

41951- ANÁLISE DE SISTEMAS

Desenho por objetos: da análise para o código

Ilídio Oliveira

V2024-05-07

universidade de aveiro
departamento de eletrónica,
telecomunicações e informática



deti

Objetivos de aprendizagem

Explicar como os casos de utilização podem ser usados para orientar as atividades de desenho

Explicar os princípios do baixo acoplamento e alta coesão em OO

Explicar os princípios GRASP propostos por Larman

“Tipos de coisas” da AOO: candidatos

Papéis de pessoas?

Cliente, Empregado, Cozinheiro

Lugares e pontos de serviço?

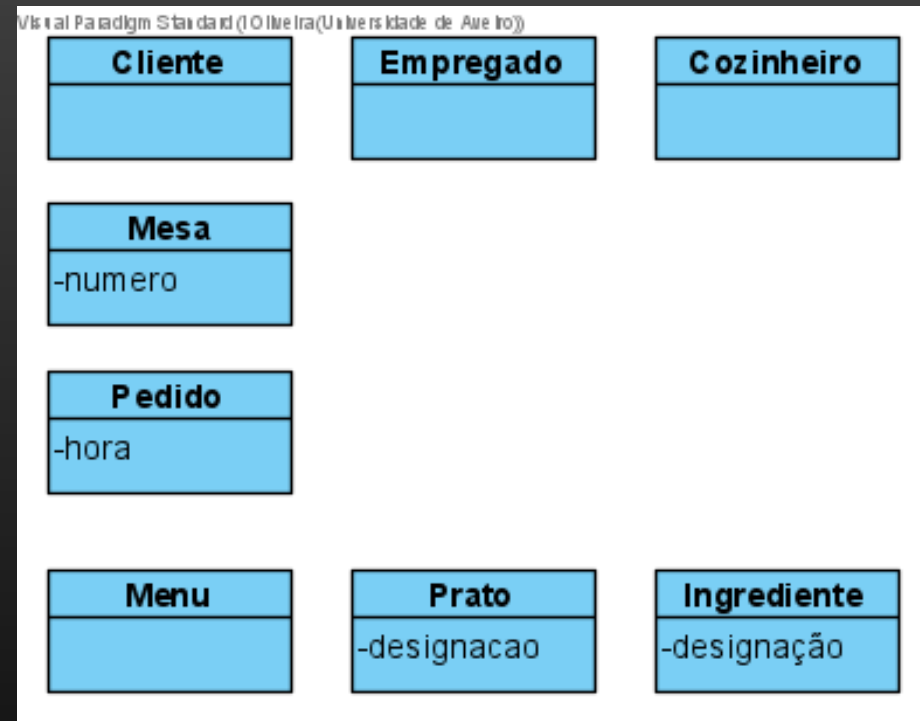
Mesa? Restaurante? Sala?

Transações de bens/serviços?

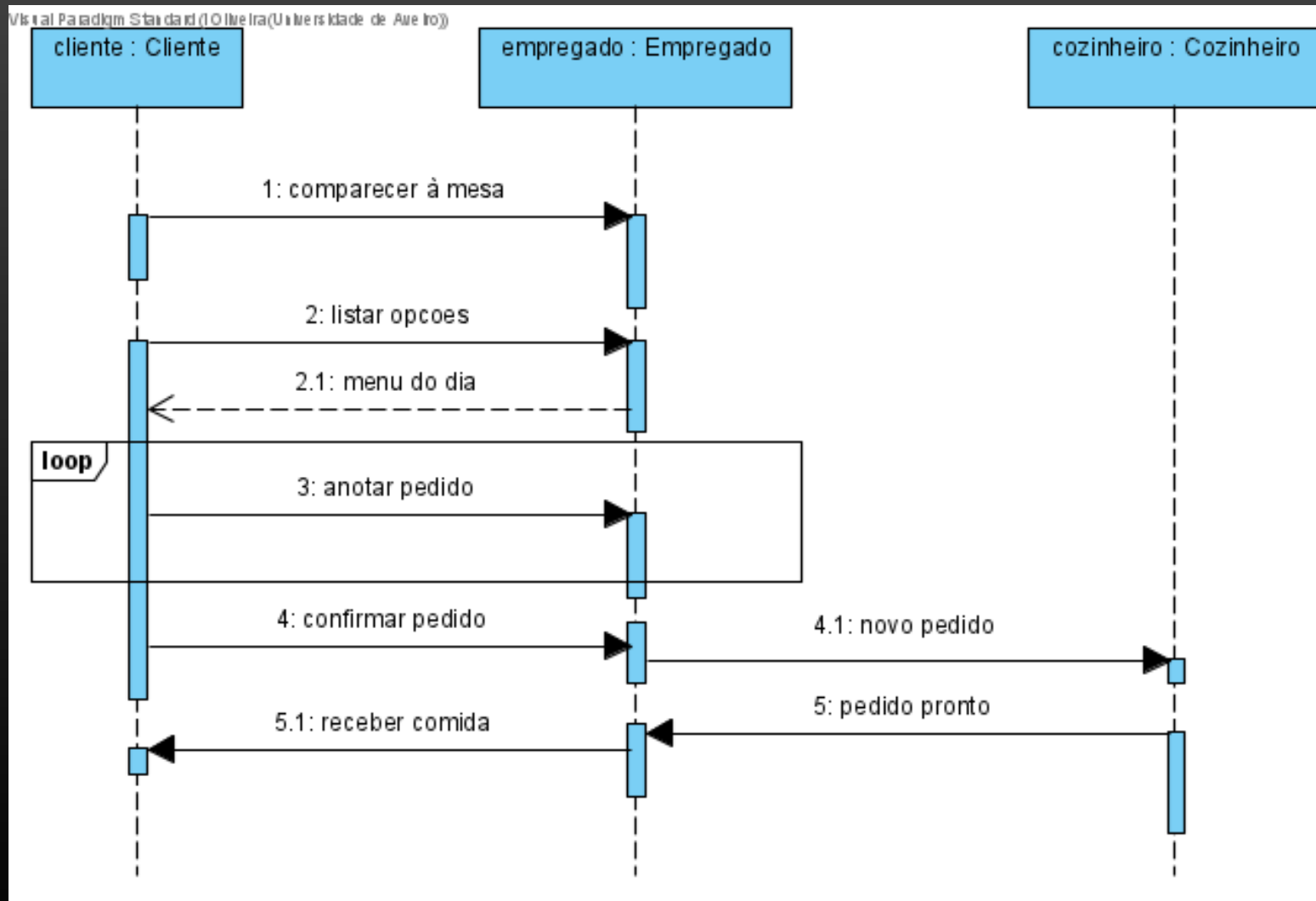
Pedido; comanda/talão?

Itens numa transação?

Prato/Opção; Menu;
Ingredientes?



Interação entre “participantes”



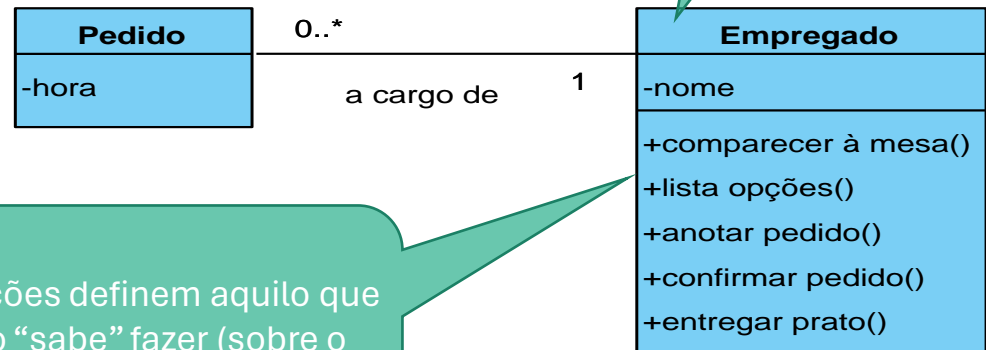
Os diagramas de classes e os de interação “distribuem” responsabilidades

Análise por classes define dois grupos de **responsabilidades**:

- O que é que cada tipo é responsável por conhecer/guardar
- O que é que cada tipo é responsável por fazer

Atributos e objetos associados definem o âmbito do que o objeto guarda.

