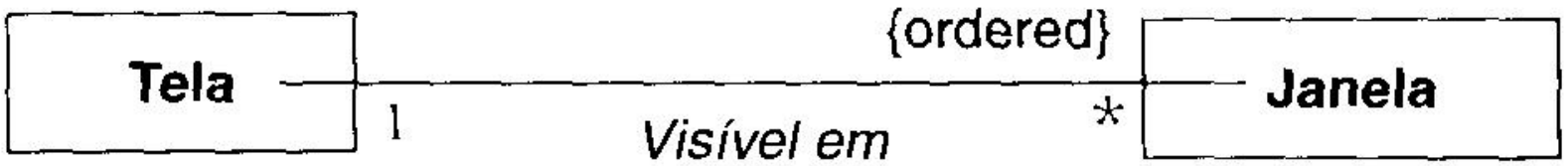


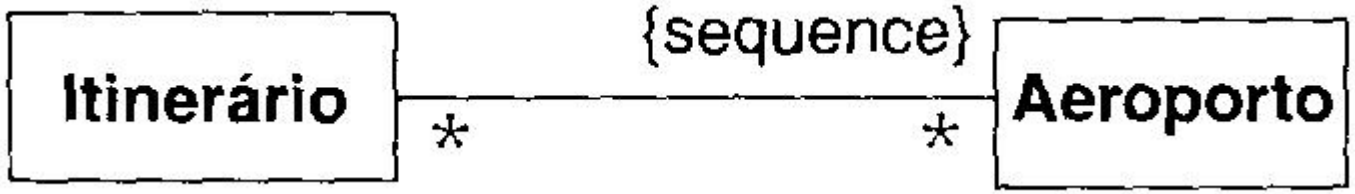
Modelo errado



Modelo correto

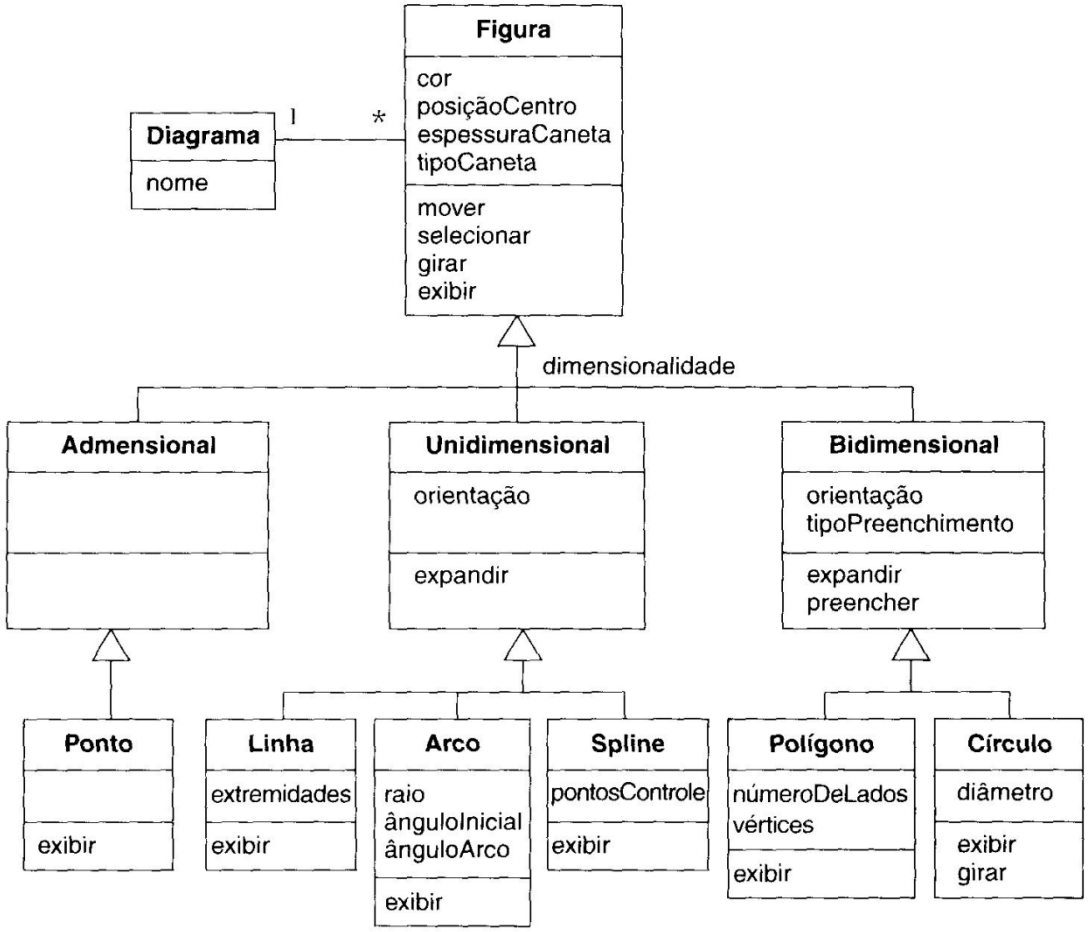
Seqüência, bag e ordenação



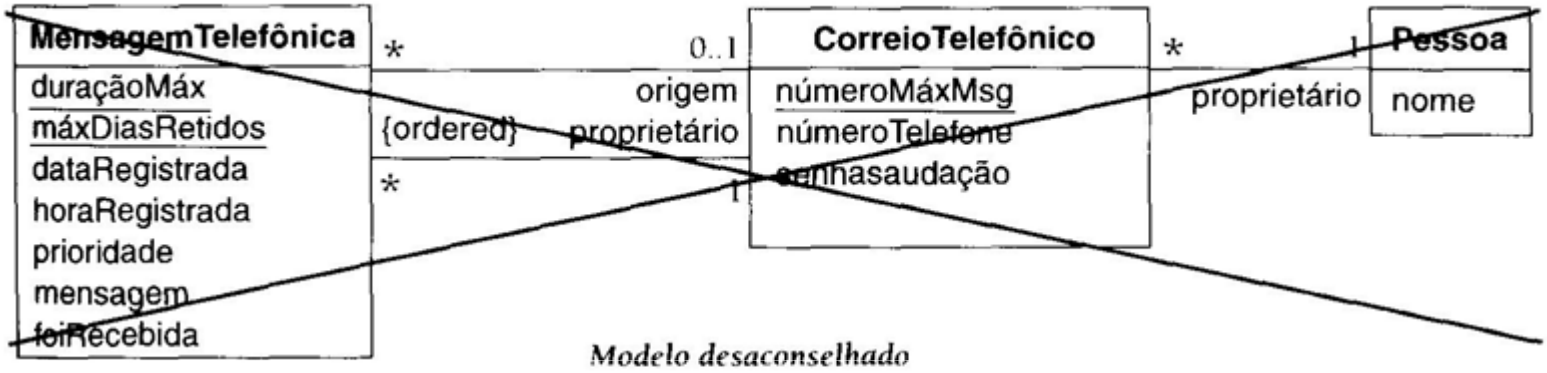


Ordenação = conjunto com ordenação e sem duplicatas
Seqüência= conjunto com ordenação e com duplicatas
Bag = conjunto sem ordenação com duplicatas

Generalização e Herança

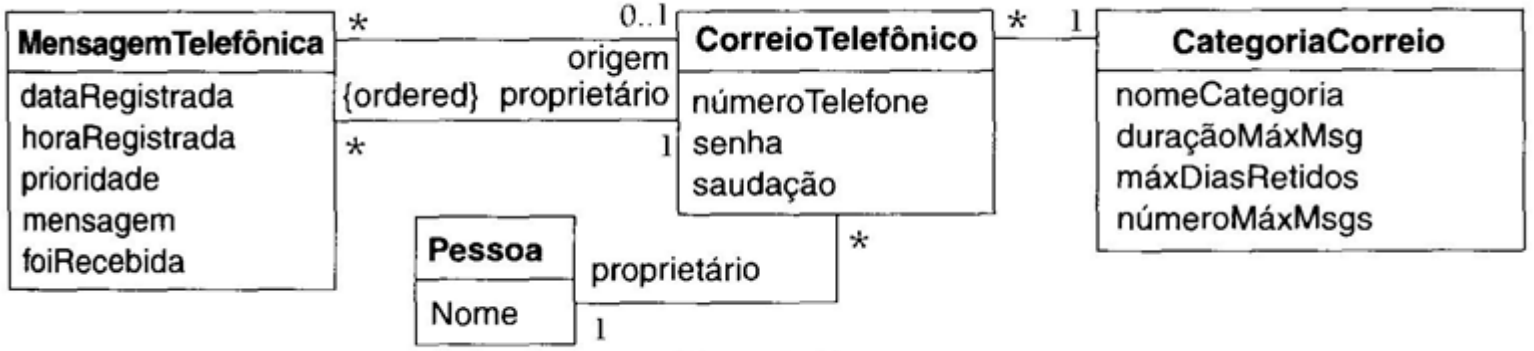


Um Exemplo de Modelo de Classes de um serviço de mensagens telefônicas. Qual o problema? Observe os atributos sublinhados...



