Fase 2 do trabalho

- Uses cases a implementar
  - Criar pedido (actor: Cliente)
  - Entregar pedido pronto (actor: Funcionário)
  - Consultar indicadores (actor: COO)

Solução **não é** só o código. É necessário seguir o processo proposto!!

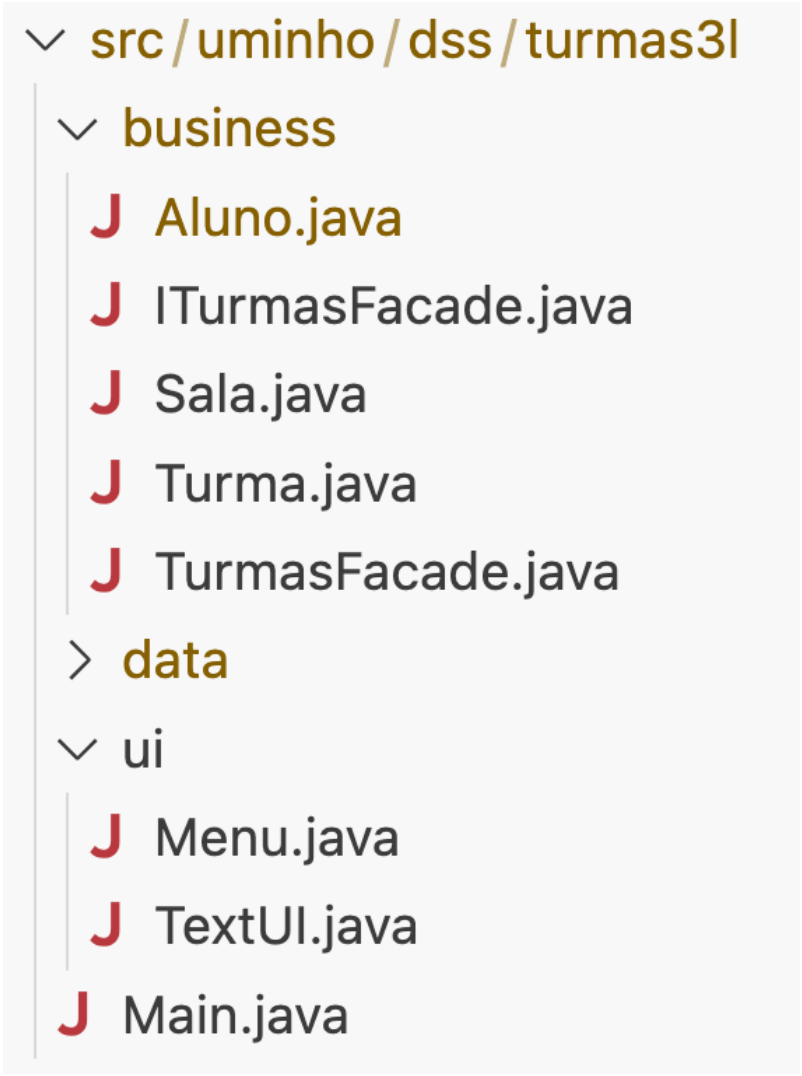
Dúvidas?

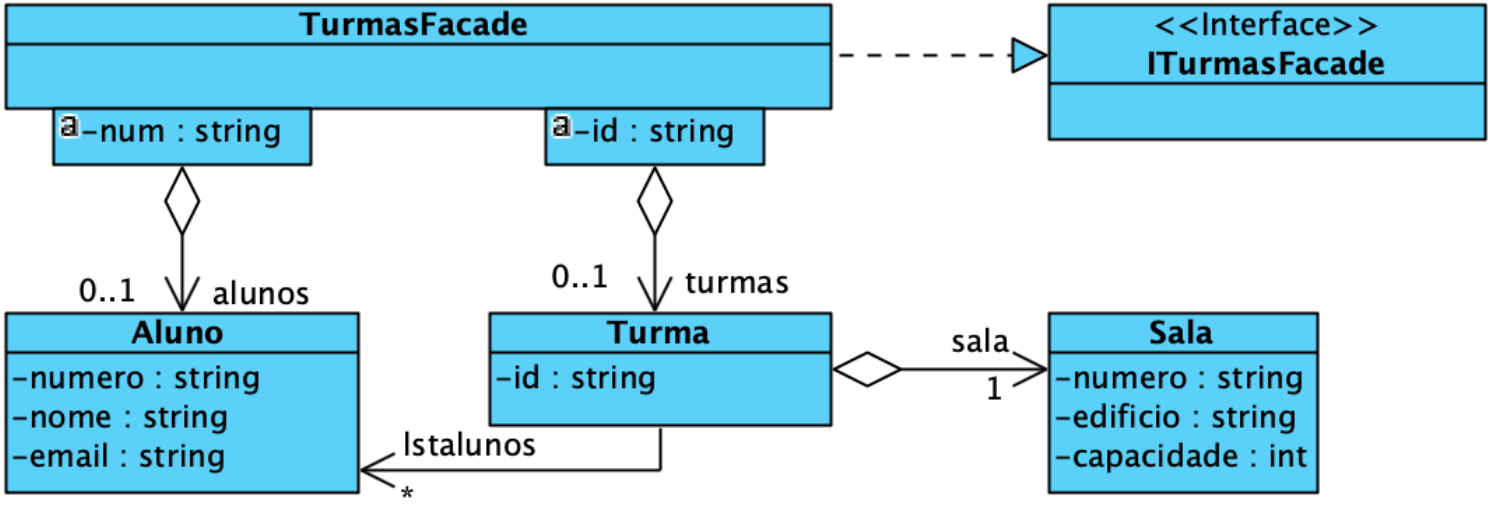
- Dúvidas sobre diagramas de classe?
- Dúvidas sobre identificação de responsabilidades da LN?
- Exemplo (que passos são responsabilidades da LN?)

```
Bem vindo ao Sistema de Gestão de Turmas!

 *** Menu ***
1 - Operações sobre Alunos
2 - Operações sobre Turmas
3 - ---
4 - ---
5 - Listar Alunos de Turma
0 - Sair
Opção: 1

 *** Gestão de Alunos ***
1 - Adicionar Aluno
2 - Consultar Aluno
3 - Listar Alunos
0 - Sair
Opção: 1
Número da novo aluno:
4141
Nome da novo aluno:
José FCF Campos
Email da novo aluno:
a4141@alunos.uminho.pt
Aluno adicionado

 *** Gestão de Alunos ***
1 - Adicionar Aluno
2 - Consultar Aluno
3 - Listar Alunos
0 - Sair
Opção:
```

