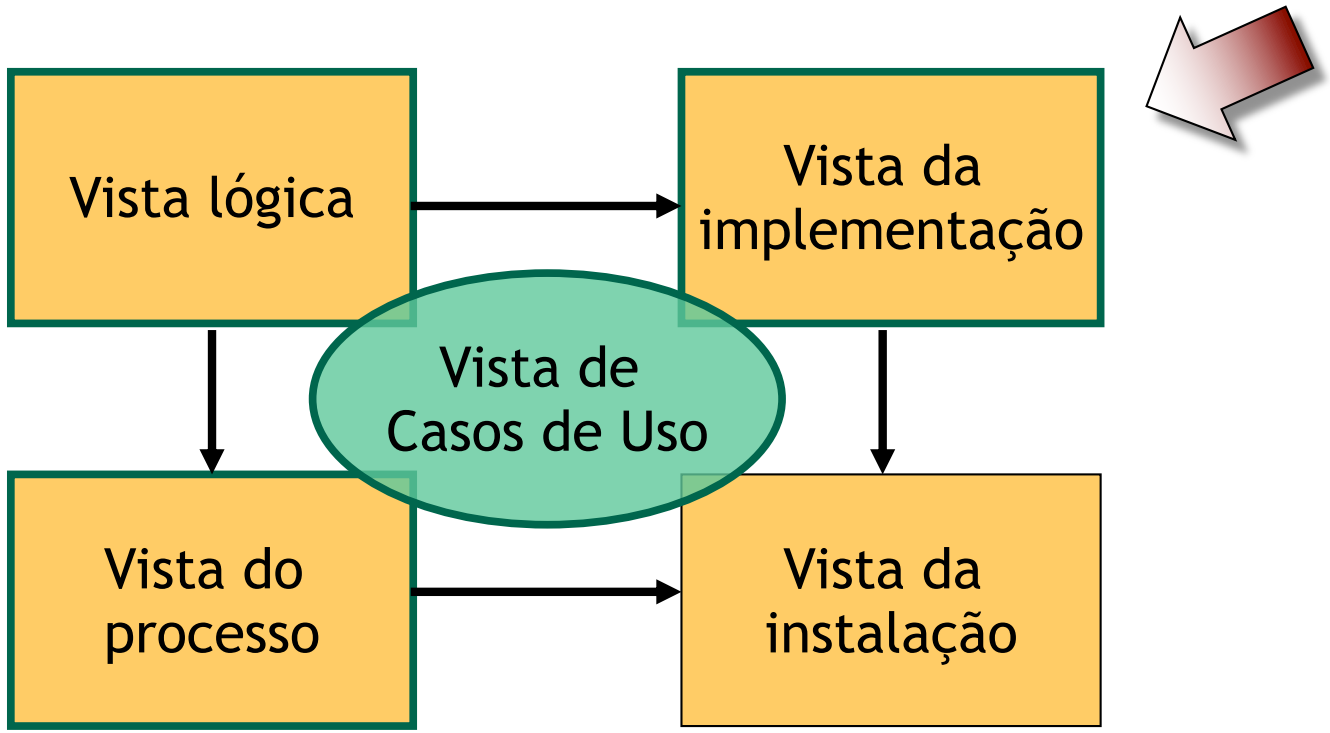
Modelação Estrutural II
(Diagramas de Package)

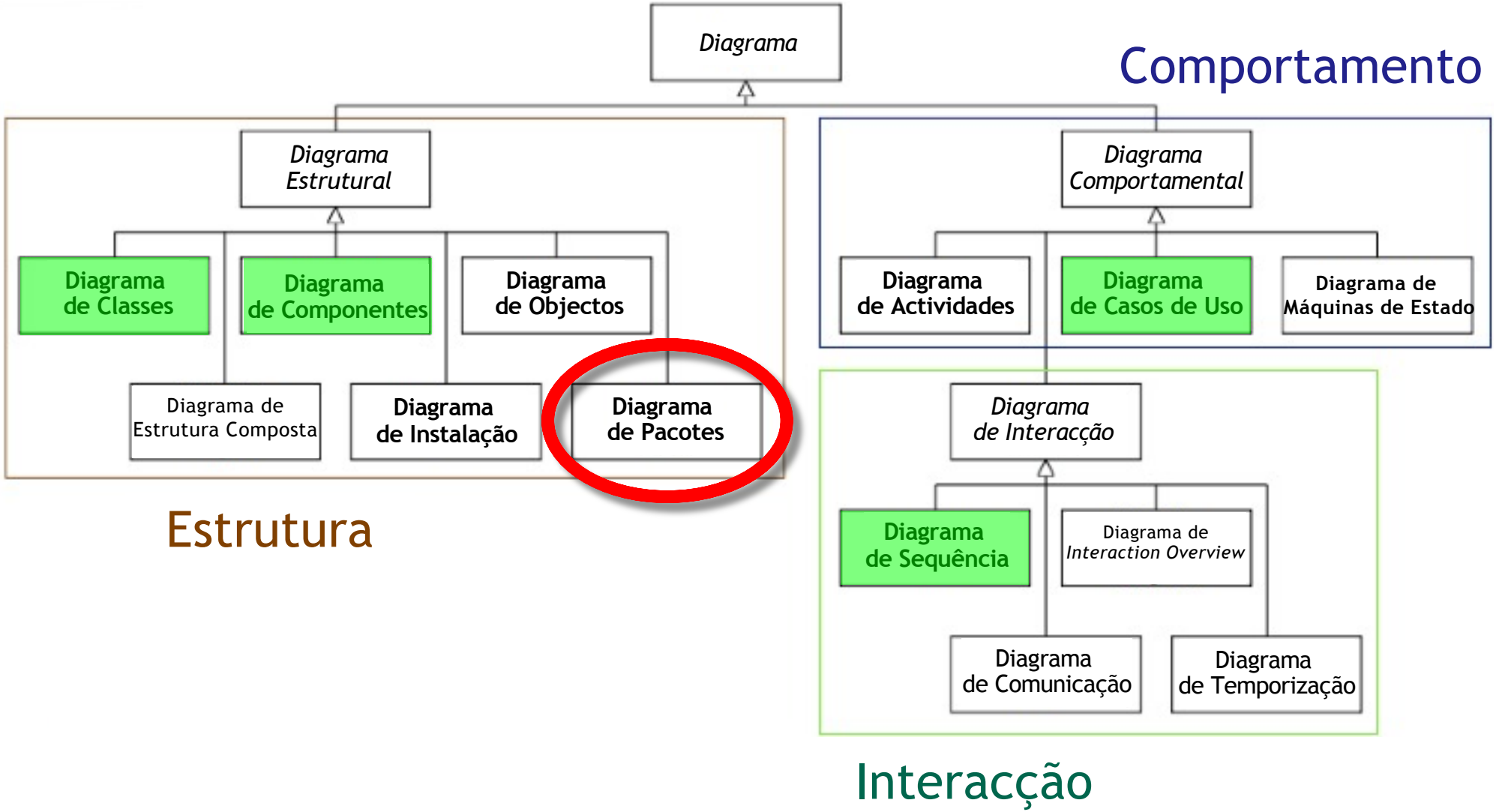
Desenvolvimento de Sistemas Software
José Creissac Campos / António Nestor Ribeiro

Onde estamos…



(Kruchten, 1995)

Diagramas da UML 2.x



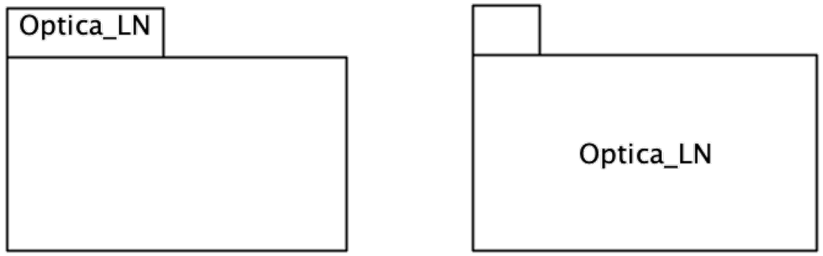
Diagramas de *Package*

Estamos a procurar trabalhar por antecipação e organizar as classes desde o início!...

- À medida que os sistemas software se tornam mais complexos:
  - Torna-se difícil efectuar a gestão de um número crescente de classes
  - A identificação de subsistemas permite organizar as classes (e as suas dependências) a nível lógico
  - É importante refletir essa organização no código, agrupando as classes de forma adequada
- Em UML os agrupamentos de classe designam-se por *packages* (pacotes) :
  - Em Java esses agrupamentos são também chamados *packages*
  - Em C++ designam-se por *namespaces*
- A identificação das dependências entre os vários *packages* permite que a equipa de projecto possa descrever informação importante para a evolução do sistema

Diagramas de *Package* (cont.)

