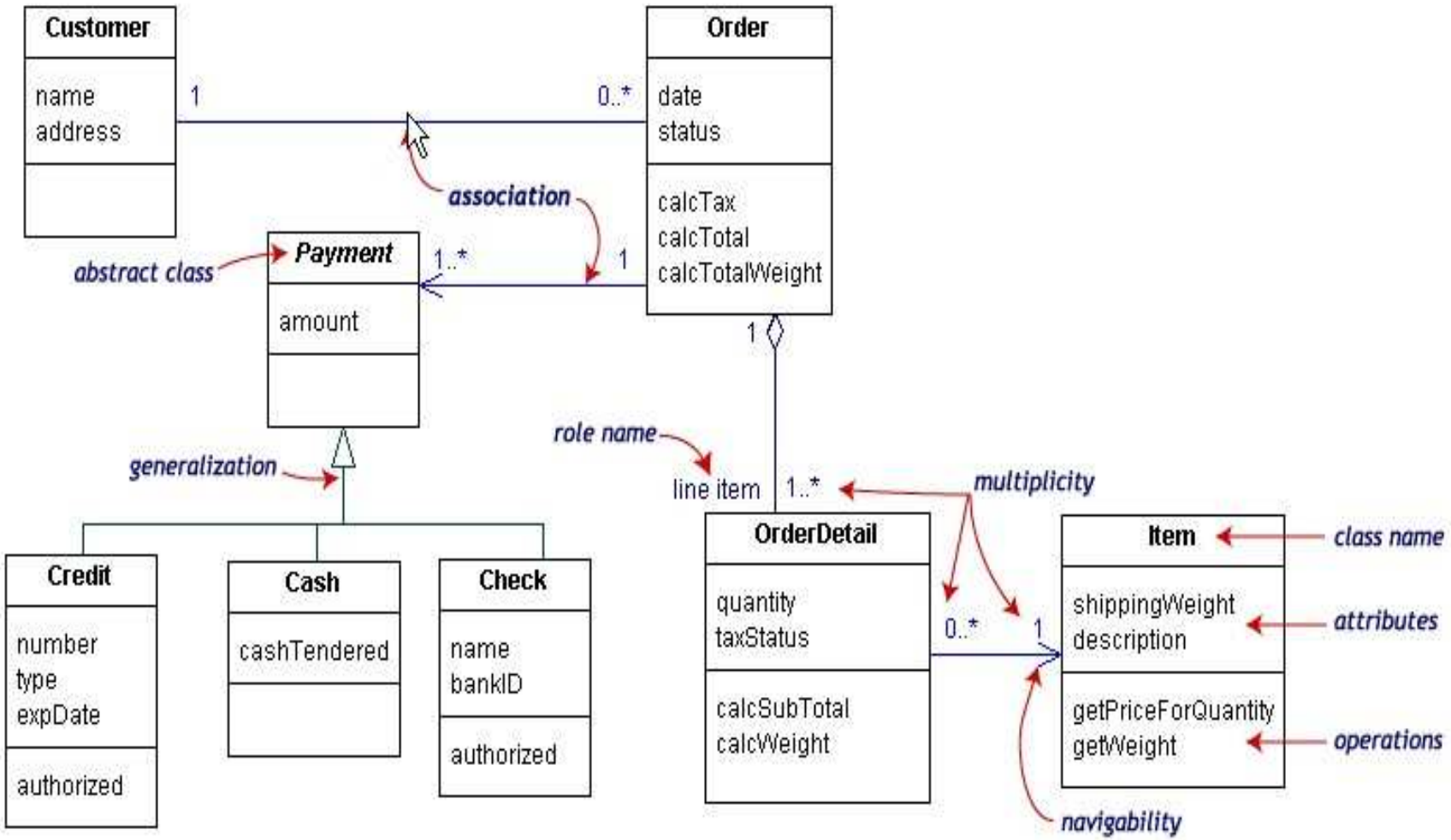
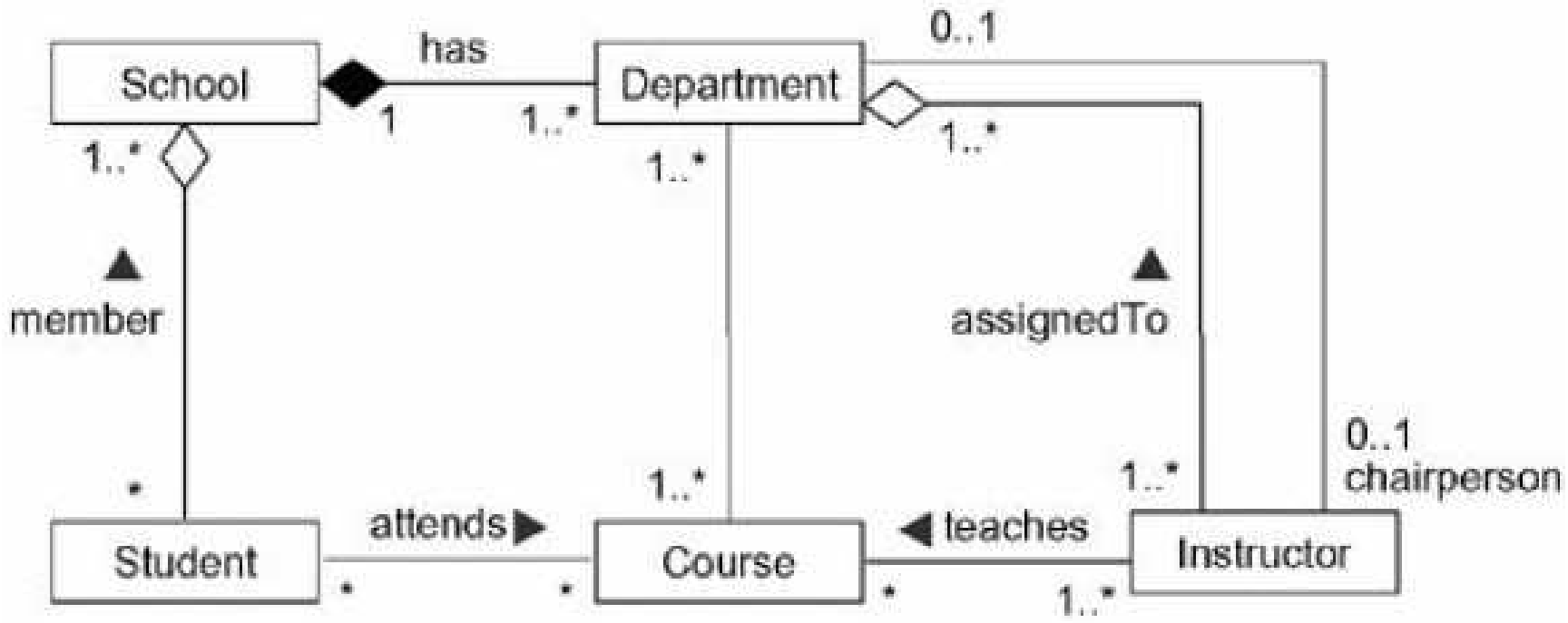
DC: Exemplo - Interpretação

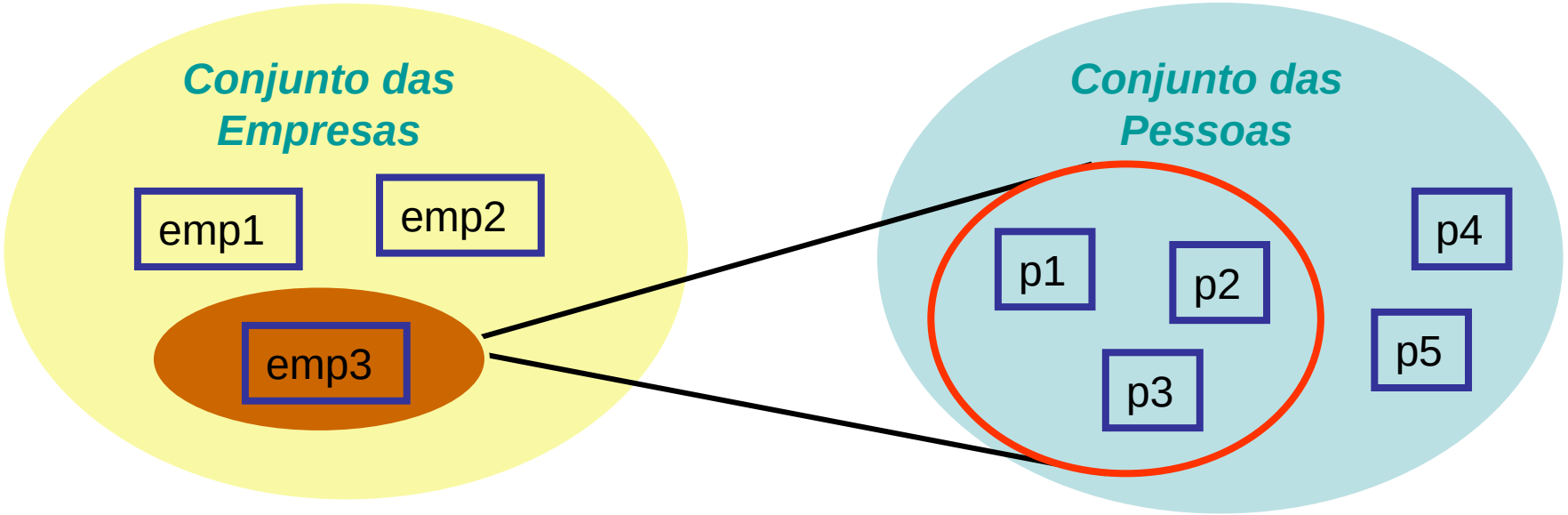


DC: Exemplo - Interpretação



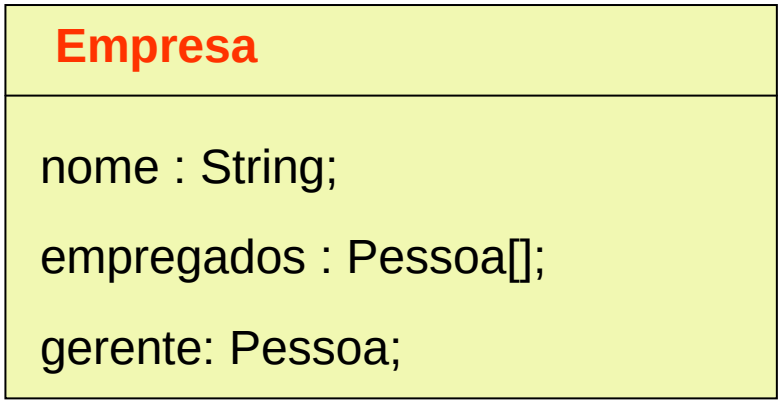
Os Diagramas de Classes podem, como já vimos anteriormente, ser usados a um nível de modelação muito alto, no qual apenas estamos a definir o domínio do problema, ou seja, o seu vocabulário e algumas regras.

DC: Exemplo - Associações

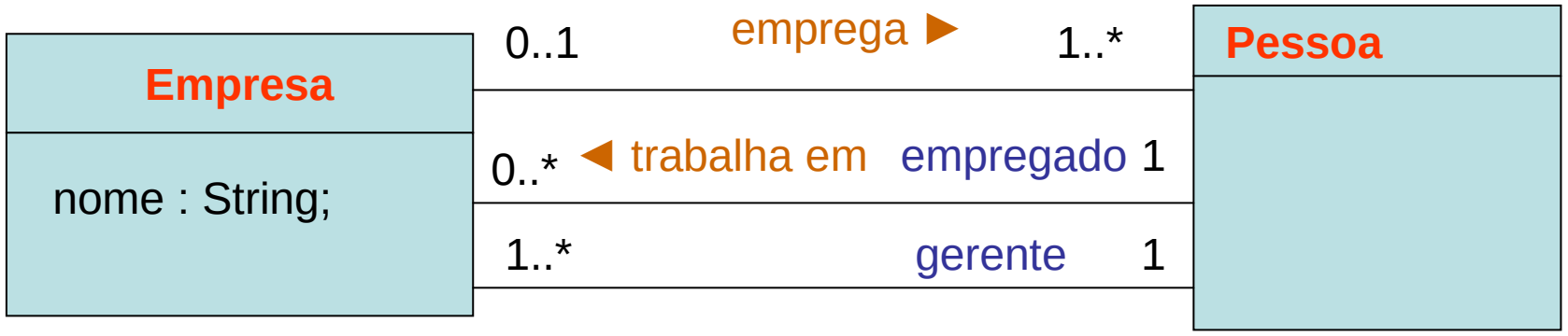


DC: Exemplo - Associações

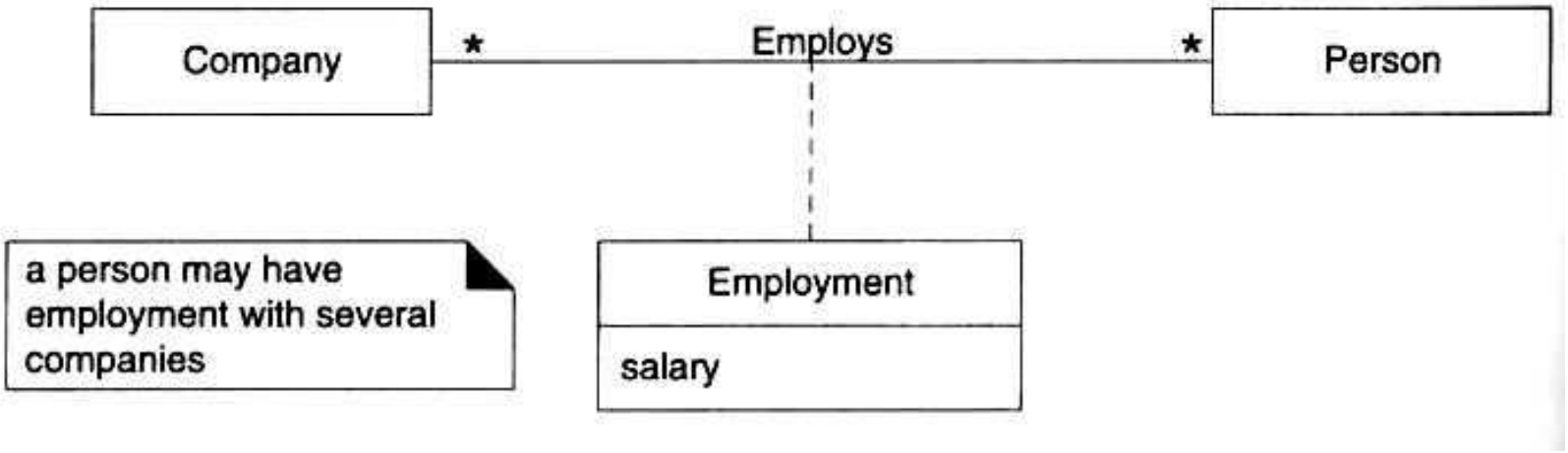
Classes podem conter referências directas entre si, pelo que a classe Empresa apresentada a seguir estaria correcta (ainda que seja "low level").



Porém, nesta fase (análise/concepção) é mais correcto usar associações !

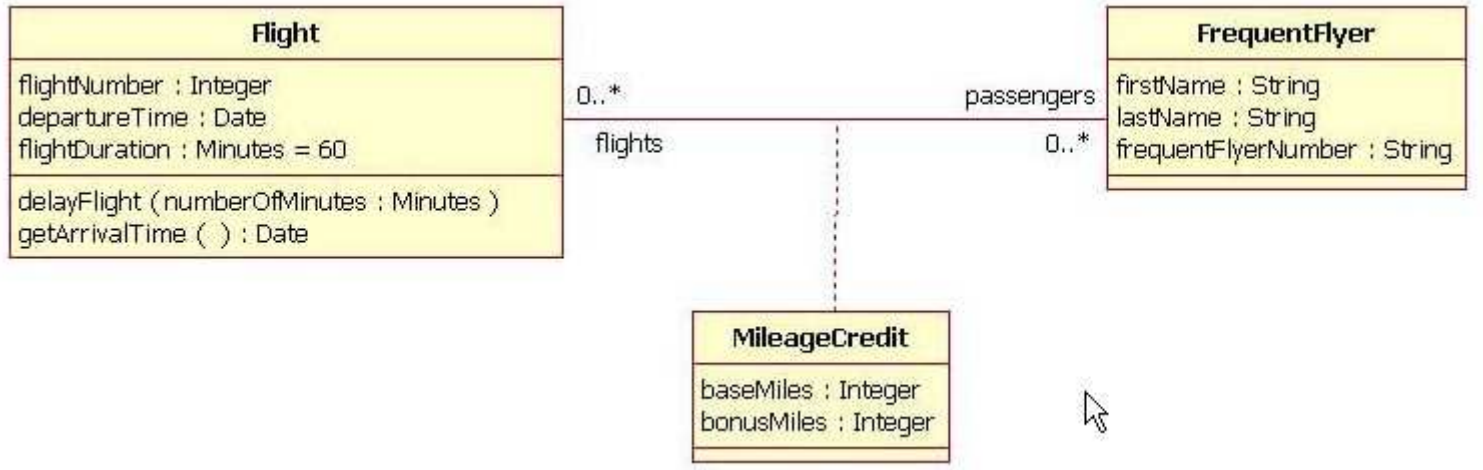


DC: Classes de Associação



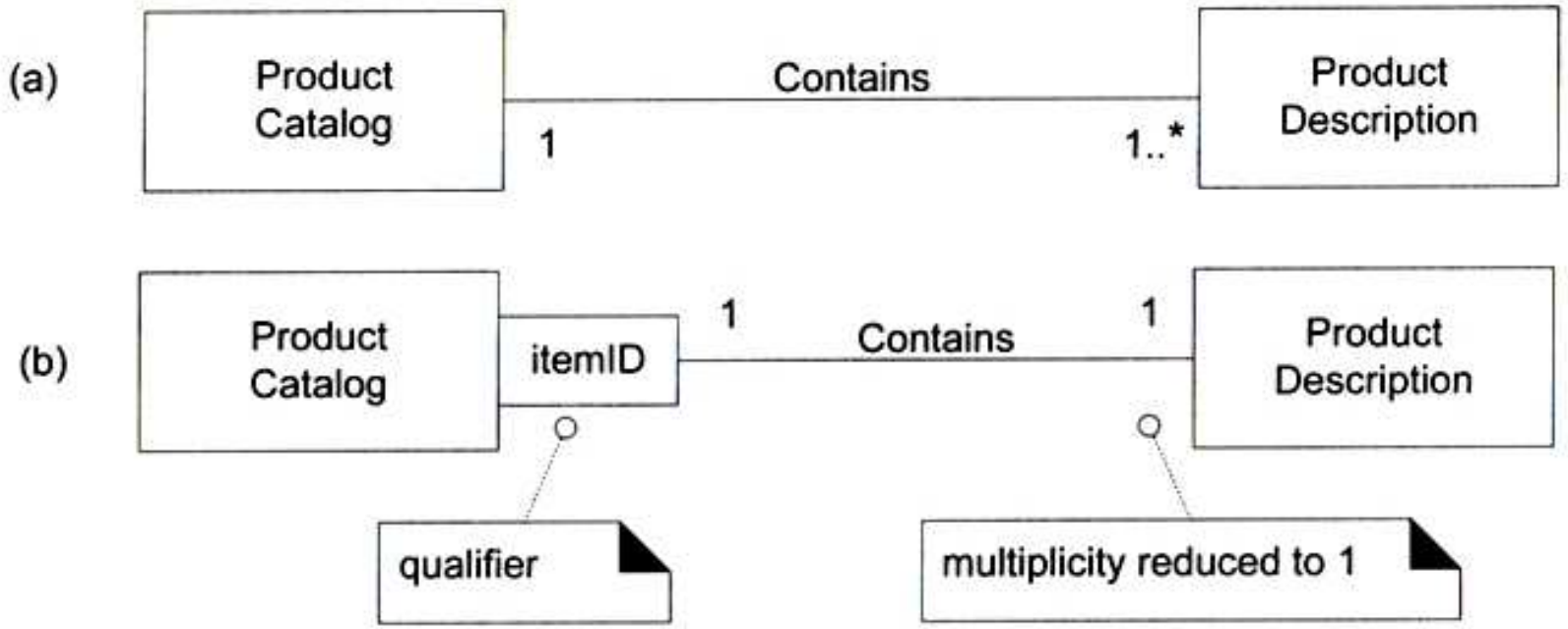
Quando se relaciona uma instância de "Empresa" com uma instância de "Pessoa" deve "juntar-se" uma instância de "Emprego", que transporta consigo o valor do salário. Assim, cada "Pessoa" poderá ter um salário distinto por emprego, ou seja, por cada uma das suas associações a uma empresa.

