

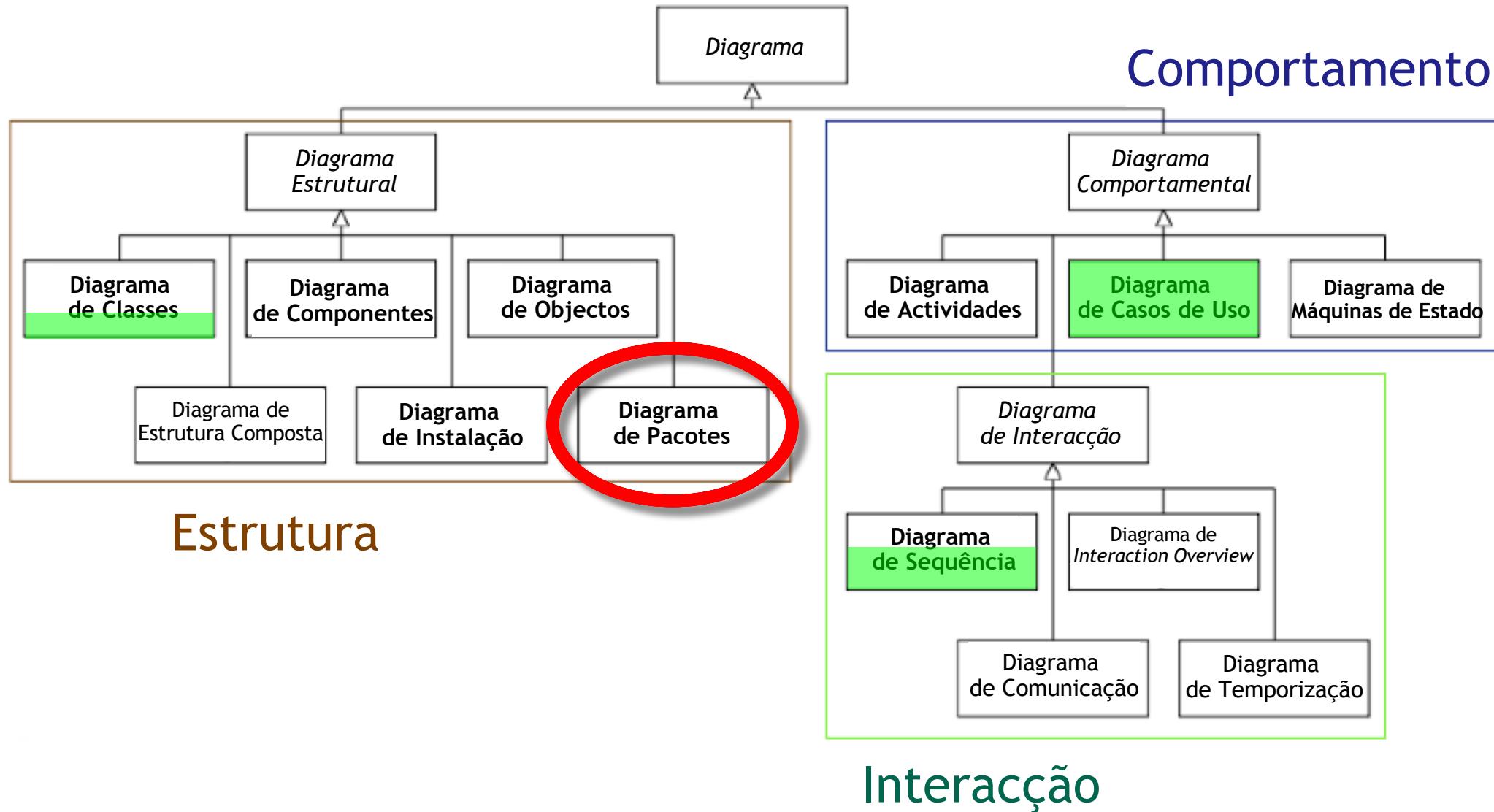


Desenvolvimento de Sistemas Software

Aula Teórica 12

Modelação Estrutural / Diagramas de Package

Diagramas da UML 2.x





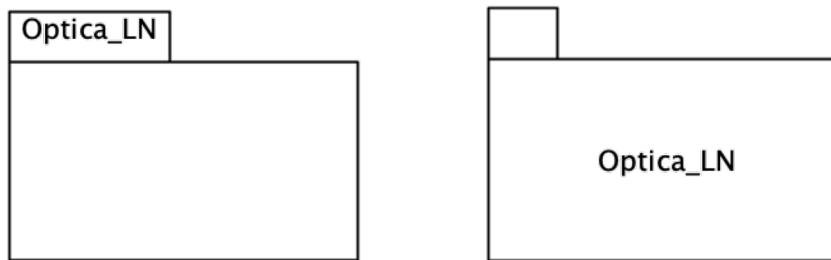
Diagramas de Package

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 uma classe e o seu papel no sistema dependem do contexto em que se encontram
 - É determinante conseguir identificar as dependências entre as diversas classes de um sistema.
- Em UML os agrupamentos de classe designam-se por packages (pacotes) e correspondem à abstracção de conceitos existentes nas linguagens de programação:
 - Em Java esses agrupamentos são os 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.)