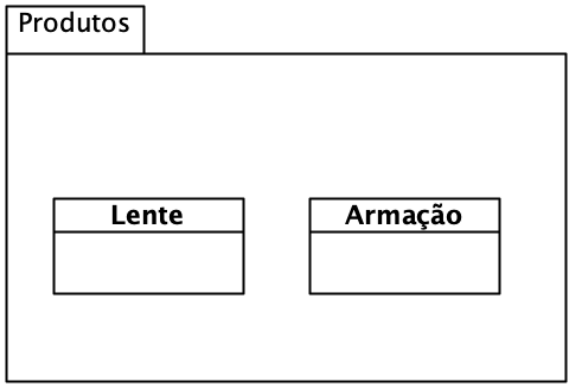
- Representam os *packages* e as relações entre *packages*

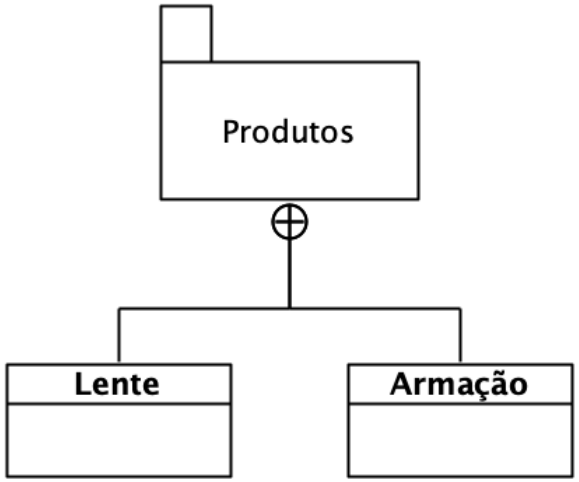


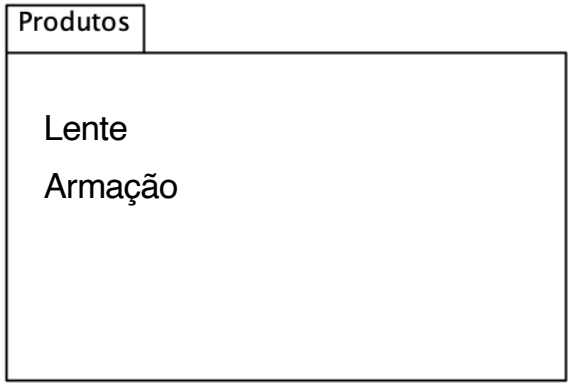
- Representam mais do que agrupamentos de classes:
  - *Packages* de classes - em diagramas de classes
  - *Packages* de componentes – em diagramas de componentes
  - *Packages* de nós – em diagramas de distribuição
  - *Packages* de casos de uso – em diagramas de *use cases*
- Um *package* é, assim, o dono de um conjunto de entidades, que vão desde outros *packages*, a classes, interfaces, componentes, *use cases*, etc.

Diagramas de *Package* (cont.)

- O conteúdo de um *package* pode ser representado de diversas formas:





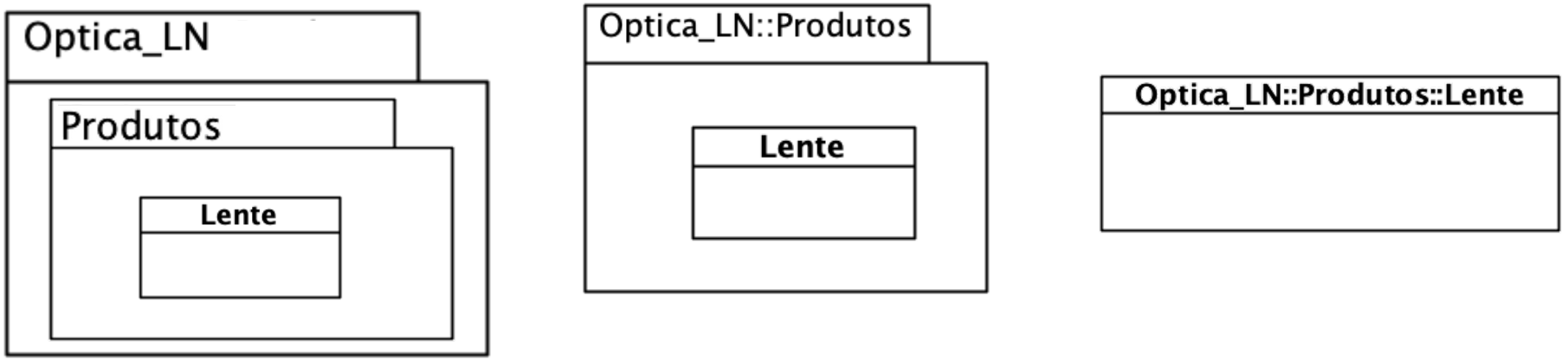



```
package Produtos;
class Lente {
      ...
}
```

```
package Produtos;
class Armação {
      ...
}
```

Diagramas de *Package* (cont.)

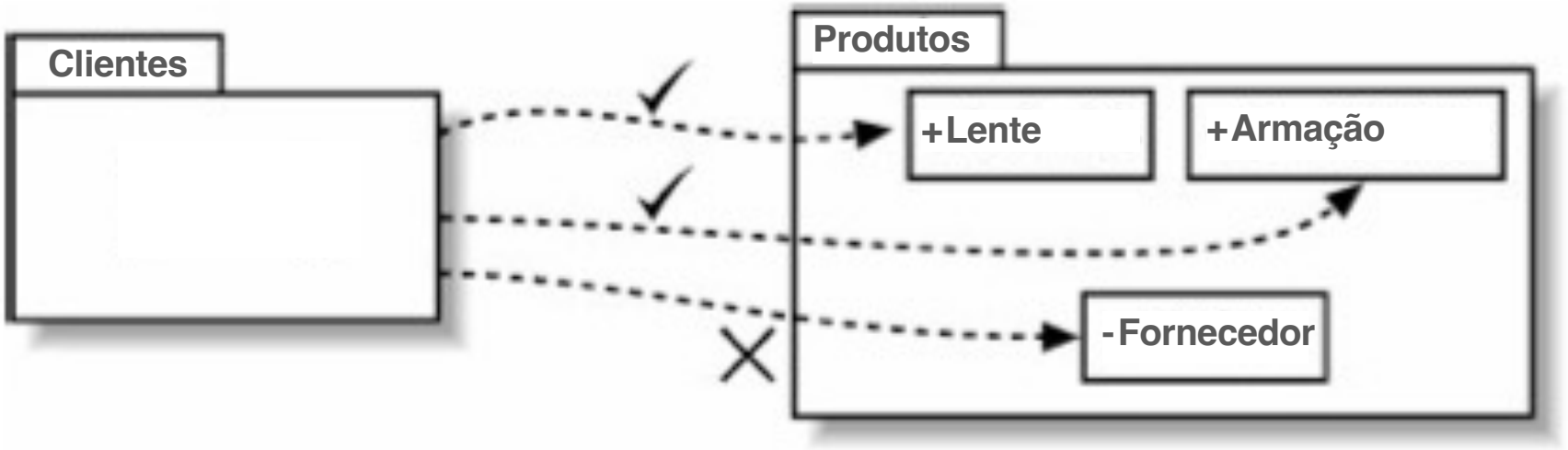
- Qualificação de classes e *packages*:



- A notação **nomePackage::nomeClasse**, identifica (qualifica) inequivocamente uma classe.
  - Tal como em Java com a utilização do nome completo (ex: java.lang.String)
  - Permite que existam classes com nome idêntico nas diversas camadas que constituem uma aplicação

Diagramas de *Package* (cont.)

- A definição da visibilidade dos elementos de um *package* utiliza a mesma notação e semântica dos diagramas de classe. Vamos usar:
  - "+" - público
  - "-" - privado



Diagramas de *Package* (cont.)

- Várias formas de especificar dependências entre pacotes:
  - Dependência (simples) – uma alteração no destino afecta a origem (e.g. slide anterior)
  - <<import>> - o *package* origem importa o contéudo público do *package* destino (conteúdo passa a estar disponível) – cf. fazer *import* de um package em Java

```
import java.util.*;
```

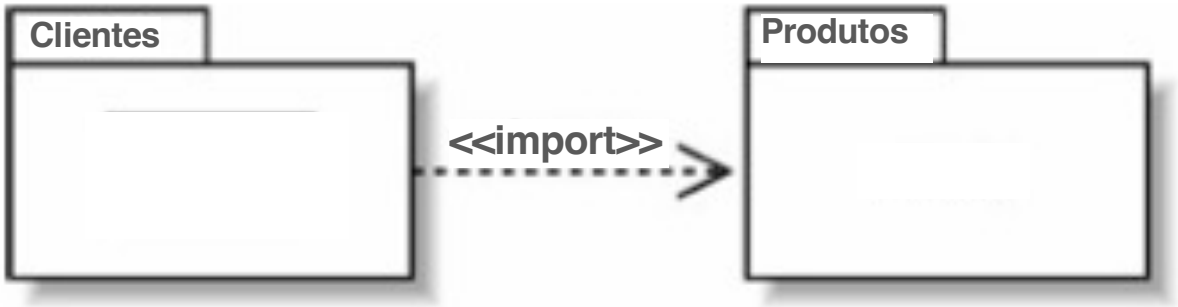
  - <<access>> - o package origem acede a elementos públicos do package destino (mas é necessário qualificar completamente os nomes desses elementos) – cf. utilizar uma classe de um *package* sem o importar:

```
private java.util.HashMap<Produto> produtos;
```

  - <<merge>> - o *package* origem é fundido com o *package* destino para gerar um novo.

Diagramas de *Package* - «import»

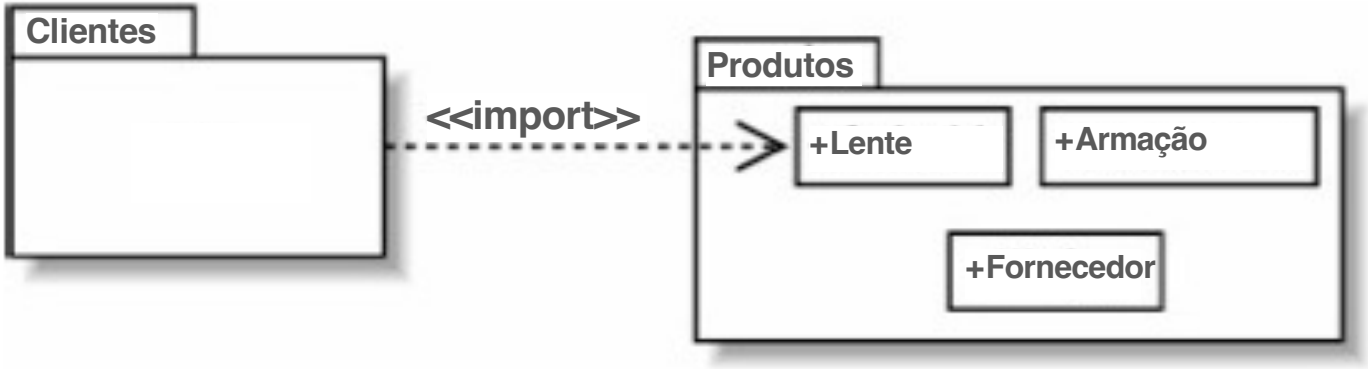
- O package Clientes importa todas as definições públicas de Produtos:




```
import Produtos.*;
```

Definições privadas de packages importados não são acessíveis por quem importa.