DC: Classes de Associação



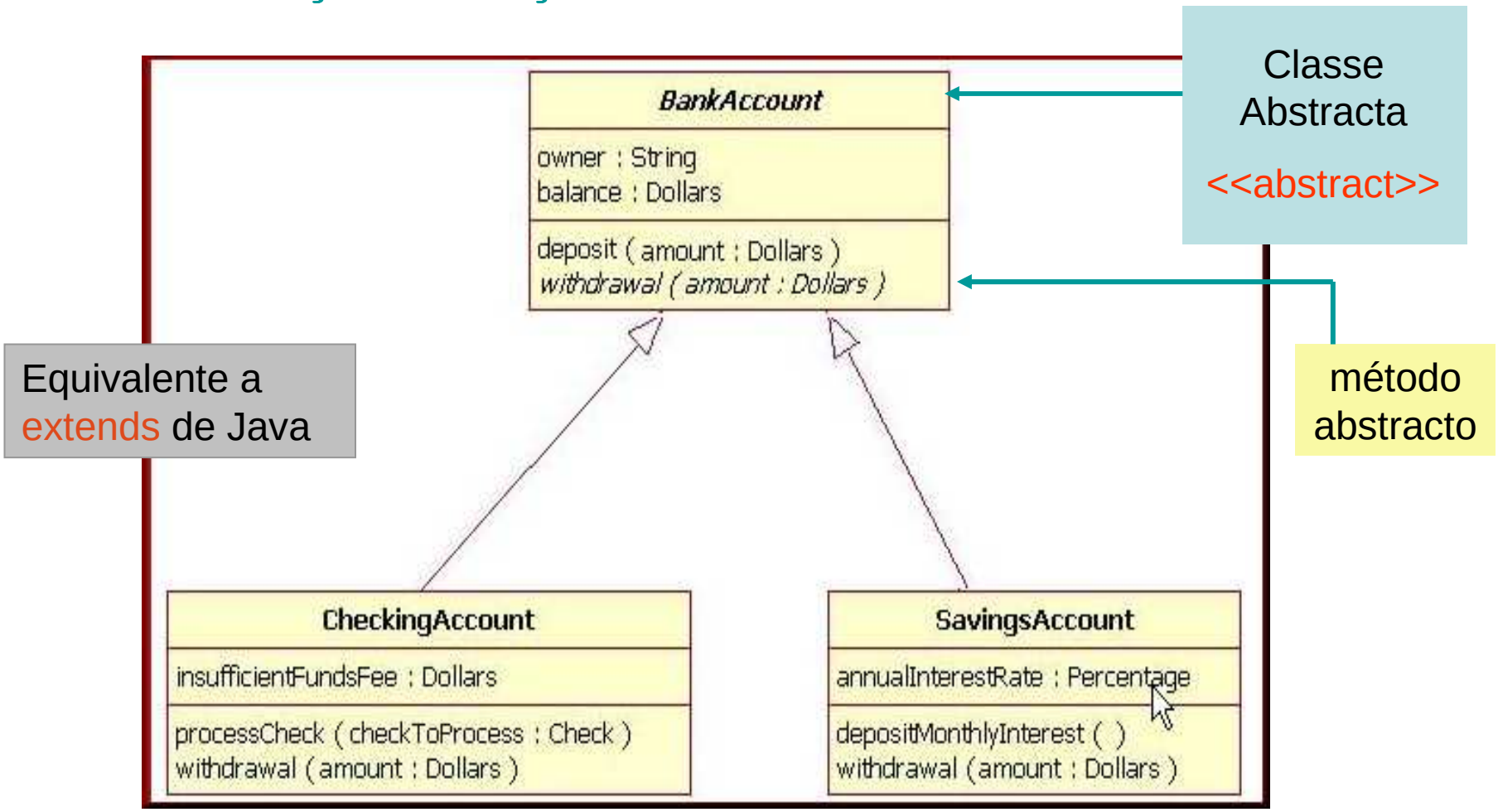
Quando se relaciona uma instância de "Voo" com uma instância de "Passageiro Frequente" deve "juntar-se" uma instância de "Crédito em Milhas" (isto porque a relação é O..*).

DC: Associação Qualificada

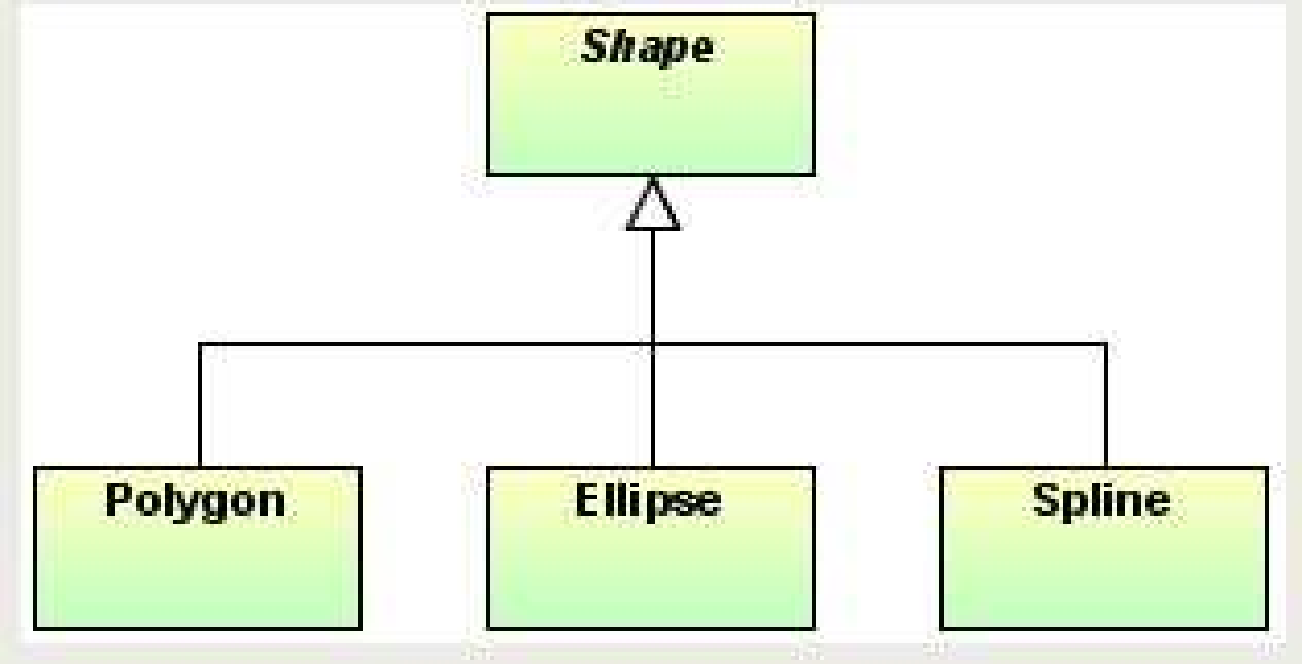


Em b) é explicitamente especificado agora que, o qualificador itemID é uma "chave única" de relacionamento entre "Product Catalog" e "Product Description". Sendo única, o relacionamento passa a ser agora lido como "Um catálogo de produtos contém 1 e só uma descrição de produto para cada valor de itemID".

Generalização - Herança



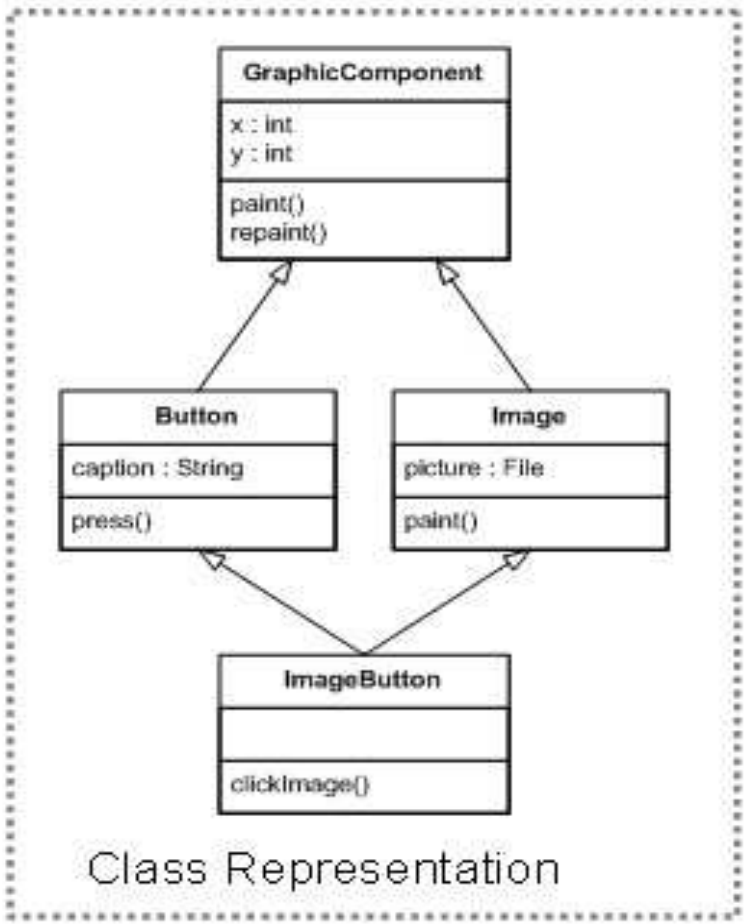
DC: Generalização

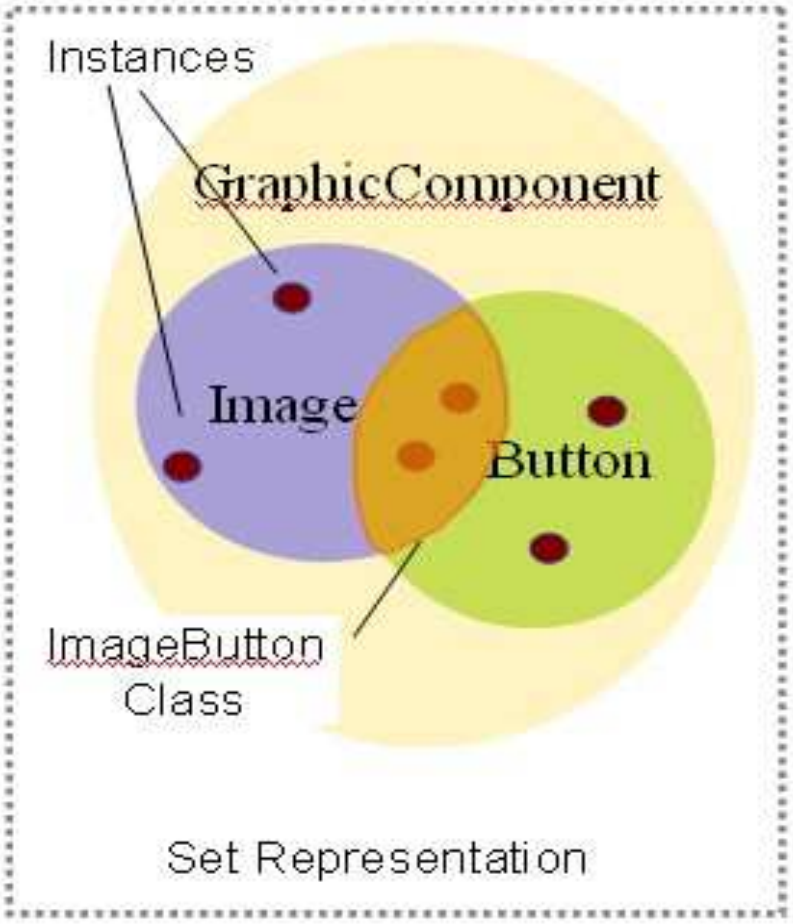


São portanto subclasses que correspondem a várias implementações (ou tipos) da superclasse abstracta.

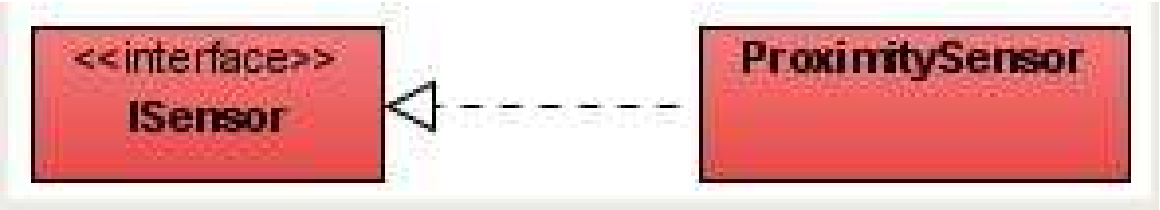
Nota: O polimorfismo é muito importante na codificação. É mesmo um mecanismo fundamental. Porém, na análise e na concepção há que o empregar com bastante moderação.