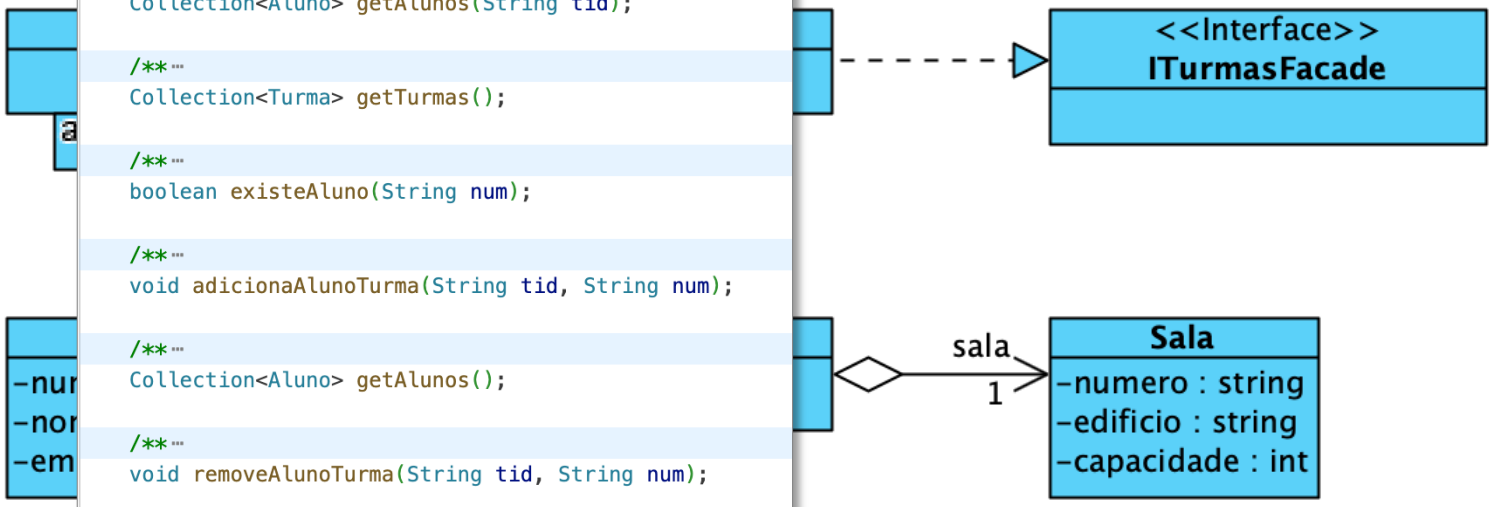
```
Bem vindo ao Sistema de Gestão de Turmas!

 *** Menu ***
1 - Operações sobre Alunos
2 - Operações sobre Turmas
3 - ---
4 - ---
5 - Listar Alunos de Turma
0 - Sair
Opção: 1

 *** Gestão de Alunos ***
1 - Adicionar Aluno
2 - Consultar Aluno
3 - Listar Alunos
0 - Sair
Opção: 1
Número da novo aluno:
4141
Nome da novo aluno:
José FCF Campos
Email da novo aluno:
a4141@alunos.uminho.pt
Aluno adicionado

 *** Gestão de Alunos ***
1 - Adicionar Aluno
2 - Consultar Aluno
3 - Listar Alunos
0 - Sair
Opção:
```

```
Use Case: Adicionar Aluno
Fluxo normal:
 1. Actor indica número, nome e email do aluno a adicionar
 2. Sistema regista o aluno
Fluxo de exceção 1: [aluno já existe] (passo 2)
 2.1. Sistema verifica que número de aluno já existe
 2.2. Sistema avisa o actor
```


src / uminho / dss / turmas3l
business
Aluno.java
ITurmasFacade.java
Sala.java
Turma.java
TurmasFacade.java
data
ui
Menu.java
TextUI.java
Main.java


TurmasFacade
<<Interface>>
ITurmasFacade
-num : string
-id : string
0..1
alunos
0..1
turmas
Aluno
-numero : string
-nome : string
-email : string
Turma
-id : string
sala
1
Sala
-numero : string
-edificio : string
-capacidade : int
lstalunos
*

```java
public interface ITurmasFacade {

    /**…
    void adicionaAluno(Aluno a);

    /**…
    void adicionaTurma(Turma t);

    /**…
    void alteraSalaDeTurma(String tid, Sala s);

    /**…
    boolean existeTurma(String tid);

    /**…
    Aluno procuraAluno(String num);

    /**…
    Collection<Aluno> getAlunos(String tid);

    /**…
    Collection<Turma> getTurmas();

    /**…
    boolean existeAluno(String num);

    /**…
    void adicionaAlunoTurma(String tid, String num);

    /**…
    Collection<Aluno> getAlunos();

    /**…
    void removeAlunoTurma(String tid, String num);

    /**…
    boolean existeAlunoEmTurma(String tid, String num);

    /**…
    boolean haAlunos();

    /**…
    boolean haTurmas();

    /**…
    boolean haTurmasComAlunos();
}
```



```
public interface ITurmasFacade {

    /**…
```