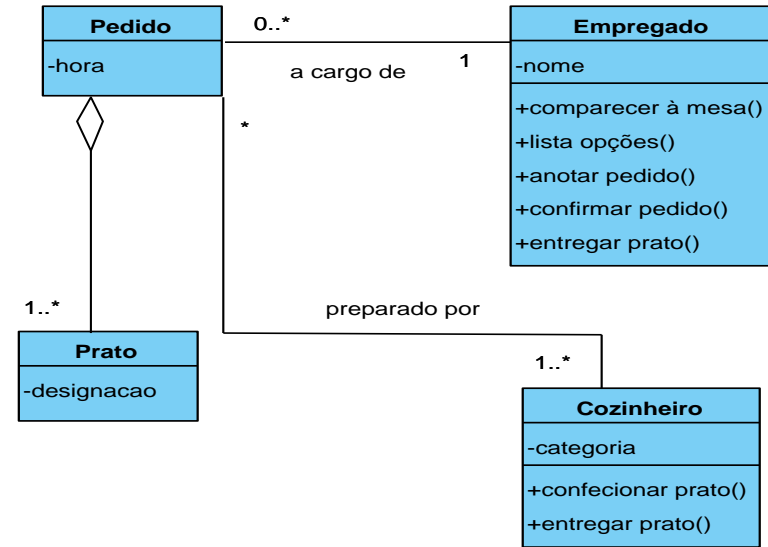
Visual Paradigm Standard (I Oliveira (Universidade de Aveiro))



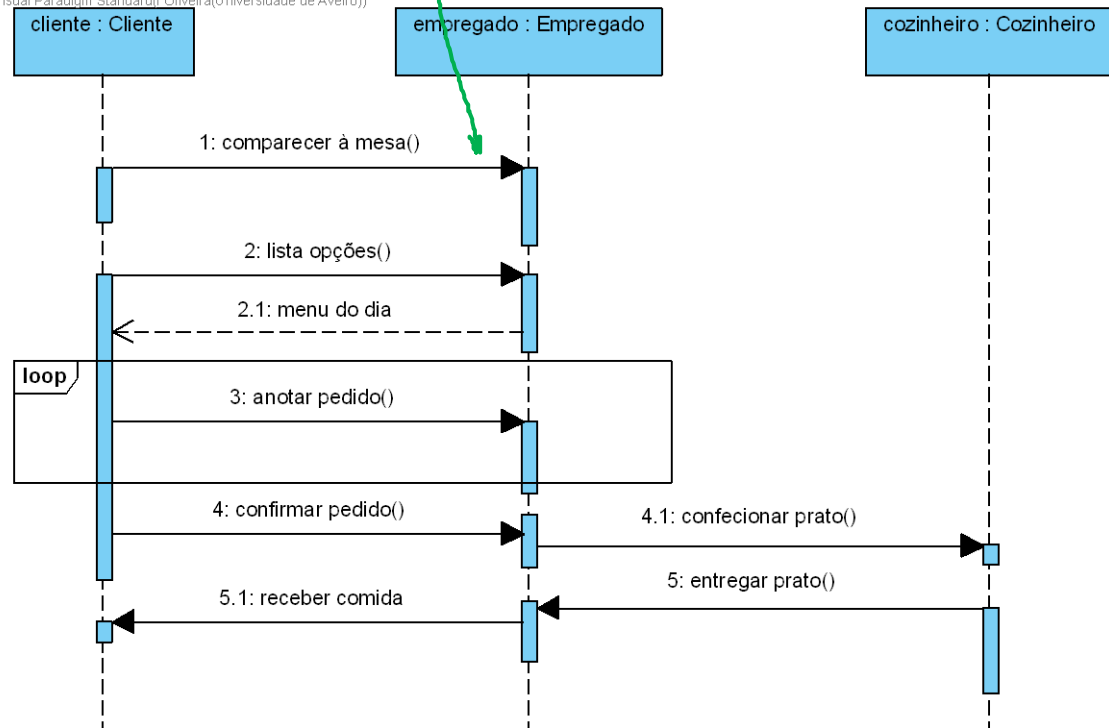
As operações definem aquilo que o objeto “sabe” fazer (sobre o conhecimento que guarda)

Vista complementares

Visual Paradigm Standard (I. Oliveira (Universidade de Aveiro))



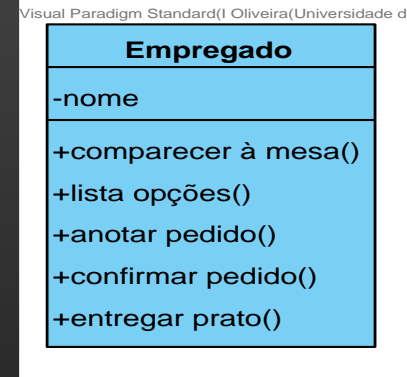
Visual Paradigm Standard (I. Oliveira (Universidade de Aveiro))



Este raciocínio, no domínio do problema, pode ser aplicado para o para o código?

Aproveitar o modelo do domínio para “inspirar” a implementação do código!

- Modelo do domínio explora o vocabulário do problema
- Modelo do domínio explica os relacionamentos relevantes e algumas regras (formas de associar objetos)
- A implementação **não usa diretamente** as representações do domínio do problema...



```
public class Employee {

    private long id;
    private String firstName;
    private String lastName;
    private String emailId;

    public Employee() {

    }

}
```

Em código....

Não é um conceito do domínio, mas uma entidade/unidade que faz sentido no programa.

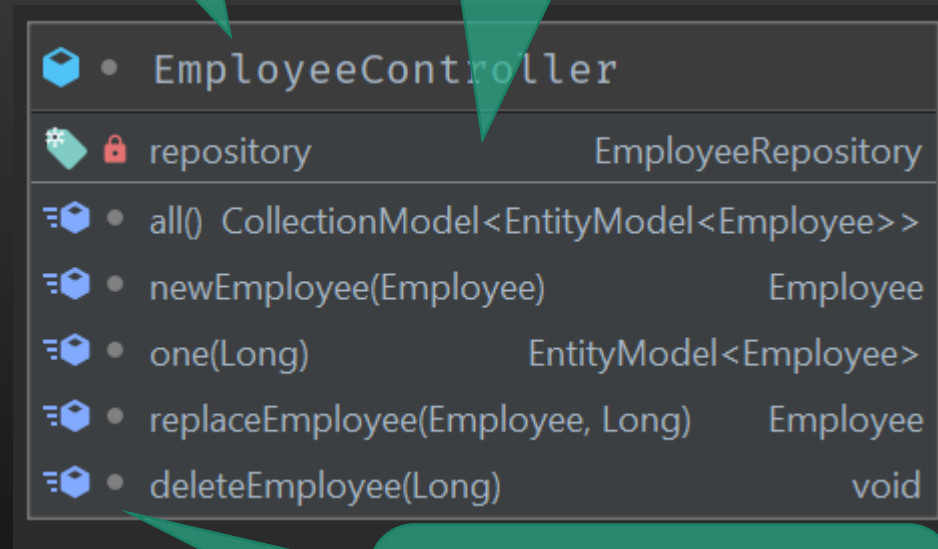
Atributos e objetos associados definem o âmbito do que o objeto guarda.

A classe passa a representar uma entidade do software

- Pode ser o “mesmo conceito” do domínio
- Mas pode ser outro tipo de entidade, com significado apenas para o “universe” do software

Mesmo mecanismo mental

- classificar em tipos (=classes)
- a classe funciona como uma unidade modular, especializada, com conhecimento e operações limitadas
- os objetos são instâncias de classes
- os objetos colaboram “em rede” trocando mensagens!



•	EmployeeController	
repository		EmployeeRepository
•	all()	CollectionModel<EntityModel<Employee>>
•	newEmployee(Employee)	Employee
•	one(Long)	EntityModel<Employee>
•	replaceEmployee(Employee, Long)	Employee
•	deleteEmployee(Long)	void

As operações definem aquilo que o objeto “sabe” fazer (sobre o conhecimento que guarda/tem acesso)

Como evoluir os resultados da análise para o desenho?