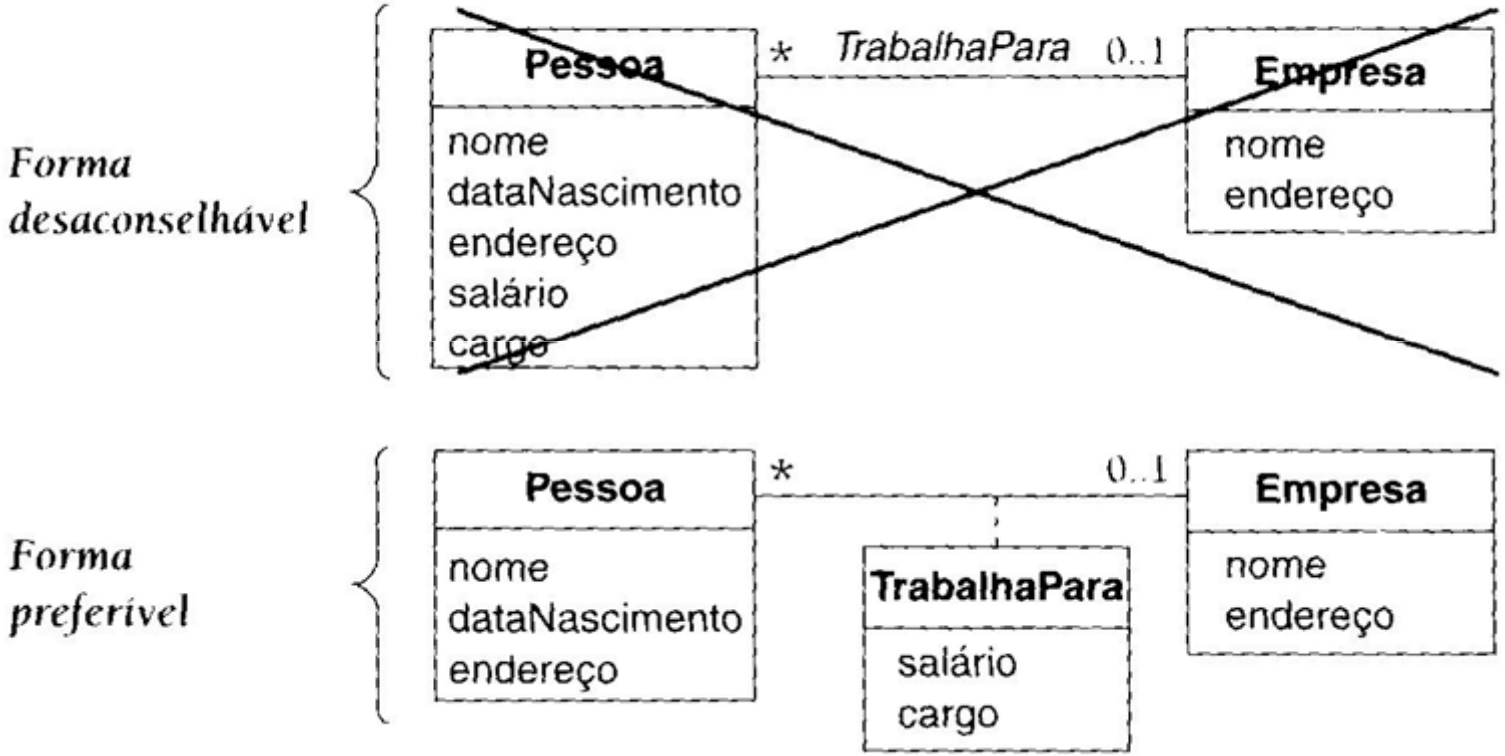
Modelo desaconselhado

- Introduzindo o conceito de grupo ou categoria….

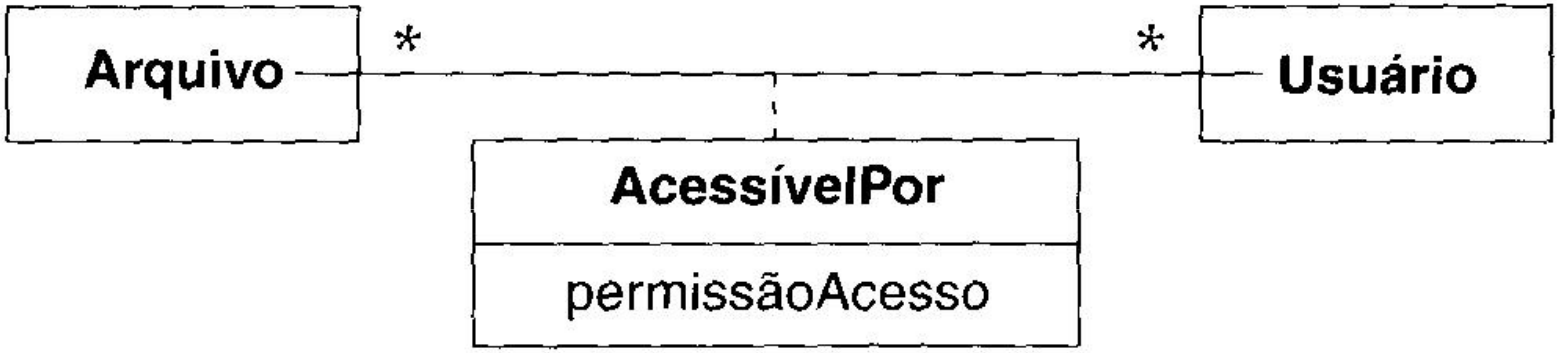


Modelo preferido

Classes de Associação com dados sobre associação

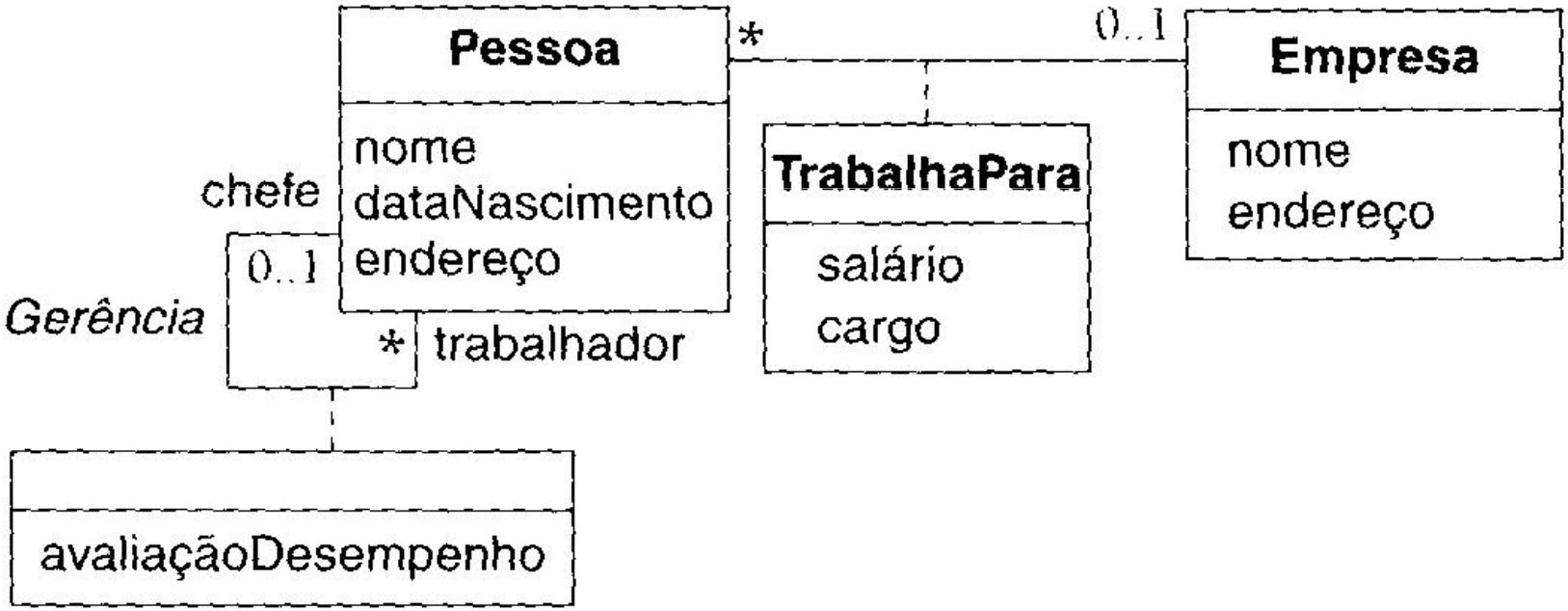


Classes de Associação

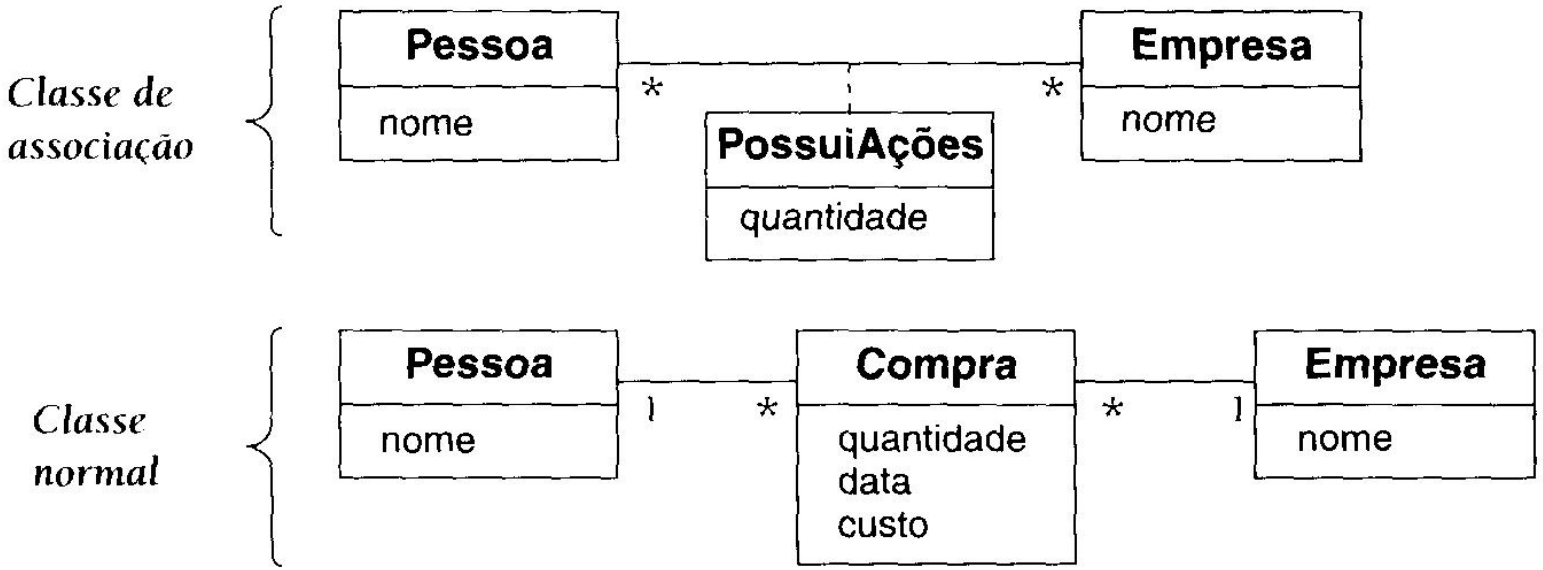


/etc/termcap	read	João da Silva
/etc/termcap	read-write	Maria Brito
/usr/silva/.login	read-write	João da Silva