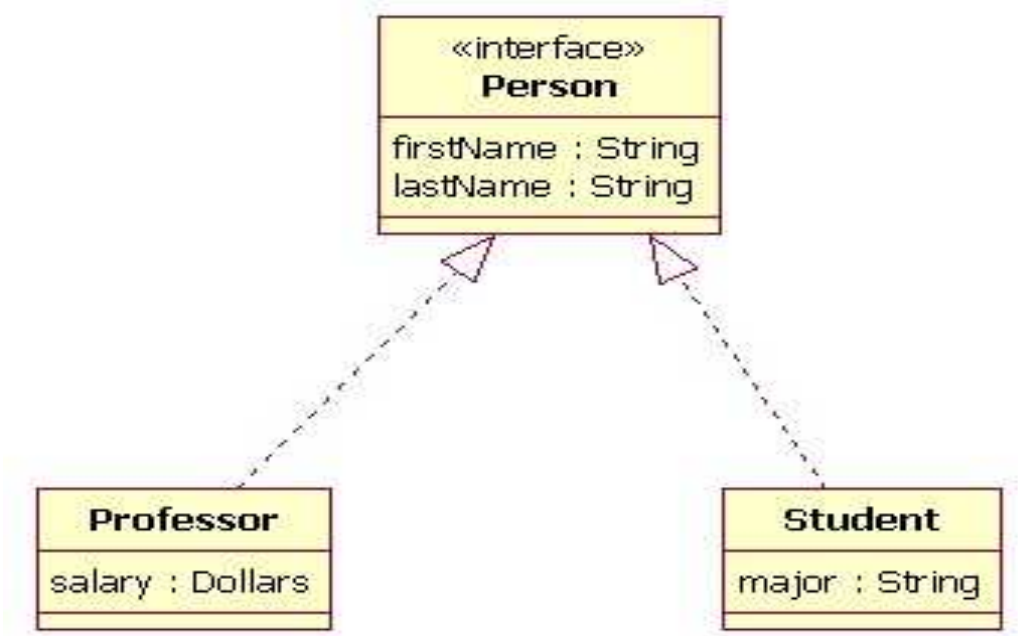
Generalização e Conjuntos de Instâncias





DC: Interfaces e Realização



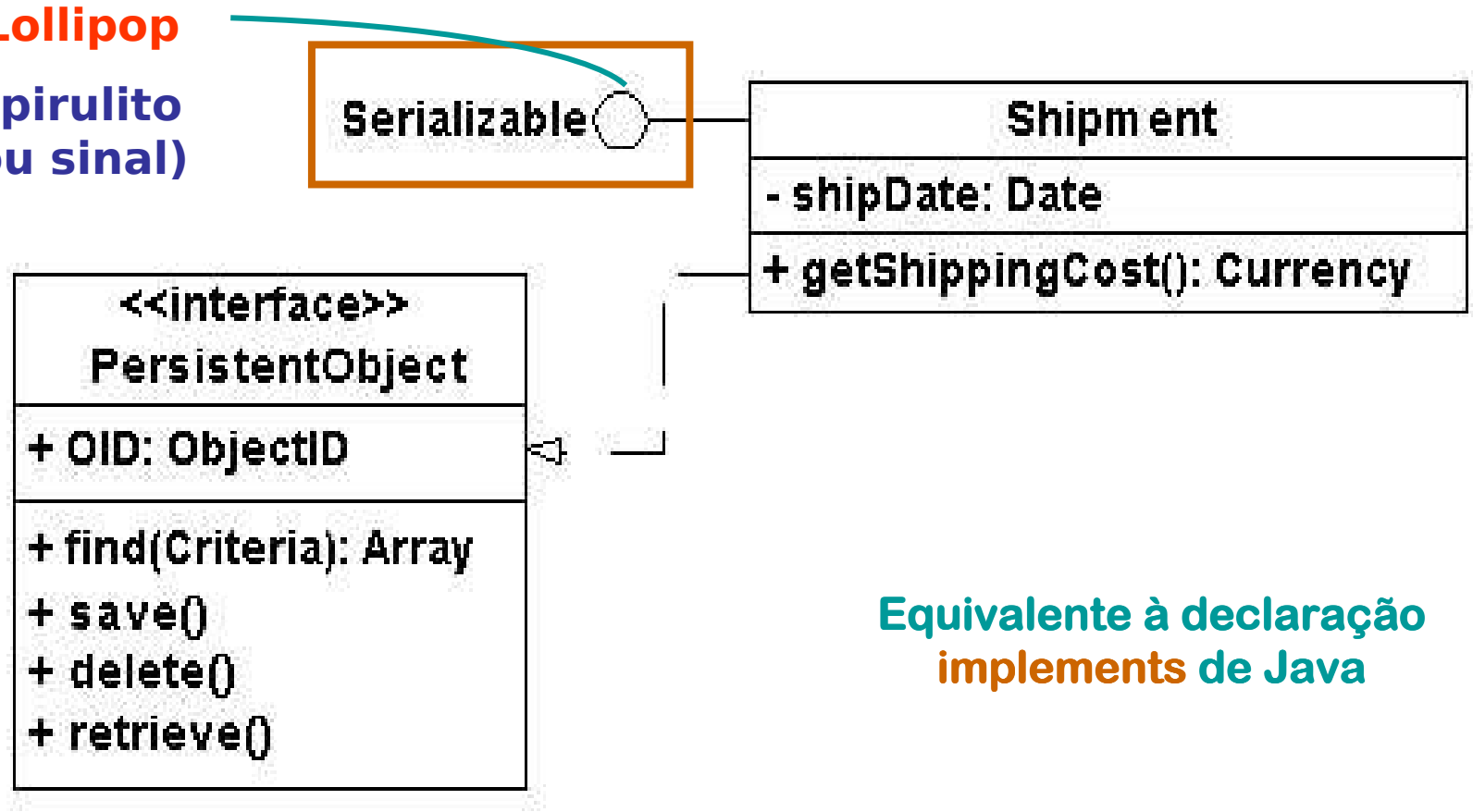


Equivalente à declaração implements de Java

DC: Interfaces e Realização

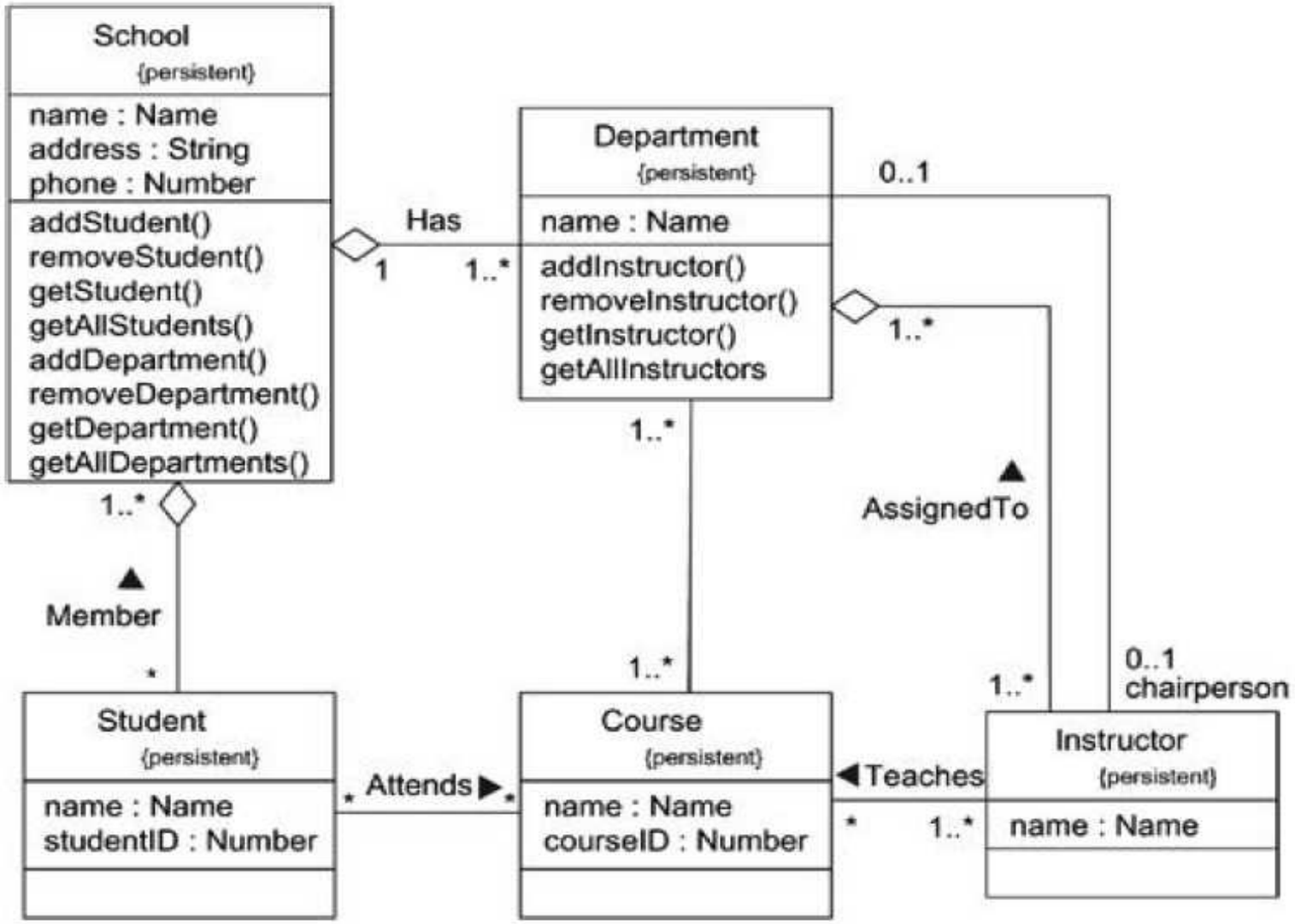
Lollipop

(pirulito ou sinal)

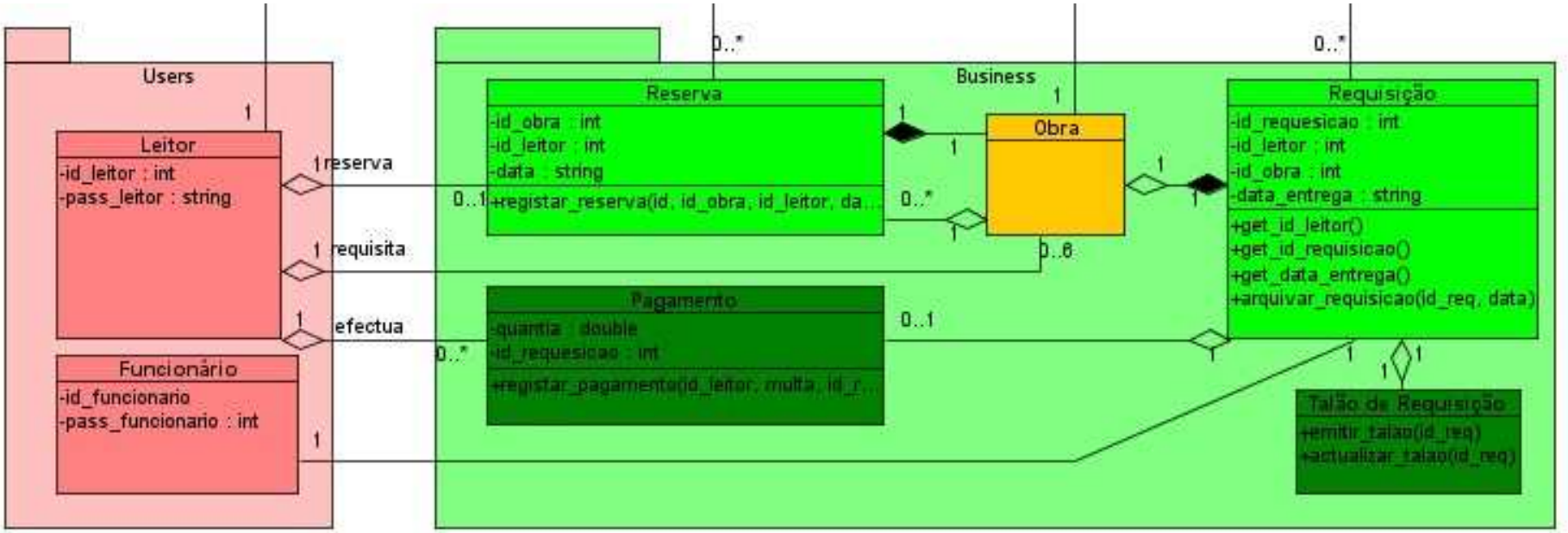


Equivalente à declaração implements de Java

DC: Exemplo - Interpretação



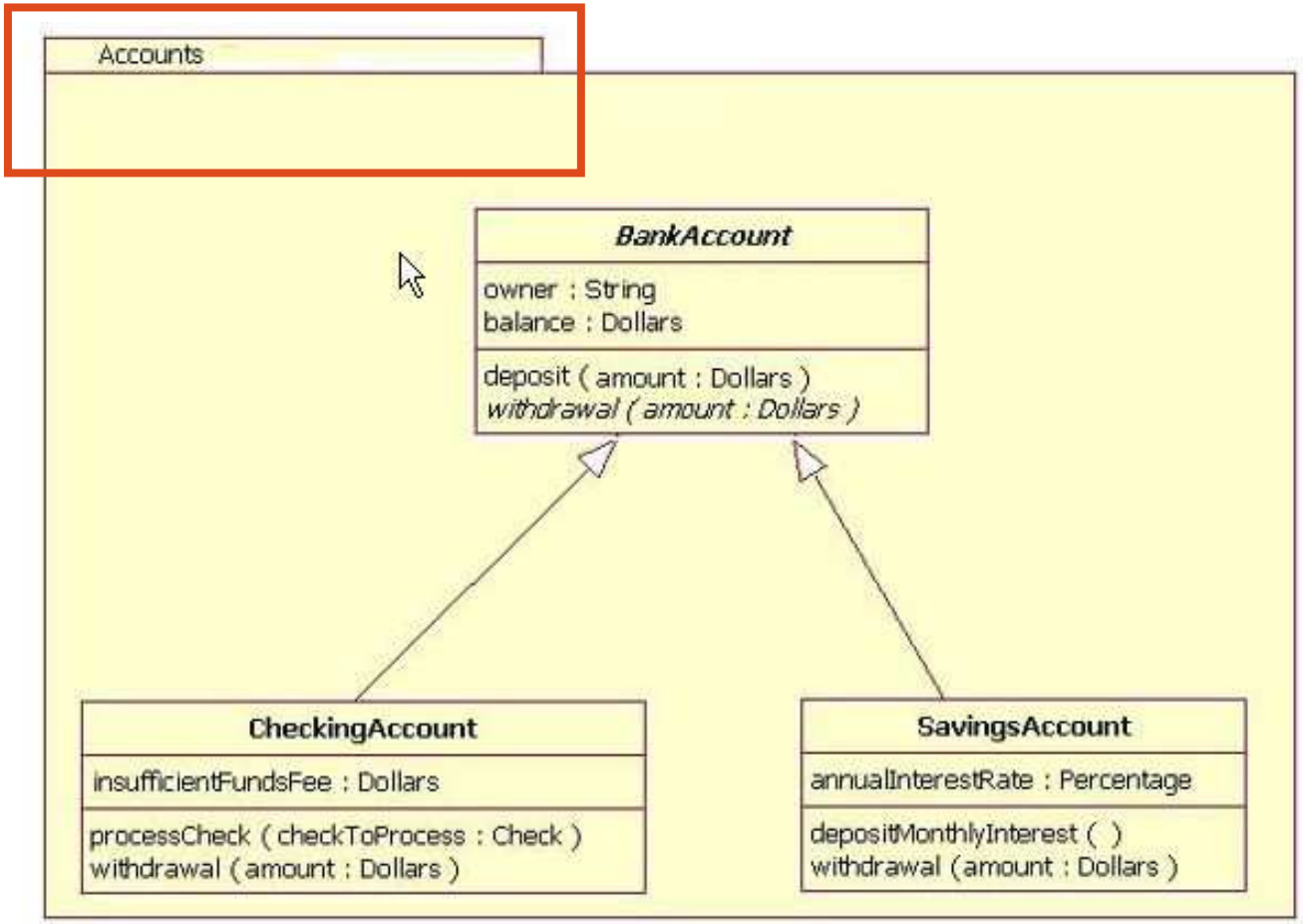
DC e Packages



Diagramas de Classes que se tornam complexos devem ser divididos, e funcional ou semanticamente estruturados em Packages, que agrupam classes com tais afinidades.

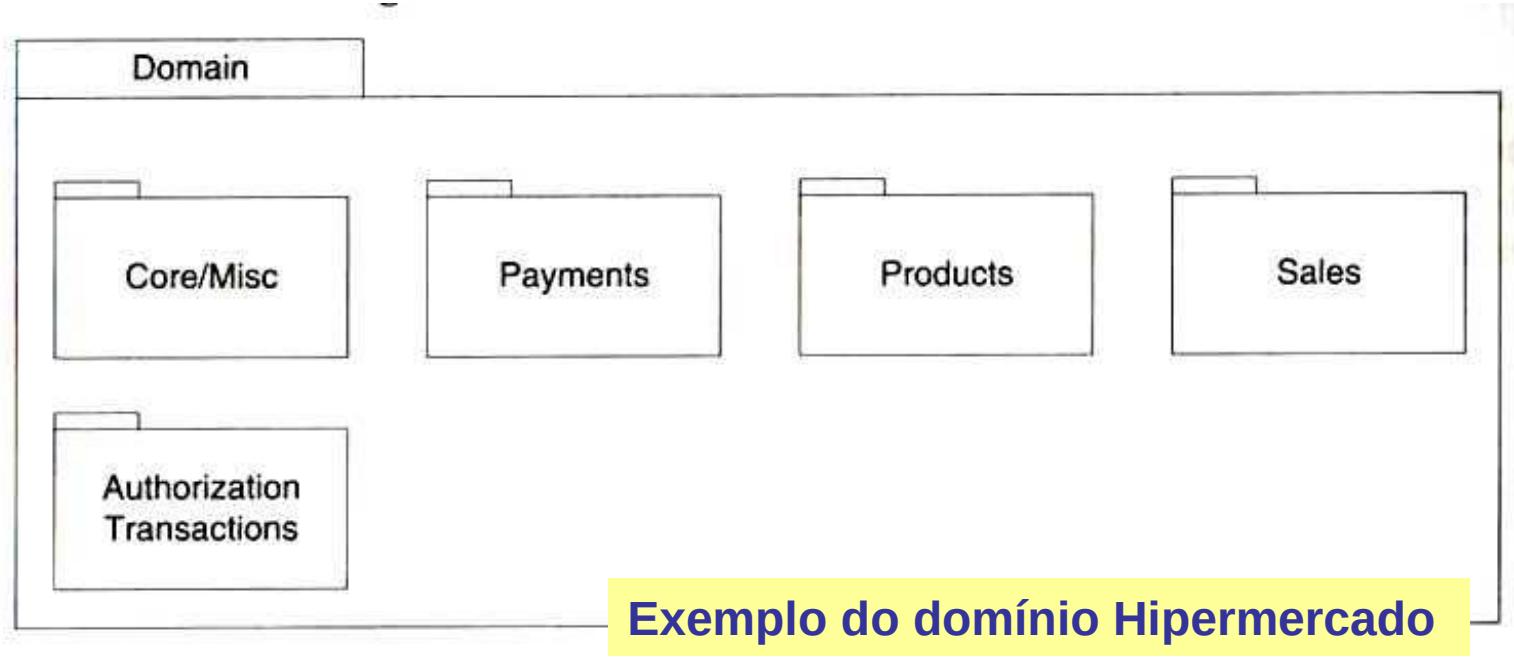
Packages podem ser vistos como subsistemas ou apenas como "name spaces" (contextos de identificação), mas são muito importantes na especificação da arquitectura lógica do sistema.

DC e Packages



Package que agrupa todos os tipos possíveis de contas definidas no negócio de uma dada instituição

DC e Packages

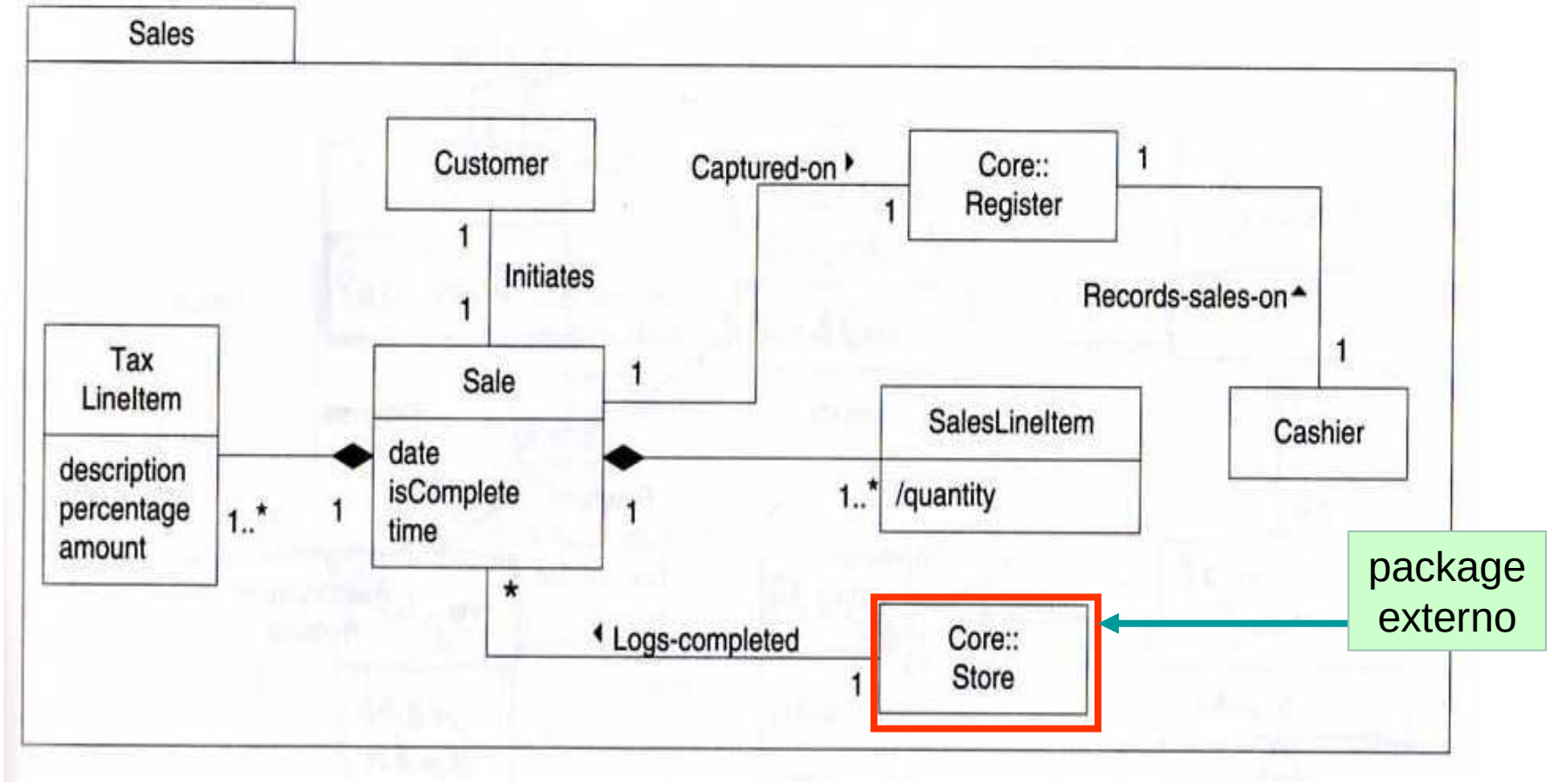


Exemplo do domínio Hipermercado

- Os **packages** de domínio, que poderão resultar da análise de alto nível, correspondem a sub-domínios particulares e bem identificados