APPLYING UML AND PATTERNS

An Introduction to Object-Oriented Analysis and Design and Iterative Development

THIRD EDITION

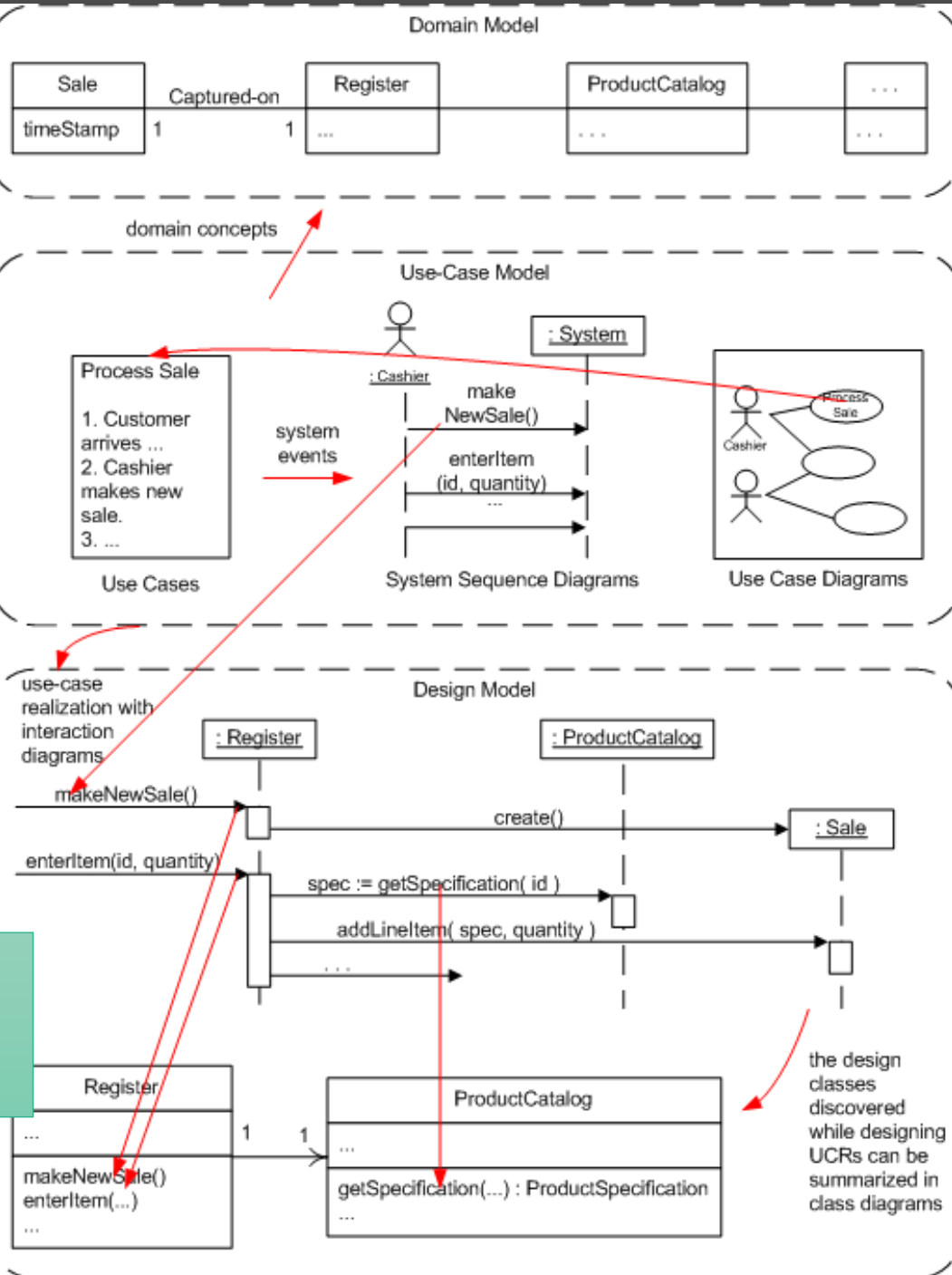


"People often ask me which is the best book to introduce them to the world of OO design. Ever since I came across K. Applying UML and Patterns has been my unreserved choice."

CRAIG LARMAN
Foreword by Philippe Kruchten

conceptual classes in the domain inspire the names of some software classes in the design

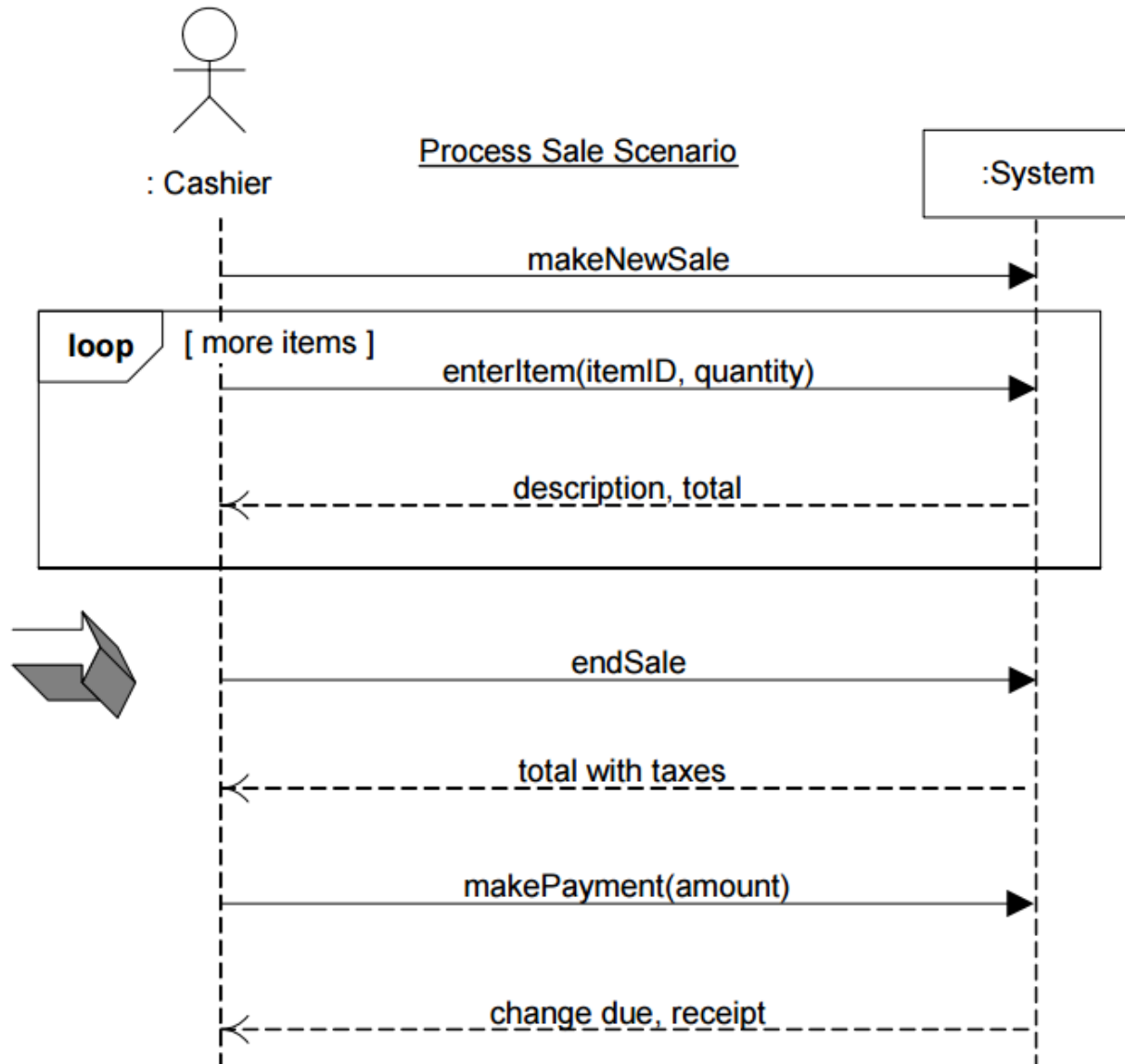
@Larman:
Da análise para o desenho: utilização dos resultados preparados pelo Analista (modelo do domínio, descrição funcional)



From the use cases into software design

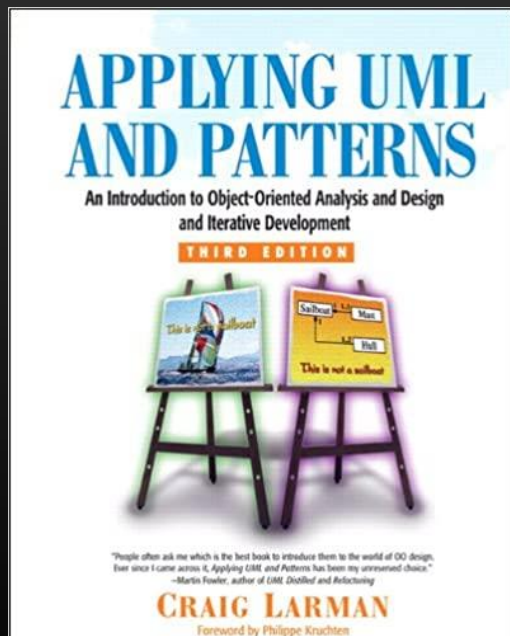
Simple cash-only Process Sale scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
 2. Cashier starts a new sale.
 3. Cashier enters item identifier.
 4. System records sale line item and presents item description, price, and running total.
- Cashier repeats steps 3-4 until indicates done.
5. System presents total with taxes calculated.
 6. Cashier tells Customer the total, and asks for payment.
 7. Customer pays and System handles payment.
- ...