- O package Clientes apenas importa a classe Lente do package Produtos:

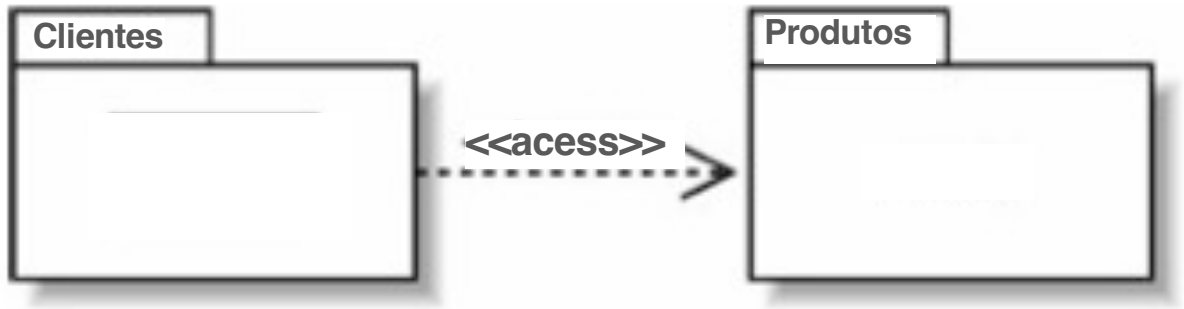



```
import Produtos.Lente;
```

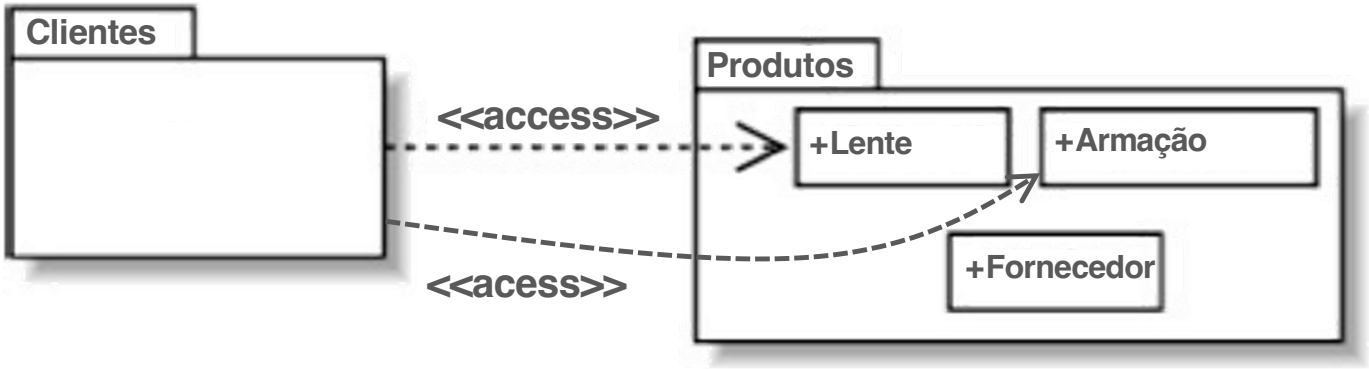
Diagramas de *Package* - «access»

- Classes no package Clientes usam definições públicas de Produtos, sem as importar:

```
private Produtos.Lente lenteEsq, lenteDir;
private Produtos.Armação arm;
```



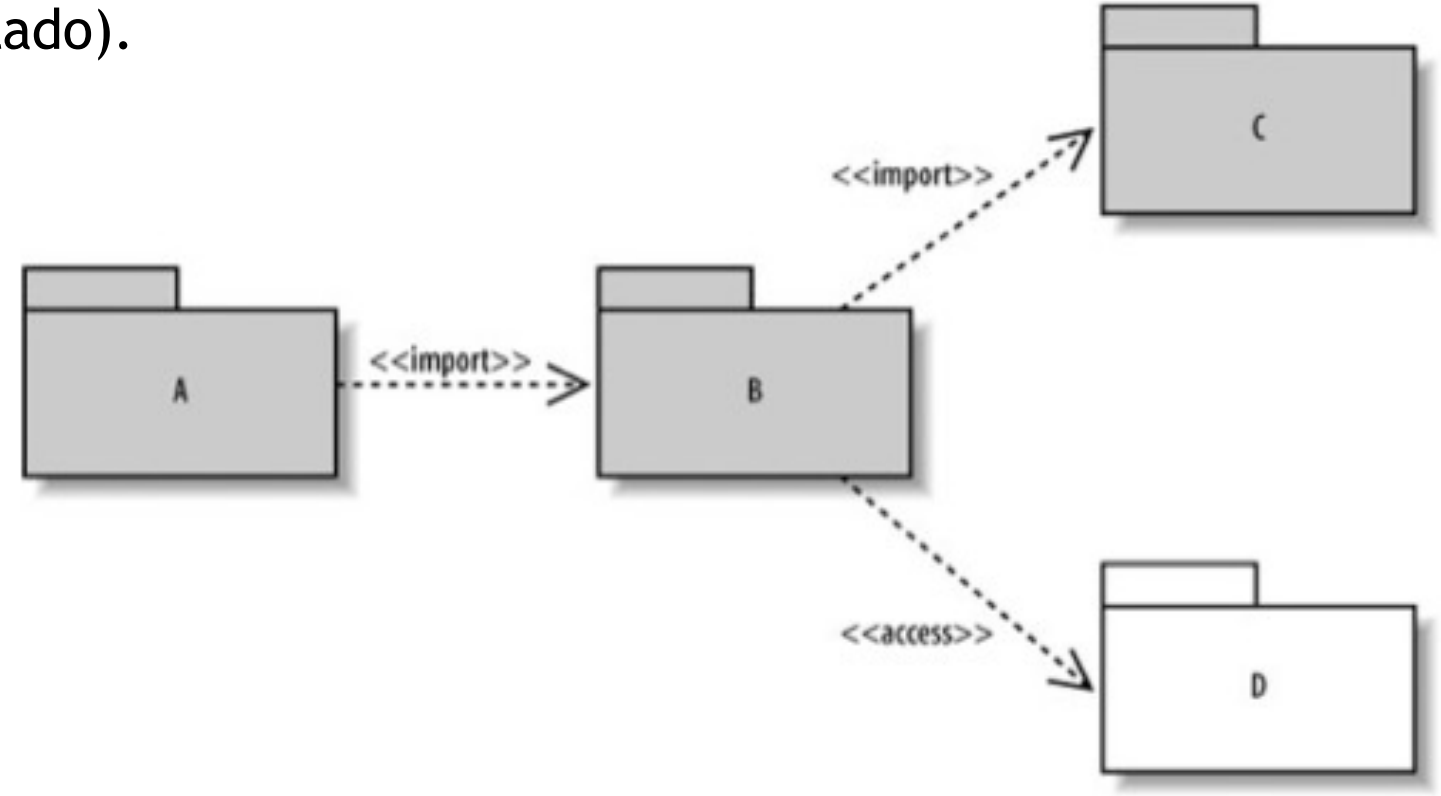
Nesta versão do modelo, optamos por não especificar exatamente o que é acedido.