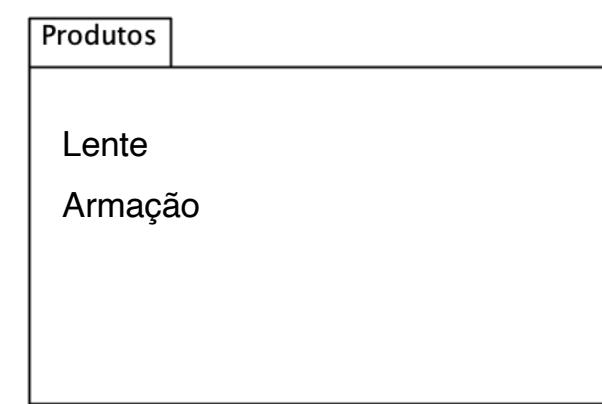
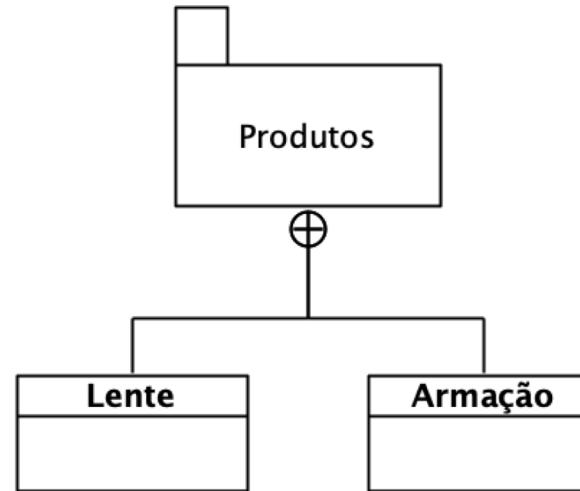
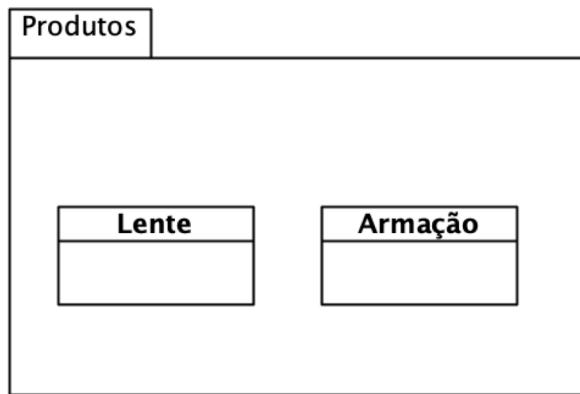
- Um diagrama de package representa os packages e as relações entre packages



- Os diagramas de packages em UML representam mais do que relações entre classes:
 - Packages de classes (packages lógicos) - 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.)

- Essa forma de agregação pode ser representada de diversas formas:



```
package Produtos;

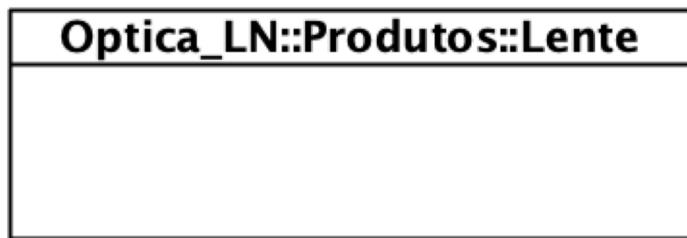
public/private/protected(?) class Lente {

    ...

}
```

Diagramas de Package (cont.)

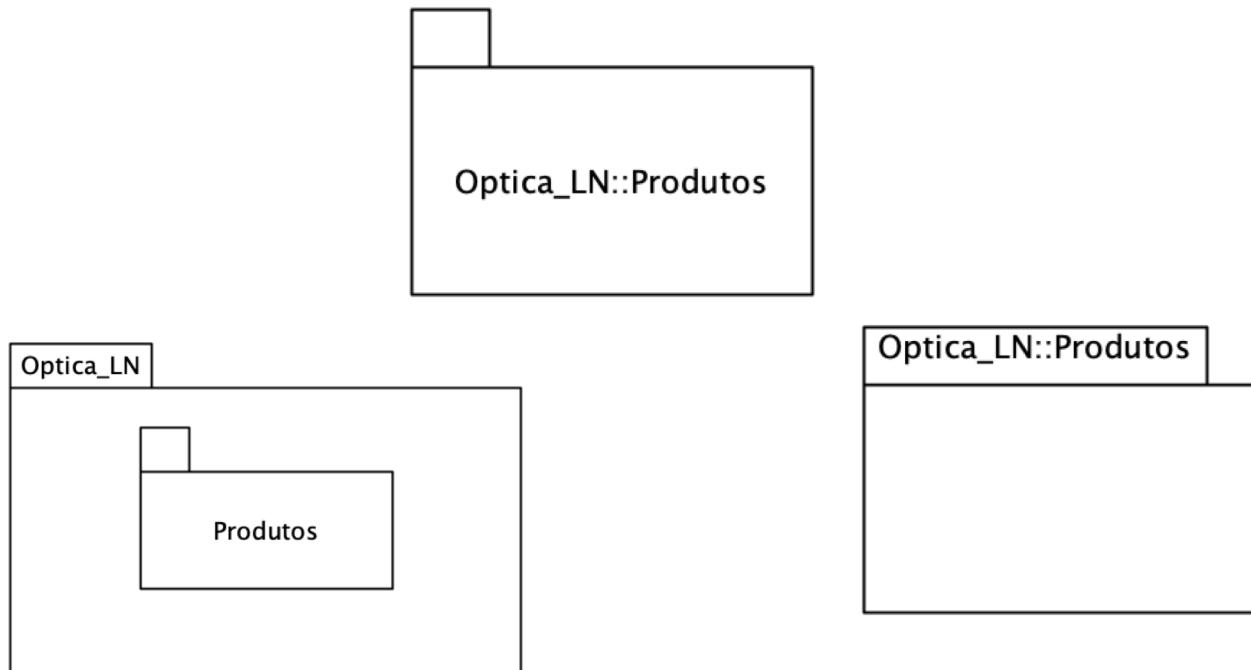
- Por uma questão de identificação do contexto de uma classe (o seu namespace) é usual que as ferramentas identifiquem no diagrama de classes, qual é o agrupamento lógico a que uma classe pertence.