In Larman:

Passo de transição intermédio: Diagrama de Sequência de Sistema (levantamento das funções “externas” de entrada no Sistema, a partir do CaU)



Iniciado quando um cliente telefona para o callCenter para solicitar uma reserva.

O operador pesquisa o cliente por código ou nome.

Se o cliente ainda não existe no sistema, os dados desse novo cliente são recolhidos e o cliente registado.

Os elementos da reserva são recolhidos pelo operador, que verifica se existe disponibilidade para o período pretendido. Nesse caso, a reserva é confirmada.

O cliente é informado do código de reserva (gerado pelo sistema).

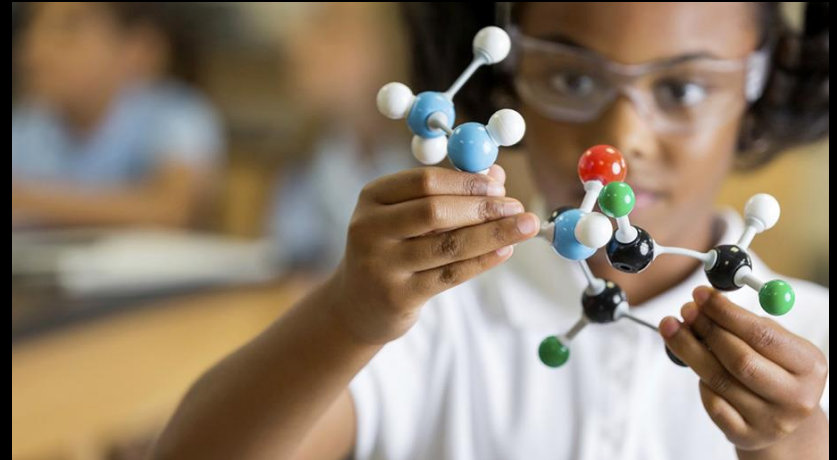


Operação de sistema: `pesquisarClientePorCodigo`

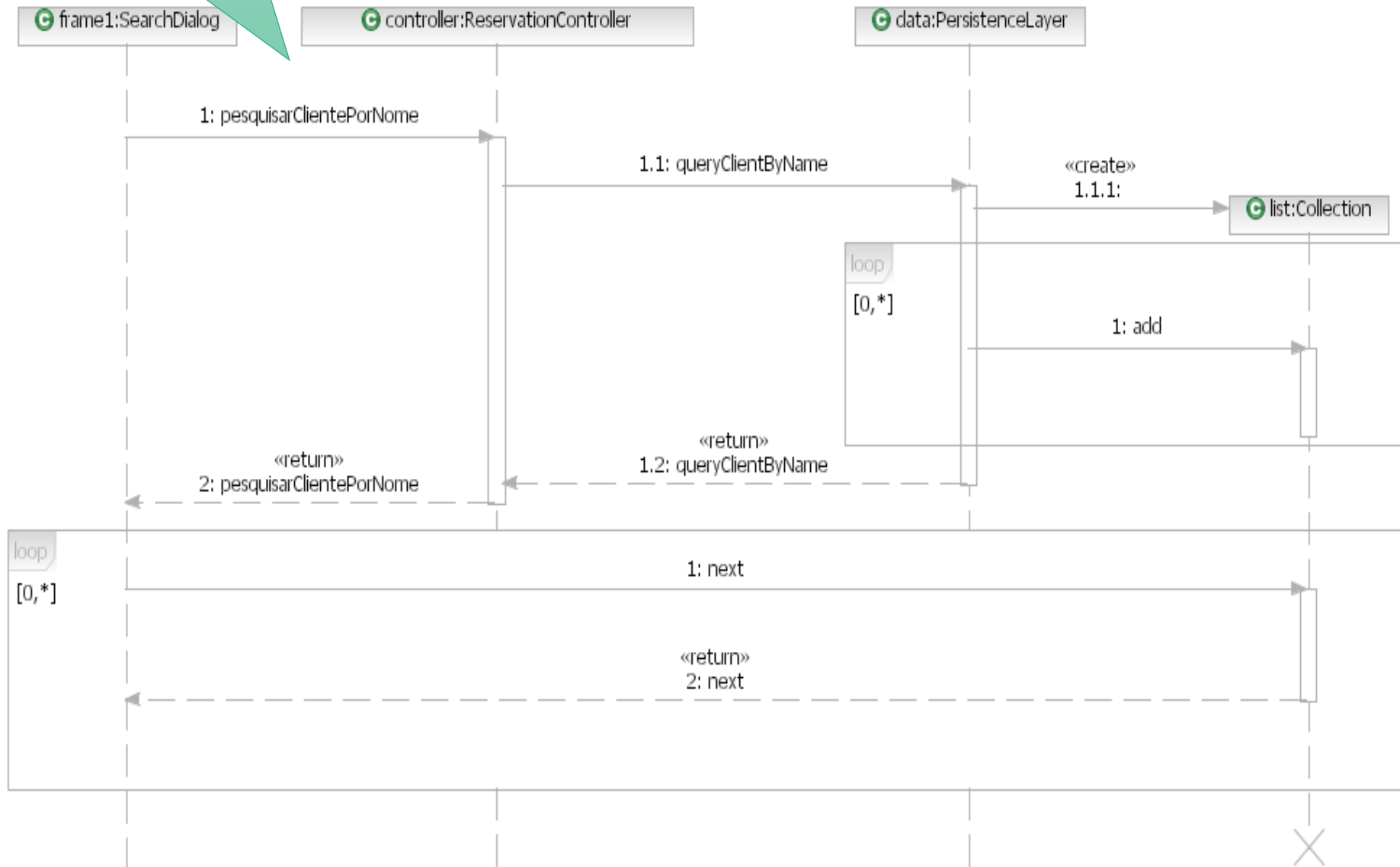
Quem deve controlar a interação? E a visualização?

Que outros objetos são necessários? Com que métodos?

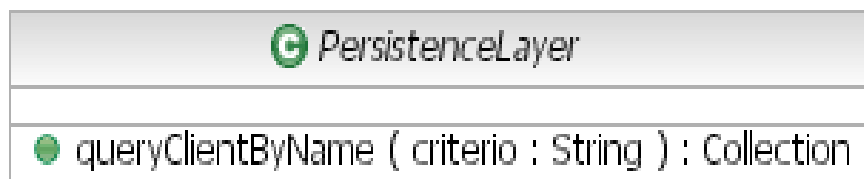
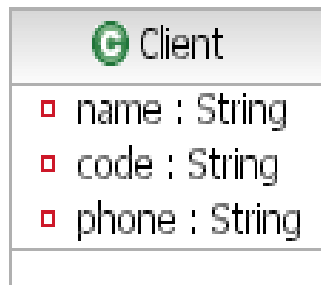
Qual é a ordem das mensagens entre objetos?



Expansão de cada operação de sistema:
qual a colaboração concreta de objetos
que a realiza? Processo de descoberta.