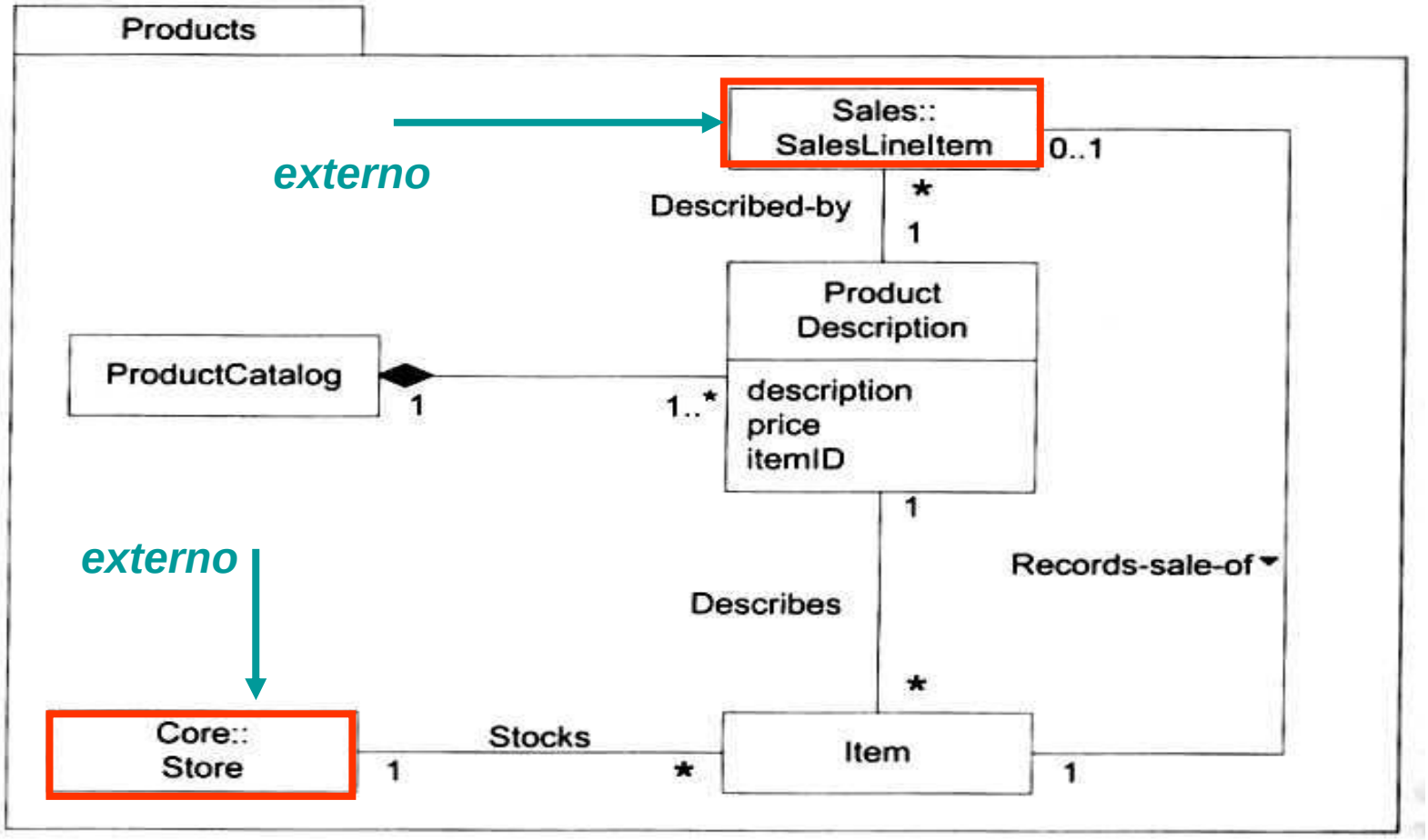
- Cada **package** de sub-domínio agrupa e estrutura um conjunto de classes.

DC e Packages



Package do sub-domínio Vendas

DC e Packages



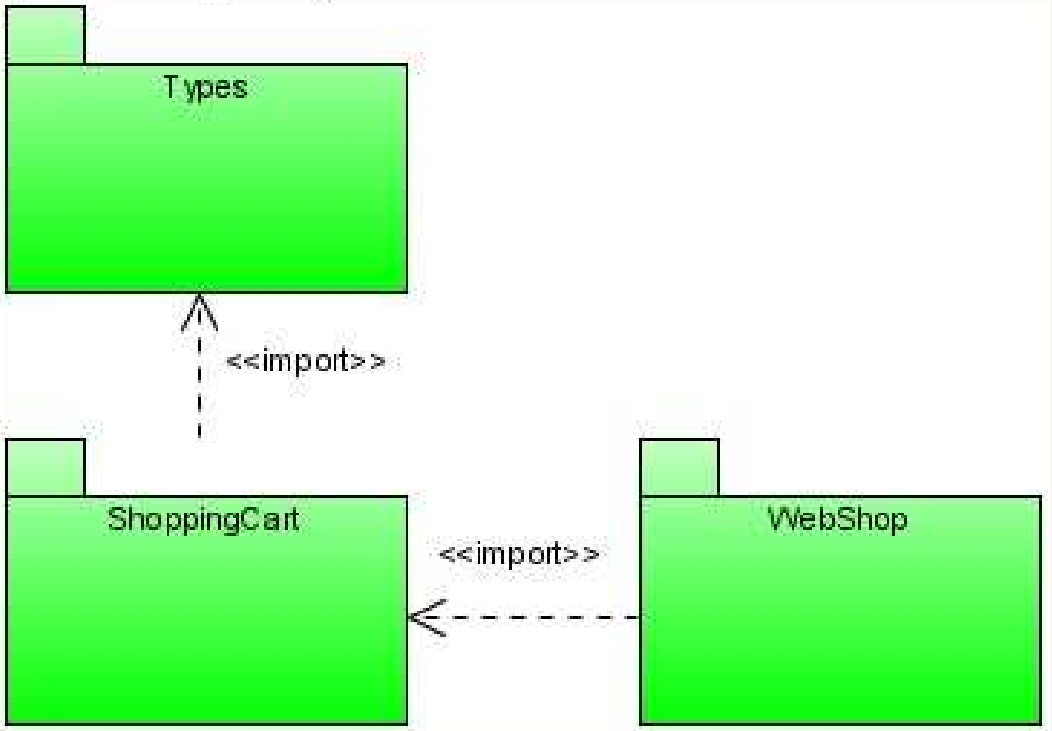
Package do subdomínio Produtos

Packages: Dependências

Exemplos retirados da página de tutoriais do Visual Paradigm.

PackageImport(public)

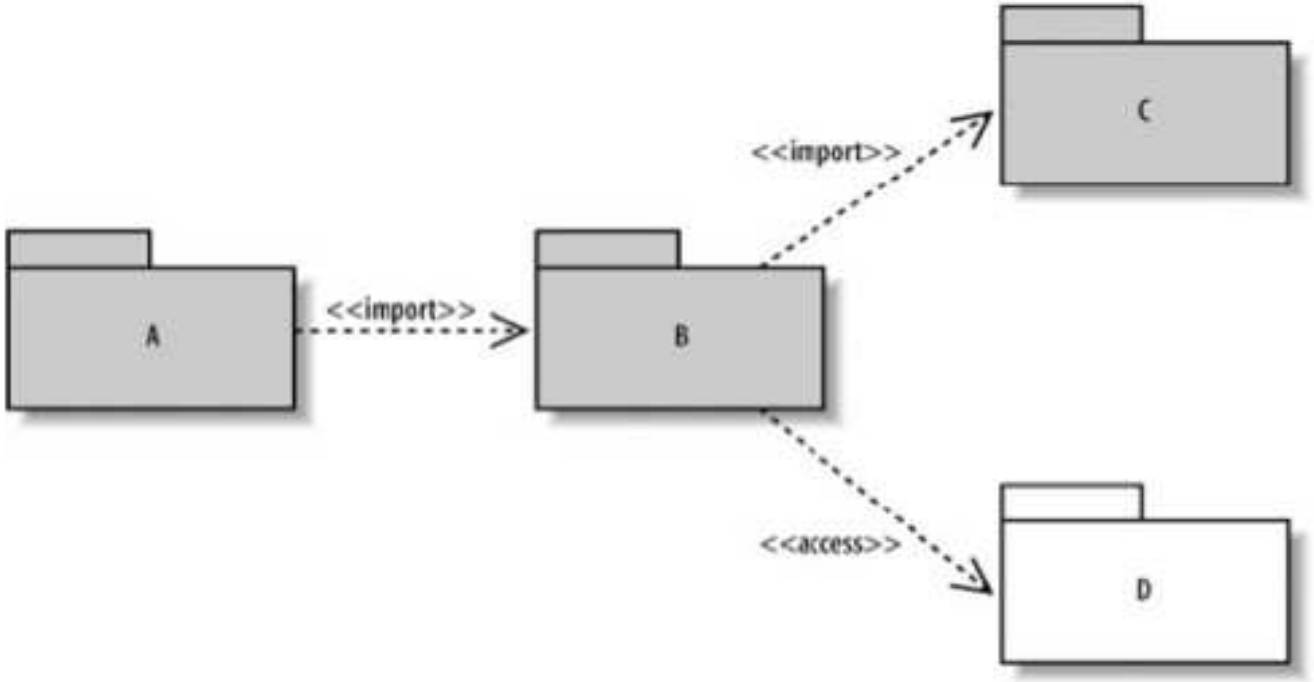
A package import is defined as a directed relationship that identifies a package whose members are to be imported by a namespace.(OMG Unified Modeling Lanauge Specification - UML 2.0 Superstructure Specification, p. 113)



<<import>>

Packages: Dependências

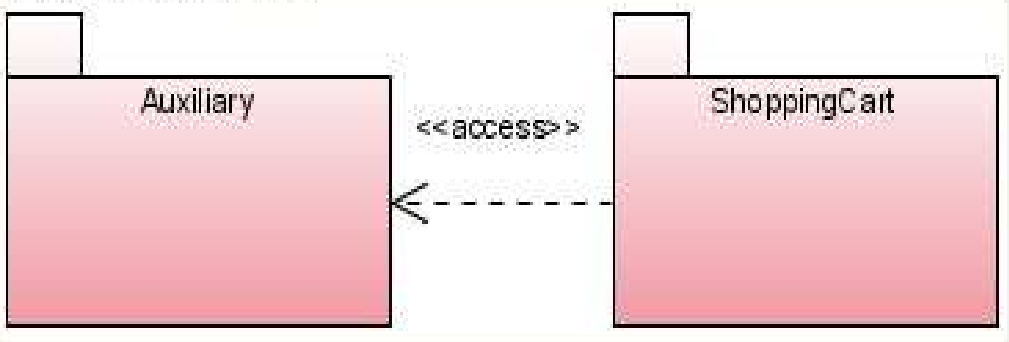
- Utilização de *<<import>> e <<access>>*
  - O package *B* vê os elementos públicos em *C* e *D*.
  - *A* importa *B*, pelo que vê os elementos públicos em *B* e em *C* (porque este é importado por *B*)
  - *A* não tem acesso a *D* porque *D* só é acedido por *B* (e não é importado).



Packages: Dependências

PackageImport(private)

A package import is defined as a directed relationship that identifies a package whose members are to be imported by a namespace.(OMG Unified Modeling Lanauge Specification - UML 2.0 Superstructure Specification, p. 113)

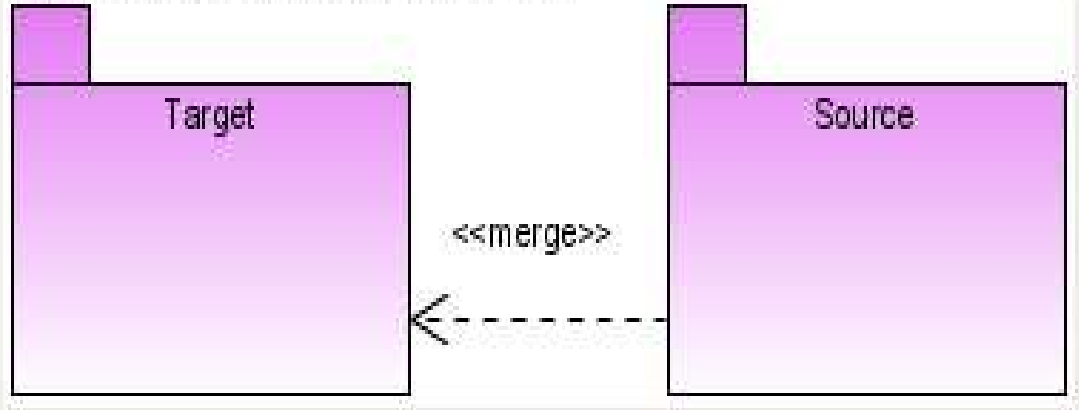


<<access>>

Packages: Dependências

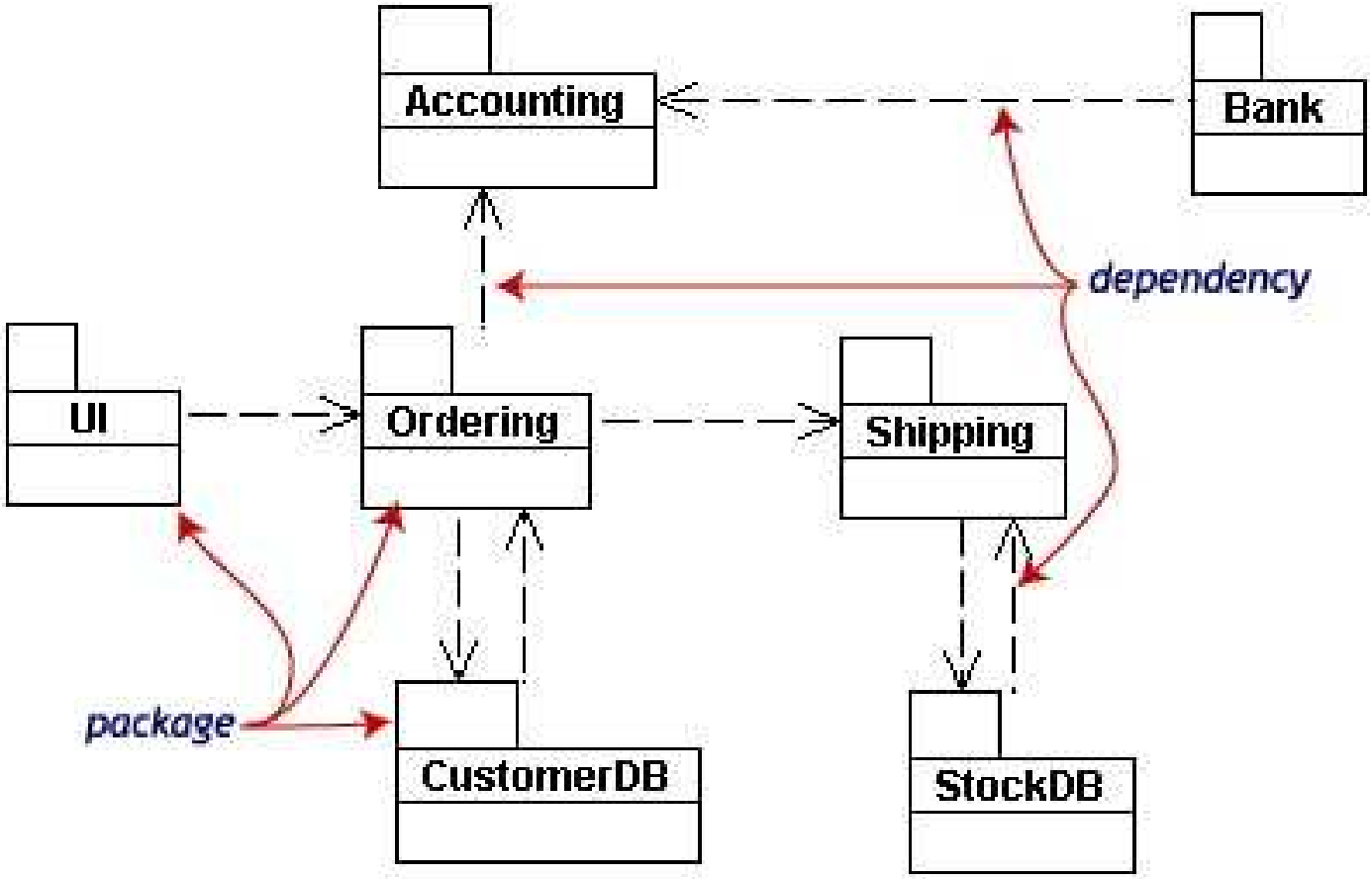
PackageMerge

A package merge is a directed relationship between two packages, that indicates that the contents of the two packages are to be combined. It is very similar to Generalization in the sense that the source element conceptually adds the characteristics of the target element to its own characteristics resulting in an element that combines the characteristics of both.(OMG Unified Modeling Lanauge Specification - UML 2.0 Superstructure Specification, p. 114)



<<merge>>

Packages: Dependências



- Dependências definem-se com ---> + << >> estereótipo

Subsistemas: Importância

▣ **Subsistemas** são partições de um sistema que possuem algumas das seguintes propriedades e/ou características:

☑ **Representam unidades independentes**, ou seja, que são autónomas, que podem ser internamente alteradas sem que tal implique alterações nas outras unidades, porque mantêm as API ou interfaces;

☑ **Podem portanto ser independentemente desenhados, testados e instalados;** As suas mudanças, adaptações, etc. não implicam com as outras unidades do sistema software (**mantendo a API**);

☑ **Podem representar sistemas externos** à concepção;

☑ **Podem representar/modelar "componentes" (já existentes)**;

☑ **Podem representar agrupamentos de classes** que em conjunto representam funcionalidade de nível superior de granularidade, que são depois "usadas" através de uma API conjunta, encapsulando a sua estrutura interna, e que é capaz de criar instâncias em "run-time" !!

▣ Como identificar subsistemas com estas propriedades na concepção ?