Method Summary

All Methods | **Instance Methods** | **Abstract Methods**

Modifier and Type	Method and Description
void	**adicionaAluno**(**Aluno** a) Método que adiciona um aluno.
void	**adicionaAlunoTurma**(java.lang.String tid, java.lang.String num) Método que adiciona um aluno à turma.
void	**adicionaTurma**(**Turma** t) Método que adiciona uma turma
void	**alteraSalaDeTurma**(java.lang.String tid, **Sala** s) Método que altera a sala da turma.
boolean	**existeAluno**(java.lang.String num) Método que verifica se um aluno existe
boolean	**existeAlunoEmTurma**(java.lang.String tid, java.lang.String num) Método que verifica se o aluno existe na turma
boolean	**existeTurma**(java.lang.String tid) Método que verifica se uma turma existe
java.util.Collection<**Aluno**>	**getAlunos**() Método que devolve todos os alunos registados.
java.util.Collection<**Aluno**>	**getAlunos**(java.lang.String tid) Método que devolve os alunos de uma turma.
java.util.Collection<**Turma**>	**getTurmas**() Método que devolve todas as turmas
boolean	**haAlunos**() Método que verifica se há alunos no sistema
boolean	**haTurmas**() Método que verifica se há turmas no sistema
boolean	**haTurmasComAlunos**() Método que verifica se há turmas com alunos registados
Aluno	**procuraAluno**(java.lang.String num) Método que procura um aluno
void	**removeAlunoTurma**(java.lang.String tid, java.lang.String num) Método que remove um aluno da turma.

```java
public class TextUI {
    // O model tem a 'lógica de negócio'.
    private final ITurmasFacade model;
```

```java
/**
 *  Estado - Gestão de Alunos
 */
private void gestaoDeAlunos() {
    Menu menu = new Menu(titulo:"Gestão de Alunos", new String[]{
            "Adicionar Aluno",
            "Consultar Aluno",
            "Listar Alunos"
    });

    // Registar os handlers
    menu.setHandler(i:1, ()->adicionarAluno());
    menu.setHandler(i:2, ()->consultarAluno());
    menu.setHandler(i:3, ()->listarAlunos());

    menu.run();
}
```

```
public class Tex
    // O model t
    private fina
```

```
/**
 *  Estado - Gestão de
 */
private void gestaoDeA
    Menu menu = new Me
            "Adicionar
            "Consultar
            "Listar Al
    });

    // Registar os han
    menu.setHandler(i:
    menu.setHandler(i:
    menu.setHandler(i:

    menu.run();
}
```

```
/**
 *  Estado - Adicionar Aluno
 */
private void adicionarAluno() {
    try {
        System.out.println(x:"Número da novo aluno: ");
        String num = scin.nextLine();
        if (!this.model.existeAluno(num)) {
            System.out.println(x:"Nome da novo aluno: ");
            String nome = scin.nextLine();
            System.out.println(x:"Email da novo aluno: ");
            String email = scin.nextLine();
            this.model.adicionaAluno(new Aluno(num, nome, email));
            System.out.println(x:"Aluno adicionado");
        } else {
            System.out.println(x:"Esse número de aluno já existe!");
        }
    }
    catch (NullPointerException e) {
        System.out.println(e.getMessage());
    }
}
```

```
public class Tex
    // O model t
    private fina
```

```
/**
 *  Estado - Gestão de
 */
private void gestaoDeA
    Menu menu = new Me
            "Adicionar
            "Consultar
            "Listar Al
    });

    // Registar os han
    menu.setHandler(i:
    menu.setHandler(i:
    menu.setHandler(i:

    menu.run();
}
```

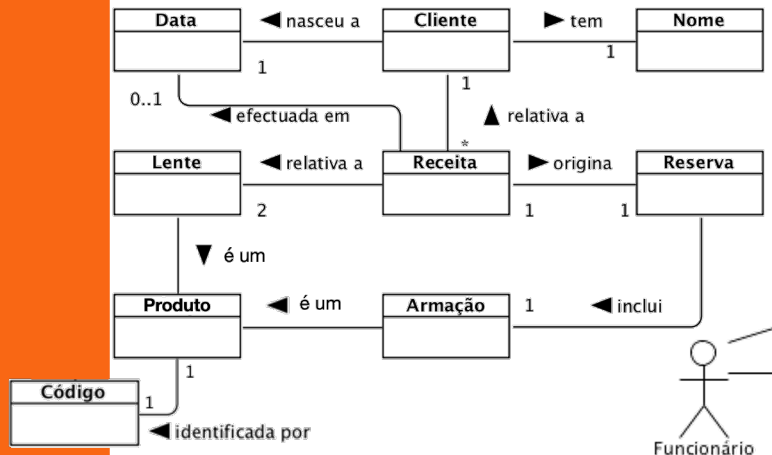
```
/**
 *  Estado - Adicionar Aluno
 */
private void adicionarAluno() {
    try {
        System.out.println(x:"Número da novo aluno: ");
        String num = scin.nextLine();
        if (!this.model.existeAluno(num)) {
            System.out.println(x:"Nome da novo aluno: ");
            String nome = scin.nextLine();
            System.out.println(x:"Email da novo aluno: ");
            String email = scin.nextLine();
            this.model.adicionaAluno(new Aluno(num, nome, email));
            System.out.println(x:"Aluno adicionado");
        } else {
            System.out.println(x:"Esse número de aluno já existe!");
        }
    } ca

    }
}
```

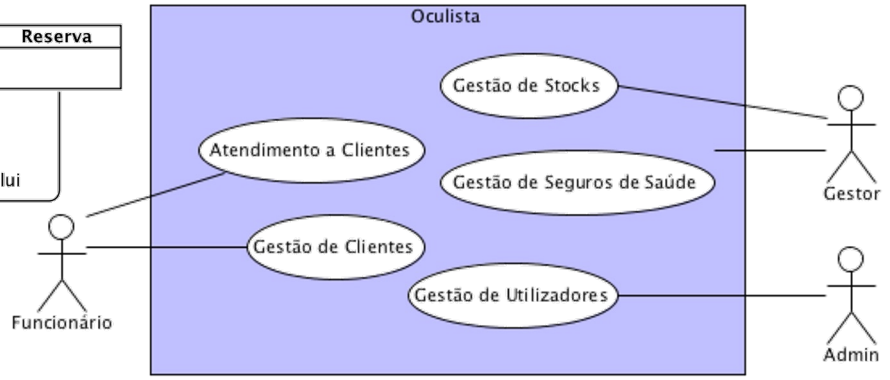
```
Use Case: Adicionar Aluno
Fluxo normal:
 1. Actor indica número, nome e email do aluno a adicionar
 2. Sistema regista o aluno
Fluxo de exceção 1: [aluno já existe] (passo 2)
 2.1. Sistema verifica que número de aluno já existe
 2.2. Sistema avisa o actor
```

Modelação Comportamental
(Diagramas de Sequência)

Desenvolvimento de Sistemas Software
José Creissac Campos / António Nestor Ribeiro