- 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.)

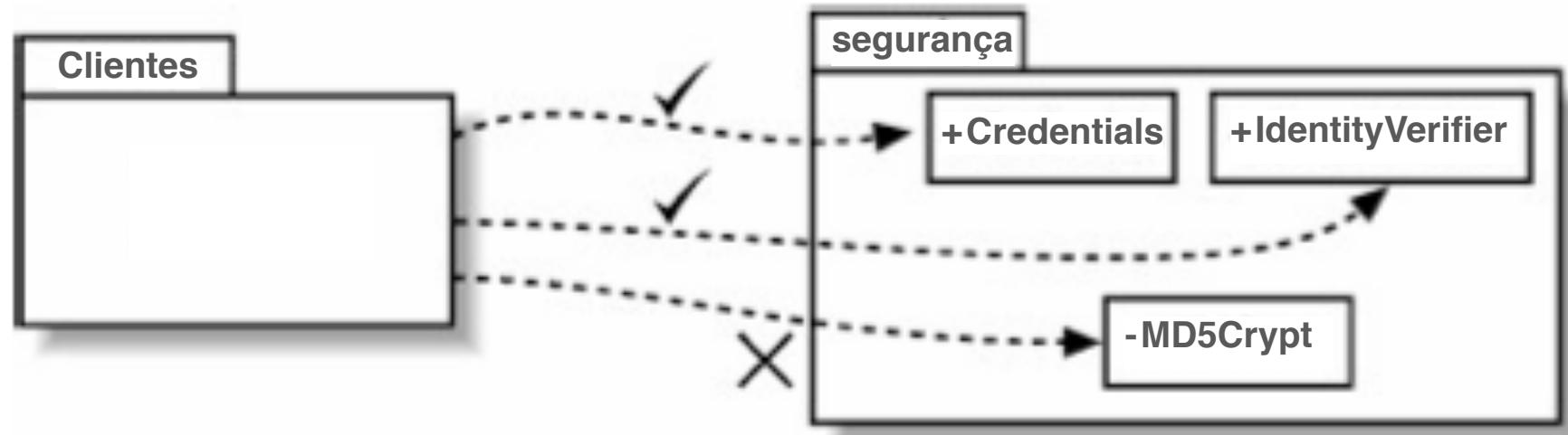
- Existem várias formas de representar a agregação de packages
- Essas regras definem também a qualificação dos nomes das classes



- As três representações correspondem à mesma informação

Diagramas de Package (cont.)

- A definição da visibilidade dos elementos de um package utiliza a mesma notação e semântica que veremos nos diagramas de classe:
 - “+” - público
 - “-” - privado
 - “#” - protected (só acessível/visível por filhos do package em causa)



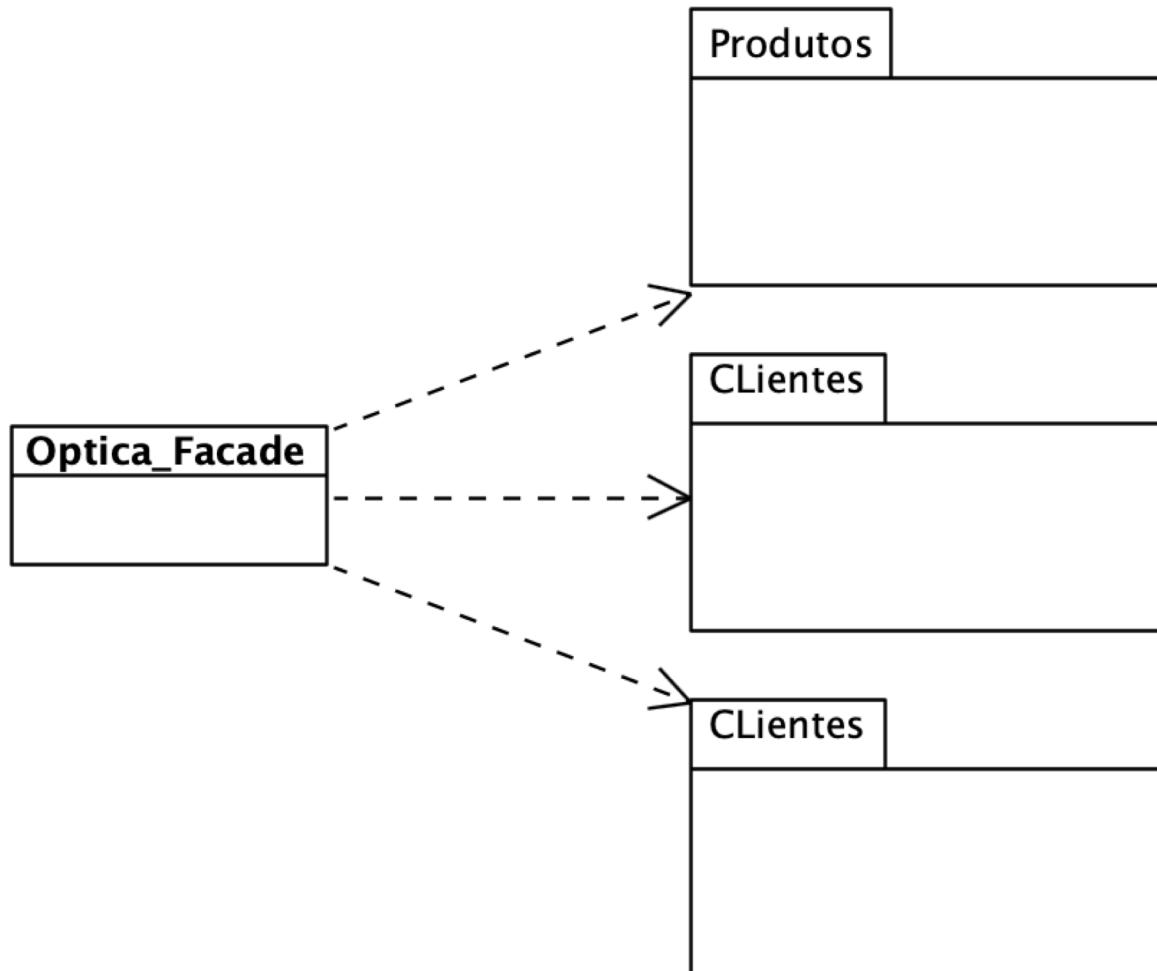
Diagramas de Package (cont.)