Subsistemas: Identificação

- Uma classe complexa resultante da análise é em geral transformada num susbsistema a desenhar se, aparentemente, apresenta demasiado comportamento para uma única classe;

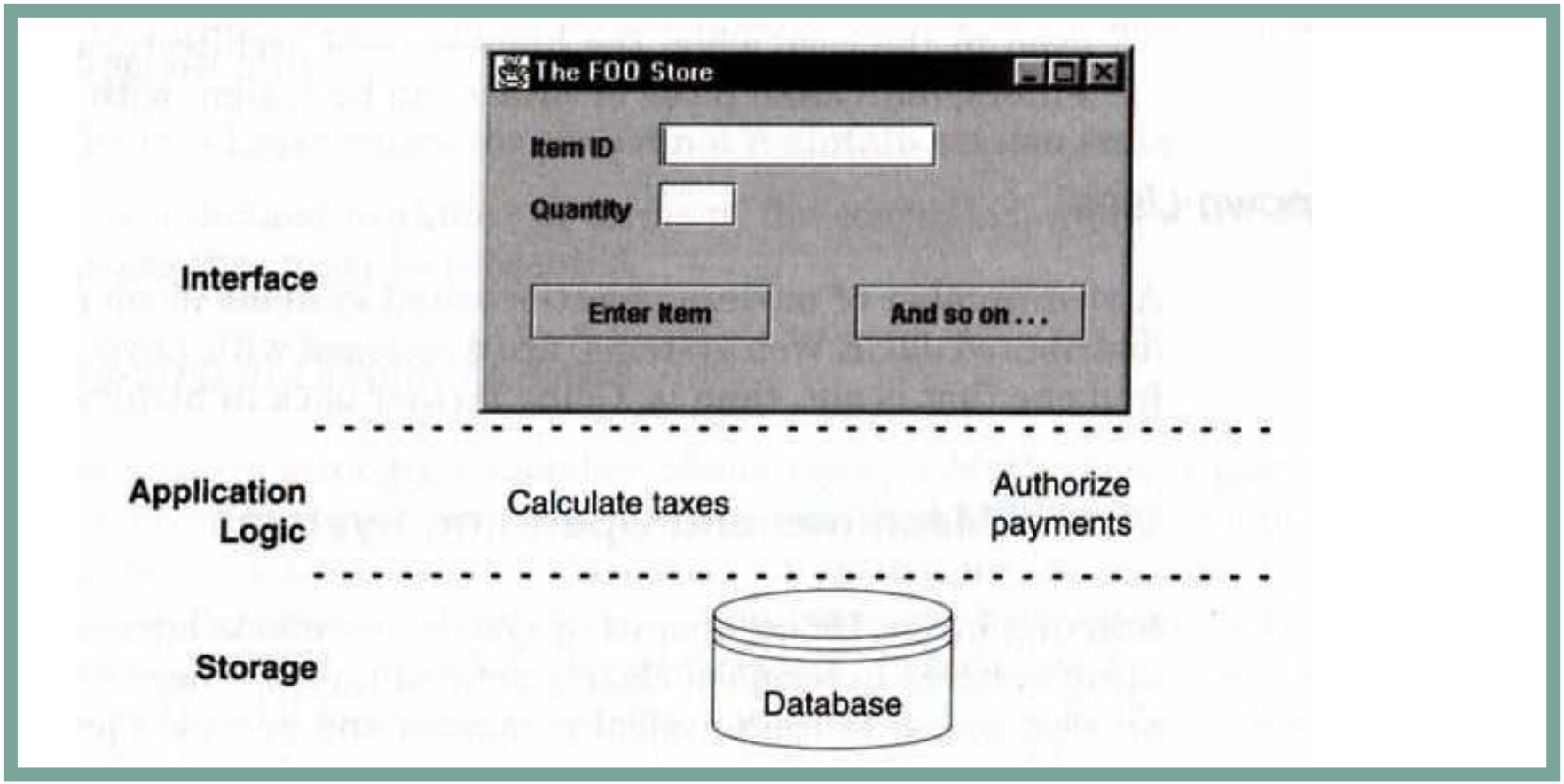
- Uma classe complexa é em geral transformada num subsistema se se verificar que pode ser implementada como um conjunto de classes que são bastante "ligadas", ou seja, "colaborativas" entre si;

- Subsistemas são também importantes porque permitem, usando certas regras, designadamente as que os tornam "independentes do contexto", que possam ser desenvolvidos até por equipas diferentes de projecto.

- Em UML 2.0 existe uma notação específica para "subsystems", mas, tal como em UML 1.0, subsistemas podem especificados usando "packages".

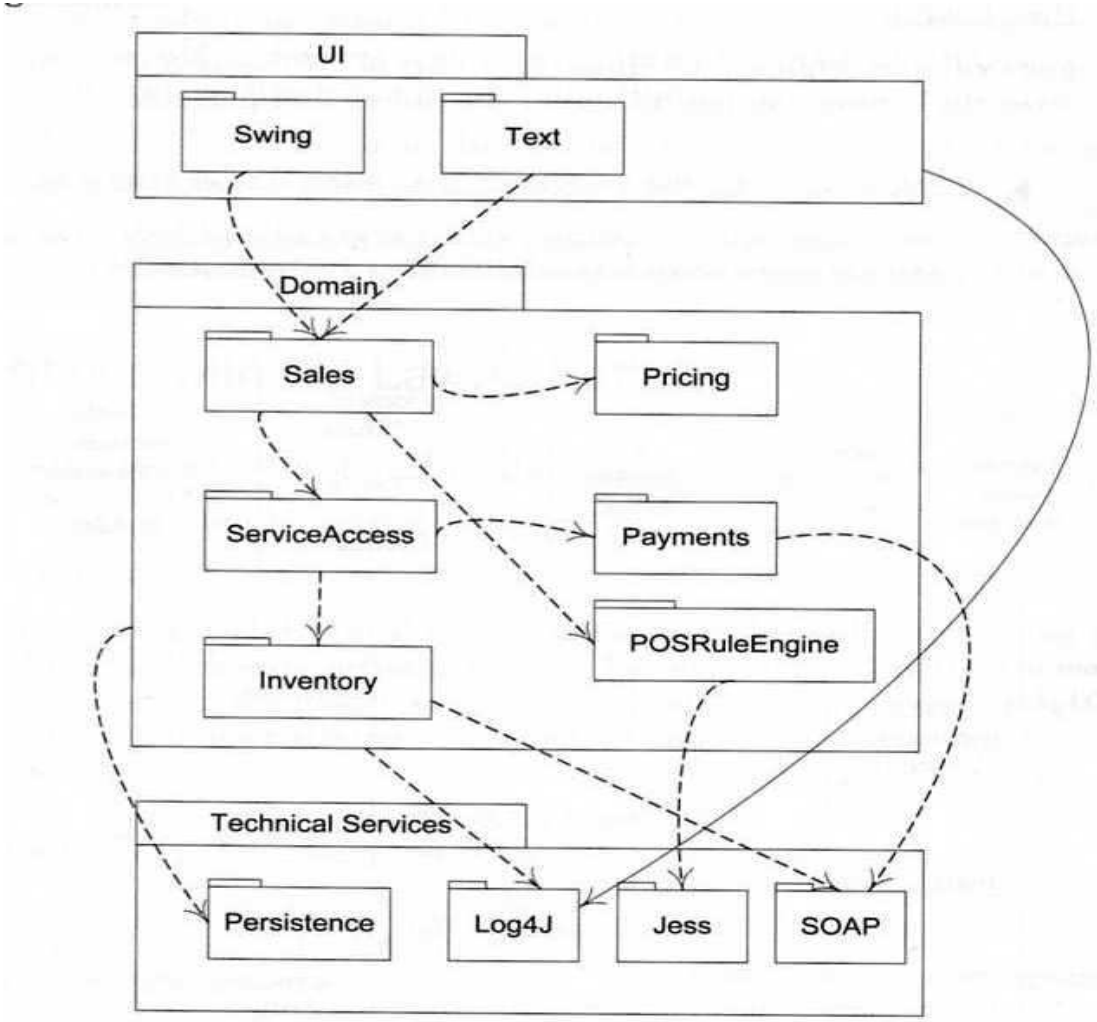
- Porém, e como há arquitecturas-objectivo bastante normalizadas e muito usadas, uma qualquer destas pode ser mesmo colocada como um requisito de solução, o que facilita a decisão (vejamos exemplos).

Subsistemas cf. Arquitectura



▣ Por exemplo, se como requisito tivermos por "target" uma arquitectura lógica de 3 camadas, típica de um **Sistema de Informação**, então será natural que tentemos especificar tendo por base 3 grandes "packages".

Subsistemas vs Arquitectura



Assim, packages podem ser desenhados usando uma estratégia de 1 package por "grupo de responsabilidades", ou por camada específica da arquitectura e dos serviços requisitados, ou seja, contratados (funcionalidade).

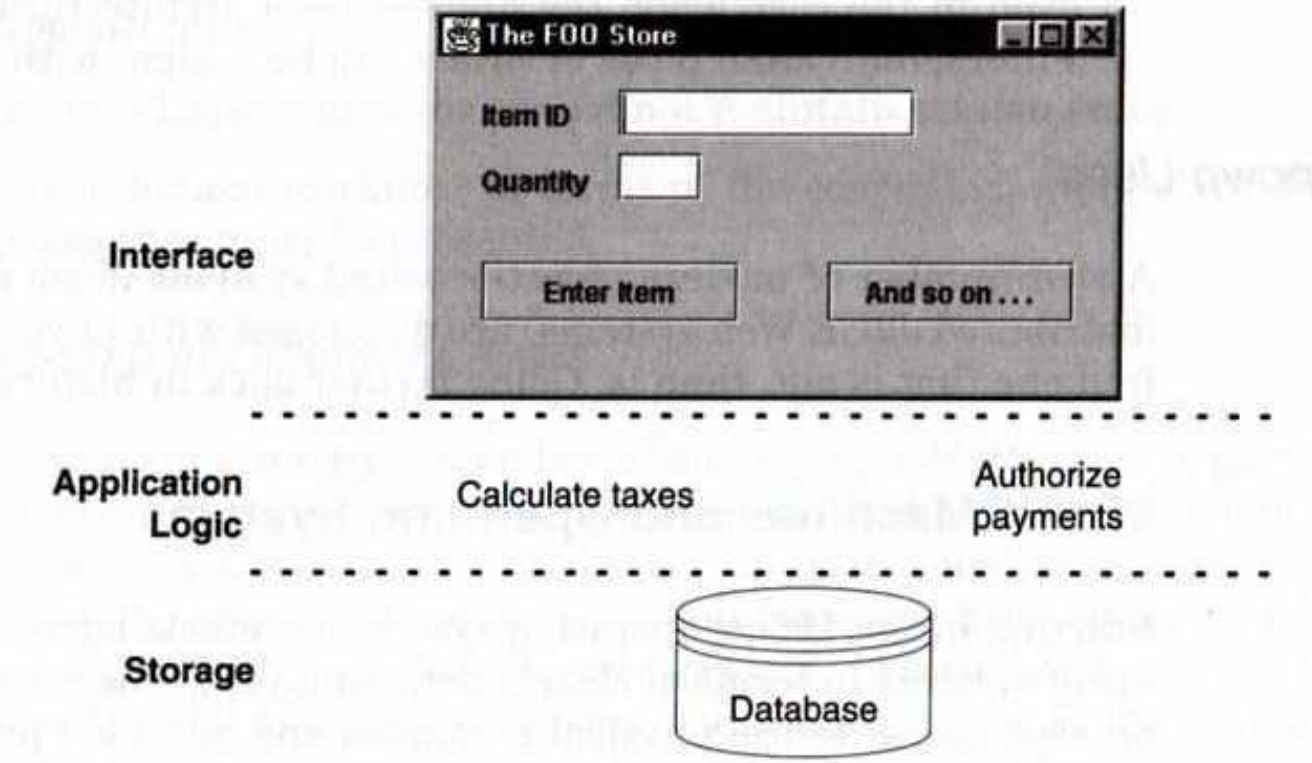
Dentro de cada mega-package, e em função da análise, podemos criar sub-packages, dentro dos quais as classes representam as informações e os comportamentos que, em colaboração entre elas, implementam os requisitos funcionais de alto nível.

Arquitectura Lógica

- A **Arquitectura Lógica** de um Sistema Software OO é uma organização, em grande escala, das suas classes, em termos de **packages** (cf. "namespaces"), **subsistemas** e **camadas**;

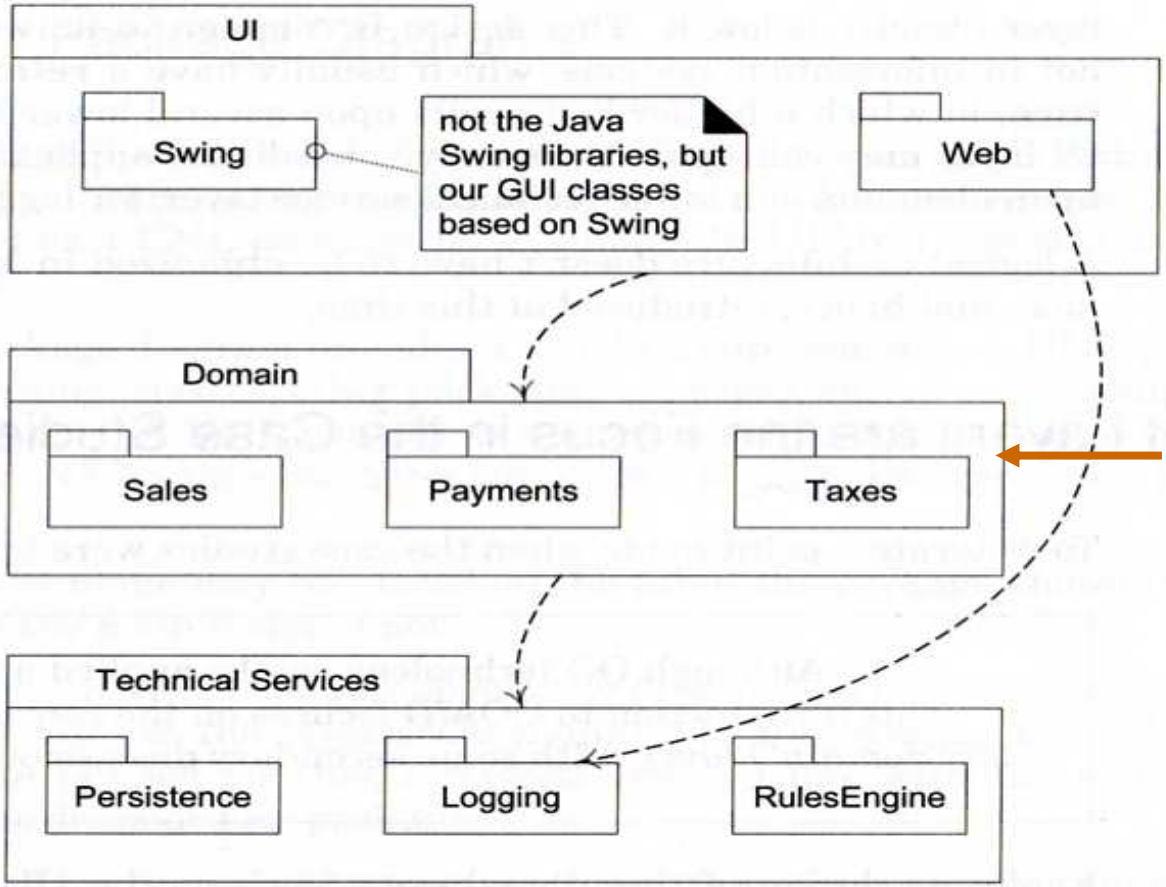
- Trata-se de uma **Arquitectura Lógica**, ao contrário do que se entende por uma **Arquitectura Física**, porque ainda não foram tomadas quaisquer decisões sobre como tais elementos são instalados ("deployed") sobre sistemas operativos, através de diferentes possíveis computadores, através de redes ou da Web, etc. (o que, em UML, é descrito através do Diagrama de "deployment" - DD);

- **Camada:** É um agrupamento de classes, packages e subsistemas que são coesos, ie., **fortemente interligados** ("strongly coupled" ou "with high cohesion"), e que, como equipa coesa e competente, assume a responsabilidade total por um dado "aspecto funcional" do sistema (cf. UI, Persistência de Dados, Segurança, Comunicações, Web, etc.).



▣ Voltando à figura anterior, cada camada exibe uma responsabilidade (funcionalidade + conhecimento) muito específica e, em geral, bem diferenciada ("aspecto" do sistema). Camadas de nível superior invocam funcionalidades das camadas inferiores, que "colaboram" para que a funcionalidade do sistema seja, de facto, correctamente prestada.