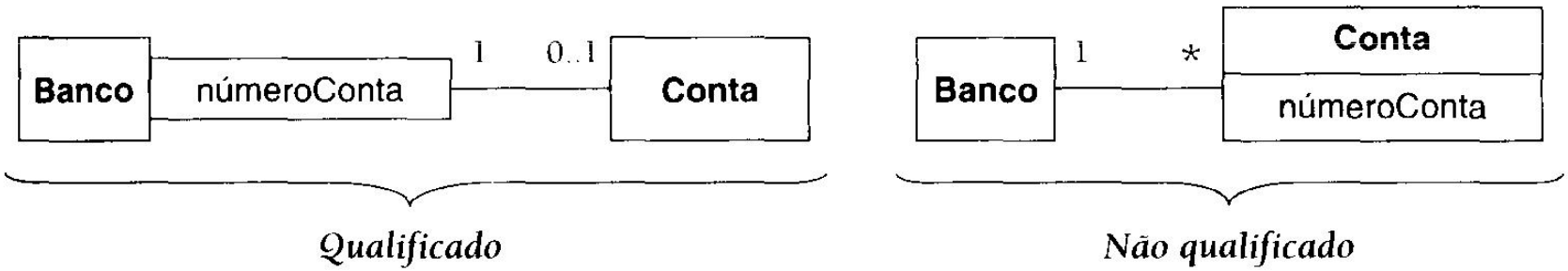
Classes de Associação - 2



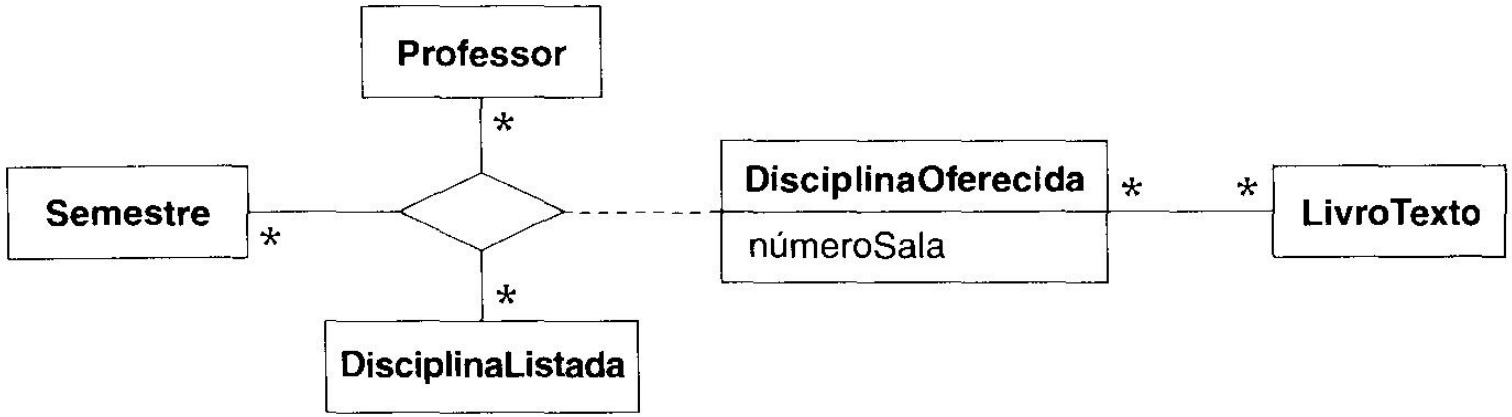
Classes de Associação e Classes Normais



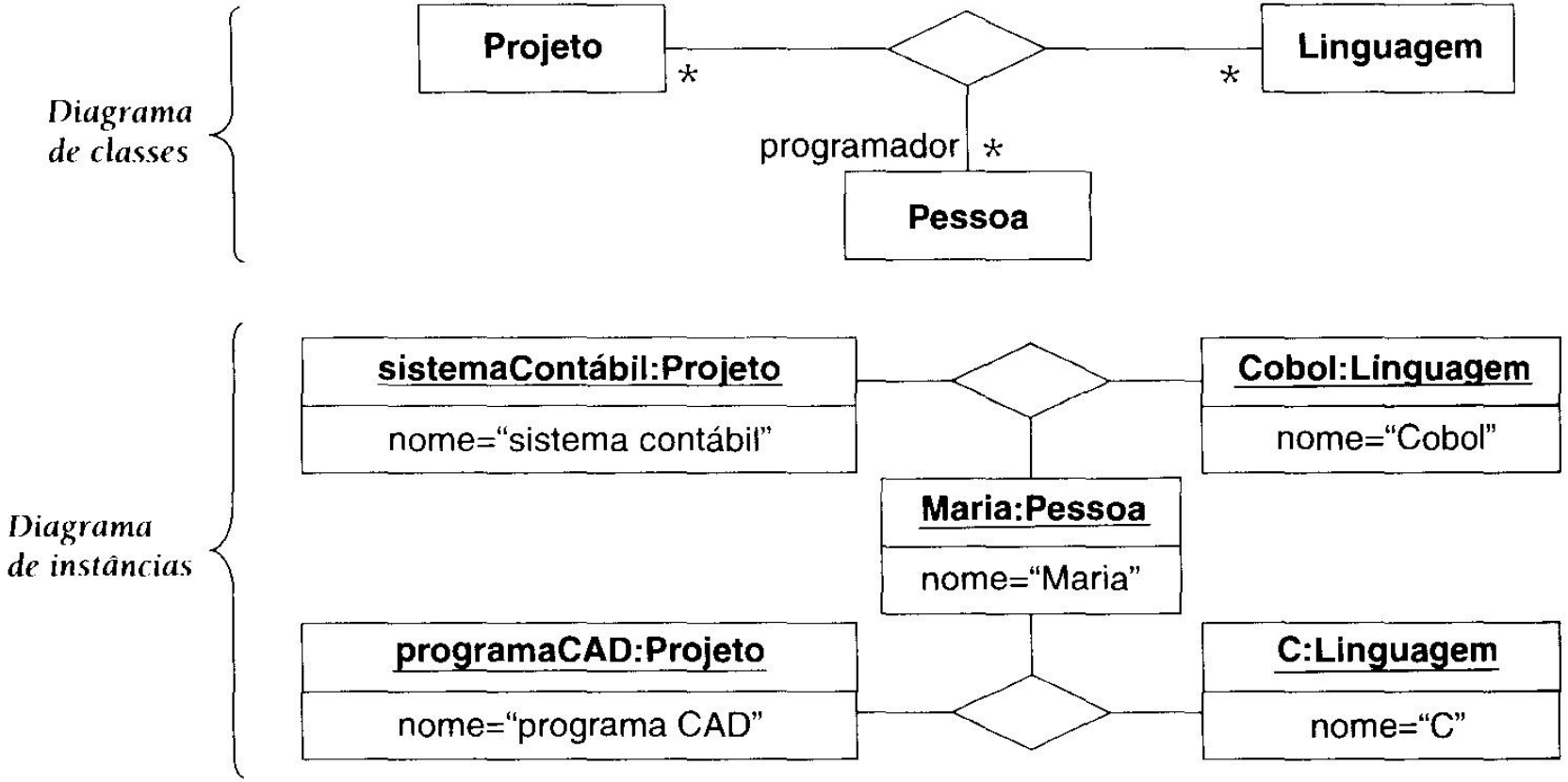
Associações Qualificadas



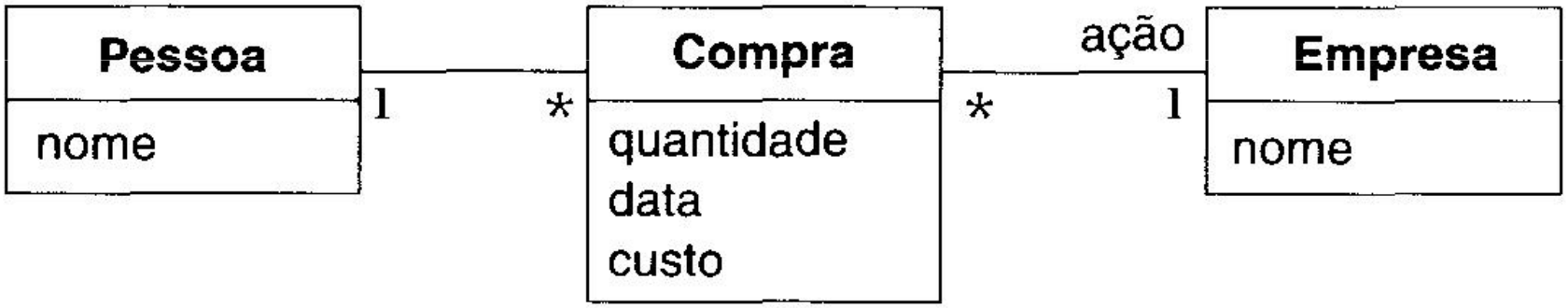
Associações N-árias Verdadeiras



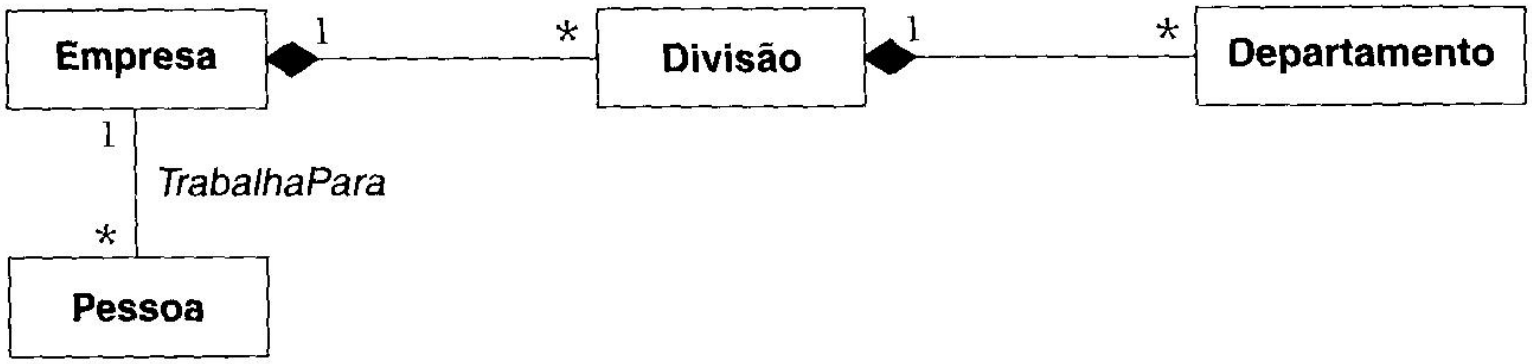
Associações n-árias



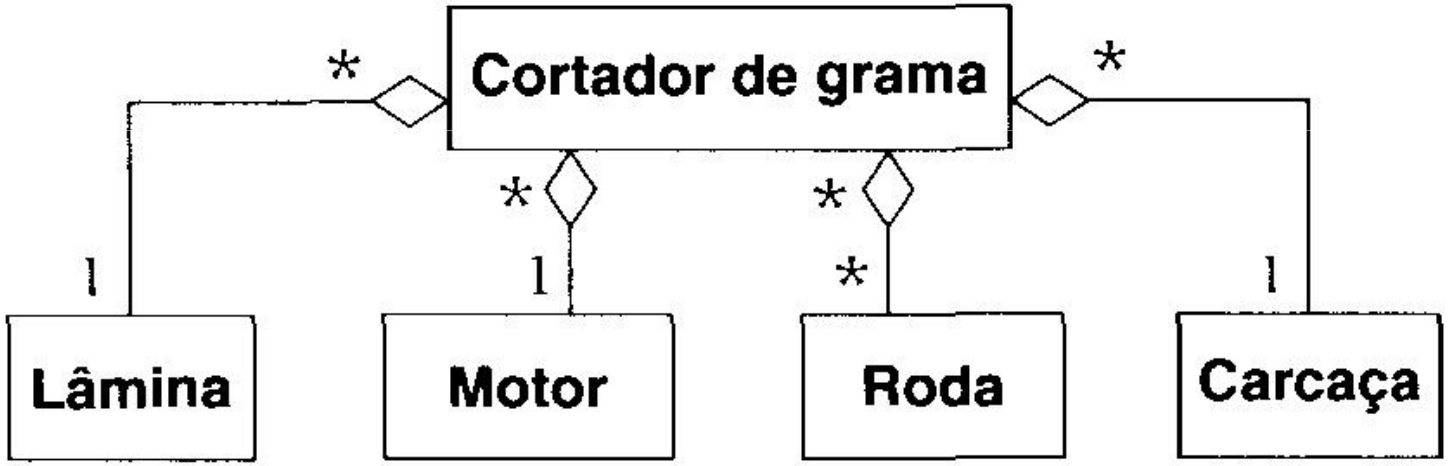
De Associação n-ária para binárias



Agregação versus Composição: Composição



Agregação versus Composição: Agregação



Sobre Associação, Composição e Agregação…

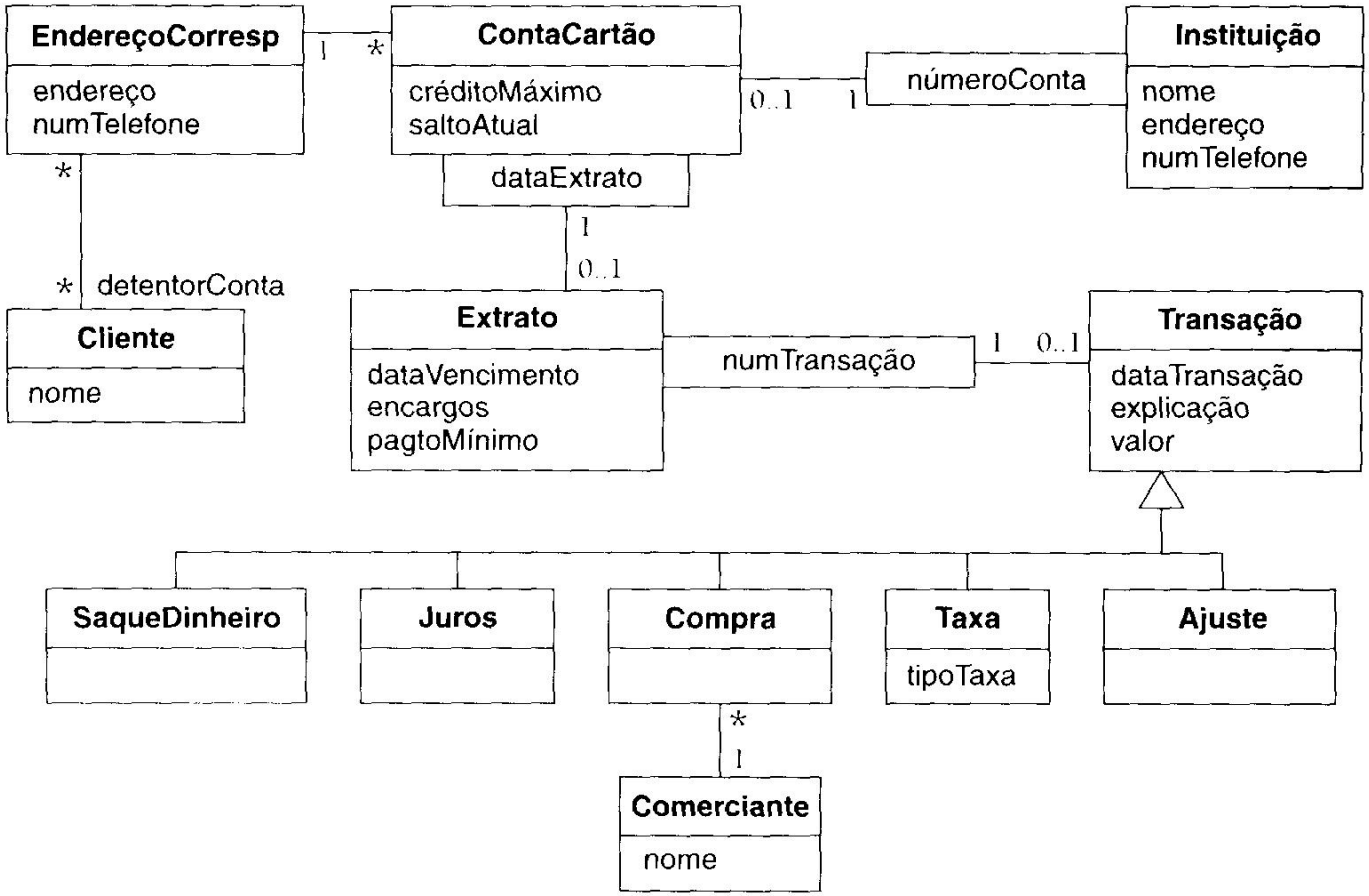
- “Francamente, as diferenças entre associação, agregação e composição confundem até mesmo projetista experientes. Se você achar que as distinções são