Em resumo…



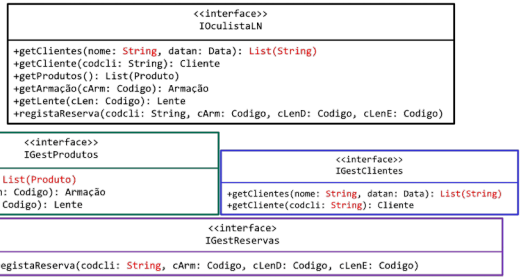


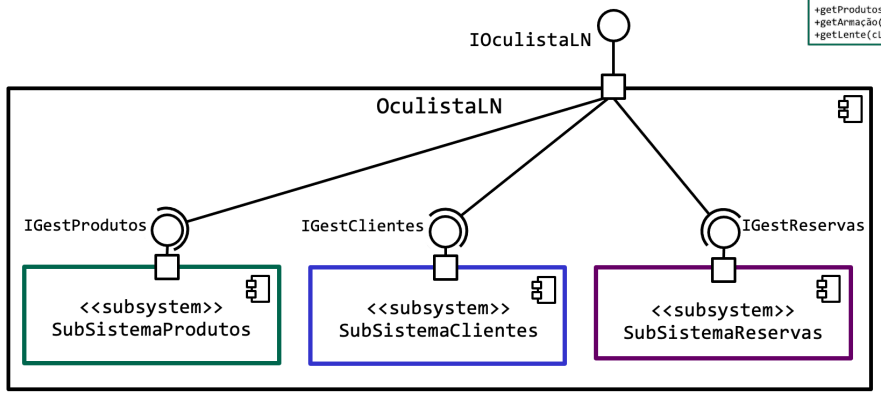
Use Case: Reservar armação e lentes

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

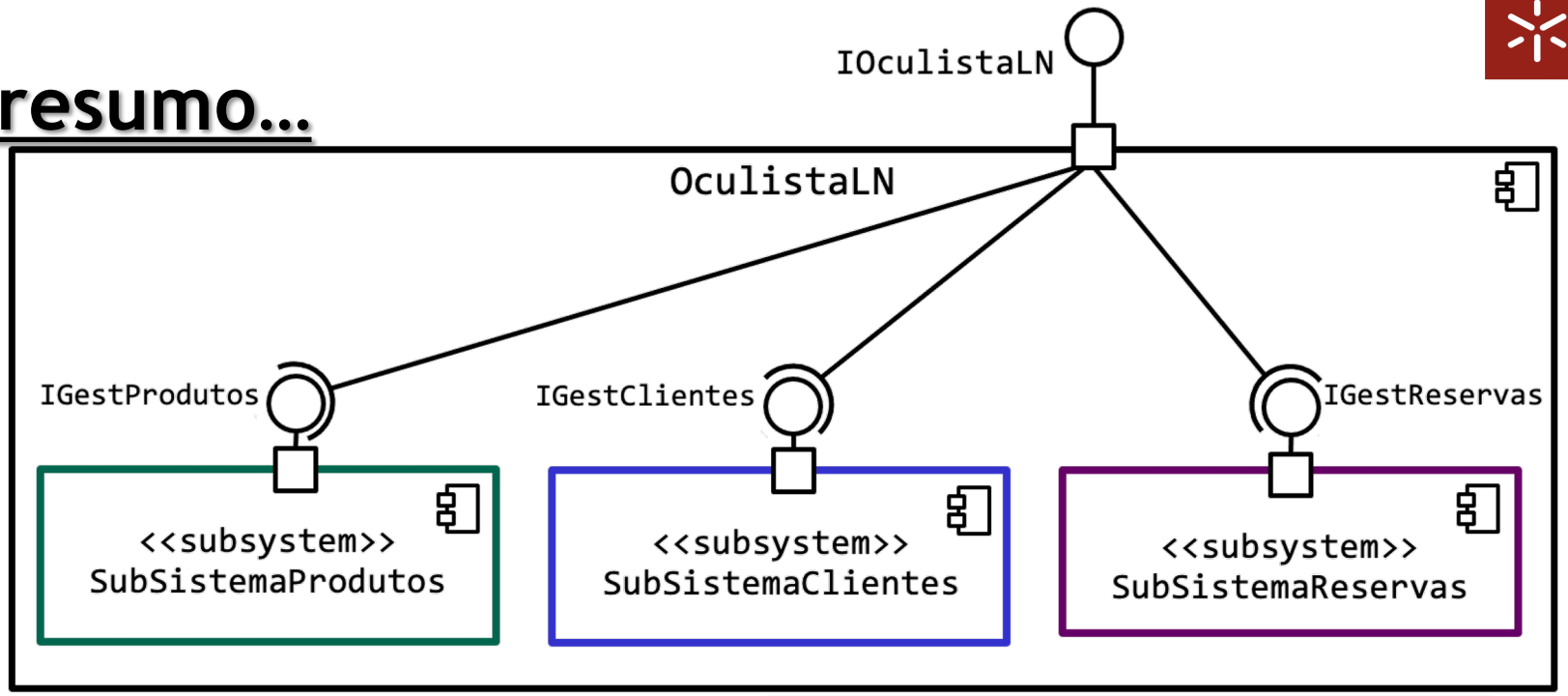
Fluxo alternativo: [lista de clientes tem tamanho 1] (passo 3)
3.1. Sistema apresenta detalhes do único cliente da lista
3.2. regressa a 7