- Existem várias formas de especificar dependências entre pacotes:
 - Dependência (simples) - quando uma alteração no package de destino afecta o package de origem
 - <<import>> - o package origem importa o conteúdo público do package destino. Logo, não necessita de qualificar completamente os elementos importados (na forma packageA::classeB).
 - Mecanismo similar ao que permite fazer import de um package em Java

```
import java.util.*;
```
 - <<access>> - o package origem acede a elementos exportados pelo package destino, mas necessita de qualificar completamente os nomes desses elementos.
 - <<merge>> - o package origem é fundido com o package destino para gerar um novo.

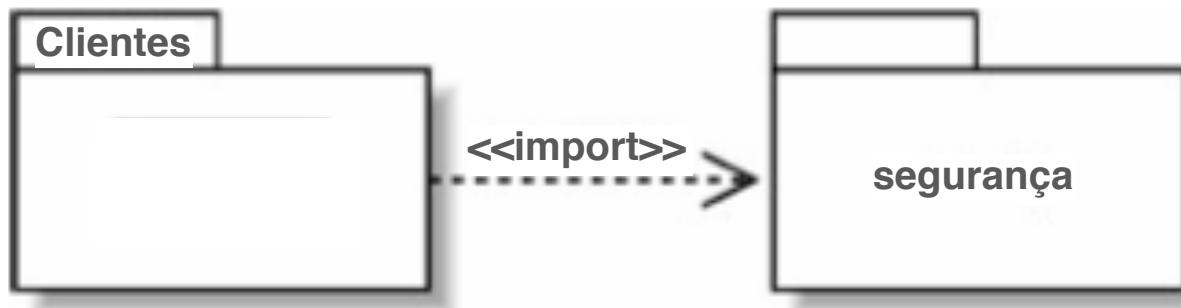
Diagramas de Package (cont.)

- Exemplos de utilização de dependências simples:

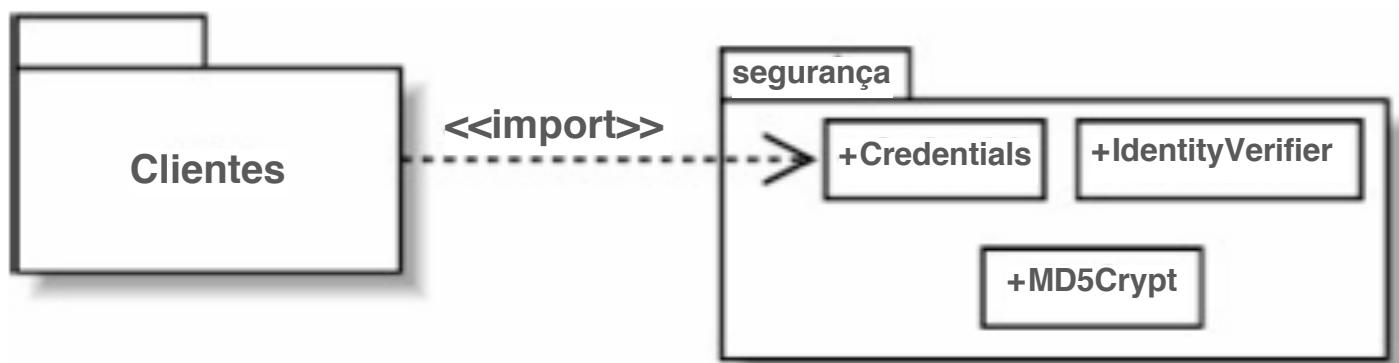


Diagramas de Package (cont.)