úteis, use-as. Porém não gaste tempo ponderando sobre diferenças sutis entre esses conceitos. Do ponto de vista prático de um programador Java, é util saber quando uma classe armazena uma referência a outra classe.”
  - Cay Horstmann em Conceitos de Computação com o essencial de Java

Exercício: sistema de cartão de crédito

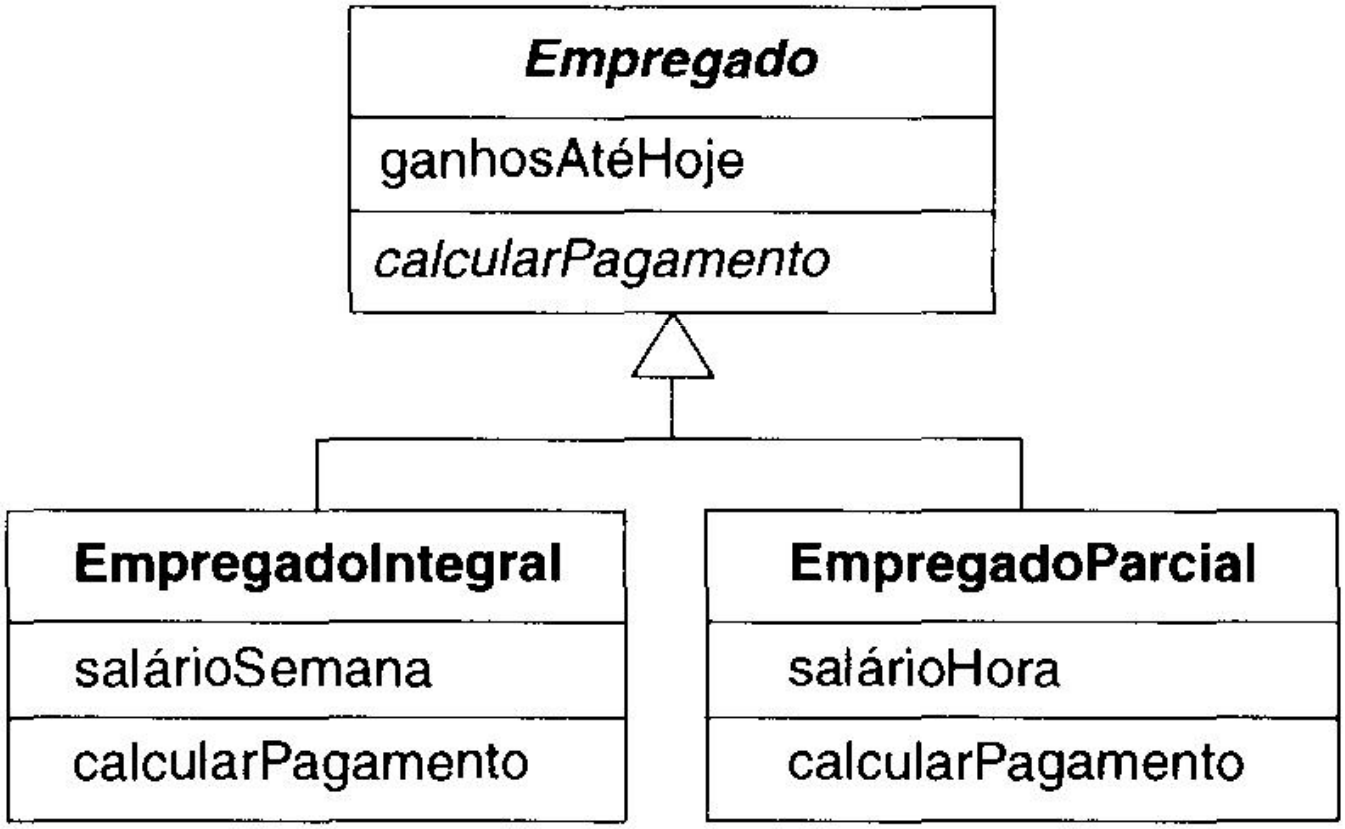
- Modele em uma diagrama de classes um subsistema de informações sobre operações de cartões de crédito contemplando cliente, comerciante, tipos de transação (compra, saque, taxa, juros, ajuste), conta bancária vinculada ao cartão e instituição financeira, extratos do cartão....

- Identifique as classes (conceitos mais relevantes), depois as relações....
- Use associações qualificadas quando for o caso....