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



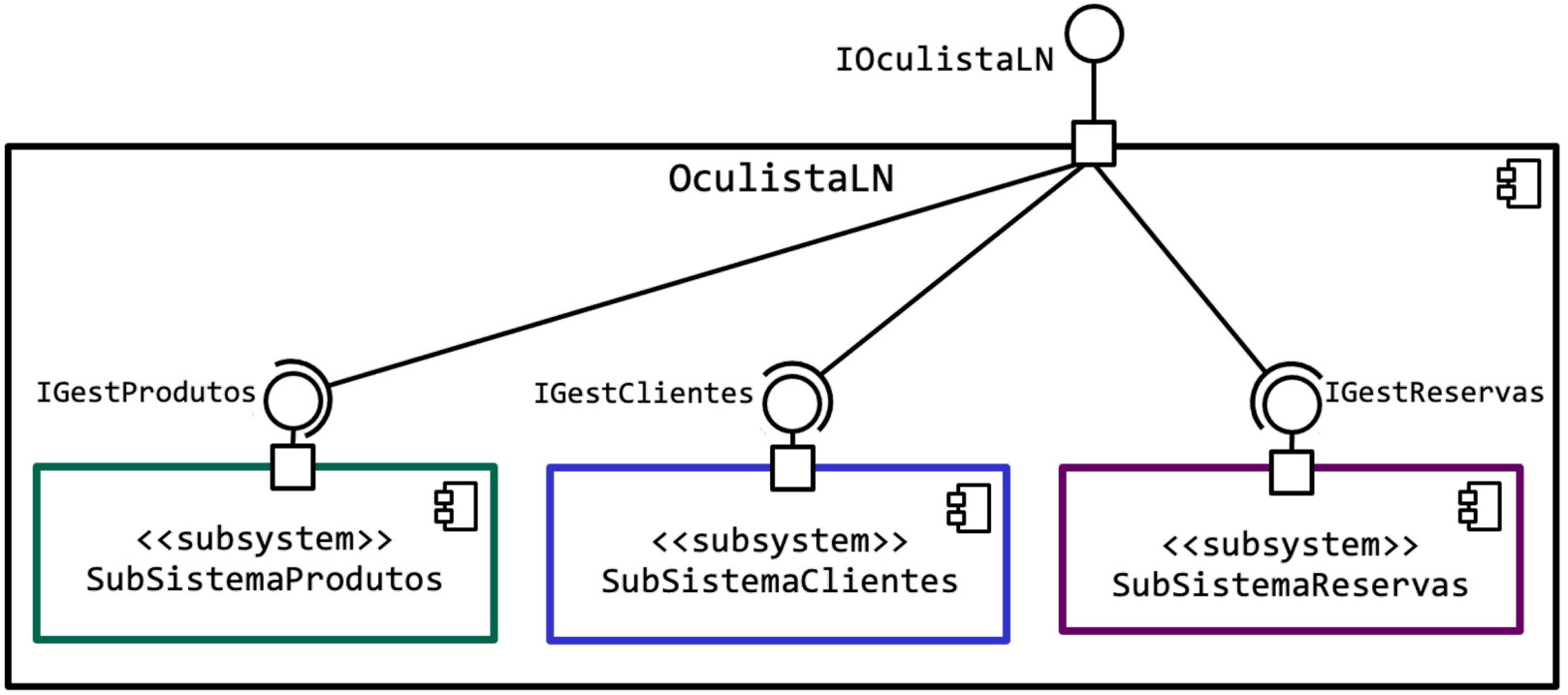


Em resumo…

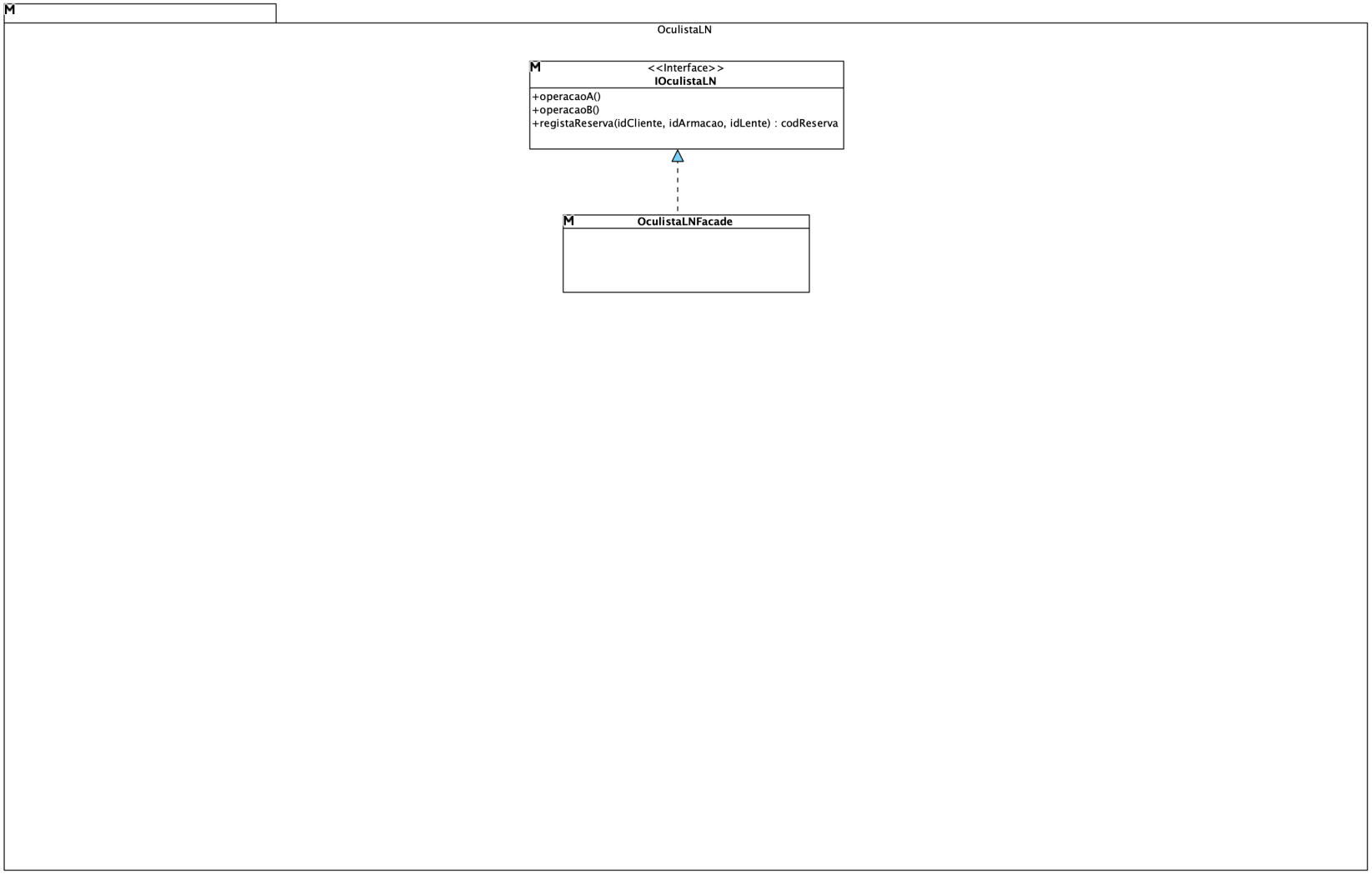

IOculistaLN
OculistaLN
IGestProdutos
IGestClientes
IGestReservas
<<subsystem>>
SubSistemaProdutos
<<subsystem>>
SubSistemaClientes
<<subsystem>>
SubSistemaReservas