Arquitectura Lógica

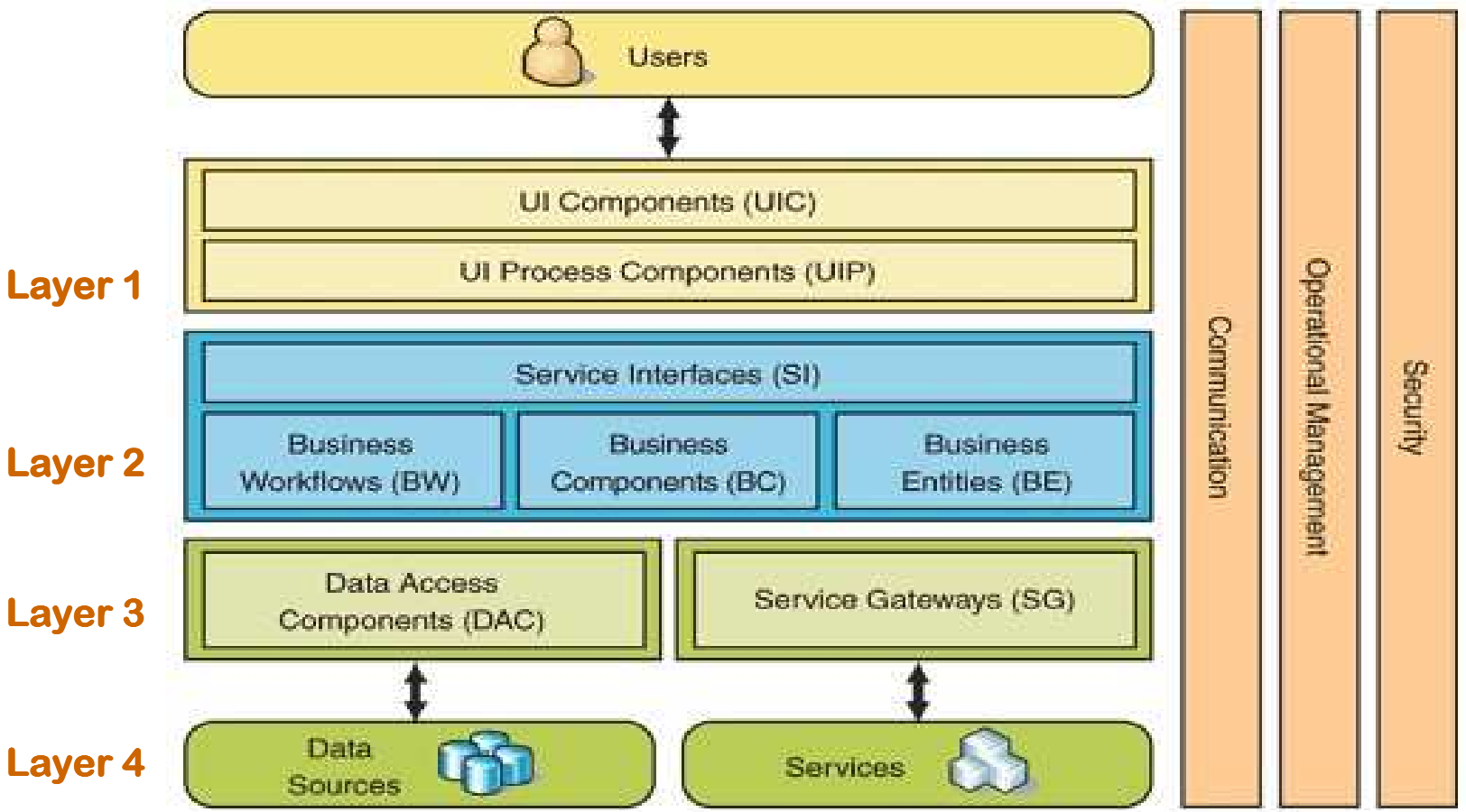


Partições

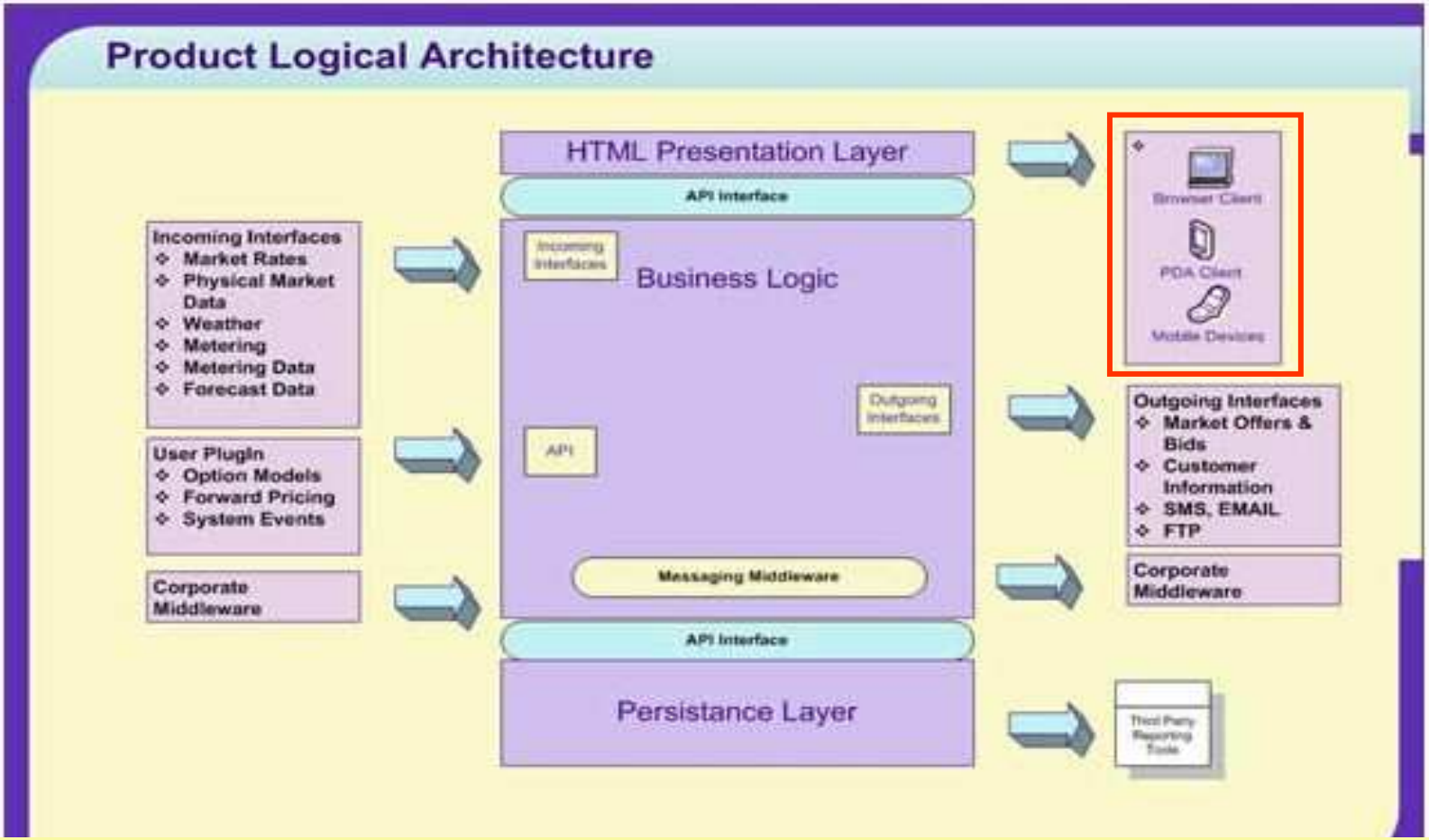
Subagrupamentos funcionais que são reconhecidos como especializações dentro da mesma camada

- Arquitectura de 3 camadas modelada a alto nível usando packages.

Uma arquitectura genérica de 3 camadas e serviços.



Subsistemas cf. Arquitectura



▣ Arquitectura de um Sistema Software de 3 camadas, e de grande complexidade. Note-se a distinção entre **API** e **"interfaces"** (protocolos) entre este e outros sistemas externos.

Subsistemas: Identificação

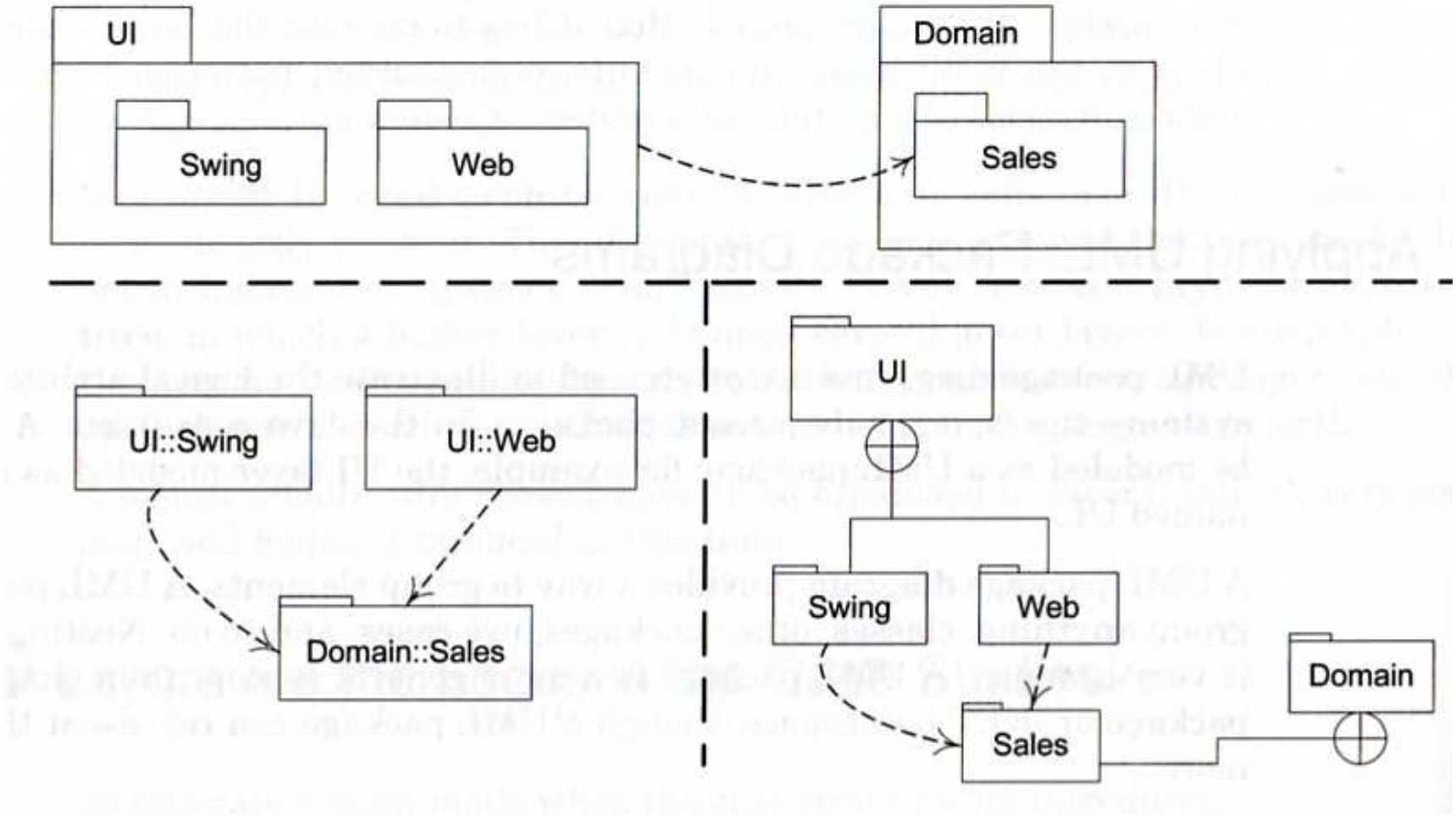
- Assim, como se comprova pelos exemplos anteriores, a primeira mais valia de um Sistema Software relativamente, em especial, aos seus actores primários (cf. Use Cases) é providenciar-lhes a realização de tal tarefa, se possível.

- O Sistema Software aparentemente ideal, seria o sistema monolítico, fechado, auto-suficiente, sem especializações, logo não explorando, por reutilização ou por "uso", modalidades de comunicação entre "sistemas", através de interfaces e protocolos bem definidos.

- Hoje não existem sistemas monolíticos ! Sistemas monolíticos cf. os nossos : System são de fraca coesão, ou seja pouco especializados, de enorme complexidade, não reutilizáveis, difíceis de manter e permanentemente a sofrer alterações. NÃO !!

- Assim, temos que pensar em SUBSISTEMAS, ou seja, funcionalidades distintas, especializadas e bem divididas !

Packages: Notações



Packages são pois uma boa 1.ª forma de representar camadas funcionais

Subsistemas: Identificação

▣ **Questão:** Uma arquitectura lógica OO, ou seja, conceptual e OO, é determinada mais pela estrutura ou mais pelo comportamento ?

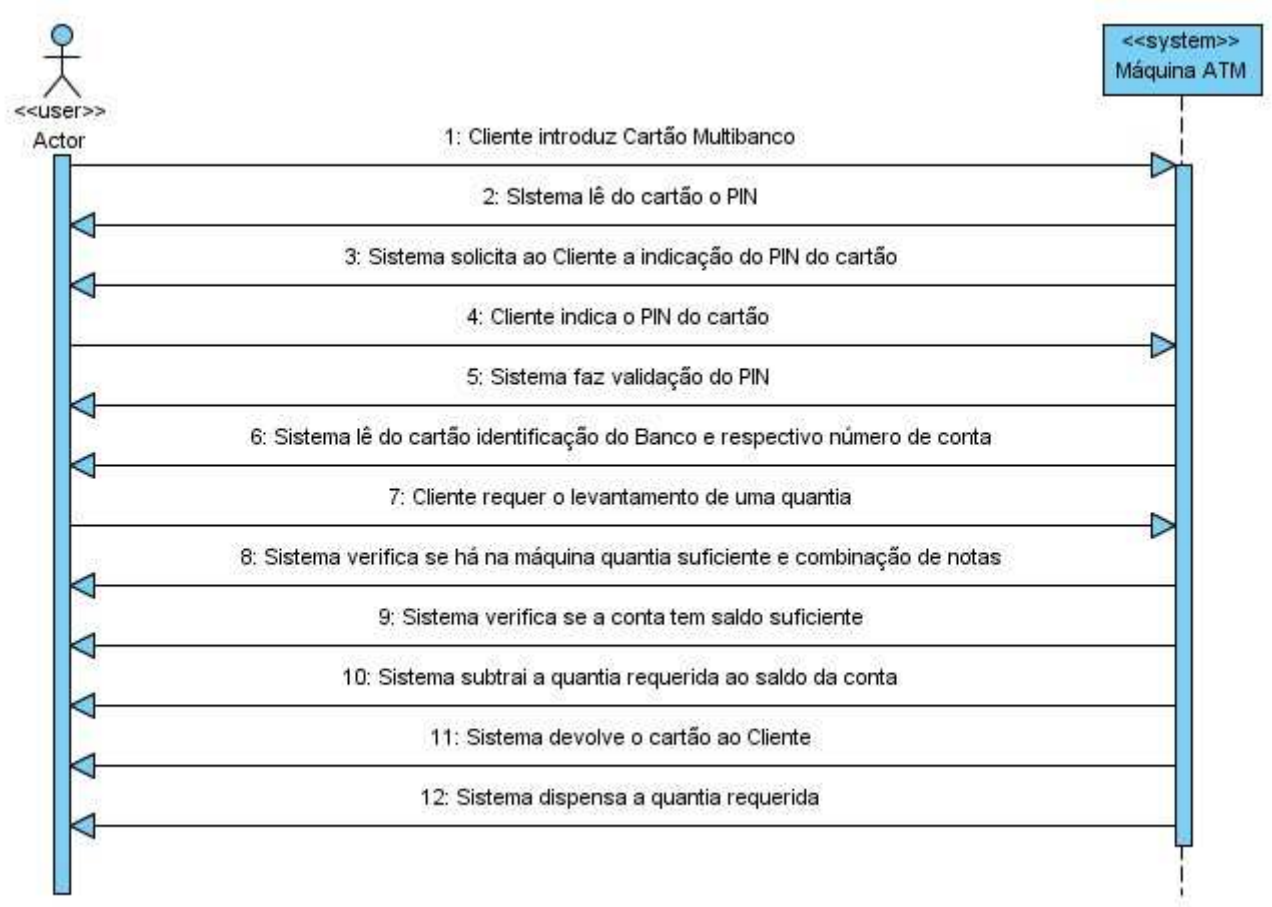
▣ **Resposta:** Pela **coesão das classes** (entidades) em termos funcionais, ou seja, pela "causa maior" para a qual todas dão a sua colaboração.

▣ **Questão:** Haverá alguma metodologia ou método que nos ajude na identificação dos subsistemas de um dado sistema depois de (+-) terminada a fase de captura de requisitos (cf. UCs, DA, DSS) ?

▣ **Resposta:** Não. Mas podemos definir algumas regras, baseadas no facto de termos tratado com método a criação dos UC a partir da modelação do domínio, e termos tratado com método a criação dos DSS a partir dos UCs textuais.

Subsistemas: Identificação

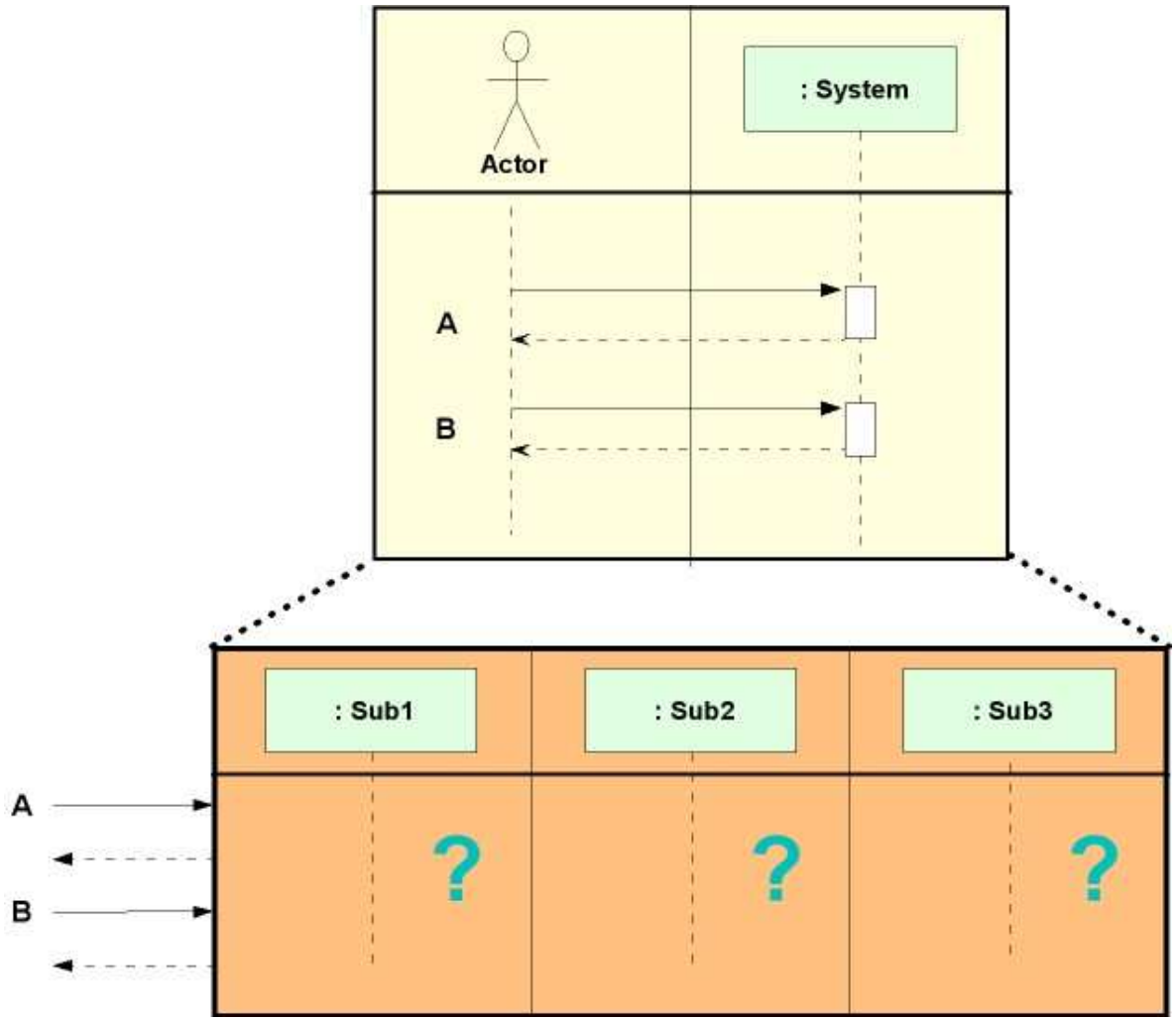
- Claro que um DSS apenas nos sugere uma classe com "pouca coesão", e é exactamente por isso que temos que fazer refinamento.



Uma classe que representa todo o sistema !

Ficção !