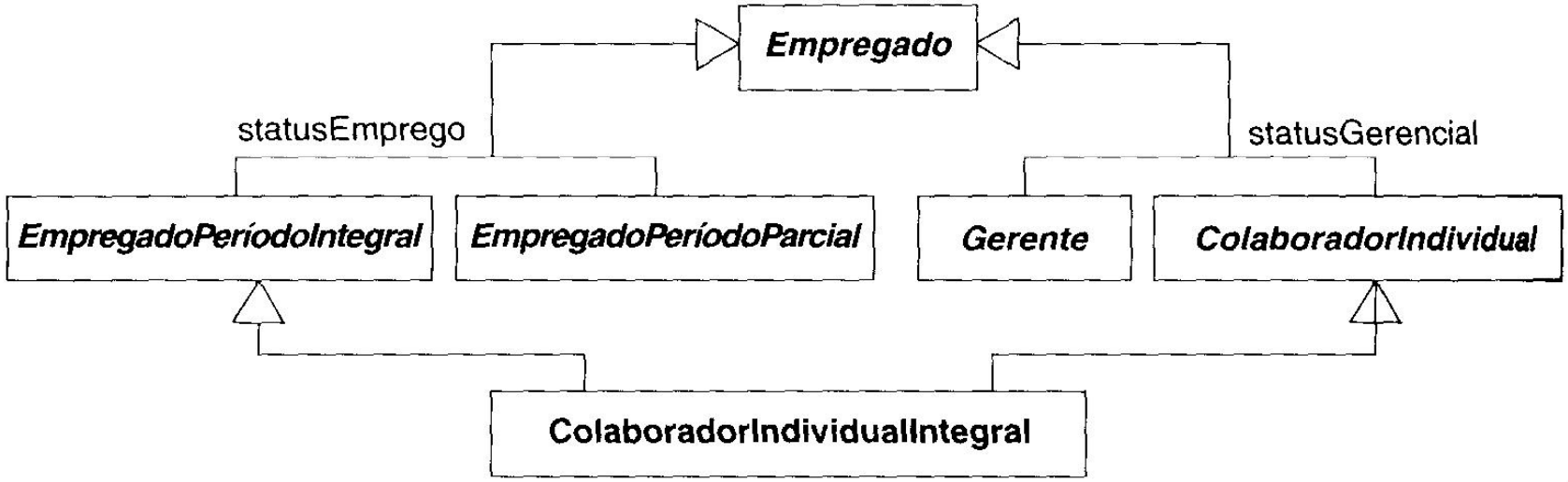
Exemplo: cartão de crédito



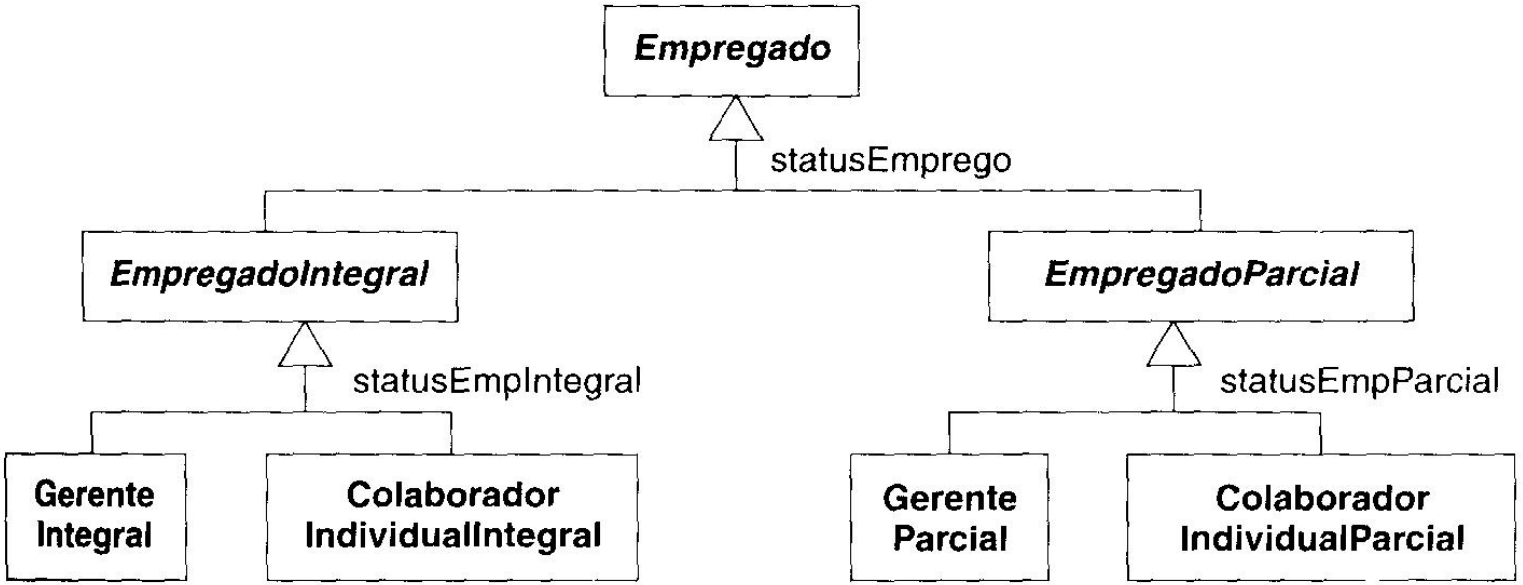
Classes Abstratas



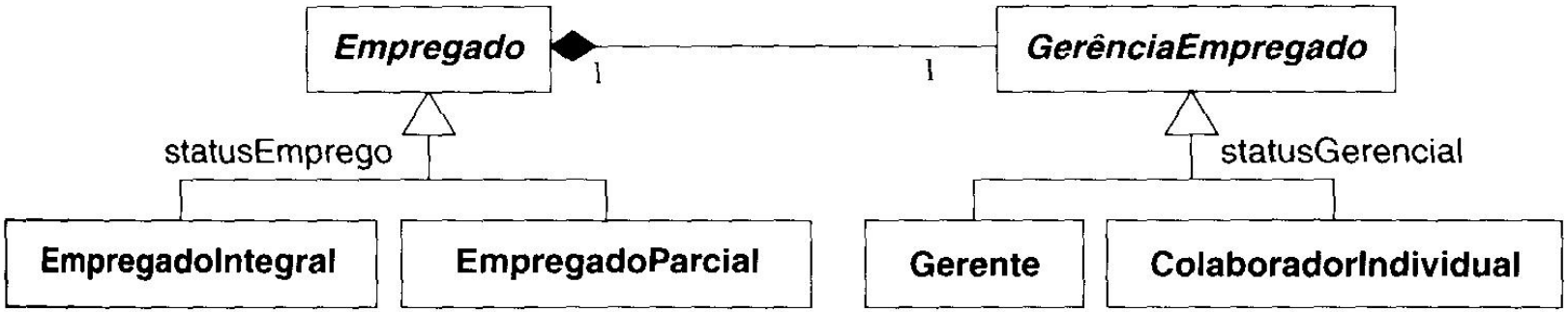
Herança Múltipla



Evitando Herança múltipla por Herança Aninhada



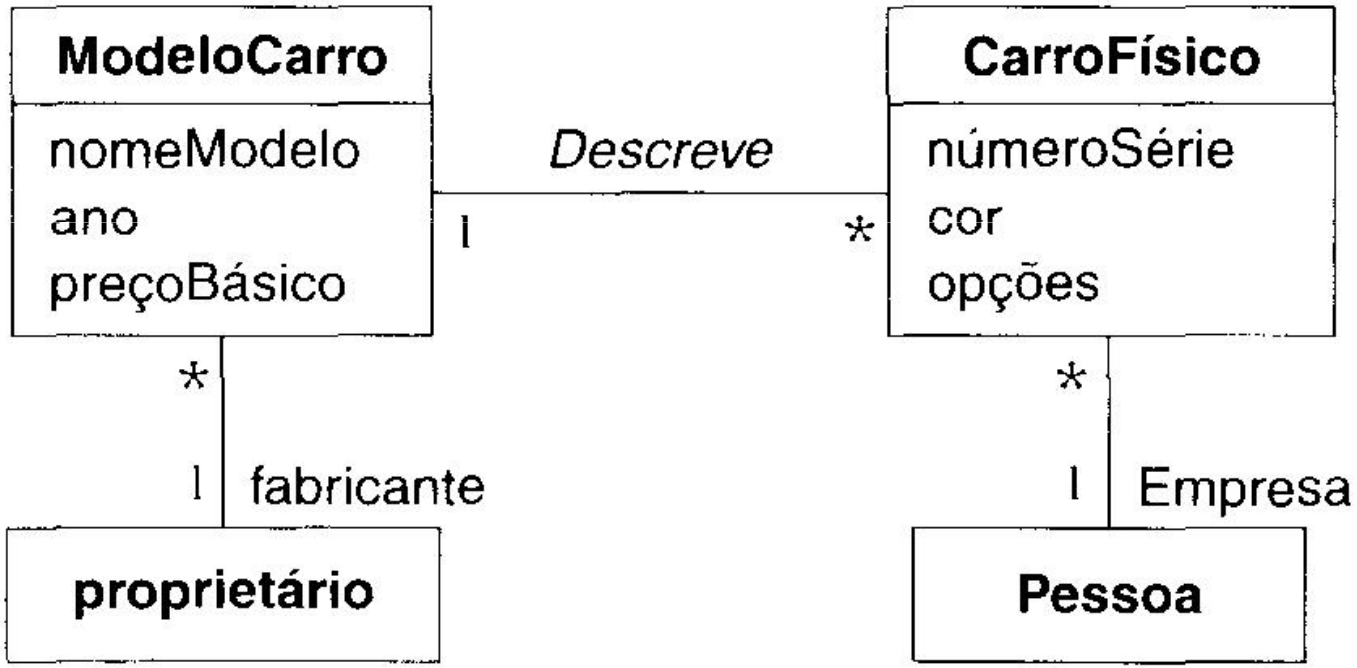
Evitando Herança múltipla com delegação



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/devem 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 alguns atributos com escopo de classe para manter a abrangência de uma classe. Via de regra, você deve evitar número excessivo de 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 (exemplo CategoriaCorreio)