Os diagramas de interacção ajudam a distribuir responsabilidades → encontrar os métodos

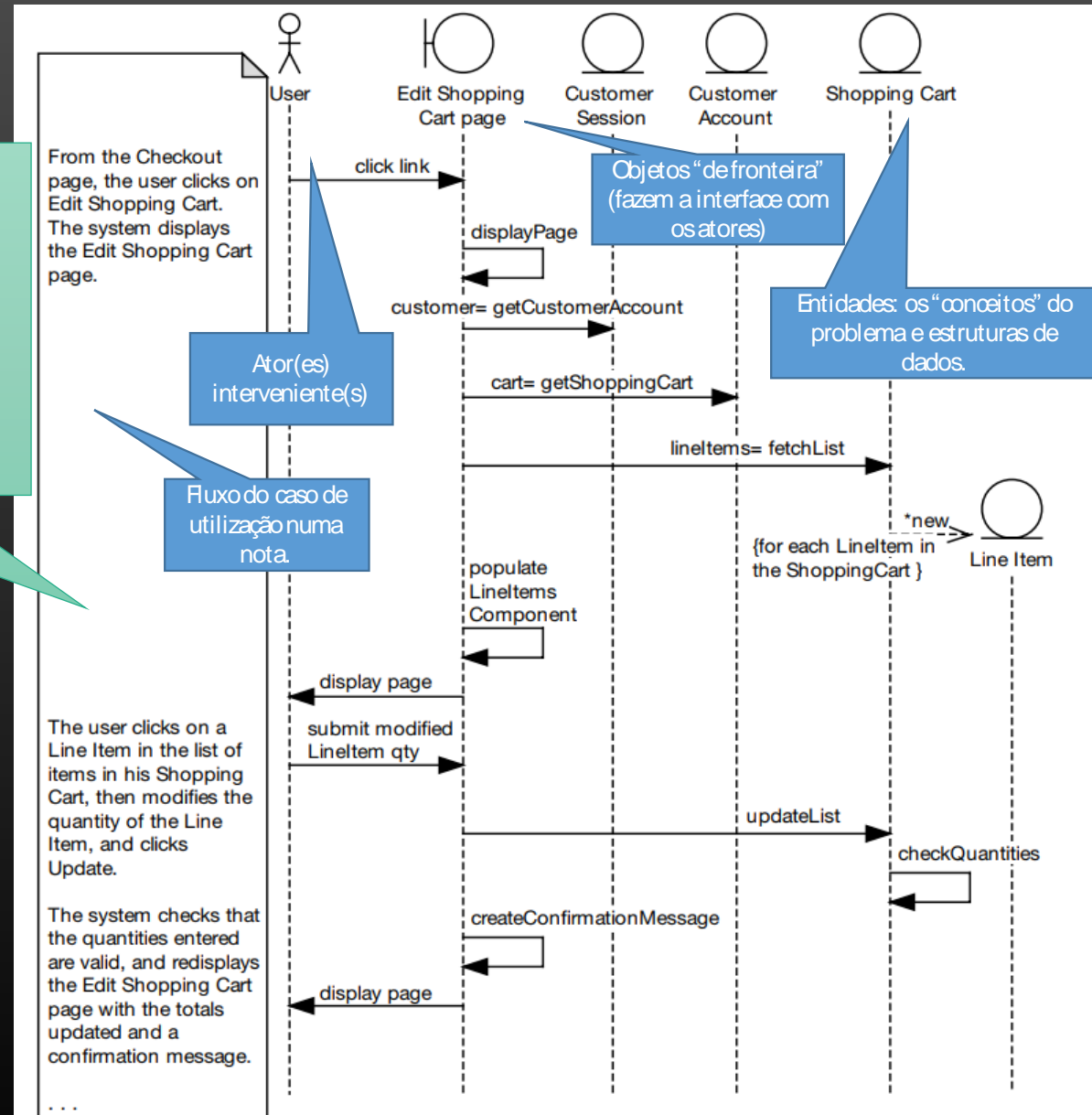
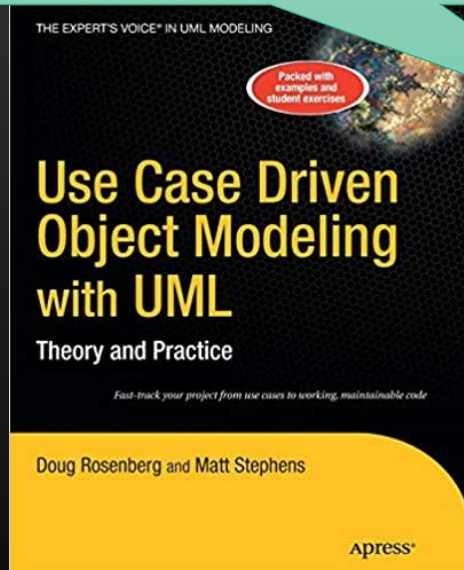


In Rosenbeg:

Da análise para o desenho: utilização dos resultados preparados pelo Analista para desenvolver o “modelo de robustez”

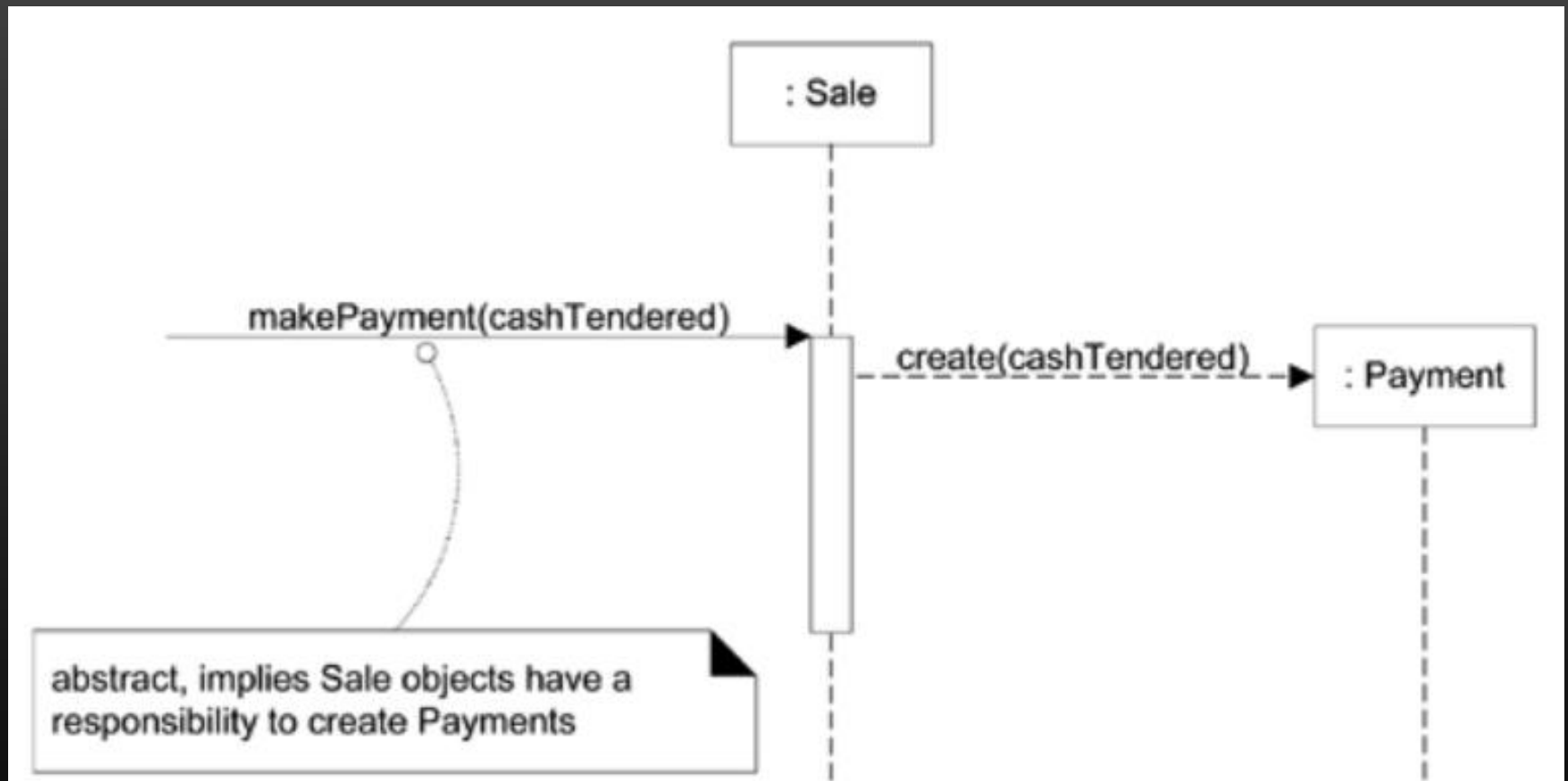
Três categorias de classes:

- Fronteiras
- Controladores
- Entidades



“Pensar por objetos” é aplicar princípios para “distribuir” as responsabilidades pelas classes

Ao desenhar um diagrama de interação, estamos a atribuir responsabilidades



Como atribuir responsabilidades aos objetos?