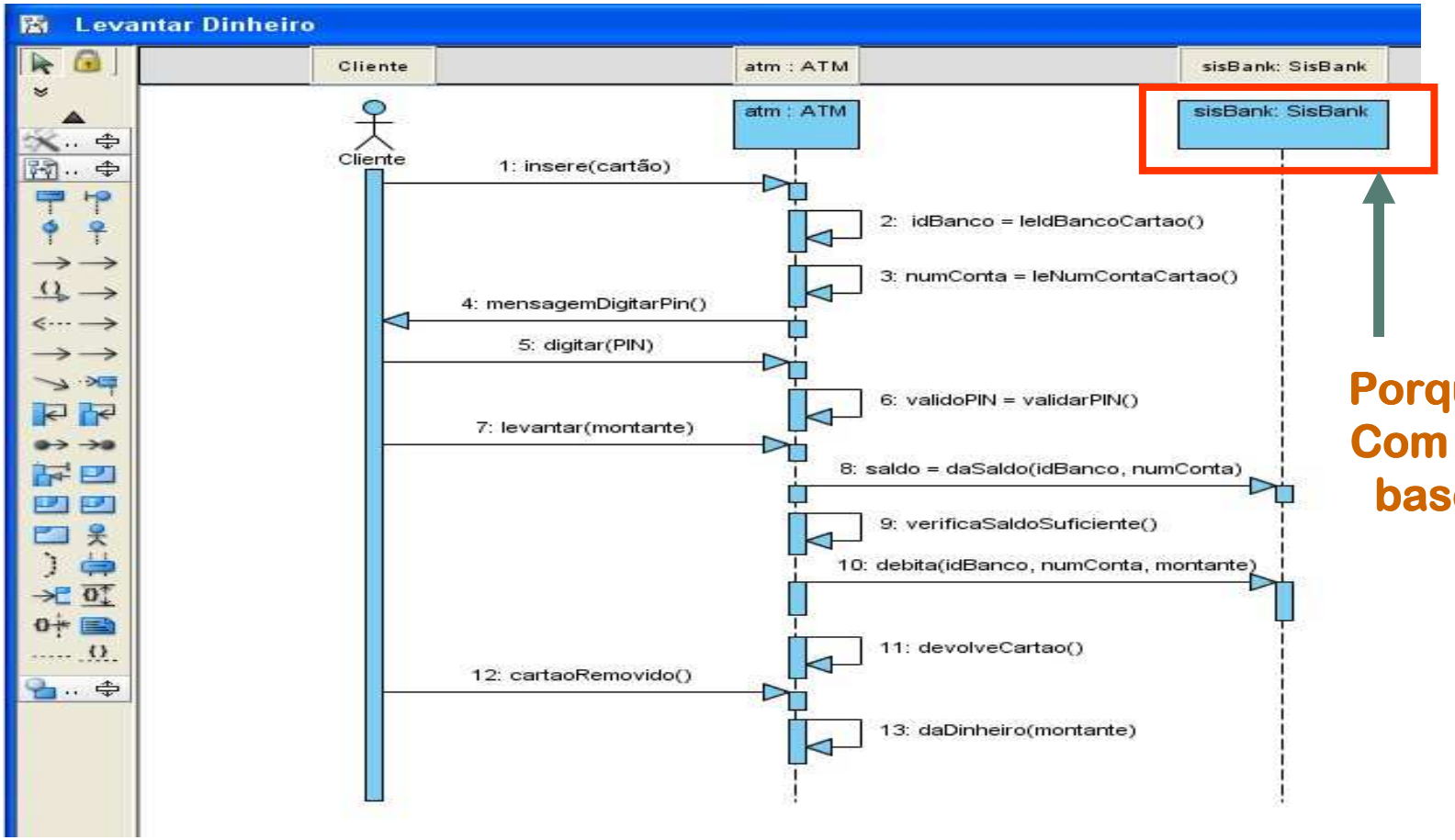
Subsistemas: Identificação



Claro que temos que refinar o comportamento ou funcionalidade mas com que base e com que regras ?

Subsistemas: Identificação

▣ **Mesmo que saibamos refinar cada um dos DSS, tal não corresponde a uma identificação de subsistemas, porque estamos apenas a refinar e a realizar 1 UC. Falta uma perspectiva global !**



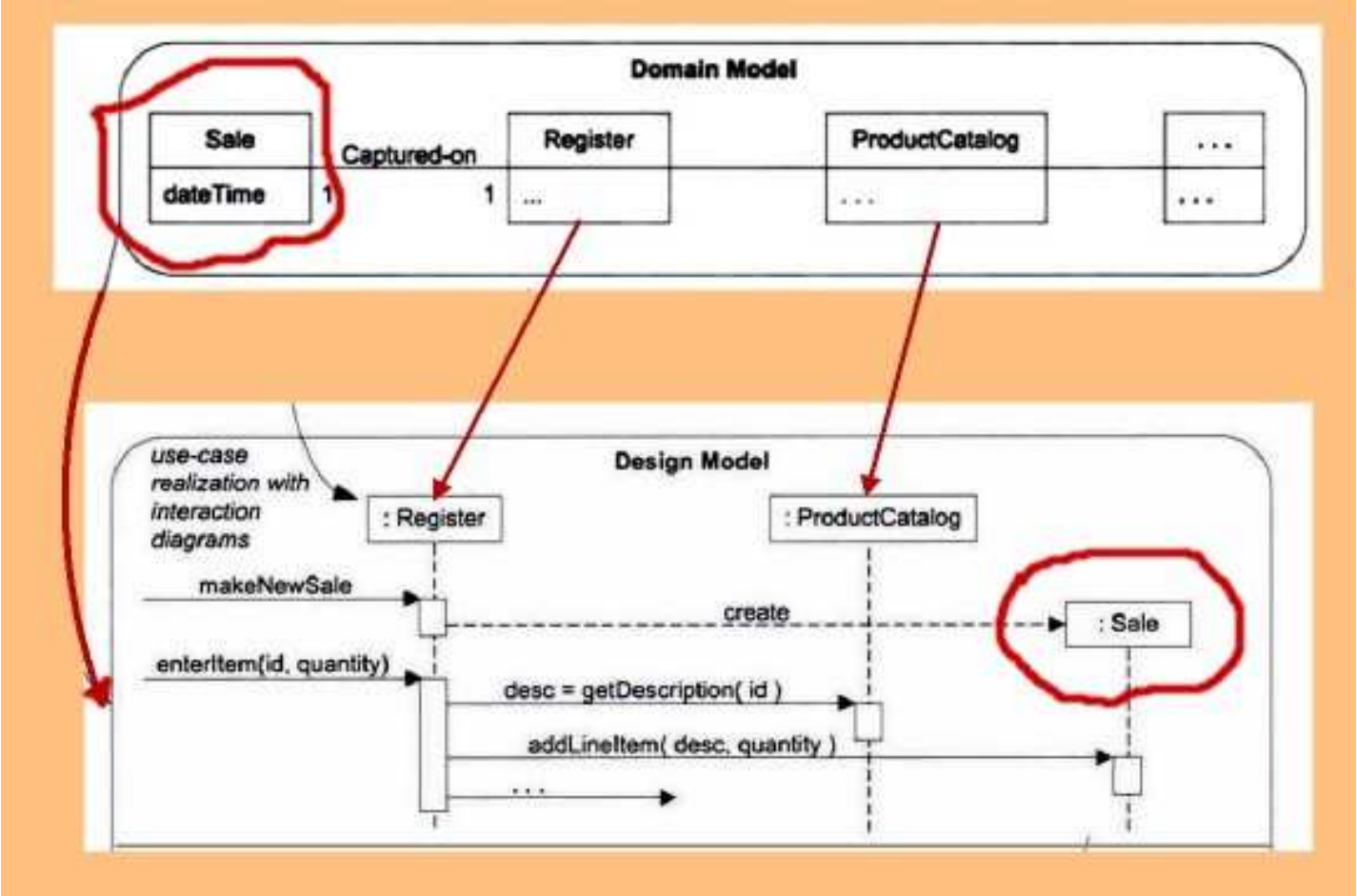
Porquê ?
Com que base ?

Procura de uma estratégia

- Um DSS representa, para cada UC, as interacções (mensagens) entre os actores e o sistema;
- O sistema é, a este nível, uma "black box", abstraindo-se da sua estrutura interna e do modo como realiza a sua funcionalidade;
- Mas, o total da funcionalidade "requisitada" e observável do sistema a desenhar, pode ser vista como todas as respostas do sistema às mensagens que os actores "enviam" ao sistema na sua interacção com ele, tal como especificado nos DSS obtidos dos UCs;
- Assim, deveríamos olhar para cada mensagem/evento que os actores iniciam para obter do sistema as suas "mais valias", e catalogá-las sob a forma de "responsabilidade funcional total" do sistema;
- Um bom modelo de domínio seria fundamental para uma tal abordagem ser "naive" mas coerente (cf. classes do domínio importantes no desenho do sistema são as "mesmas", mas refinadas, ao nível do desenho).

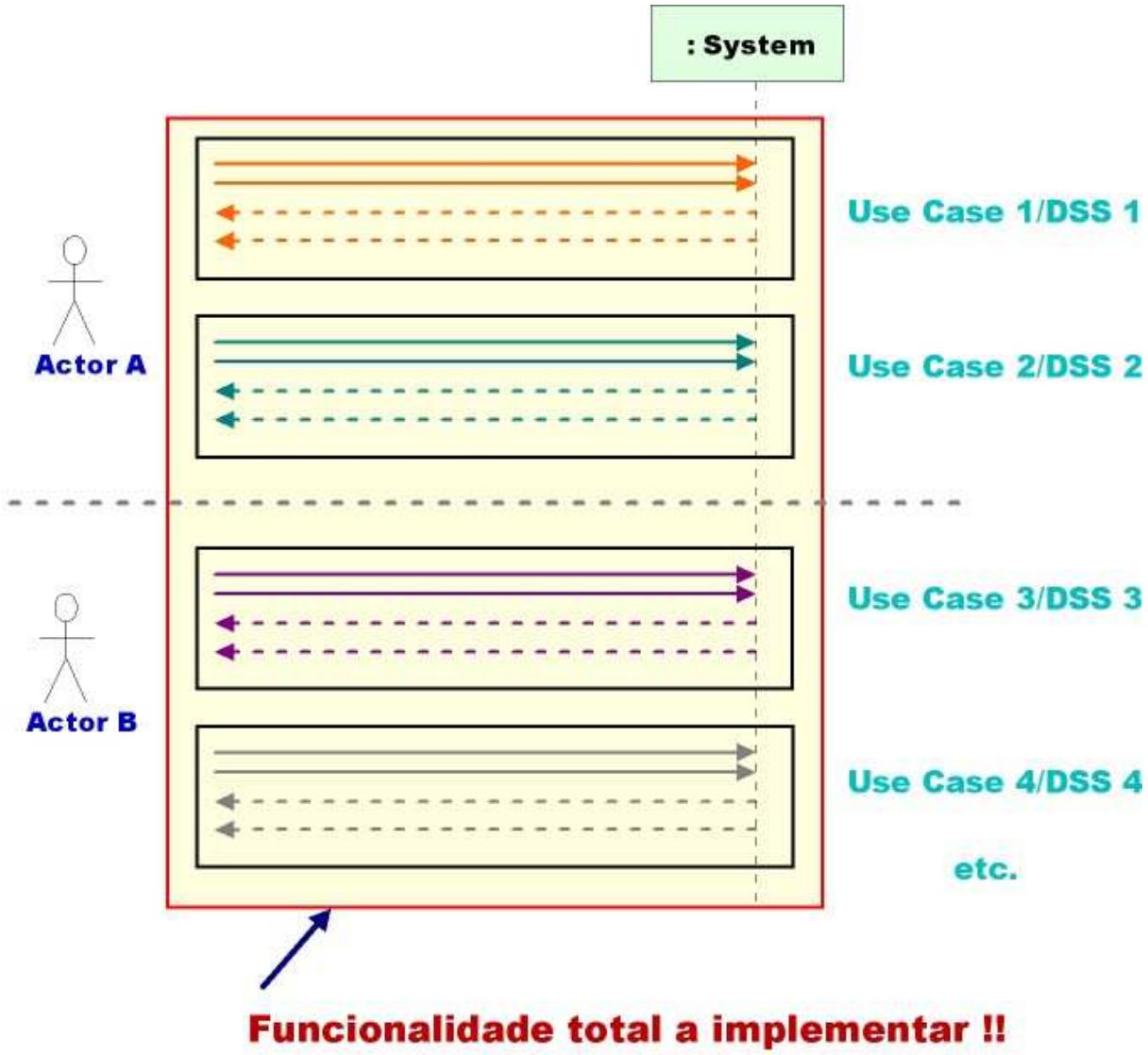
Procura de uma estratégia

Importância do Modelo de Domínio vs. Coerência dos projectos



Na criação dos Diagramas de Sequência de mais baixo nível

Procura de uma estratégia

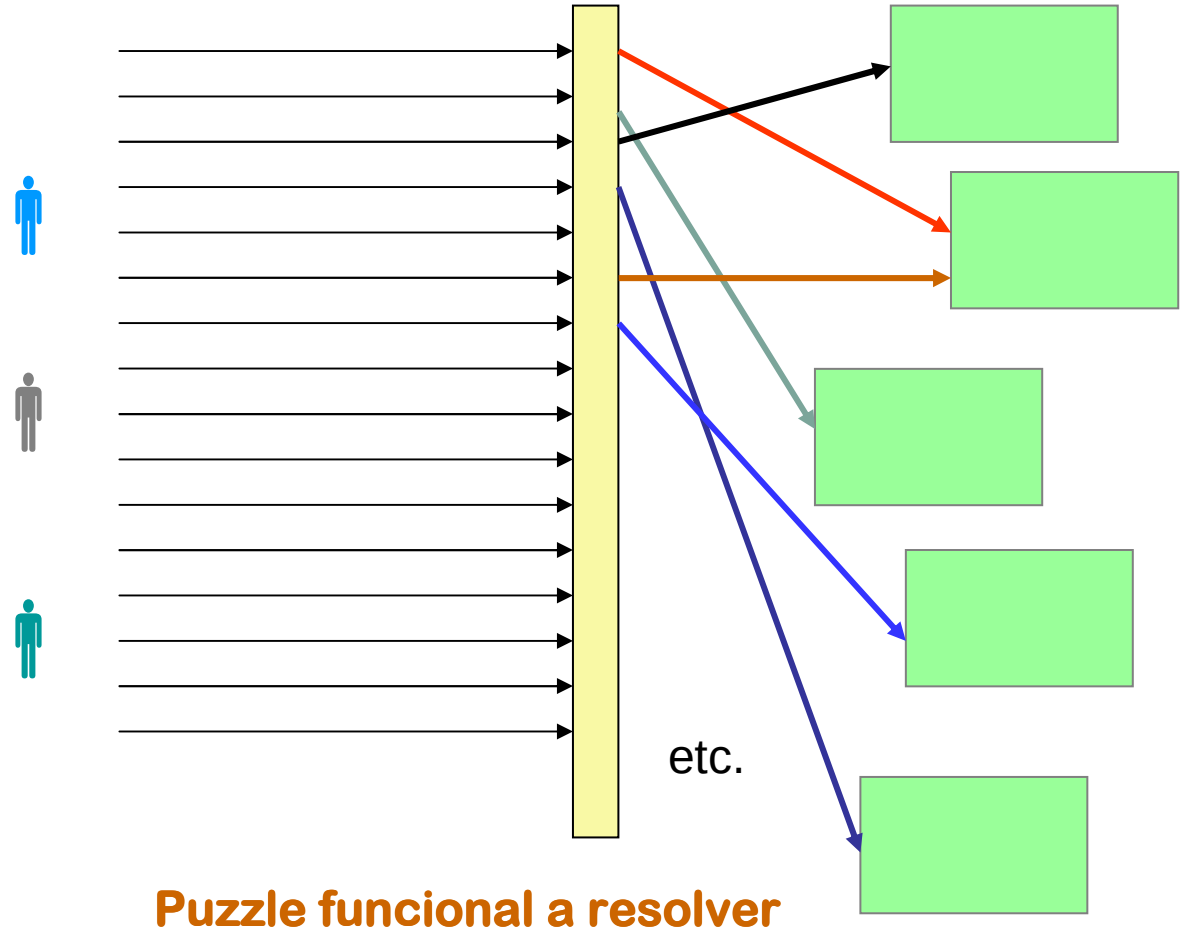


Somando todos os eventos iniciados pelos actores de um sistema que foram registados em todos os DSS do projecto, e sabendo que cada evento acciona uma operação do sistema, e que para cada operação teremos que ter um método directamente invocável que a resolve, então sabemos exactamente qual a carteira de responsabilidades funcionais

Procura de uma estratégia

O que temos então de momento ?

Uma carteira bem identificada de "responsabilidades funcionais", ou seja, de operações que as classes que vão representar o sistema terão que implementar em métodos.



Puzzle funcional a resolver

Concepção OO: a base

▣ **Questão:** Como se concebe a camada lógica de uma aplicação usando objectos ?

▣ **Resposta1:** Cria-se uma classe **:System** e nela se colocam todos os métodos necessários para que se tenha a funcionalidade pretendida

☹ **ERRADO, É EXACTAMENTE ISTO QUE NÃO QUEREMOS**

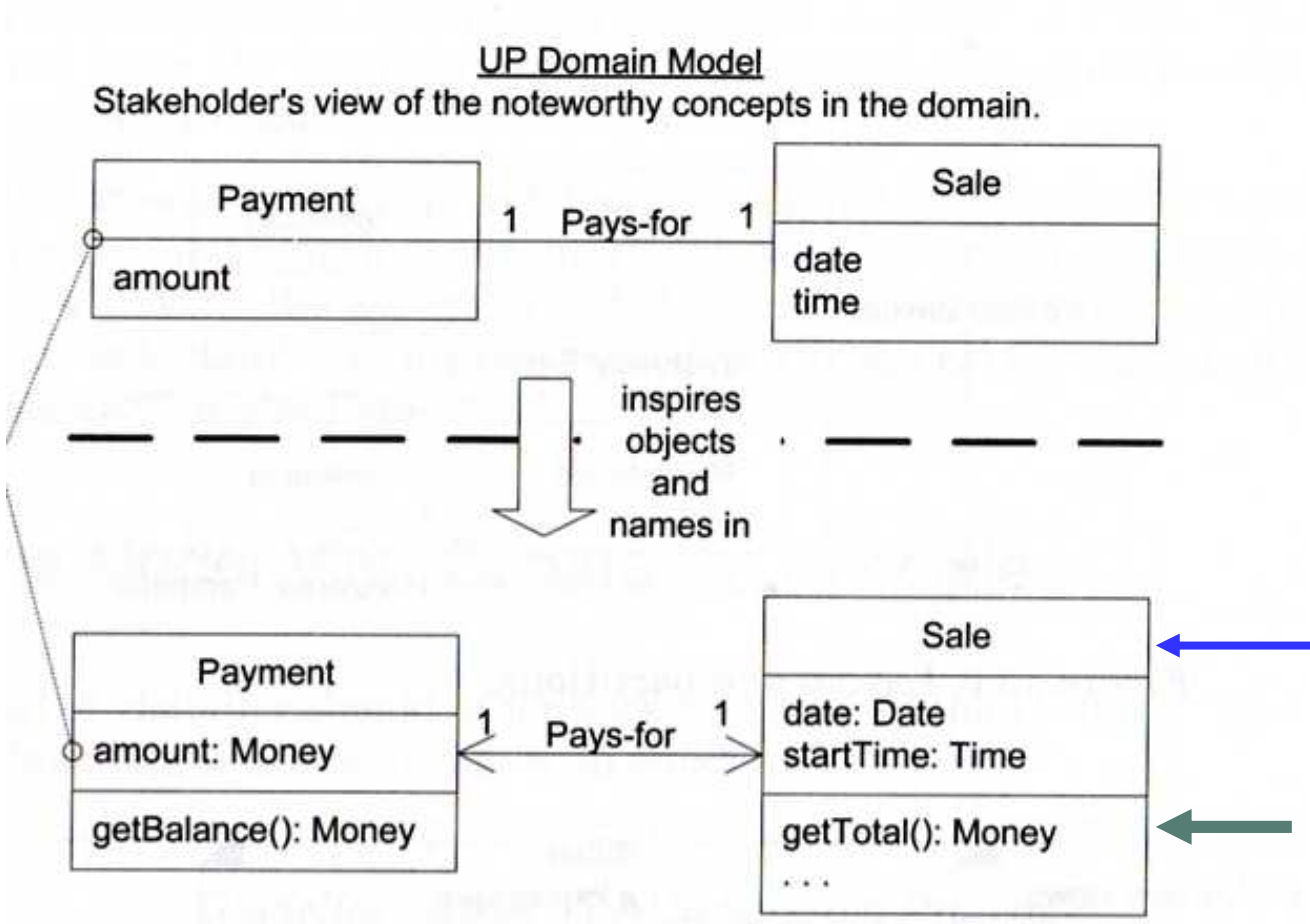
▣ **Resposta2:** Criam-se "objectos de software" com os mesmos nomes e atributos similares aos dos objectos/entidades identificadas no mundo real, que estão representados no modelo do domínio, e atribuem-se-lhes responsabilidades funcionais (ou seja, métodos adequados).