- Utilização de `<<import>>`
- O package *Clientes* importa todas as definições públicas de *segurança*

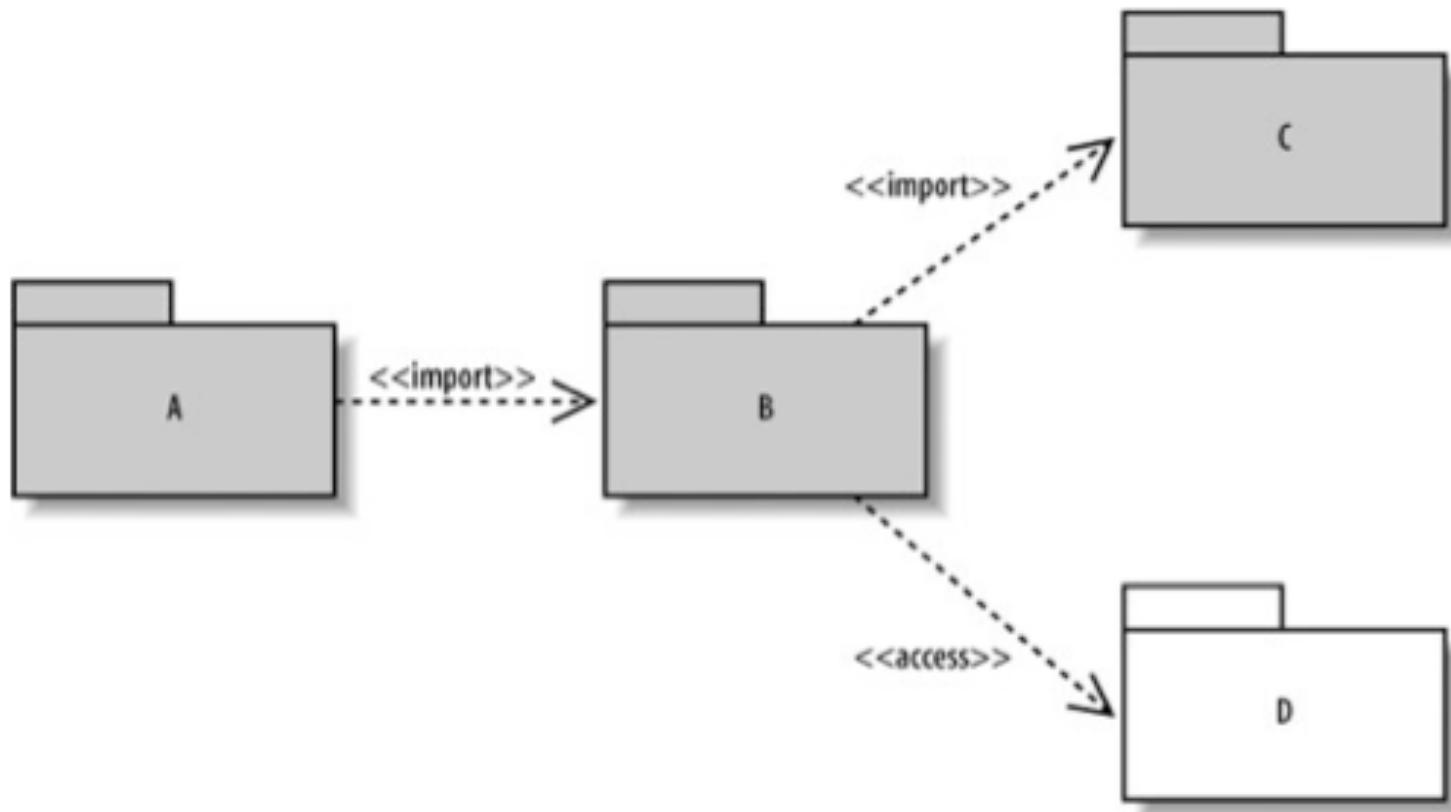


- Definições privadas de packages importados não são acessíveis por quem importa.
- O package *Clientes* apenas importa a classe *Credentials* do package *segurança*