Não é uma ciência exata


Por isso temos...

Bom e mau desenho

Desenho eficiente e ineficiente

Desenho elegante e tenebroso...

Implicações na facilidade de
manter e evoluir uma solução



“Desenho”, no ciclo de engenharia do software, significa o processo de planejar/idealizar o código. A pessoa que lidera o desenho é o “arquiteto de software”.

Sempre que, mesmo num problema simples, começamos por nos interrogar: quais as classes? Como é que elas vão estar interdependentes?, estamos a “desenhar” o o código (fazendo escolhas).

Responsabilidades de um objeto

Fazer

Fazer alguma coisa sobre o seu estado, como calcular alguma coisa, criar objetos,...

Iniciar uma ação em outros objetos

Coordenar/controlar as ações em outros objetos

Saber

Conhecer o seu estado interno (“escondido”)

Conhecer os objetos relacionados

GRASP (Larman)

Generic Responsibility Assignment Principles

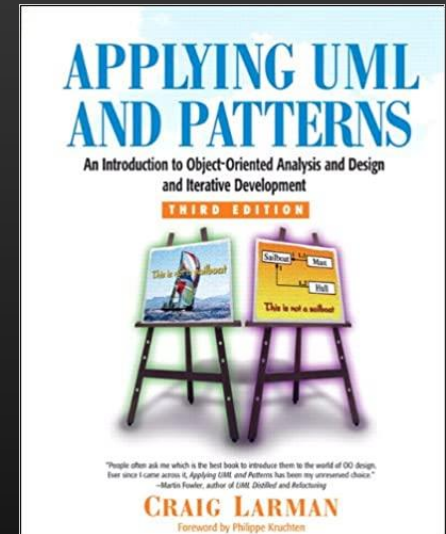
↓ Coupling

↑ Cohesion

Information Expert

Creator

Controller



Existem recomendações/princípio para orientar a distribuição de responsabilidades pelos objetos.
E.g.: GRASP

Critérios para o desenho

- Um conjunto de métricas para avaliar o desenho
- Acoplamento (*coupling*): refere-se ao grau de proximidade/interdependência da relação entre classes
- Coesão (*coesion*): refere-se ao grau com que os atributos e métodos de uma classe estão relacionados internamente.

Uma classe que tem muitos atributos que são objetos de outro tipo, tem um *coupling* elevado: **depende de** outras classes.

Uma classe que mantém, internamente, detalhes das Vendas e dos Produtos vendidos, não é coesa: em vez de ter um **foco único**, está a assumir várias responsabilidades.

Coupling

Mede a força/intensidade da dependência de uma classe de outras

A classe C1 está emparelhada com C2 se precisa de C2, direta ou indiretamente.

Uma classe que depende de outras 2 tem um “coupling” mais baixo que uma que dependa de 8.

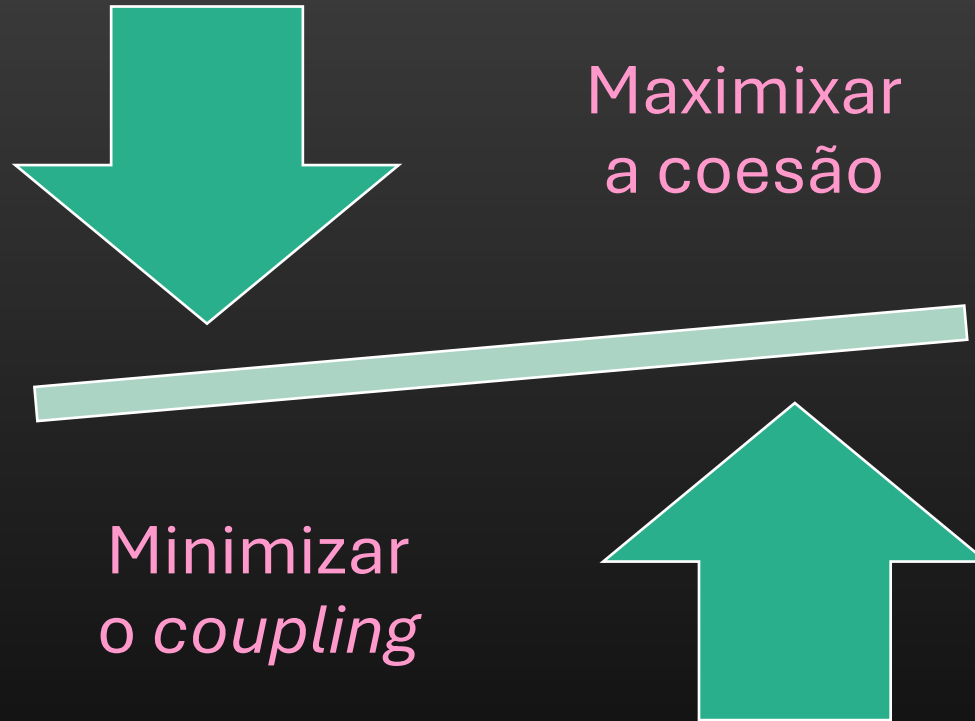
Coesão

Mede a força/intensidade do relacionamento dos elementos de uma classe entre si.

Todas as operações e dados de uma classe devem estar natural e diretamente relacionados com o conceito que a classe modela

Uma classe deve ter um foco único (vs. responsabilidades desgarradas)

Critérios gerais para um melhor desenho



Common Forms of Coupling in Java

- Type X has an attribute that refers to a type Y instance or type Y itself

```
class X{ private Y y = ...}  
class X{ private Object o = new Y(); }
```

- A type X object calls methods of a type Y object

```
class Y{f(){;}}  
class X{ X(){new Y.f();}}
```

- Type X has a method that references an instance of type Y (E.g. by means of a parameter, local variable, return type,...)

```
class Y{}  
class X{ X(Y Y){...}}  
class X{ Y f(){...}}  
class X{ void f(){Object y = new Y();}}
```

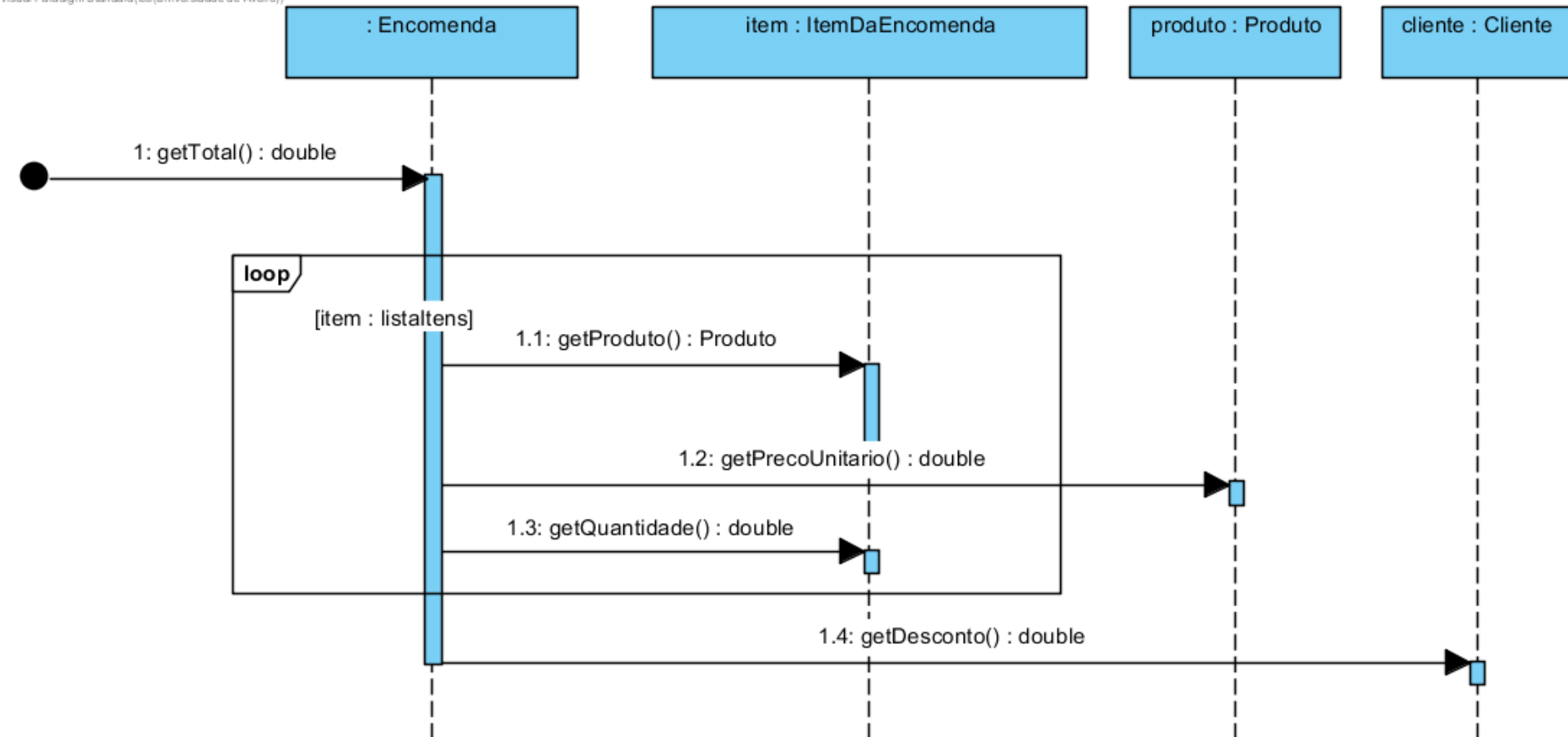
- Type X is a subtype of type Y

```
class Y{}  
class X extends Y{}
```