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

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

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

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

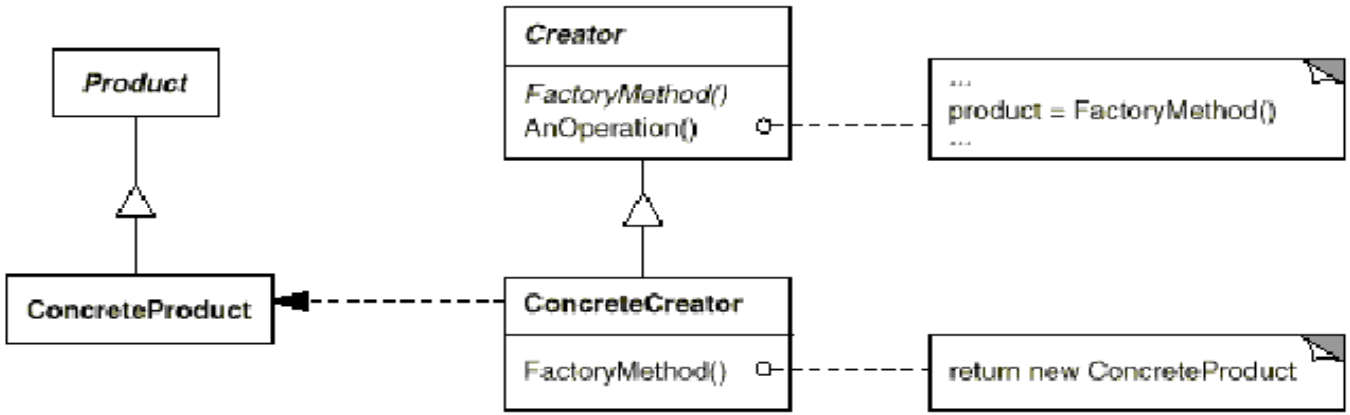
      createImage("image.jpg");

      - 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

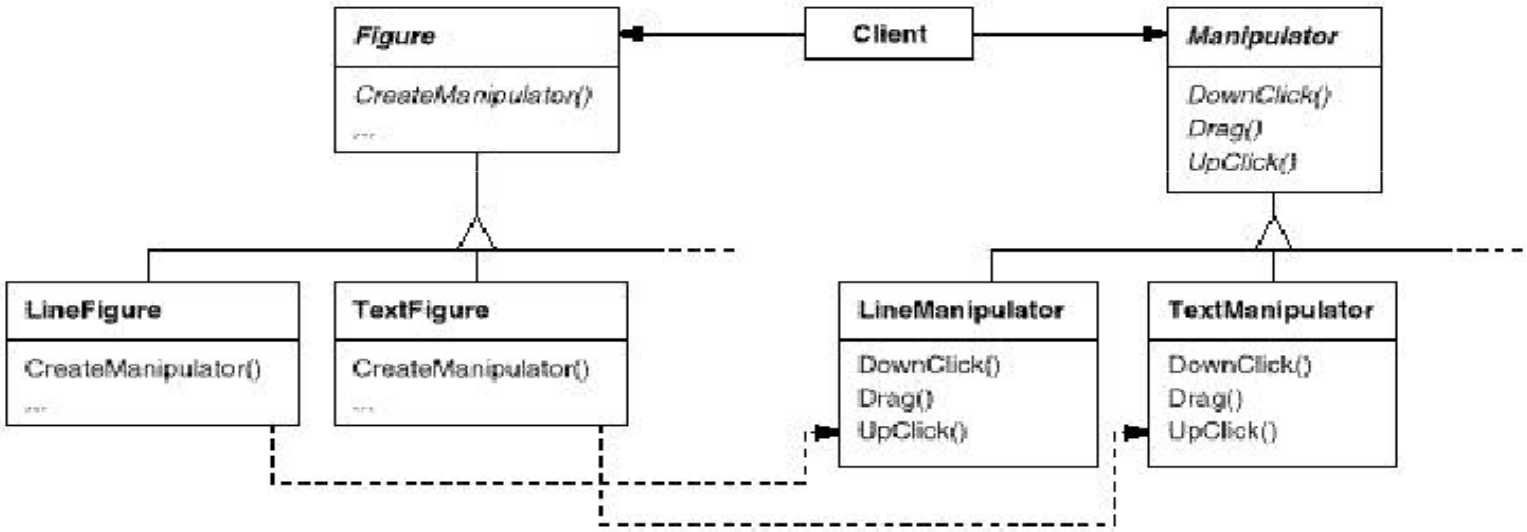
Padrão de Projeto Fábrica Abstrata (Abstract Factory Design Pattern)

- 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

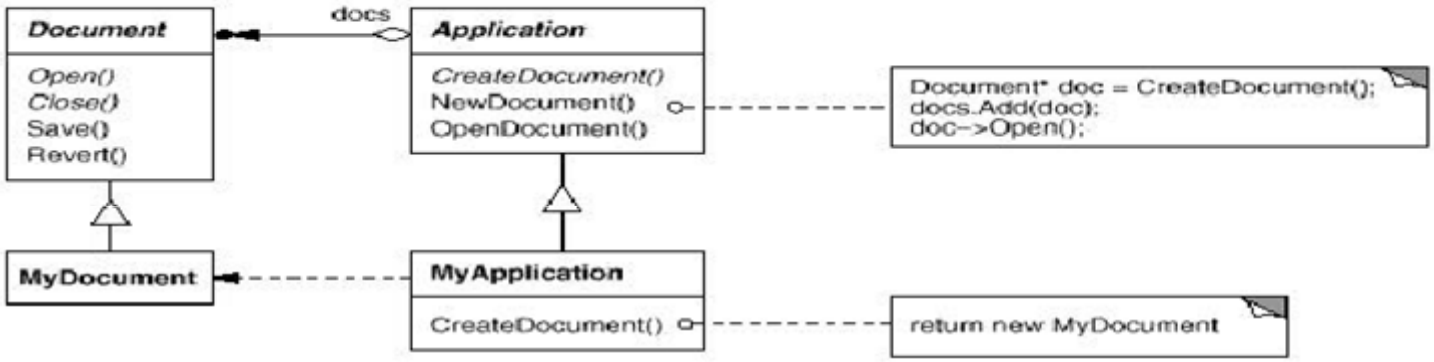
Estrutura


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

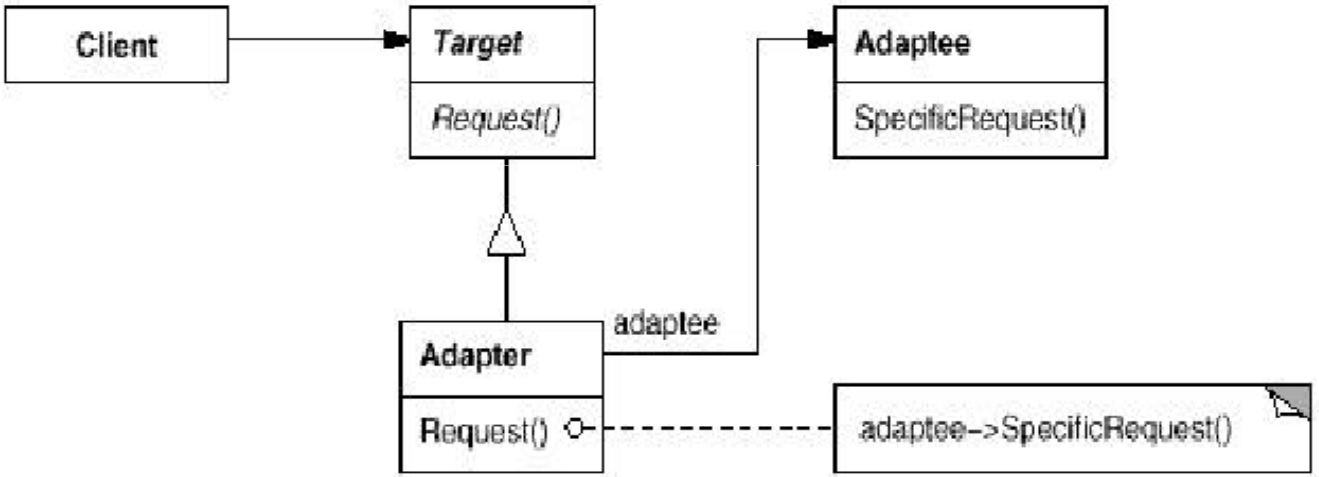
Structural Design Patterns

- 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

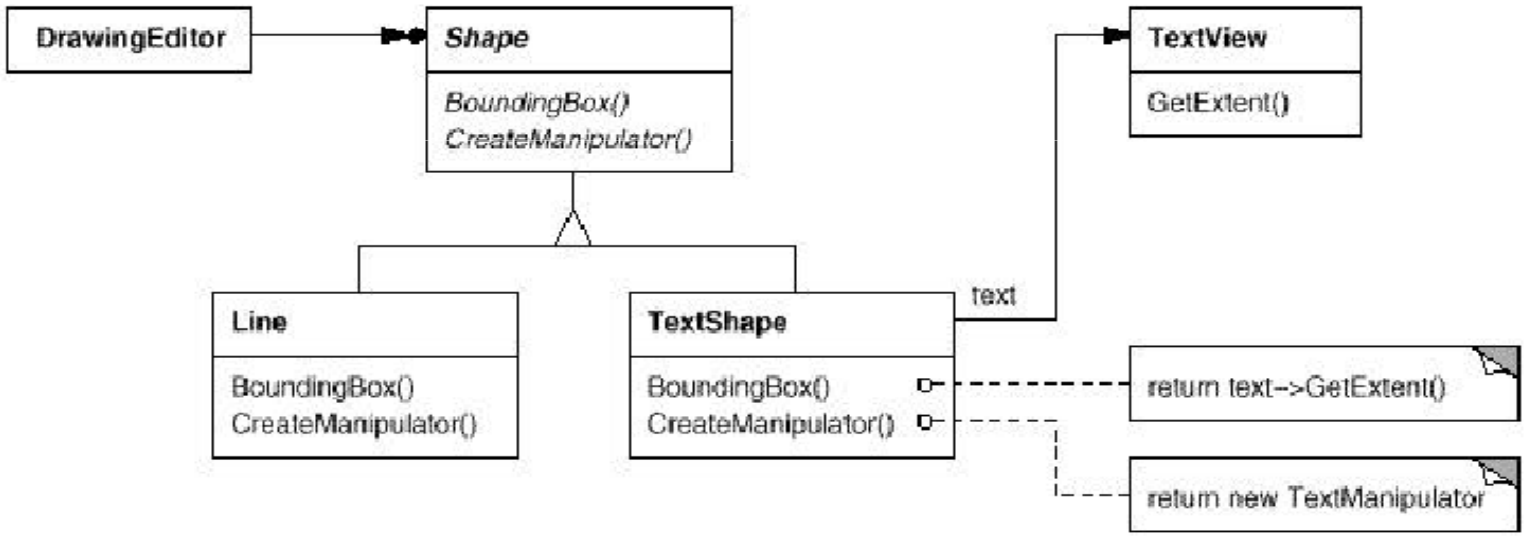
- Intent
  - ⇒ Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also Known As
  - ⇒ Wrapper
- Motivation
  - ⇒ Sometimes a toolkit or class library can not be used because its interface 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