Estes objectos de software designam-se por "objectos do domínio" pois representam uma "coisa" importante do domínio do problema.

Ou seja, concebem-se **classes** que possam criar tais objectos.

Procura de uma estratégia

▣ **A identificação prévia de um conjunto de classes que fazem parte do Modelo do Domínio, sendo certo que algumas destas se irão tornar classes software do sistema software.**



Na concepção da camada lógica

Responsabilidades funcionais

Estratégia (cont.)

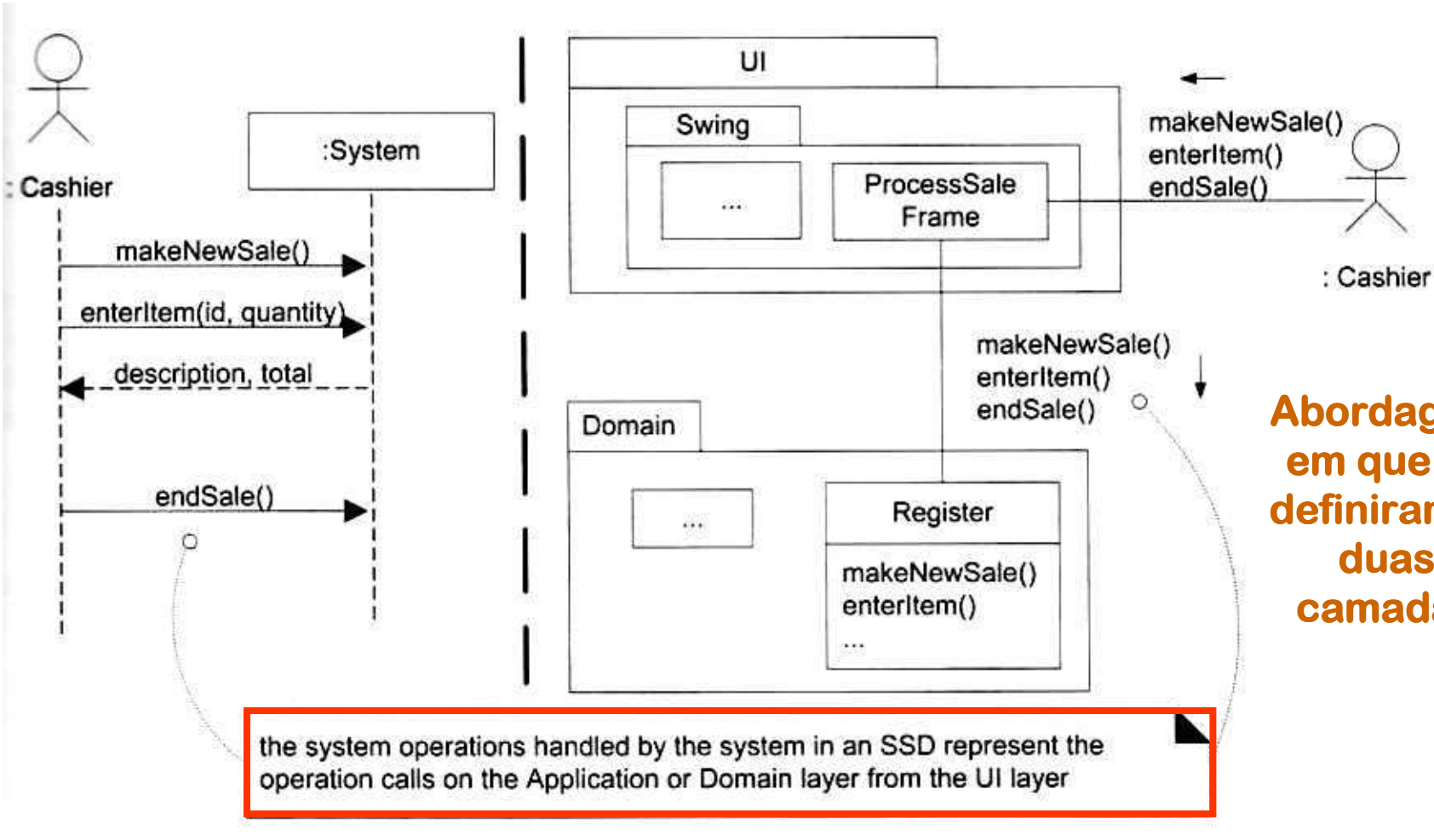
▣ **Questão: Como se identificam outras classes necessárias à implementação de tal carteira de responsabilidades funcionais, ou seja, que possuam métodos que implementem as operações de sistema que foram contadas e identificadas nos DSS ?**

Resposta1: De forma intuitiva. (☹ só "feeling"? É difícil).

Resposta2: De forma racional usando os modelos que, supostamente de forma coerente, foram já desenvolvidos e, idealmente, seguindo os seguintes passos:

1) Procurando identificar em primeiro lugar grandes grupos de funcionalidades, que se relacionam com "aspectos" do sistema ou até da arquitectura lógica escolhida (págs 198-201), sabendo-se que, como se viu atrás, os "packages" podem muito bem representar, sob a forma de agrupamentos de classes, tais "grupos de funcionalidade";

Concepção OO

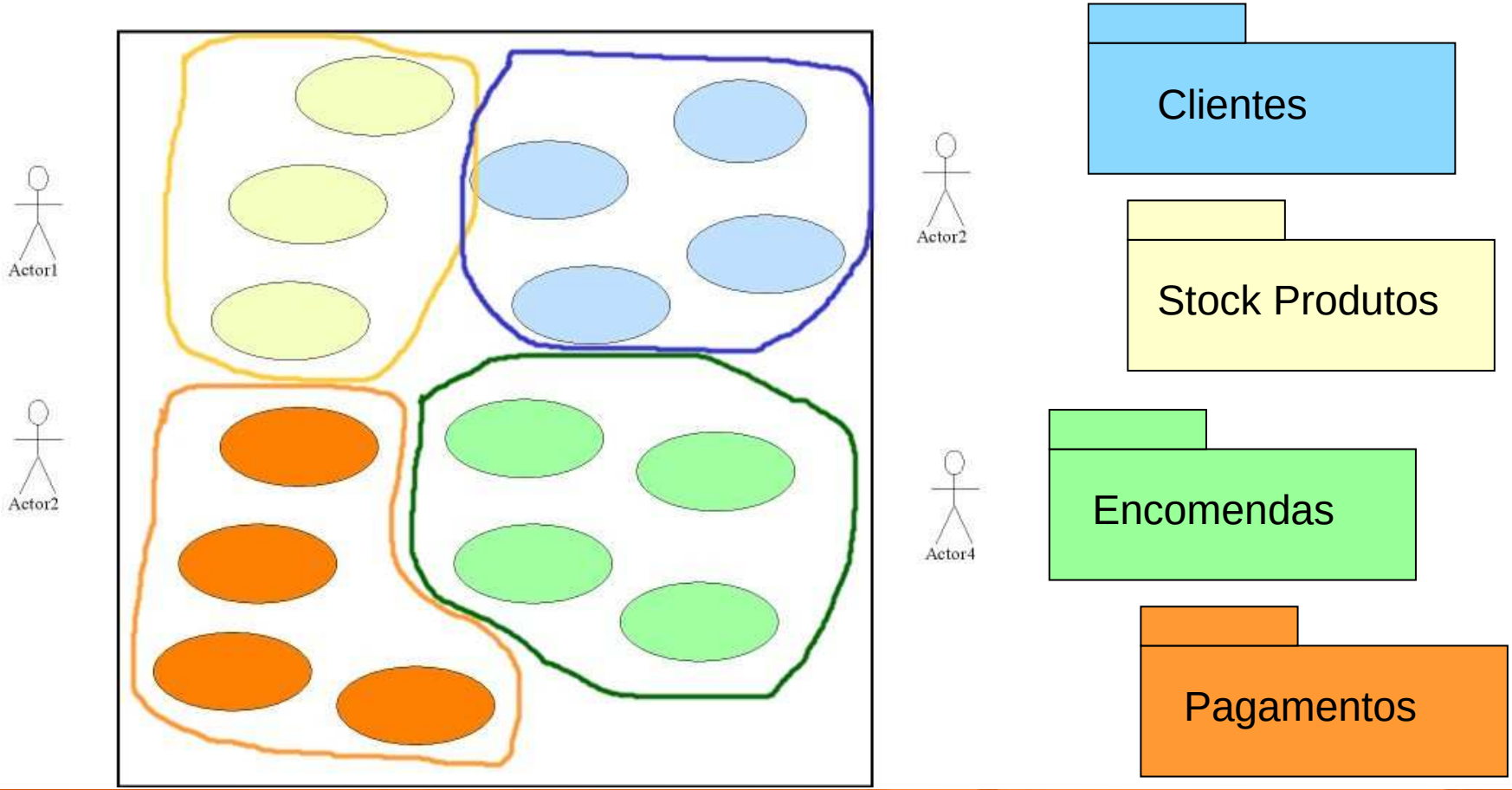


Abordagem em que se definiram já duas camadas

Obs: Note-se a coerência dos nomes dos eventos e dos métodos

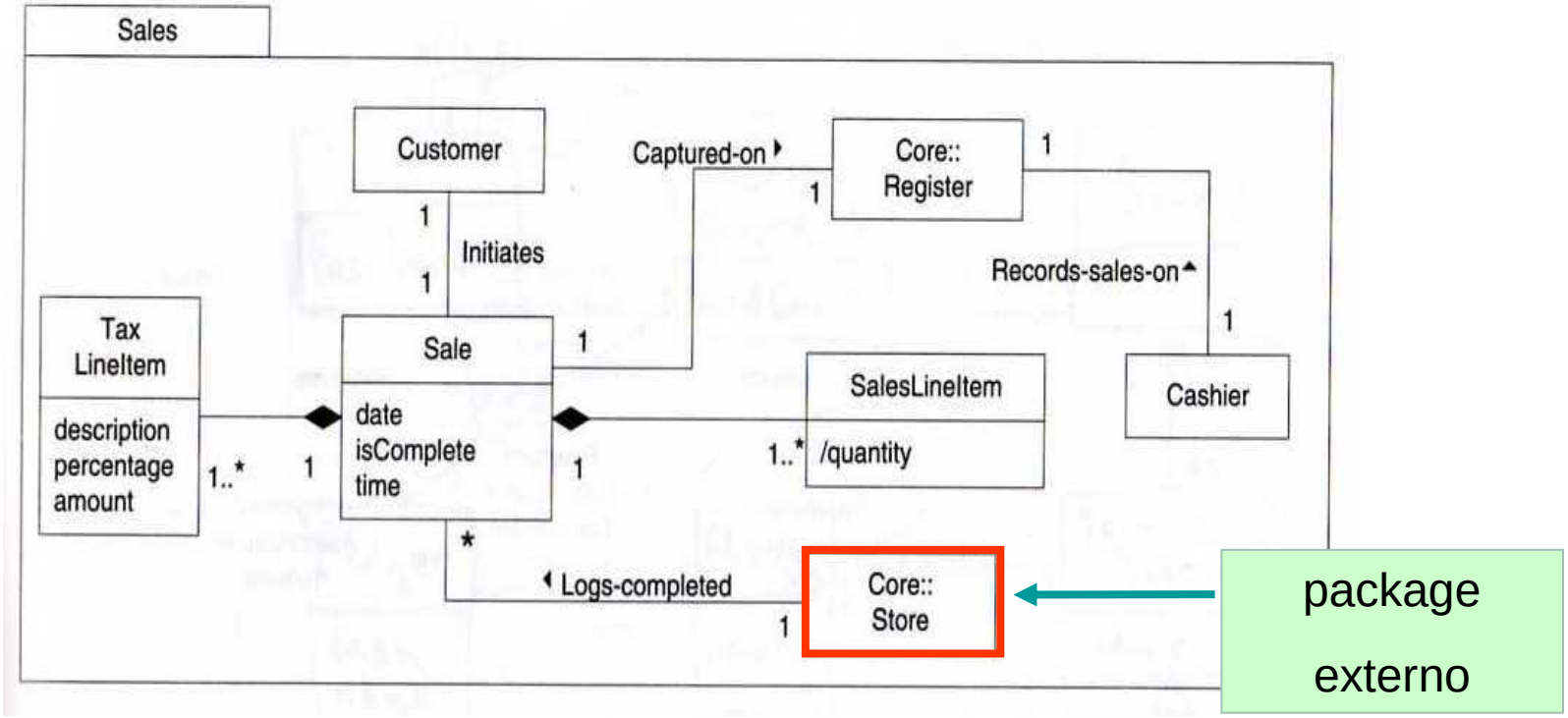
Procura de uma estratégia

▣ **Podemos usar um (ou um conjunto) de Diagrama de Use Cases para identificar tarefas que, pela sua afinidade, possam ser agrupadas num grupo de funcionalidade mais abstracta, cf. Vendas, Gestão de Clientes, Gestão de Produtos, Facturação, etc.**



- Em seguida, se assumirmos esta abordagem "top-down" a partir dos diagramas de Use Cases, cada "package" deve agora ver a sua estrutura especificada usando um diagrama de classes, ficando da forma-exemplo:

Package do sub-domínio Vendas



Responsabilidades: CRC-cards

Responsability-based modeling

A análise da "responsabilidade" (o que deve fazer; o que deve "saber") de cada entidade no funcionamento global de um sistema, permite identificar subsistemas e suas formas de colaboração.

CRC-cards permitem melhor "desenhar" as nossas classes.

CRC-cards são usados para tentar analisar a criação de possíveis subsistemas (tal como os Diagramas de Colaboração)

Responsabilidades

- O que sabe (atributos)
- O que faz (métodos)

Técnica: CRC-cards

Class Name:	
Superclasses:	
Subclasses:	
Responsibilities:	Collaborators

Coisas que sabe

Coisas que faz

Classes que devem colaborar com esta para que esta possa cumprir as suas responsabilidades

Técnica: CRC-cards

CRC Card Sample

A CRC card is an index card that is use to represent the responsibilities of classes and the interaction between the classes.

BankAccount

Super Classes :

Sub Classes : SavingAccount, MarginAccount

Description :
Store the transcation record, customer data, balance, etc.

Attributes :

Name	Description
accountNumber	A unique value to identify the accounts

Responsibilities :

Name	Collaborator
Keep the latest value of the balance	Bank controller, Transcation records

BankController

Super Classes :

Sub Classes : AccountController, TranscationController, ATMController

Description :
Control the interactions between the customer and the bank system.

Attributes :

Name	Description
status	identify the status of the controller

Responsibilities :

Name	Collaborator
withDraw	Withdraw money from the bank acco

SavingAccount

Super Classes : BankAccount

Sub Classes :

Description :
Store the cash information of the customer record.

Attributes :

Name	Description
cashBalance	latest value of the cash balance

Responsibilities :

Name	Collaborator
getBalance	TranscationController, AccountControl

ATMController

Super Classes : BankController

Sub Classes :

Description :
Control the interactions between customer and the ATM terminals.

Attributes :

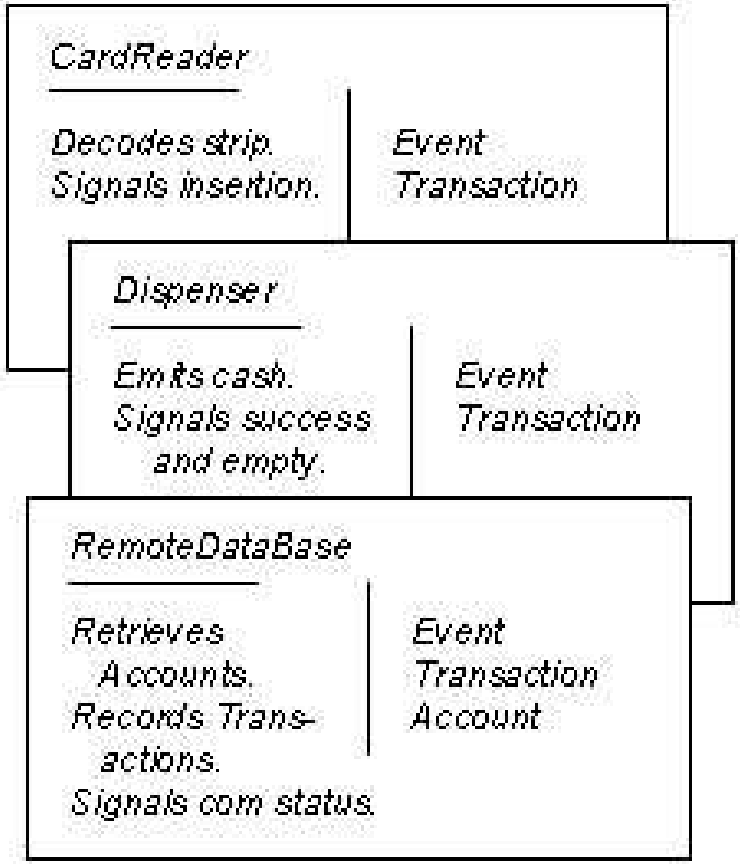
Name	Description
machineType	identify the type of the ATM terminal

Responsibilities :

Name	Collaborator
checkingPassword	

VP

Técnica: CRC-cards



Student	
Student number Name Address Phone number Enroll in a seminar Drop a seminar Request transcripts	Seminar

Técnica: CRC-cards

Professor	
Name Address Phone number Email address Salary Provide information Seminars instructing	Seminar

Transcript	
See the prototype Determine average mark	Student Seminar Professor Enrollment

Seminar	
Name Seminar number Fees Waiting list Enrolled students Instructor Add student Drop student	Student Professor

Enrollment	
Mark(s) received Average to date Final grade Student Seminar	Seminar

Student Schedule	
See the prototype	Seminar Professor Student Enrollment Room

Student	
Name Address Phone number Email address Student number Average mark received Validate identifying info Provide list of seminars taken	Enrollment

Room	
Building Room number Type (Lab, class, ...) Number of Seats Get building name Provide available time slots	Building

Building	
Building Name Rooms Provide name Provide list of available rooms for a given time period	Room

CRC cards de um sistema de inscrição de alunos num seminário

Regras fundamentais em OO

- Na próxima aula apresentaremos algumas regras e princípios que permitirão, seguindo uma abordagem OO, melhor conceber as nossas aplicações, que serão em geral, multi-camada.