- ...

Coupling de interação

Visual Paradigm Standard (co(Universidade de Aveiro))



Types of Interaction Coupling

Level	Type	Description
Good	No Direct Coupling	The methods do not relate to one another; that is, they do not call one another.
	Data	The calling method passes a variable to the called method. If the variable is composite (i.e., an object), the entire object is used by the called method to perform its function.
	Stamp	The calling method passes a composite variable (i.e., an object) to the called method, but the called method only uses a portion of the object to perform its function.
	Control	The calling method passes a control variable whose value will control the execution of the called method.
	Common or Global	The methods refer to a "global data area" that is outside the individual objects.
Bad	Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation and information hiding. However, C++ allows this to take place through the use of "friends."

Source: These types were adapted from Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, 2nd ed. (Englewood Cliffs, NJ: Yardon Press, 1988); and Glenford Myers, *Composite/Structured Design* (New York: Van Nostrand Reinhold, 1978).

Coupling...

Coupling	Features	Desirability
Data	A & B communicate by simple data only	High (use parameter passing & only pass necessary info)
Stamp	A & B use a common type of data	Okay (but should they be grouped in a data abstraction?)
Control (activating)	A transfers control to B by procedure call	Necessary
Control (switching)	A passes a flag to B to tell it how to behave	Undesirable (why should A interfere like this?)
Common environment	A & B make use of a shared data area (global variables)	Undesirable (if you change the shared data, you have to change both A and B)
Content	A changes B's data, or passes control to the middle of B	Extremely Foolish (almost impossible to debug!)

Coesão

Qual é a hipótese que oferece maior coesão?

Qual é o que é mais fácil de avariar/dar problemas?

De que é que precisamos 80% das vezes?....



Coesão

Uma classe, objeto ou método coesos têm um único “foco”

Coesão a nível dos métodos

O método executa mais do que um propósito/operação?

Realizar mais do que uma operação é mais difícil de entender e implementar

Coesão a nível da classe


Os atributos e métodos representam um único objeto?

As classes não devem misturar papéis, domínios ou objetos

Coesão na especialização/generalização

As classes numa hierarquia devem mostrar uma relação "tipo-de"

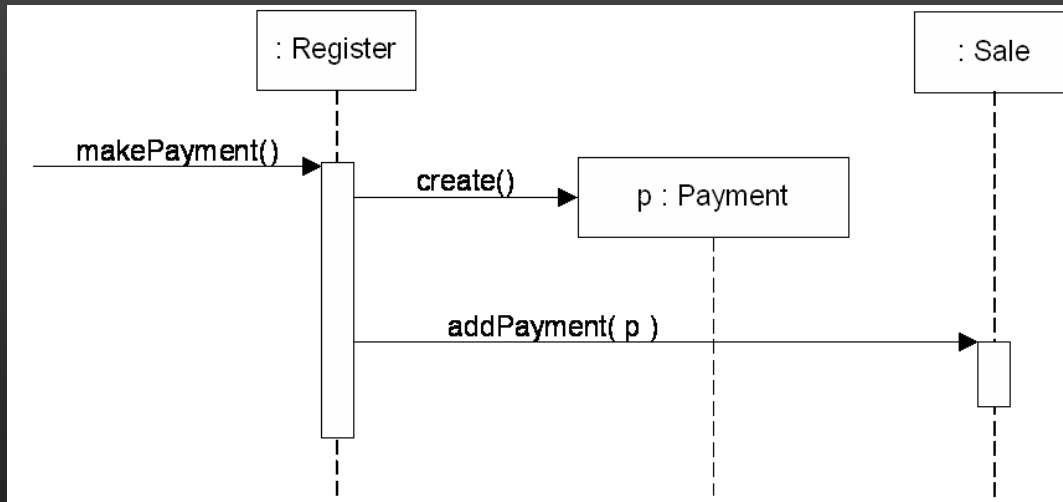
Types of Method Cohesion

Level	Type	Description
Good 	Functional	A method performs a single problem-related task (e.g., calculate current GPA).
	Sequential	The method combines two functions in which the output from the first one is used as the input to the second one (e.g., format and validate current GPA).
	Communicational	The method combines two functions that use the same attributes to execute (e.g., calculate current and cumulative GPA).
	Procedural	The method supports multiple weakly related functions. For example, the method could calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
	Temporal or Classical	The method supports multiple related functions in time (e.g., initialize all attributes).
	Logical	The method supports multiple related functions, but the choice of the specific function is chosen based on a control variable that is passed into the method. For example, the called method could open a checking account, open a savings account, or calculate a loan, depending on the message that is send by its calling method.
Bad	Coincidental	The purpose of the method cannot be defined or it performs multiple functions that are unrelated to one another. For example, the method could update customer records, calculate loan payments, print exception reports, and analyze competitor pricing structure.
Source: These types were adapted from Page-Jones, <i>The Practical Guide to Structured Systems</i> , and Myers, <i>Composite/Structured Design</i> .		

Cohesion (methods)

Cohesion	Features	Desirability
Data	all part of a well defined data abstraction	Very High
Functional	all part of a single problem solving task	High
Sequential	outputs of one part form inputs to the next	Okay
Communicational	operations that use the same input or output data	Moderate
Procedural	a set of operations that must be executed in a particular order	Low
Temporal	elements must be active around the same time (e.g. at startup)	Low
Logical	elements perform logically similar operations (e.g. printing things)	No way!!
Coincidental	elements have no conceptual link other than repeated code	No way!!