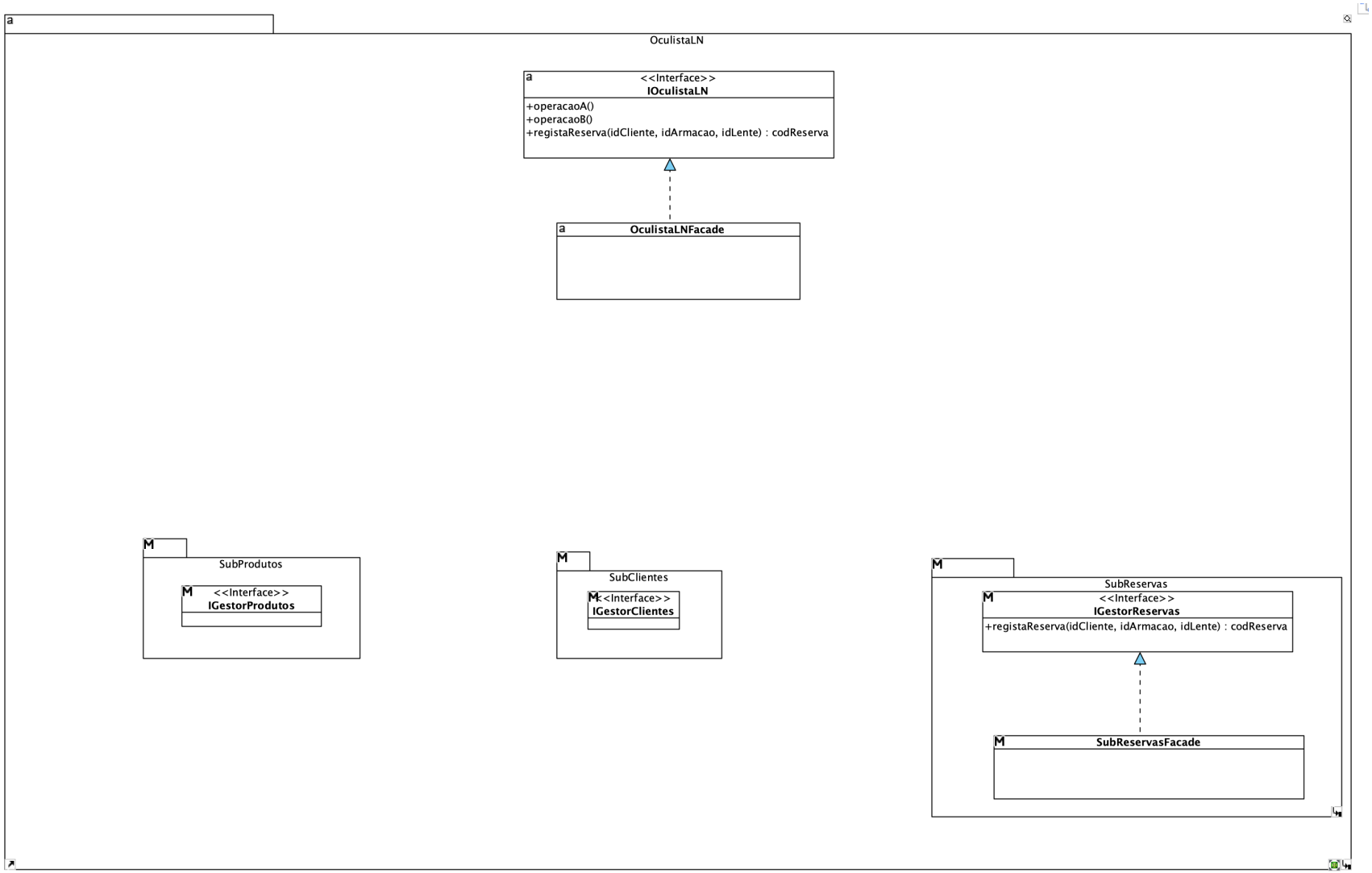

OculistaBD
OculistaLN
SubSistemaClientes
SubSistemaProdutos
SubSistemaReservas
GestReservasFacade.java
IGestReservas.java
Reservas.java
IOculistaLN.java
OculistaFacade.java
OculistaUI