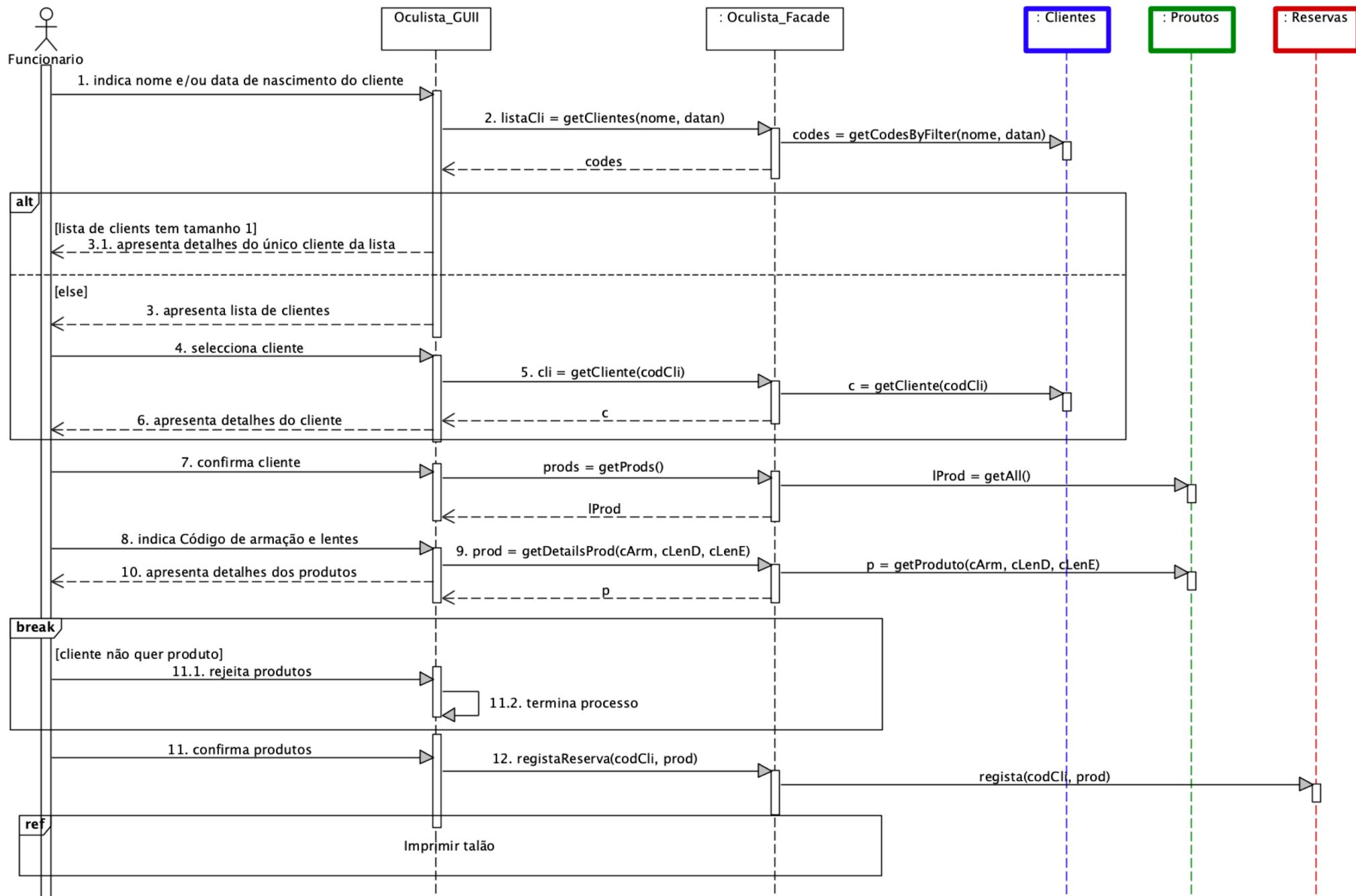


Diagramas de Package (cont.)

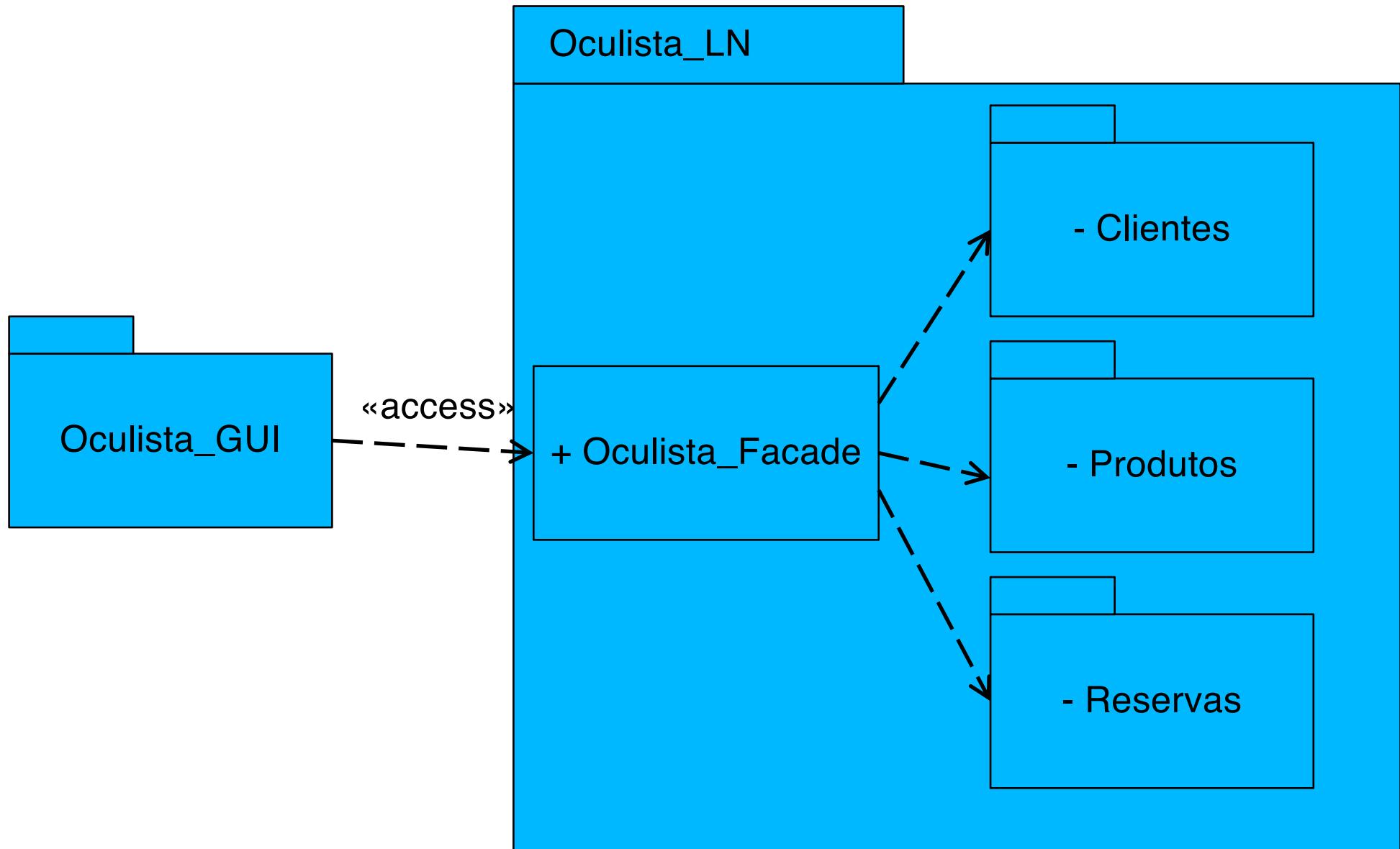
- Utilização de `<<import>>` e `<<access>>`
 - O package *B* vê os elementos públicos em *C* e *D*.
 - A importa *B*, pelo que vê os elementos públicos em *B* e em *C* (porque este é importado por *B*)
 - A não tem acesso a *D* porque *D* é apenas acedido por *B* (não é importado).