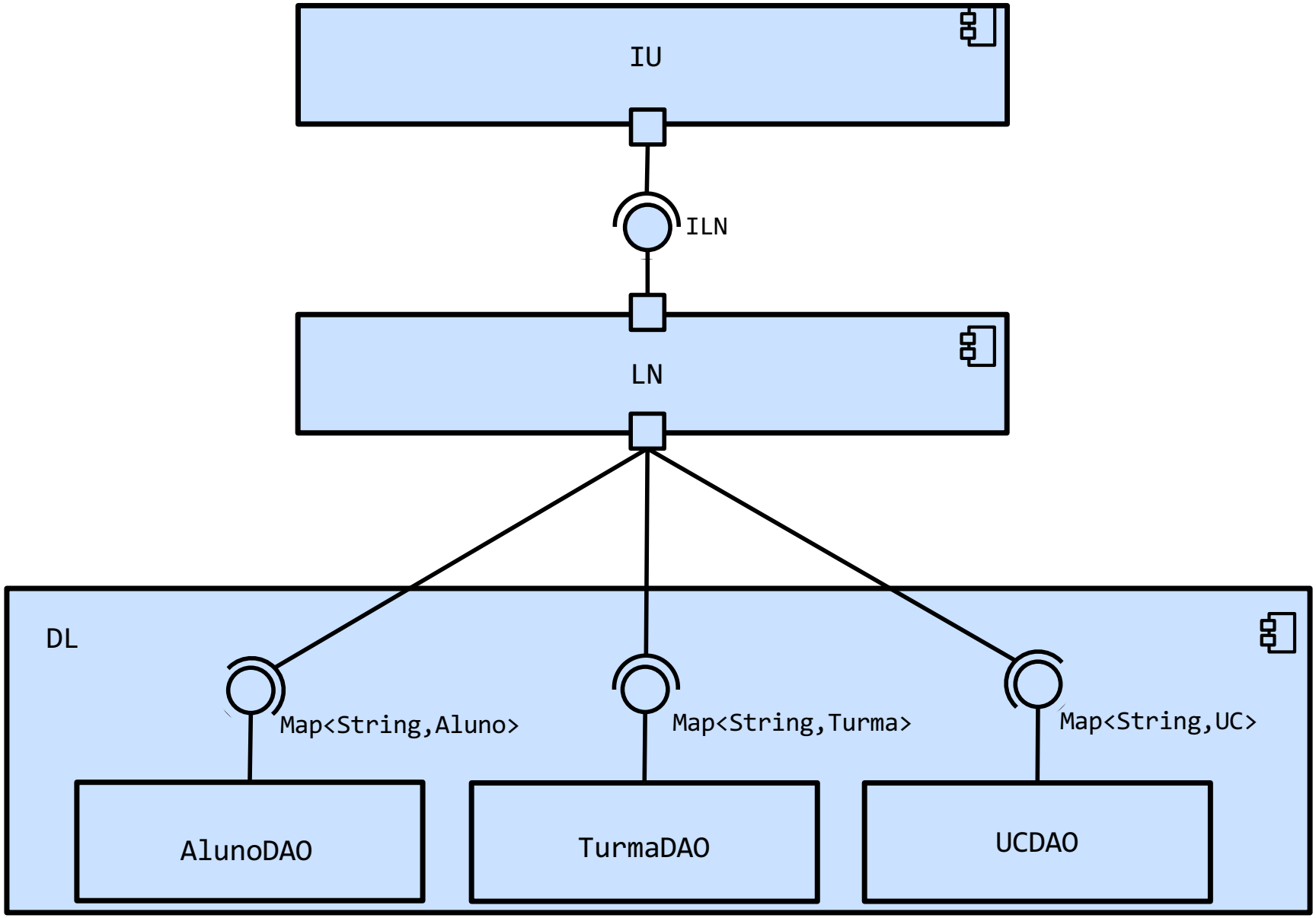
Nesta versão do modelo, deixamos explícito o que é acedido.

Diagramas de *Package* – «import» vs. «access»

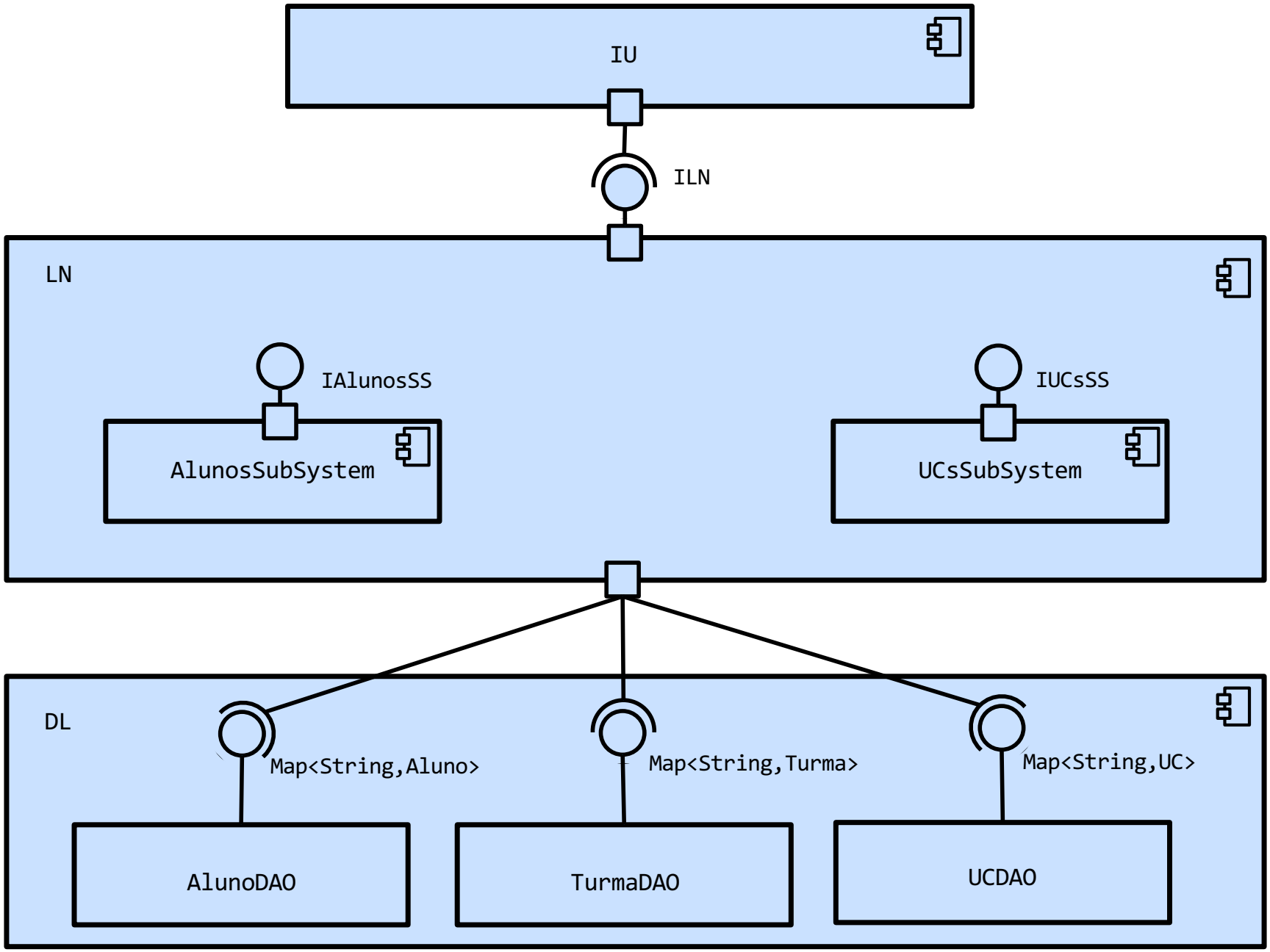
- O package B vê os elementos públicos em C e D.
- A importa B, pelo que vê os elementos públicos em B e em C (porque este é importado por B)
- A não tem acesso a D porque D é apenas acedido por B (não é importado).