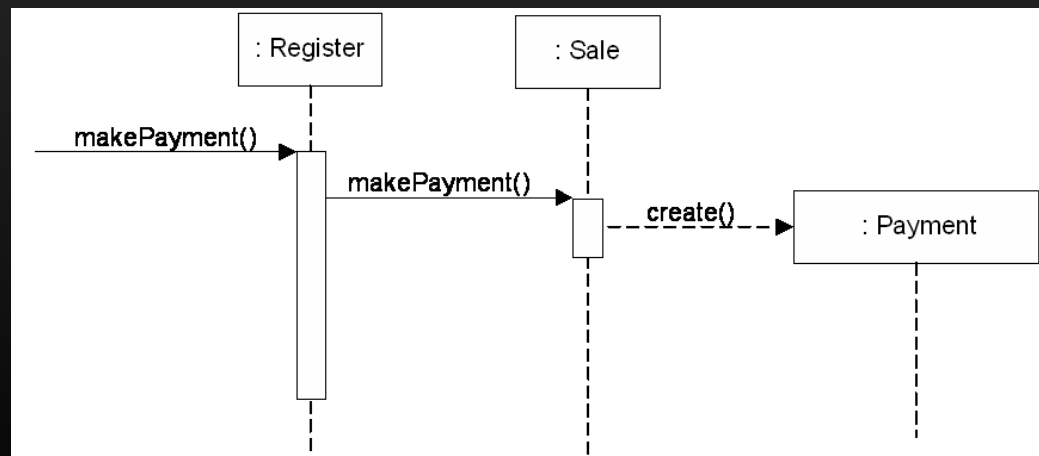
Exemplos



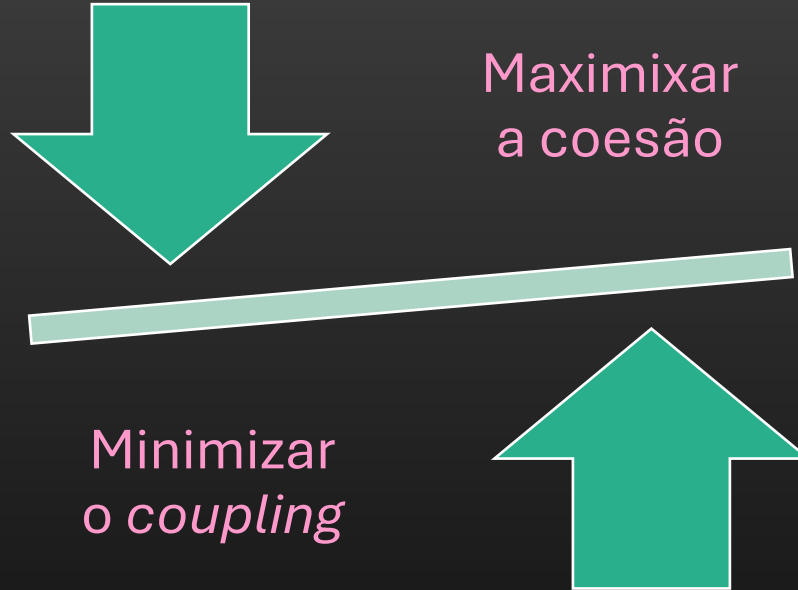
Qual é a hipótese que oferece maior coesão?

No primeiro caso, **Register** conhece informação de pagamentos e de vendas.

No segundo caso, **Register** apenas se relaciona com **Venda** (e não precisa de representar a lógica dos pagamentos)



É preciso balancear

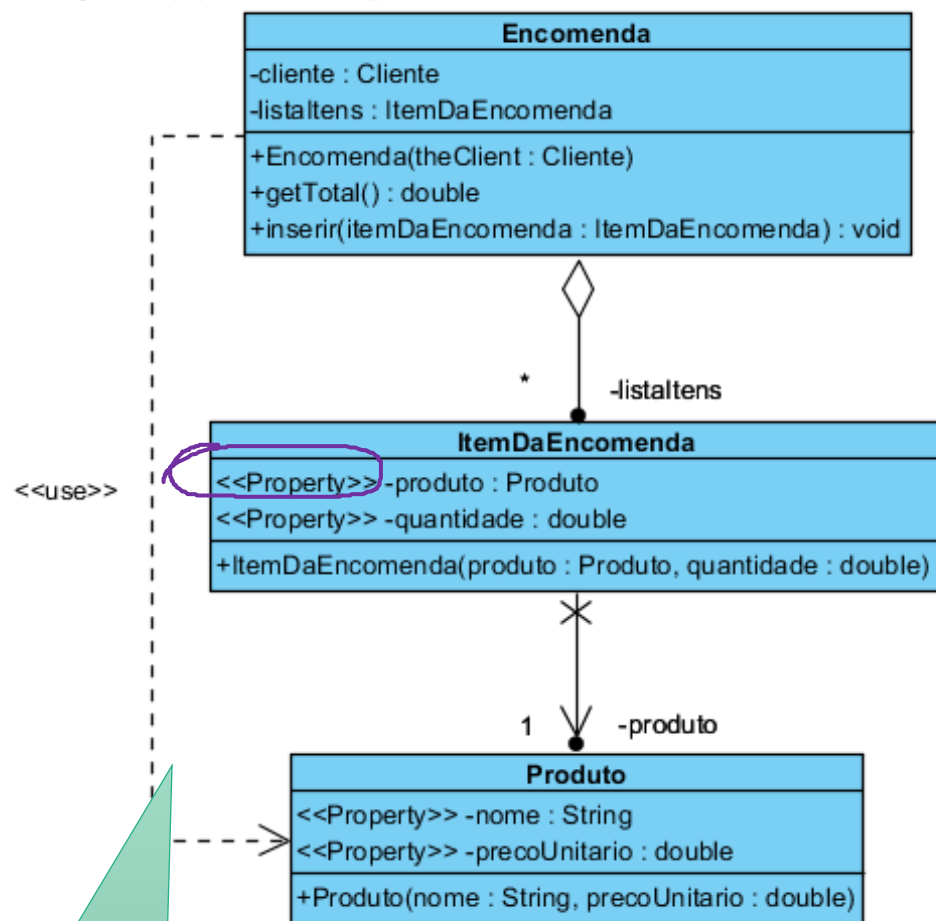


Por hipótese, a situação com melhor *coupling* (mais baixo possível) seria ter uma única classe na solução.

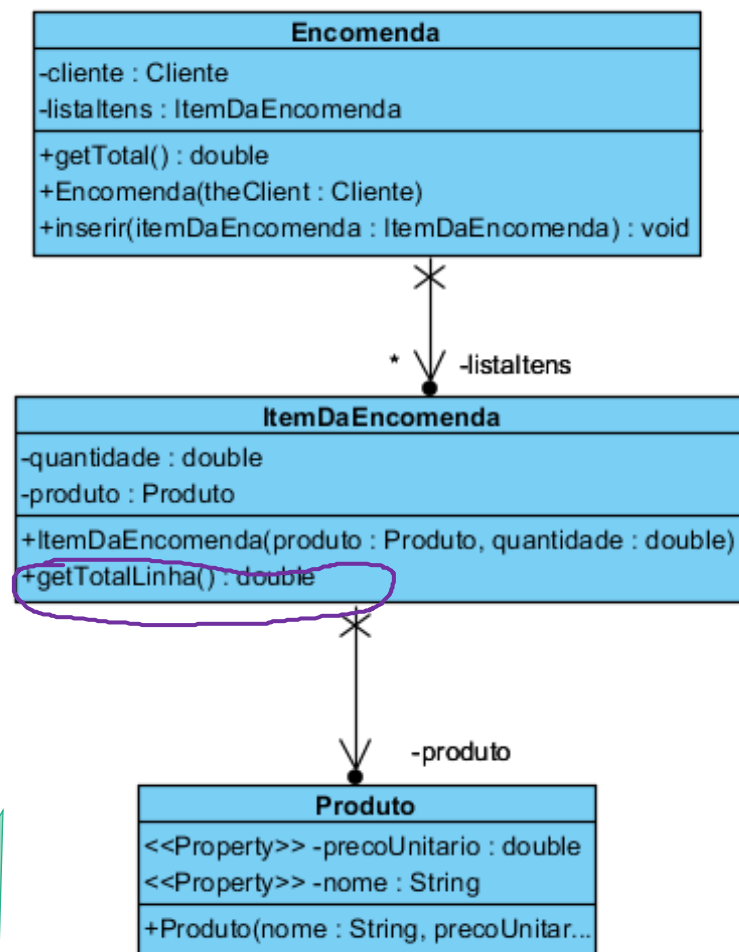
Mas essa seria a pior escolha do ponto de vista da coesão.

Avaliação de coupling/coesão: exemplo da encomenda

Visual Paradigm Standard (co(Universidade de Aveiro))



getTotal() consulta o preço unitário definido em Produto



getTotal() pede ao "item da encomenda" para lhe dar o total da linha.



SOLID

Single responsibility

Open-closed

Liskov substitution

Interface segregation

Dependency inversion

SOLID

Software Development is not a Jenga game

Referências

Core readings	Suggested readings
<ul style="list-style-type: none">• [Dennis15] – Chap. 8	<ul style="list-style-type: none">• [Larman04] – Chap. 17 and 18• Slides by M. Eichberg: SSD and OO-Design (lesson 9)