Relembrando o exemplo...



Primeira versão da arquitectura da solução...





Resumindo o exemplo...

Use Case: Reservar armação e lentes

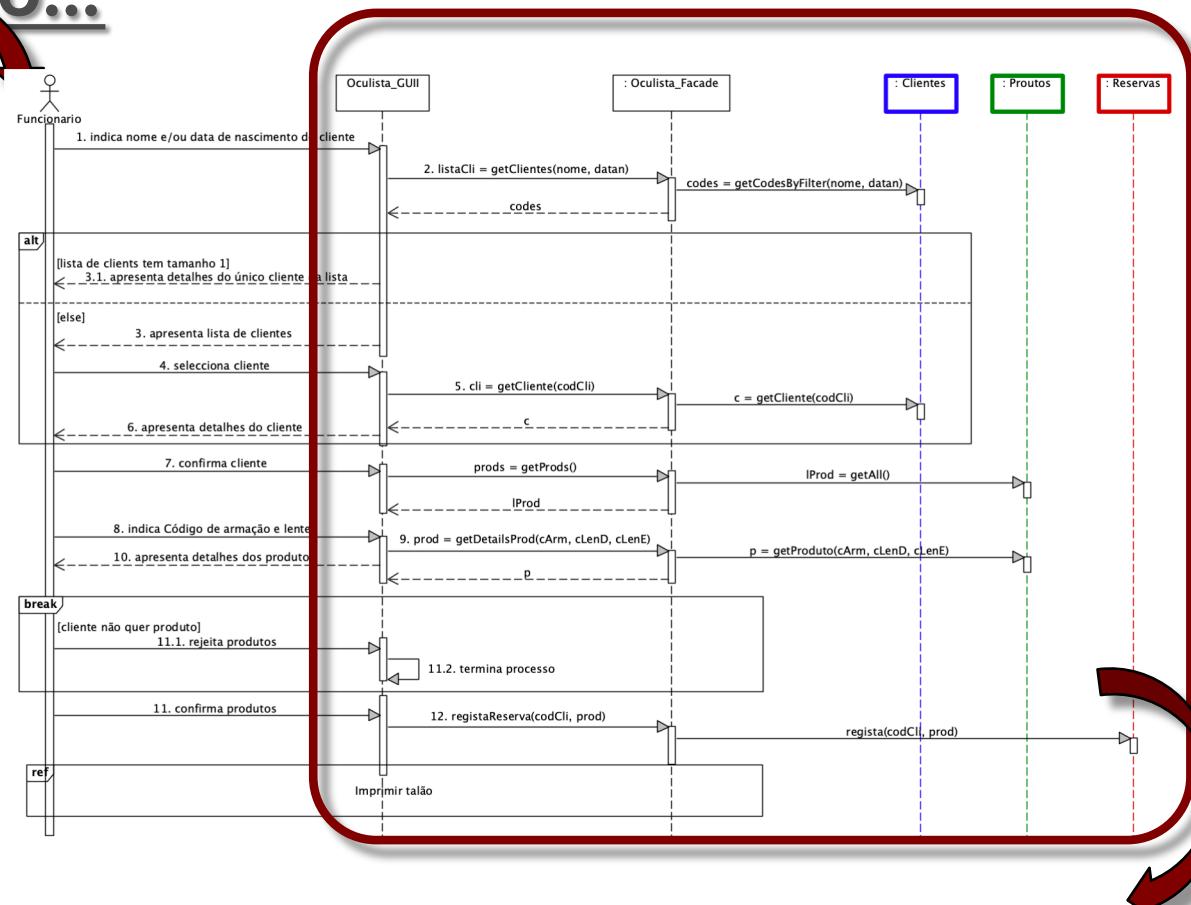
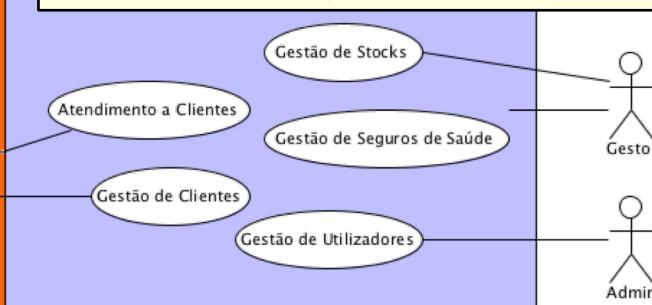
Descrição: Funcionário regista uma reserva de armação e lentes
Pós-condição: Reserva fica registada

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. Funcionário indica Código de armação e lentes
9. Sistema procura produtos
10. Sistema apresenta detalhes dos produtos
11. Funcionário confirma produtos
12. Sistema regista reserva dos produtos
13. <<include>> imprimir talão