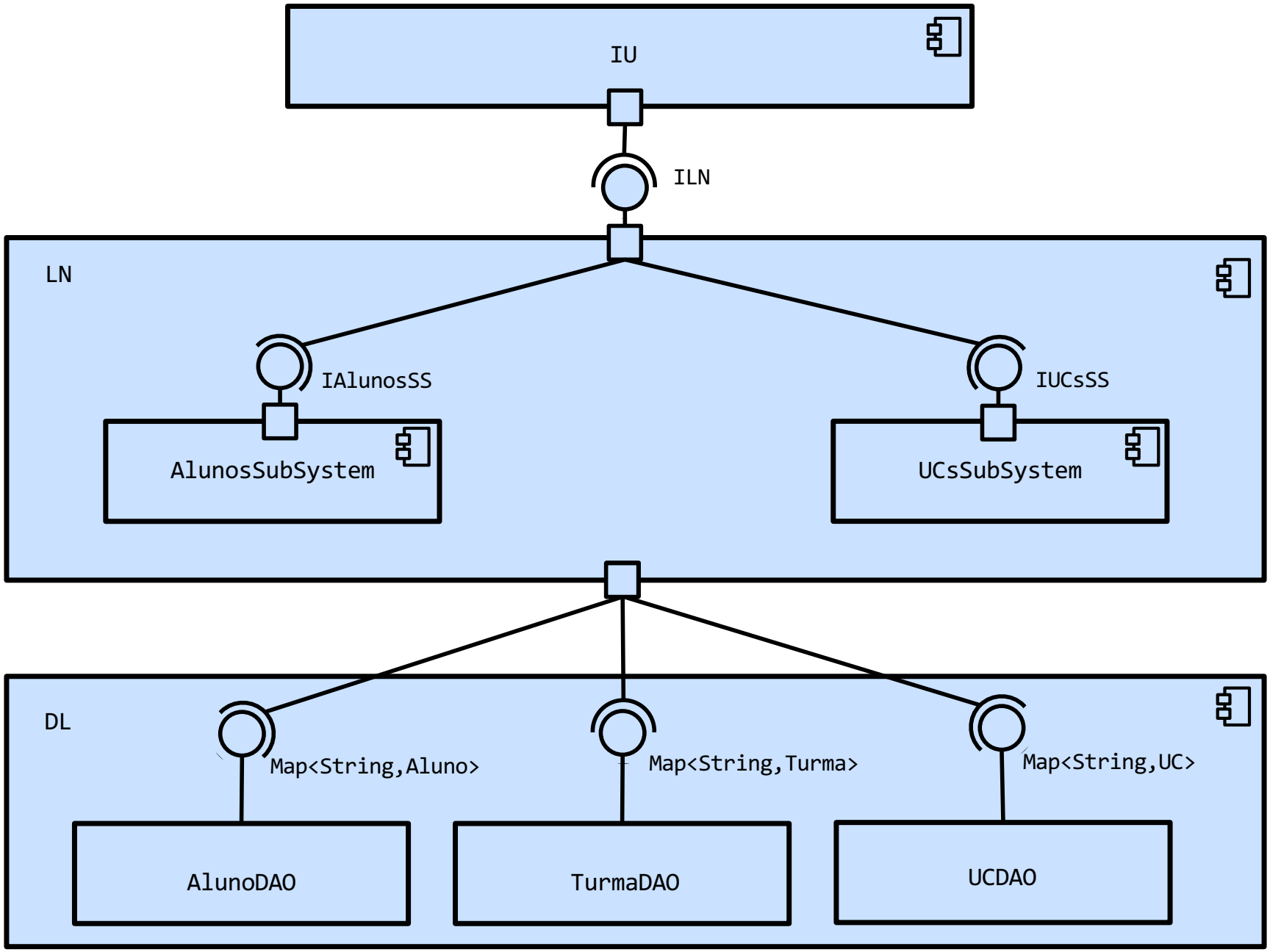
Ponto da situação



Ponto da situação

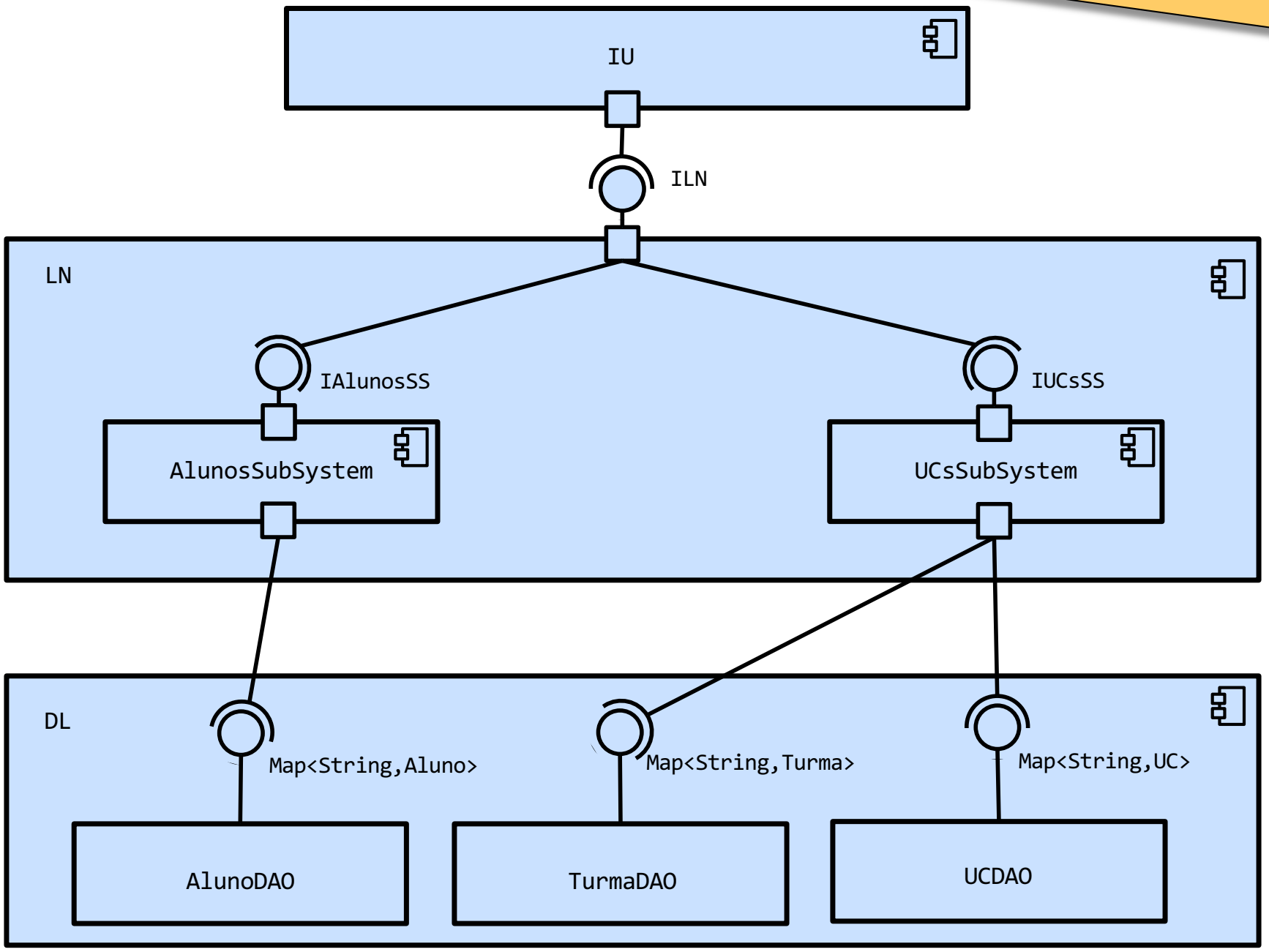


Ponto da situação

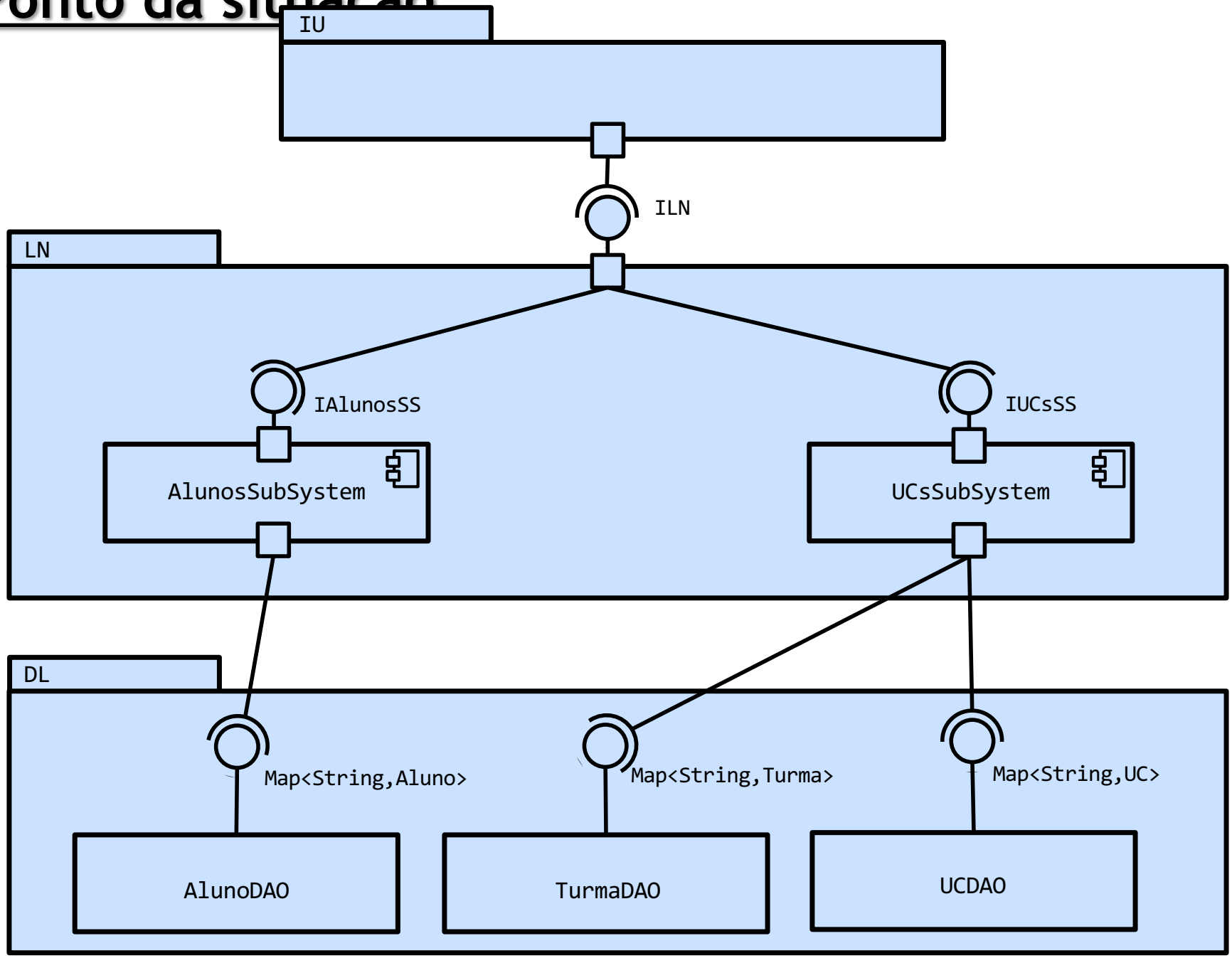


Ponto da situação

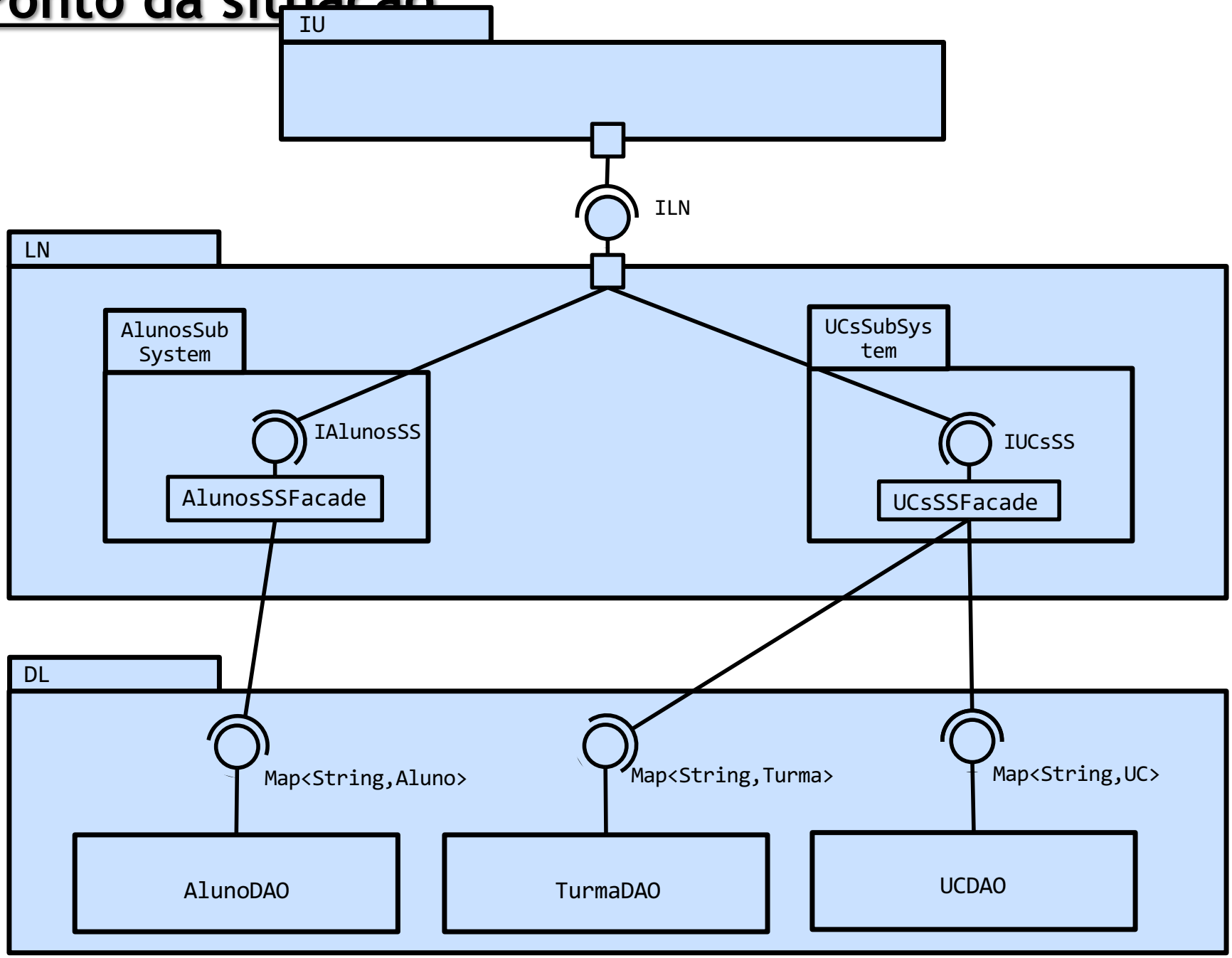
Visão lógica da arquitetura de camadas.