Estrutura


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

Exemplo

⇨ Example:

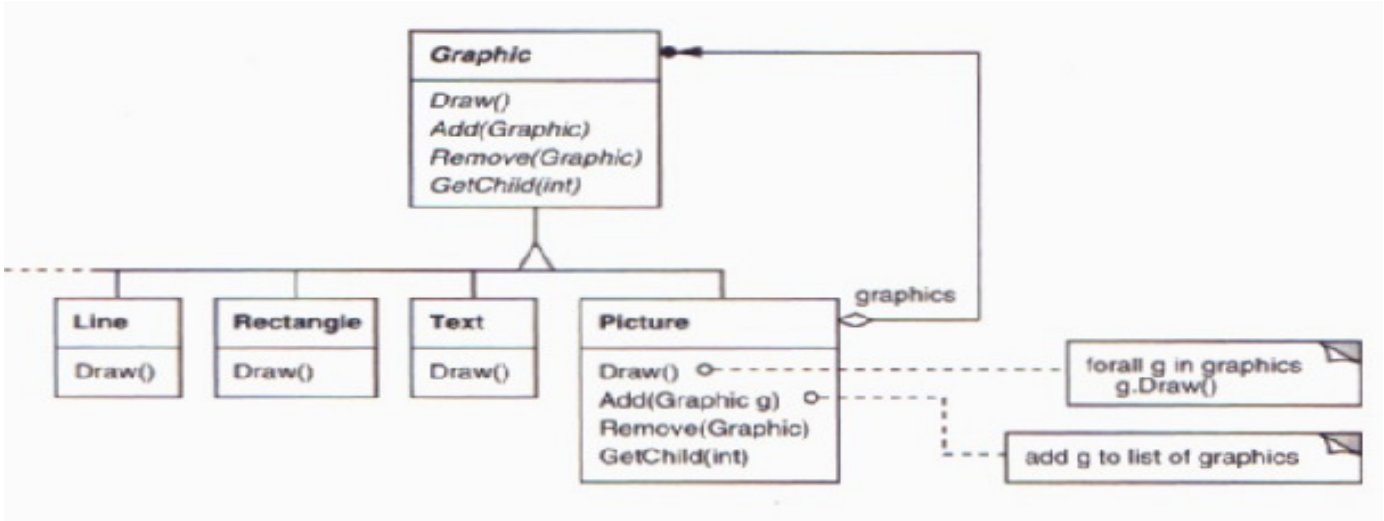


5.1.2 Structural Design Patterns

- 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

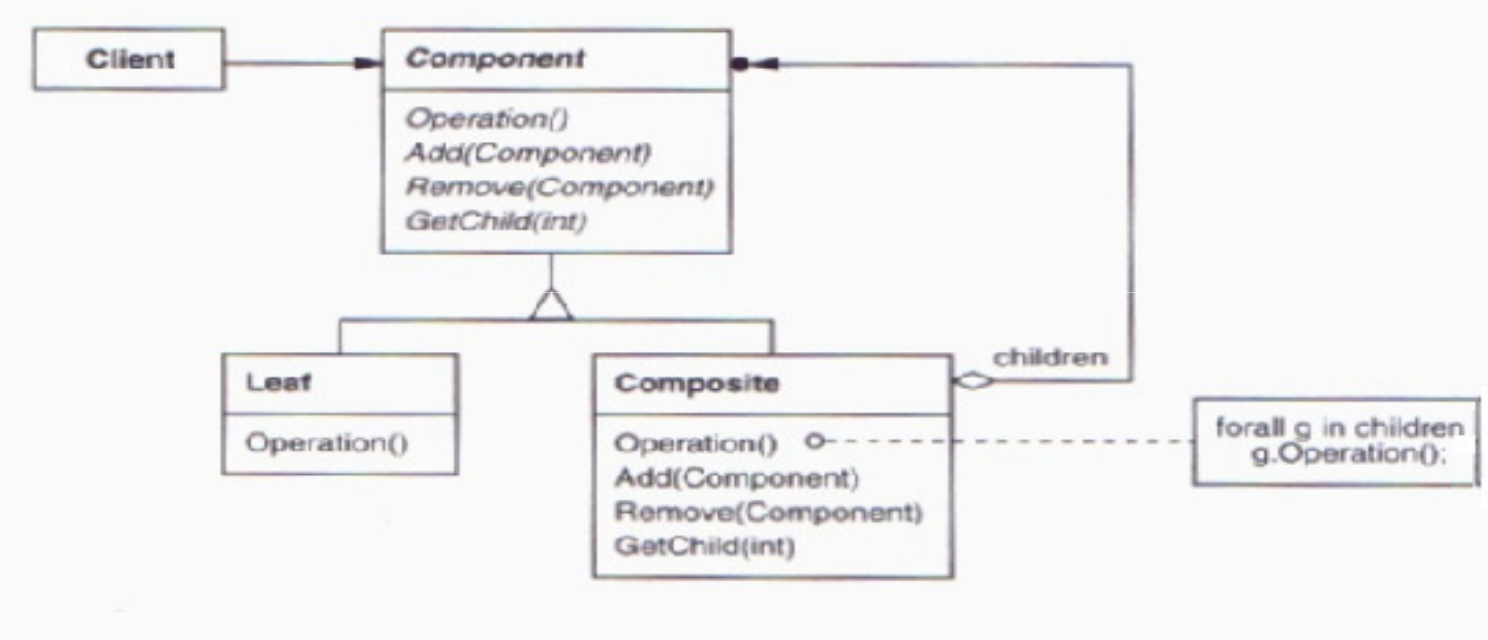
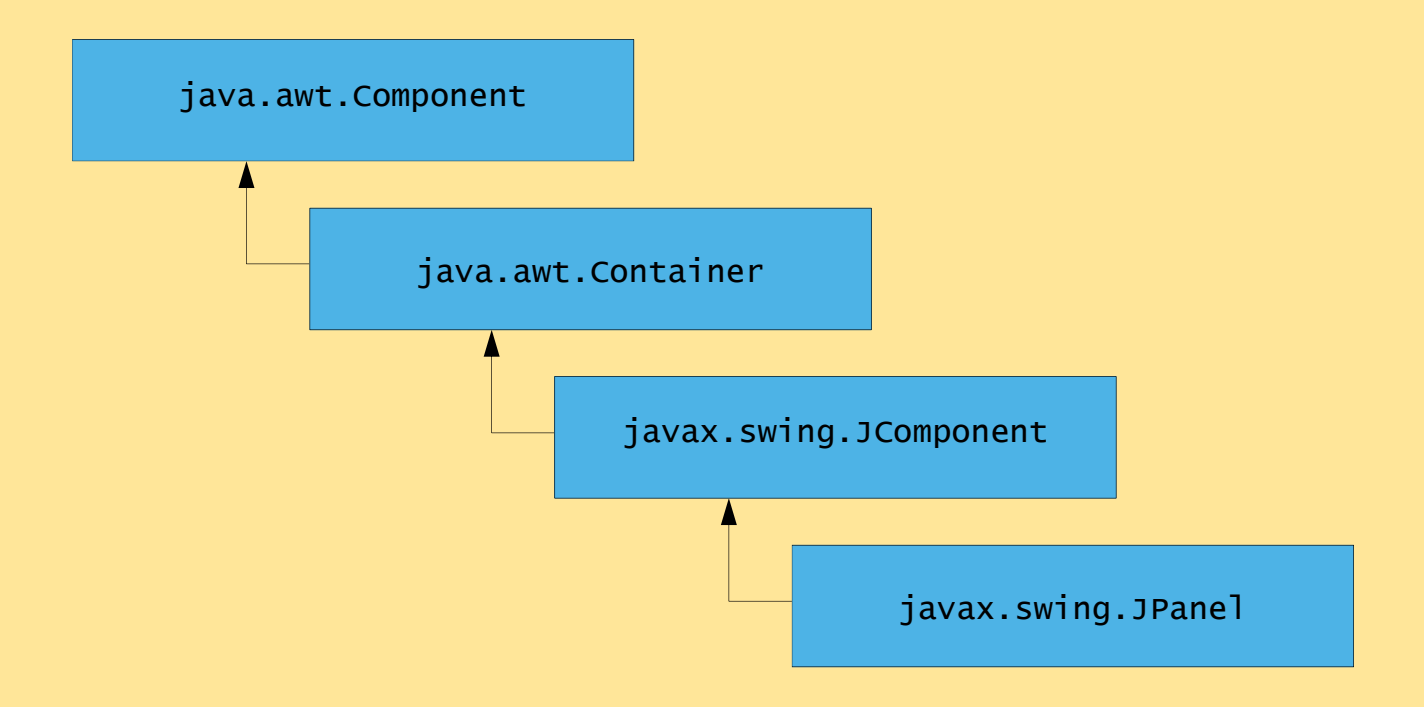

Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

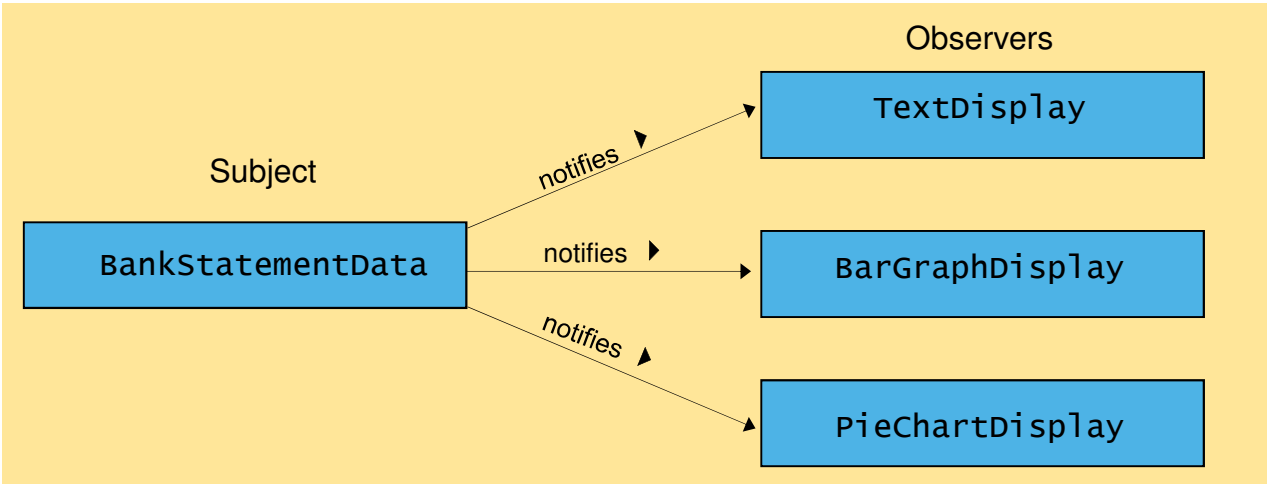
Fig. 4.27 Inheritance hierarchy for class `JPanel`