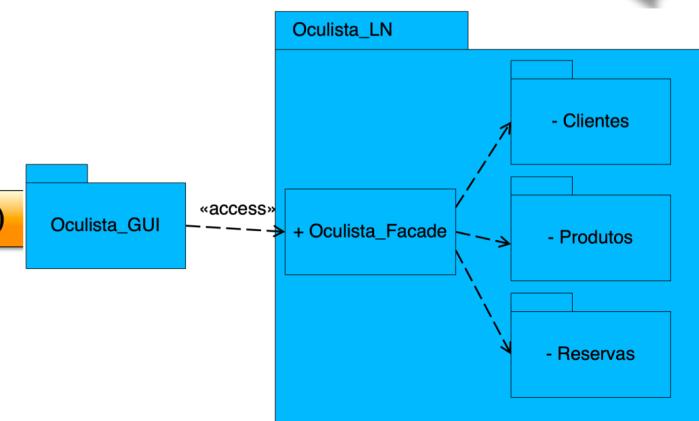
Fluxo alternativo: [lista de clients tem tamanho 1] (passo 3)
 3.1. Sistema apresenta detalhes do único cliente da lista
 3.2. regressa a 7

Fluxo de exceção: [cliente não quer produto] (passo 11)
 11.1. Funcionário rejeita produtos
 11.2. Sistema termina processo

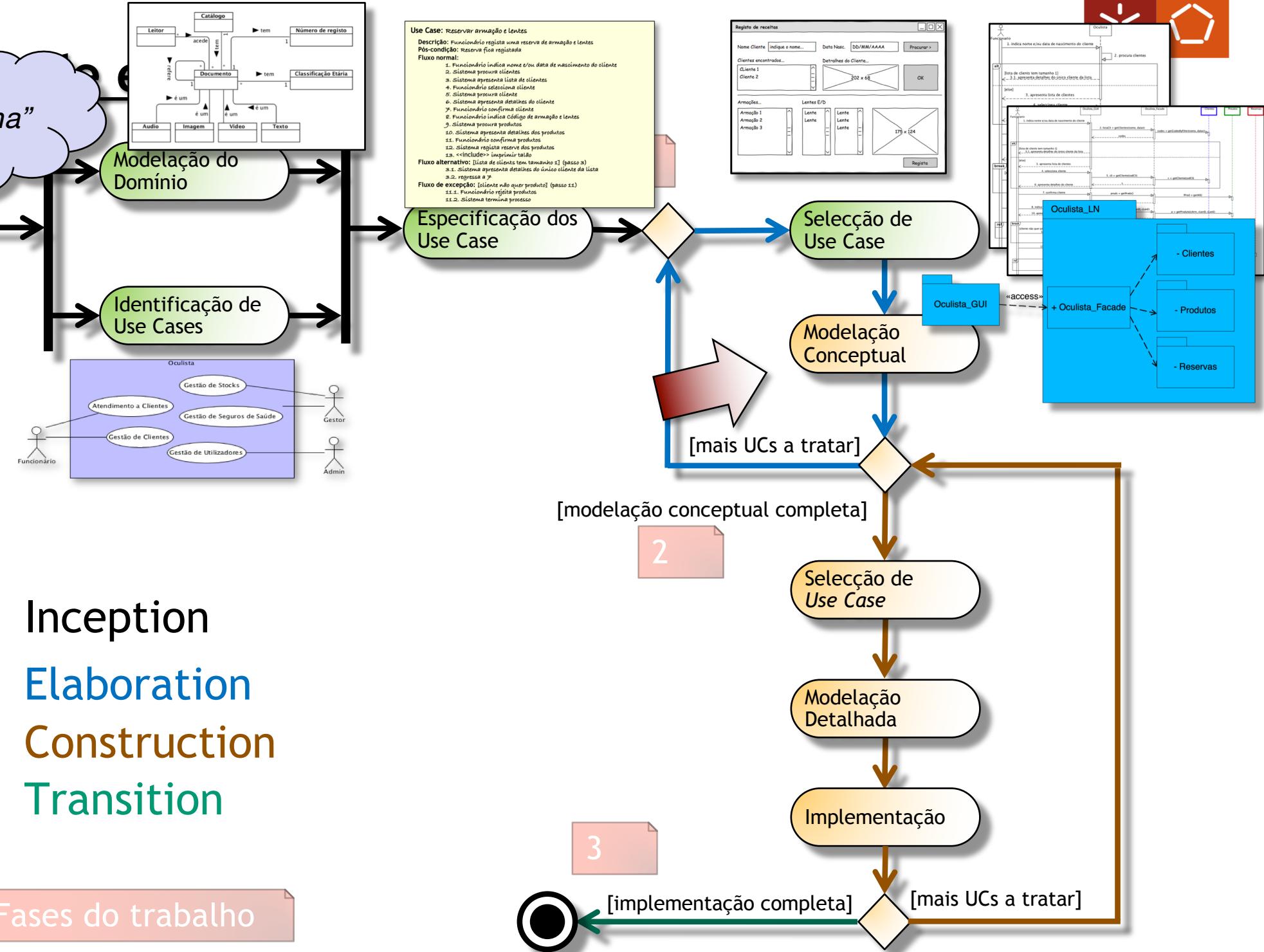


API pública (operações da Camada de Negócio)

- getClientes(nome: ?, datan: ?: ?)
- getCliente(codcli: ?: ?)
- getProds(): ?
- getProduto(cArm: ?, cLenD: ?, cLenE: ?: ?)
- registaReserva(codcli: ?, prod: ?: ?)



"Problema"





Modelação Estrutural

Sumário

- Diagramas de *Package*
 - Representação de *Packages*
 - Relações entre packages
 - Composição: diferentes representações gráficas, qualificação, visibilidade
 - Dependências: simples, «import», «access», «merge»
- Ponto da situação relativamente à abordagem ao desenvolvimento seguida