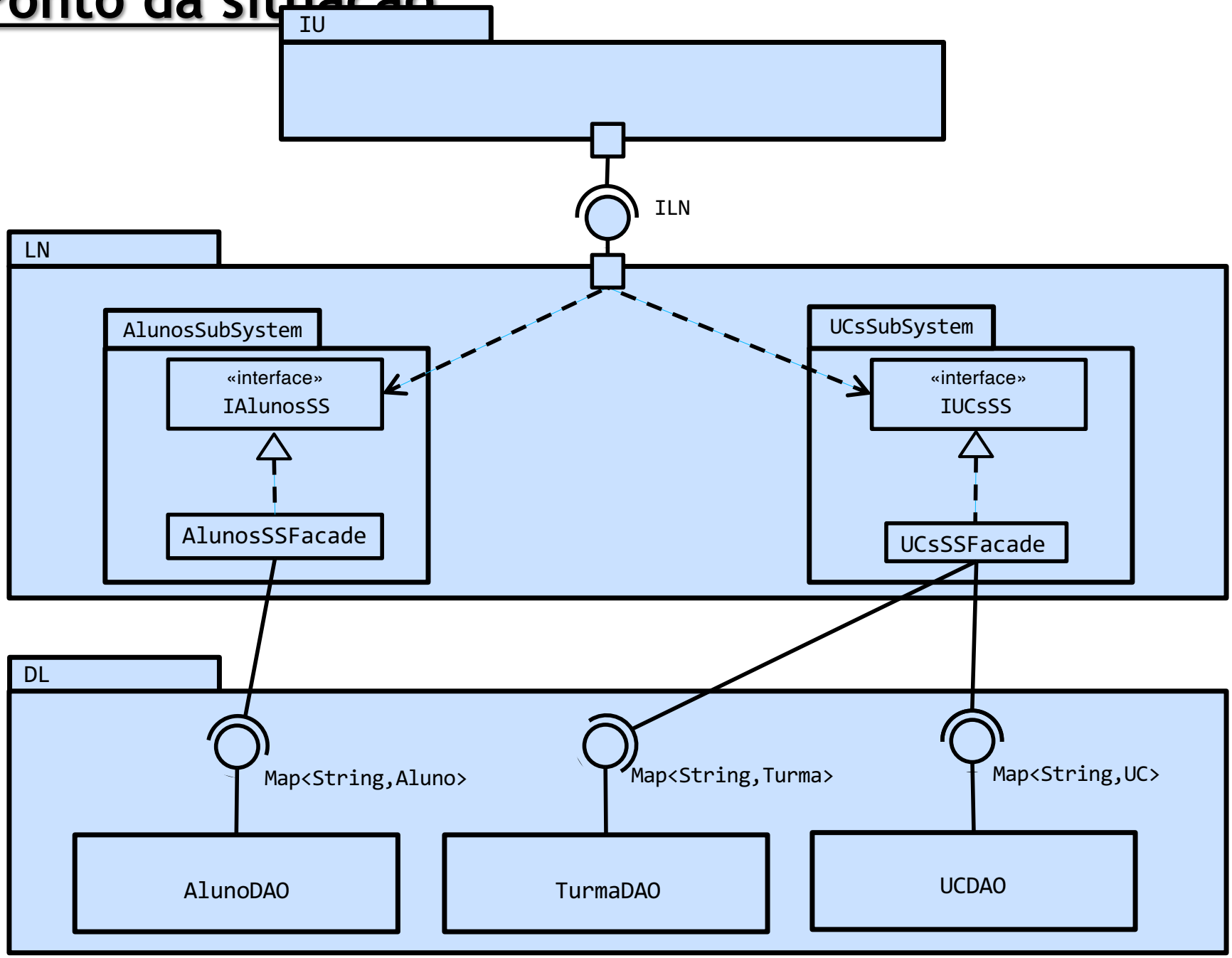
Ponto da situação



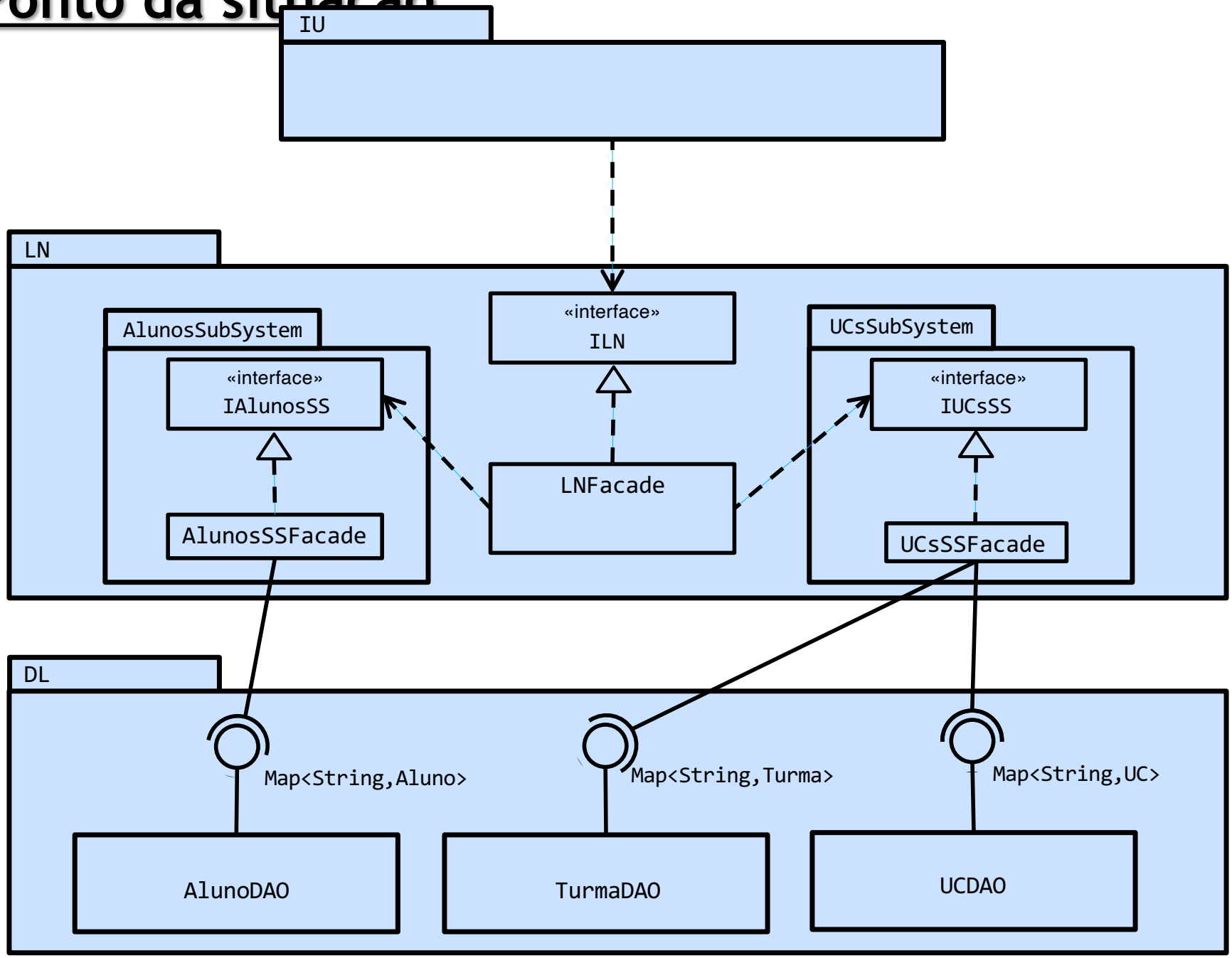
Ponto da situação



Ponto da situação

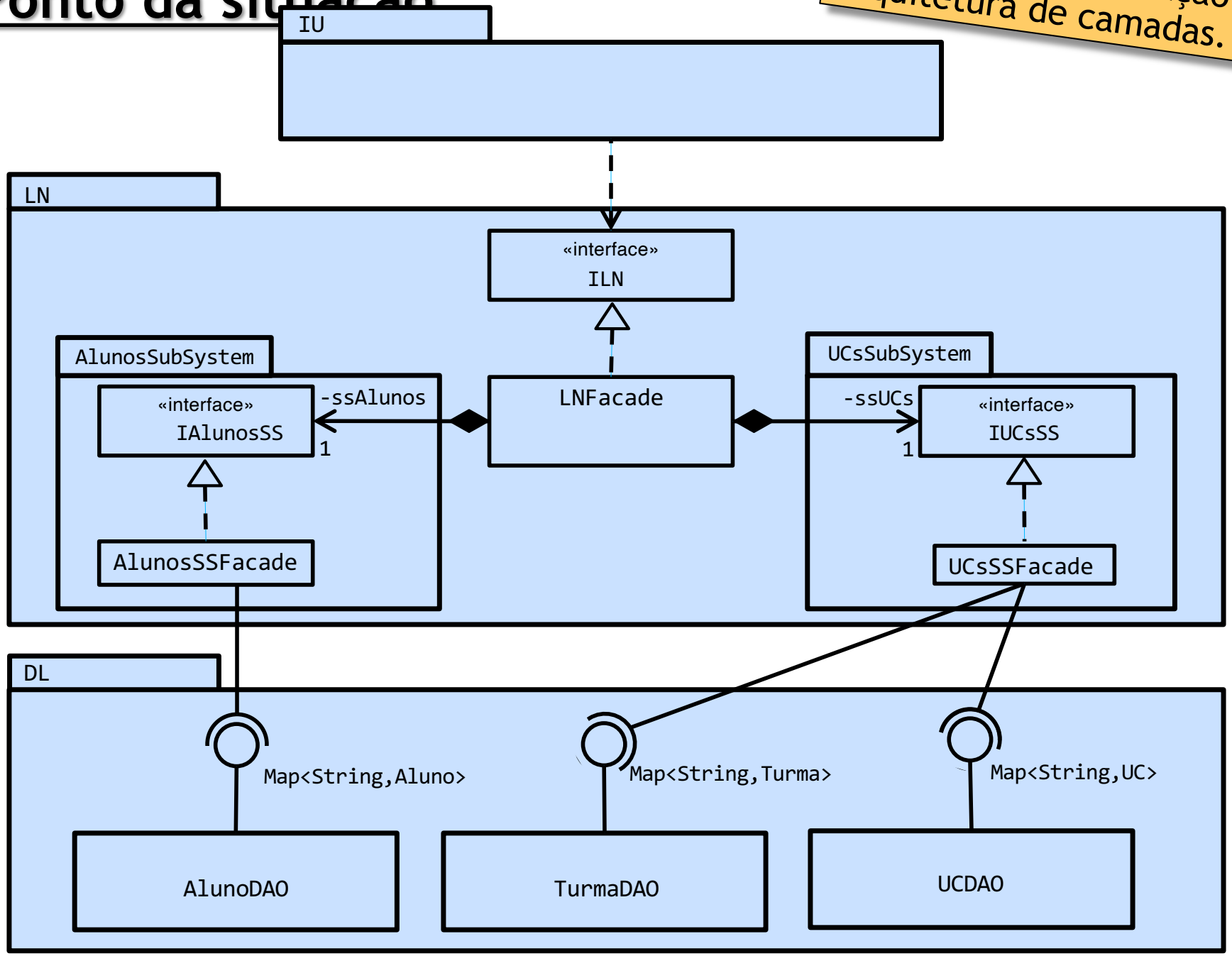


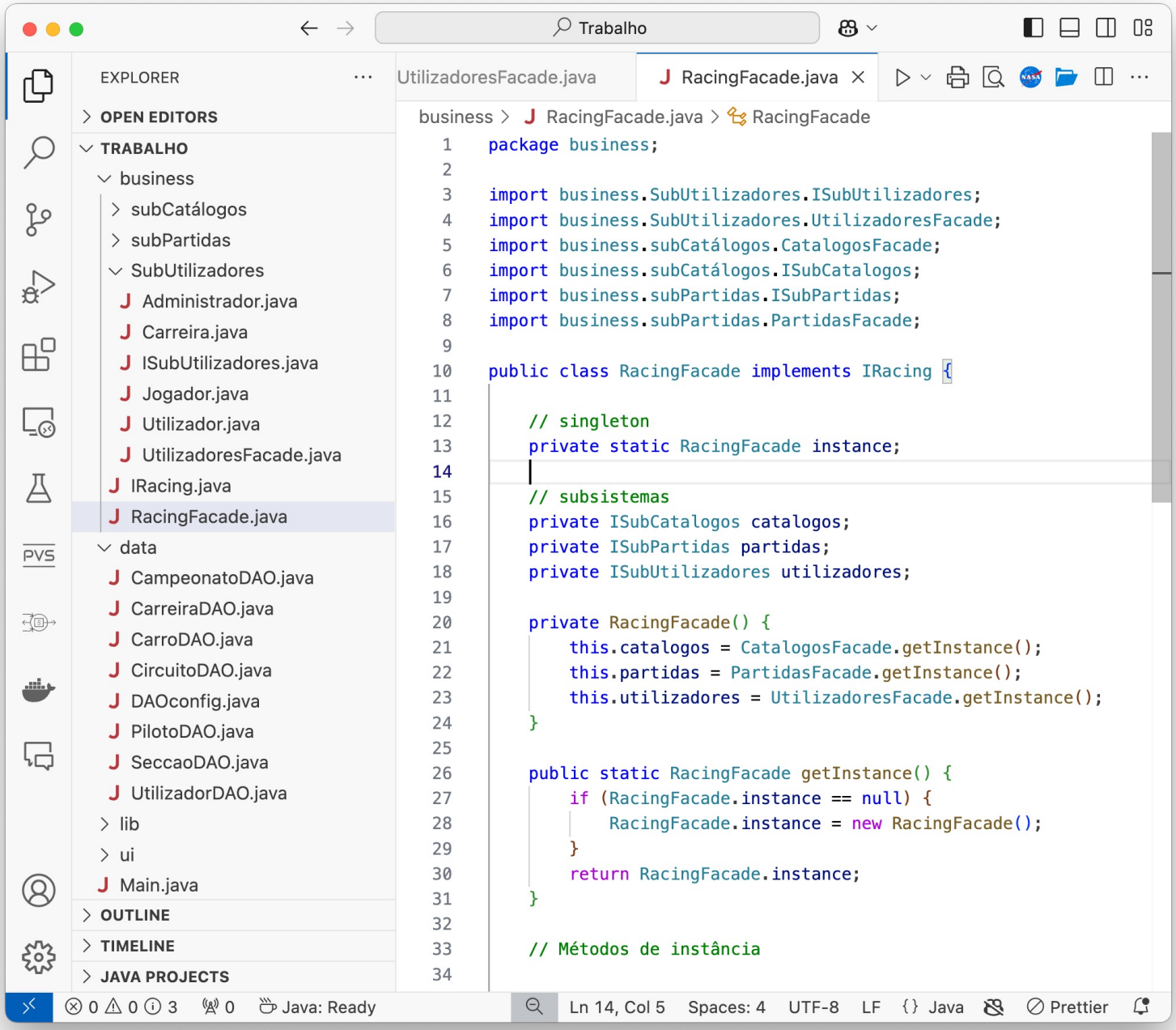
Ponto da situação



Ponto da situação

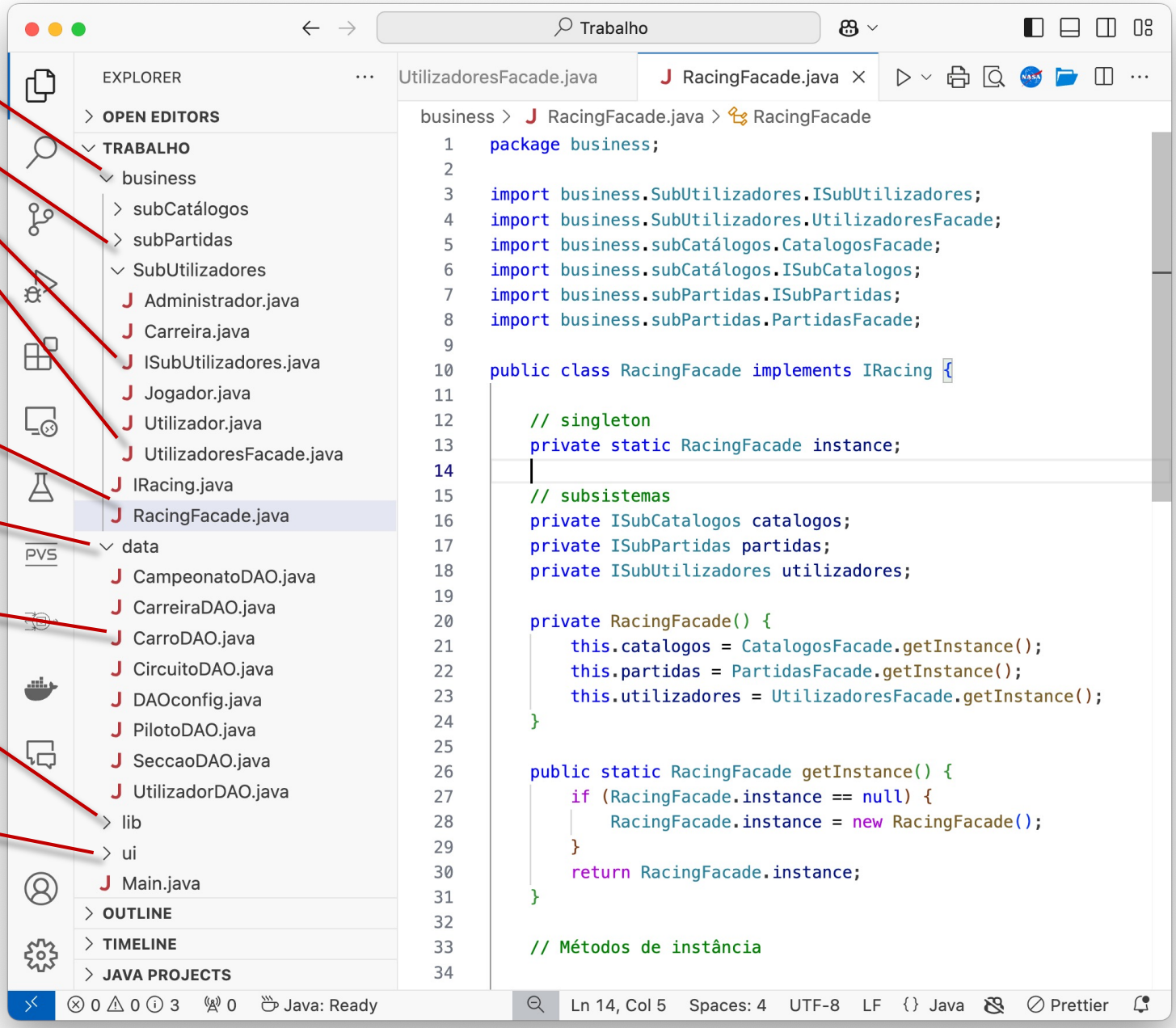
Visão de implementação da arquitetura de camadas.



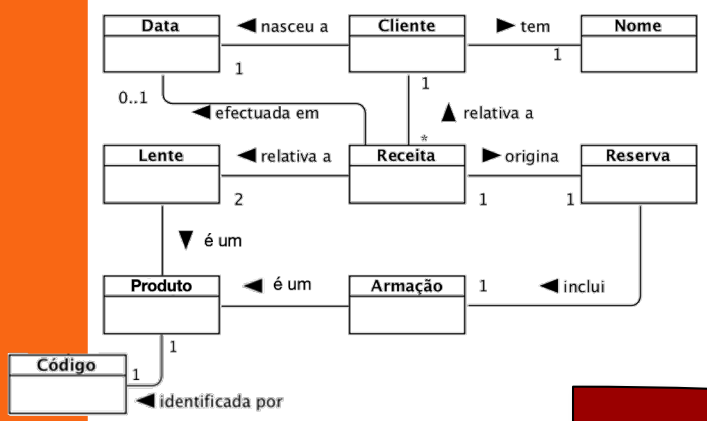

Trabalho
EXPLORER
OPEN EDITORS
TRABALHO
business
subCatálogos
subPartidas
SubUtilizadores
Administrador.java
Carreira.java
ISubUtilizadores.java
Jogador.java
Utilizador.java
UtilizadoresFacade.java
IRacing.java
RacingFacade.java
data
CampeonatoDAO.java
CarreiraDAO.java
CarroDAO.java
CircuitoDAO.java
DAOconfig.java
PilotoDAO.java
SeccaoDAO.java
UtilizadorDAO.java
lib
ui
Main.java
OUTLINE
TIMELINE
JAVA PROJECTS
UtilizadoresFacade.java
RacingFacade.java
business > RacingFacade.java > RacingFacade
1 package business;
2
3 import business.SubUtilizadores.ISubUtilizadores;
4 import business.SubUtilizadores.UtilizadoresFacade;
5 import business.subCatálogos.CatalogosFacade;
6 import business.subCatálogos.ISubCatalogos;
7 import business.subPartidas.ISubPartidas;