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

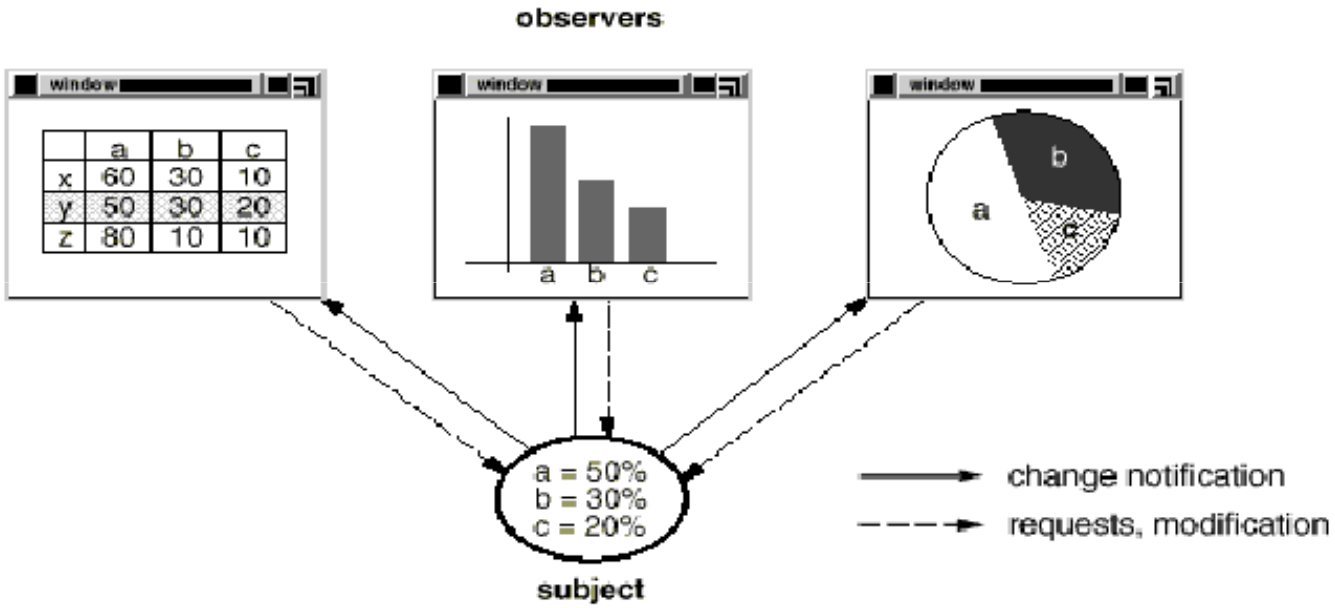
Fig. 4.28 Basis for the Observer design pattern



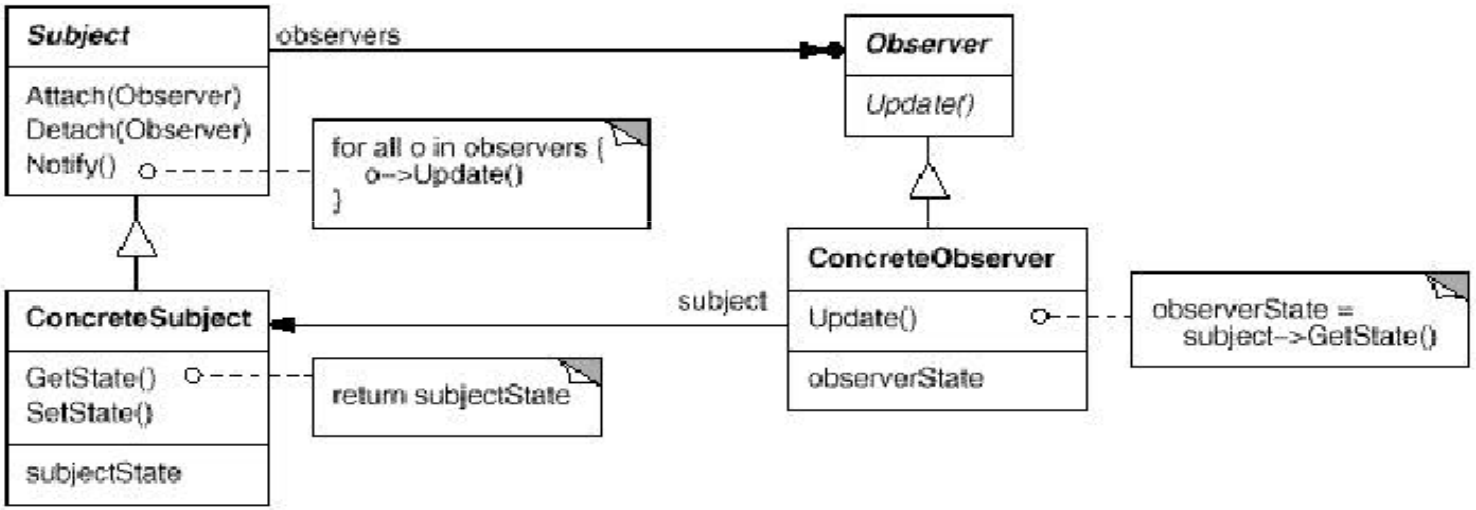
The Observer Pattern

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

Estrutura


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

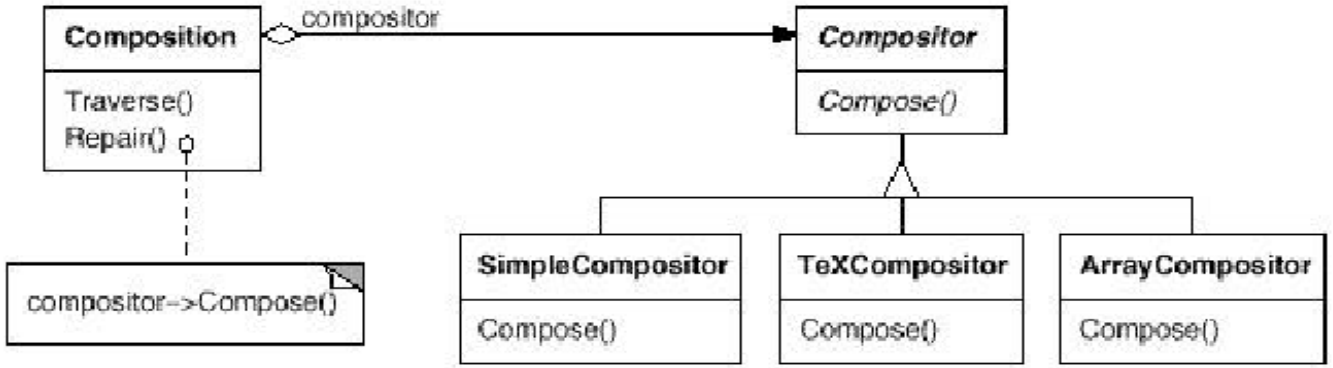
- Liabilities
  - Possible cascading of notifications
  - Observers are not necessarily aware of each other and must be careful about triggering updates
  - Simple update interface requires observers to deduce changed item

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:

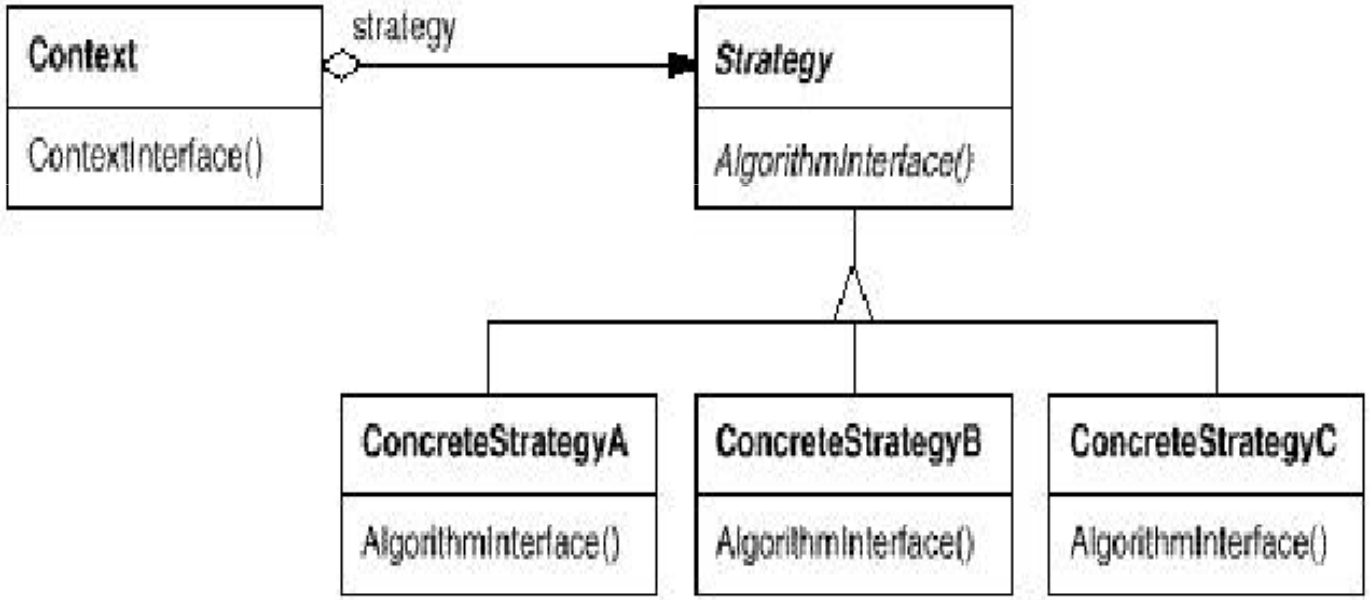


Aplicabilidade

- 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