8 import business.subPartidas.PartidasFacade;
9
10 public class RacingFacade implements IRacing {
11
12 // singleton
13 private static RacingFacade instance;
14
15 // subsistemas
16 private ISubCatalogos catalogos;
17 private ISubPartidas partidas;
18 private ISubUtilizadores utilizadores;
19
20 private RacingFacade() {
21 this.catalogos = CatalogosFacade.getInstance();
22 this.partidas = PartidasFacade.getInstance();
23 this.utilizadores = UtilizadoresFacade.getInstance();
24 }
25
26 public static RacingFacade getInstance() {
27 if (RacingFacade.instance == null) {
28 RacingFacade.instance = new RacingFacade();
29 }
30 return RacingFacade.instance;
31 }
32
33 // Métodos de instância
34
⊗ 0 ⚠ 0 ⓘ 3 0 Java: Ready
Ln 14, Col 5 Spaces: 4 UTF-8 LF { } Java Prettier

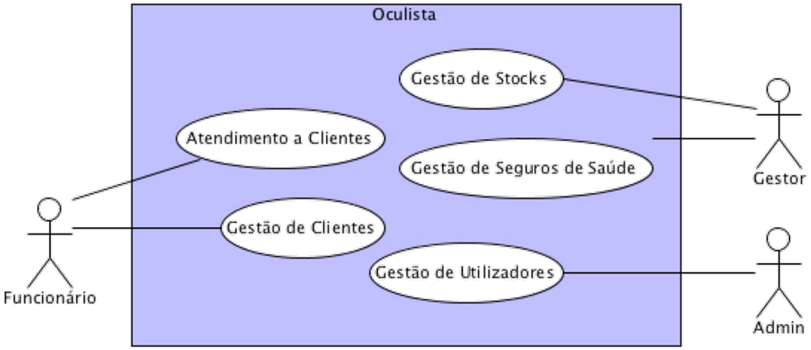
Estrutura típica do código...

- Camada de negócio
- Subsistemas
- API do subsistema Utilizadores
- Façade do subsistema Utilizadores
- API e Façade da lógica de negócio
- Camada de dados
- DAOs
- Bibliotecas auxiliares
- Camada de interface com o utilizador



Resumindo o exemplo...



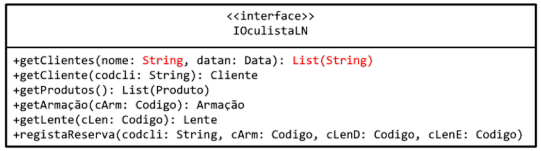


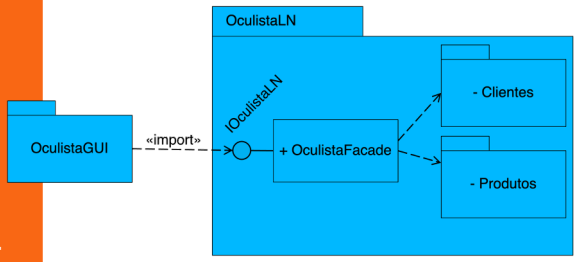
Use Case: Reservar armação e lentes

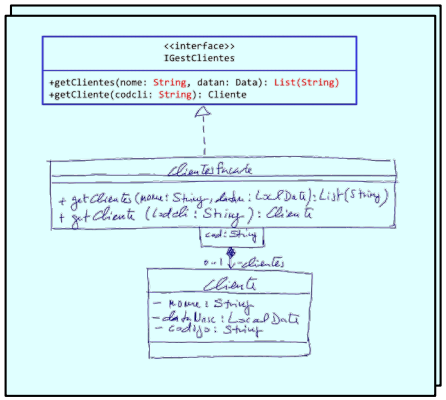
Descrição: Funcionário regista uma reserva de armação e lentes
Pós-condição: Reserva fica registada
Fluxo normal:
1. Funcionário indica nome e/ou data de nascimento do cliente
2. Sistema procura clientes
3. Sistema apresenta lista de clientes
4. Funcionário selecciona cliente
5. Sistema procura cliente
6. Sistema apresenta detalhes do cliente
7. Funcionário confirma cliente
8. Sistema procura produtos e apresenta lista
9. Funcionário indica Código de armação e lentes
10. Sistema procura detalhes dos produtos
11. Sistema apresenta detalhes dos produtos
12. Funcionário confirma produtos
13. Sistema regista reserva dos produtos
14. <<include>> imprimir talão

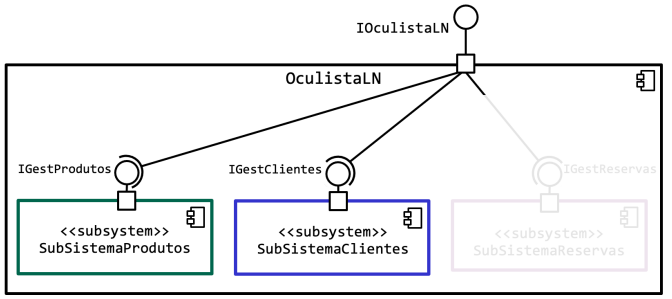
Fluxo alternativo: [lista de clients tem tamanho 1] (passo 3)
3.1. Sistema apresenta detalhes do único cliente da lista
3.2. regressa a 7

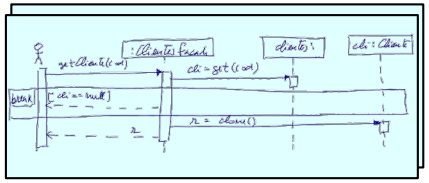
Fluxo de excepção: [cliente não quer produto] (passo 12)
11.1. Funcionário rejeita produtos
11.2. Sistema termina processo











Diagramas da UML 2.x