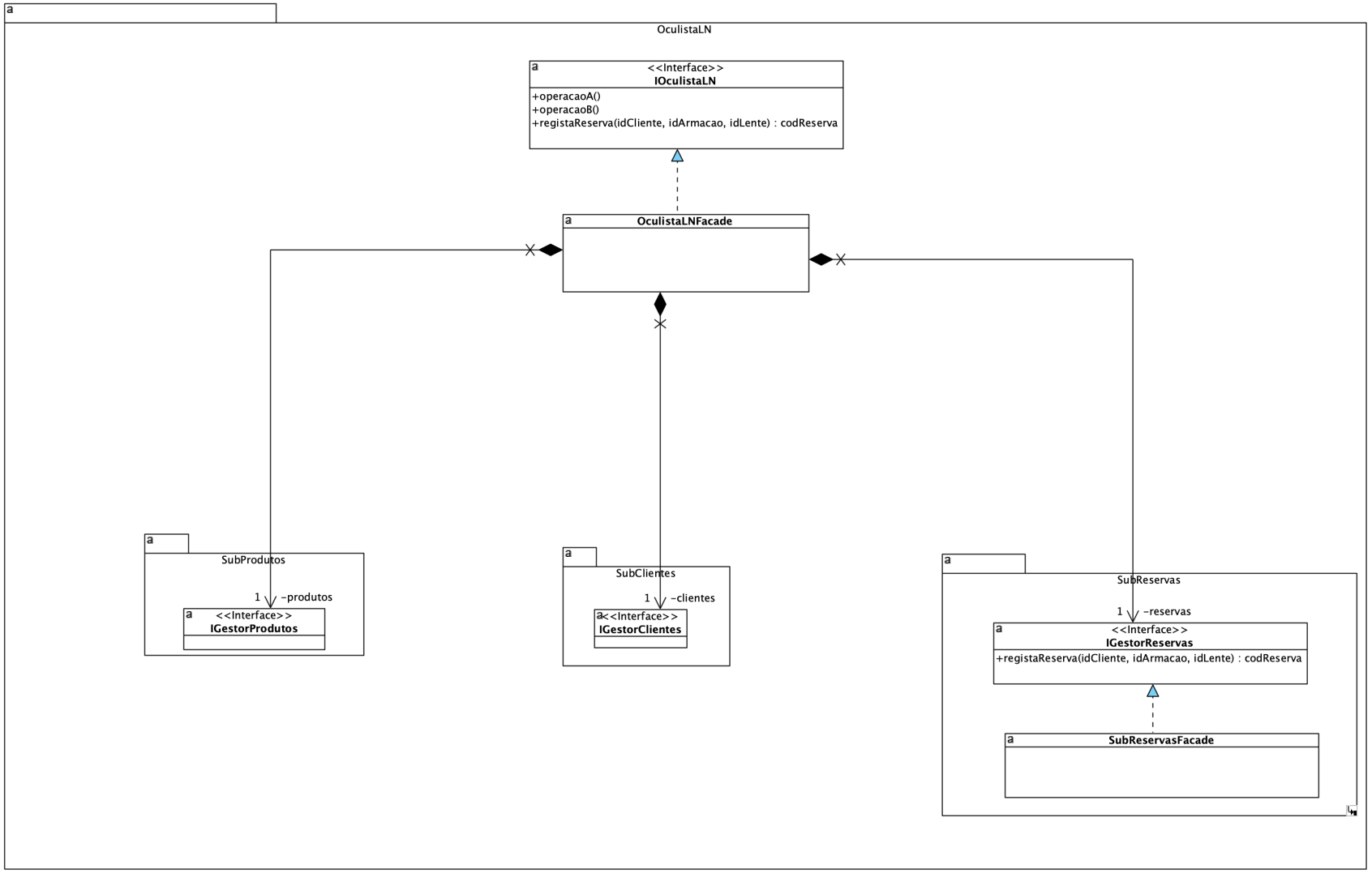
Em resumo…

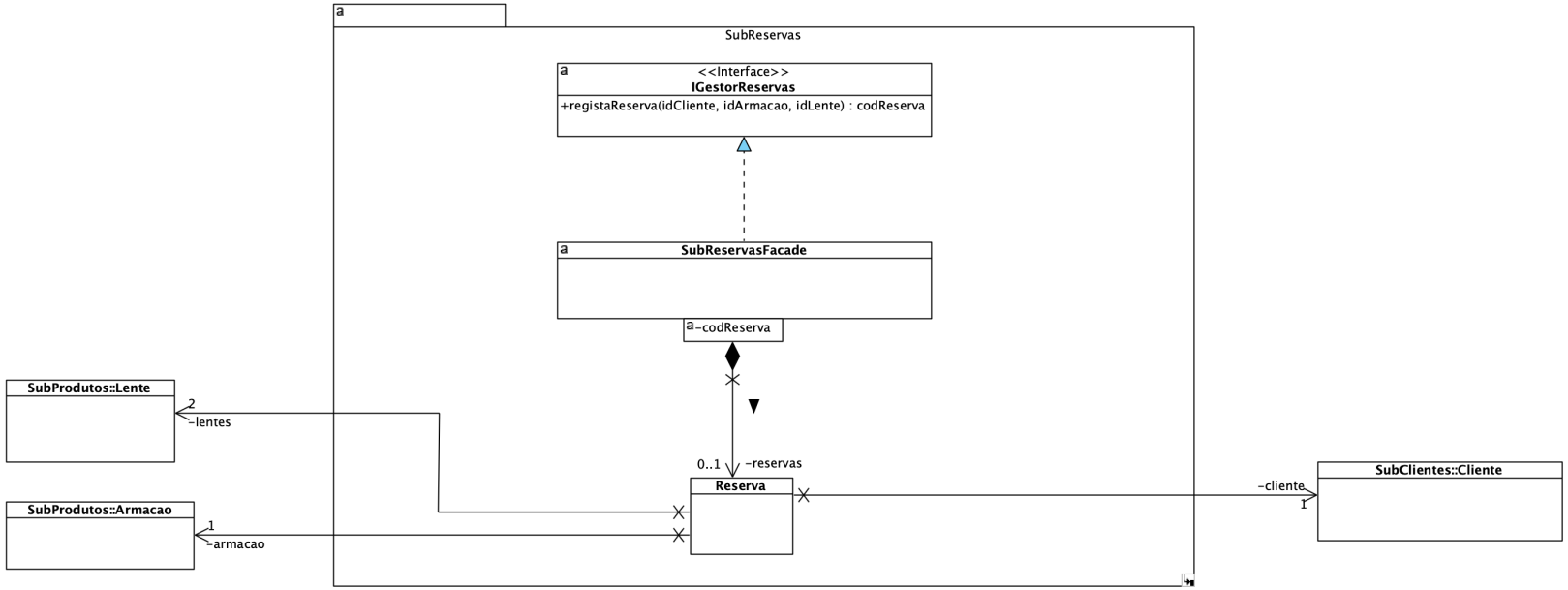


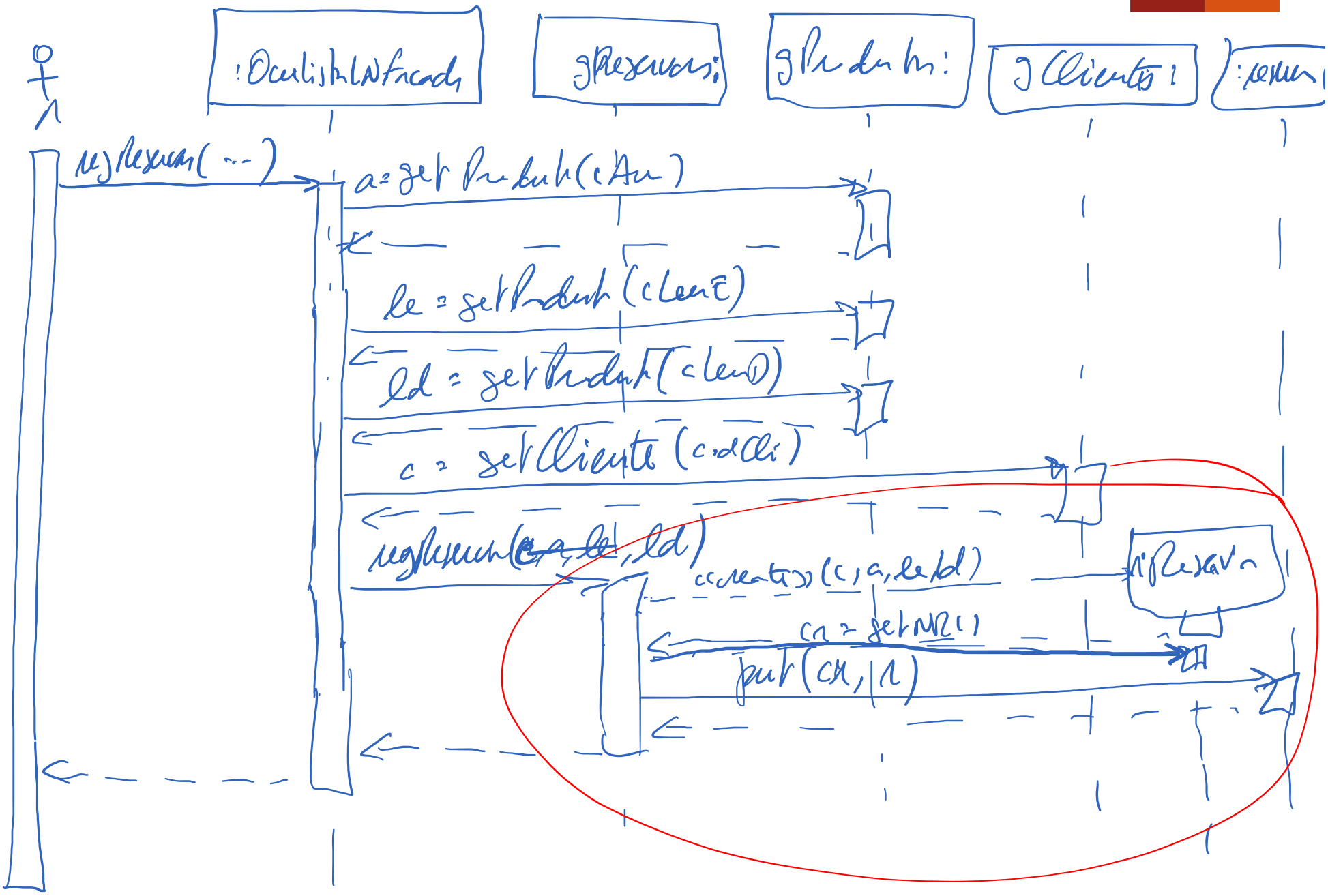
- Para tomar decisões sobre a arquitetura, é necessário considerar as operações que teremos de implementar


OculistaLN
<<Interface>>
IOculistaLN
+operacaoA()
+operacaoB()
+registaReserva(idCliente, idArmacao, idLente) : codReserva
OculistaLNFacade

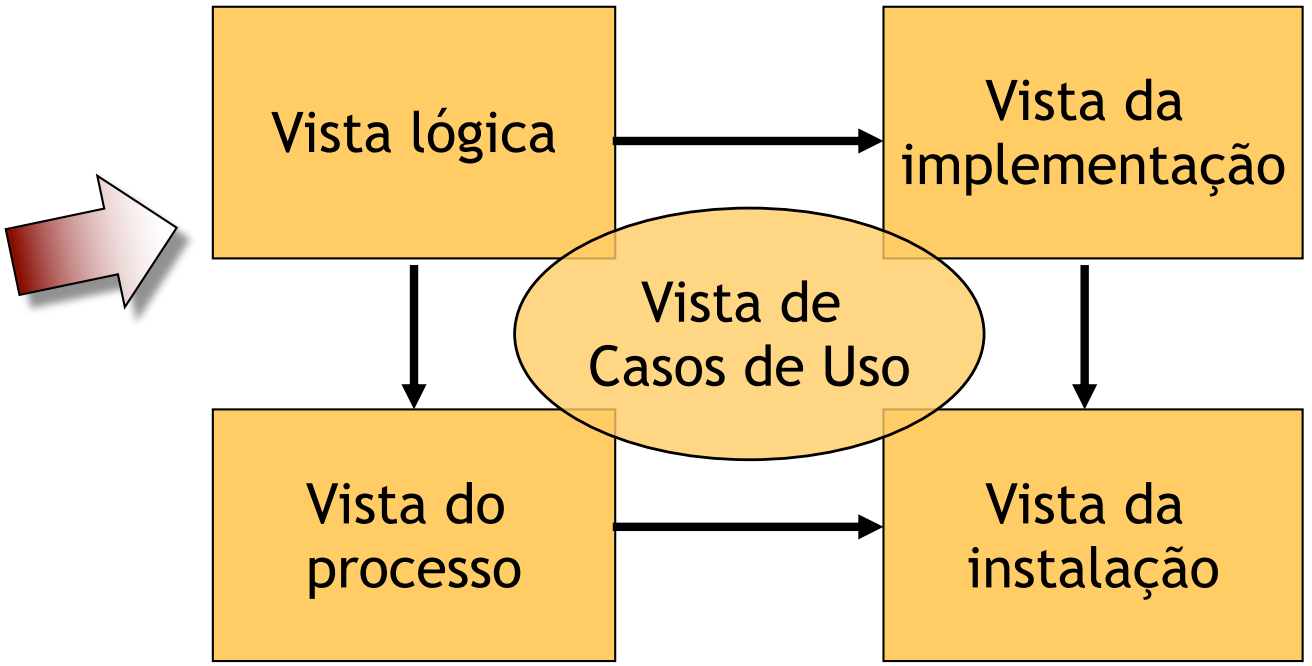

a
OculistaLN
a
<<Interface>>
IOculistaLN
+operacaoA()
+operacaoB()
+registaReserva(idCliente, idArmacao, idLente) : codReserva
a
OculistaLNFacade
M
SubProdutos
M
<<Interface>>
IGestorProdutos
M
SubClientes
M
<<Interface>>
IGestorClientes
M
SubReservas
M
<<Interface>>
IGestorReservas
+registaReserva(idCliente, idArmacao, idLente) : codReserva
M
SubReservasFacade


OculistaLN
<<Interface>>
IOculistaLN
+operacaoA()
+operacaoB()
+registaReserva(idCliente, idArmacao, idLente) : codReserva
OculistaLNFacade
SubProdutos
1
-produtos
<<Interface>>
IGestorProdutos
SubClientes
1
-clientes
<<Interface>>
IGestorClientes
SubReservas
1
-reservas
<<Interface>>
IGestorReservas
+registaReserva(idCliente, idArmacao, idLente) : codReserva
SubReservasFacade


a
SubReservas
a
<<Interface>>
IGestorReservas
+registaReserva(idCliente, idArmacao, idLente) : codReserva
a
SubReservasFacade
a
-codReserva
SubProdutos::Lente
2
-lentes
SubProdutos::Armacao
1
-armacao
0..1
-reservas
Reserva
-cliente
1
SubClientes::Cliente

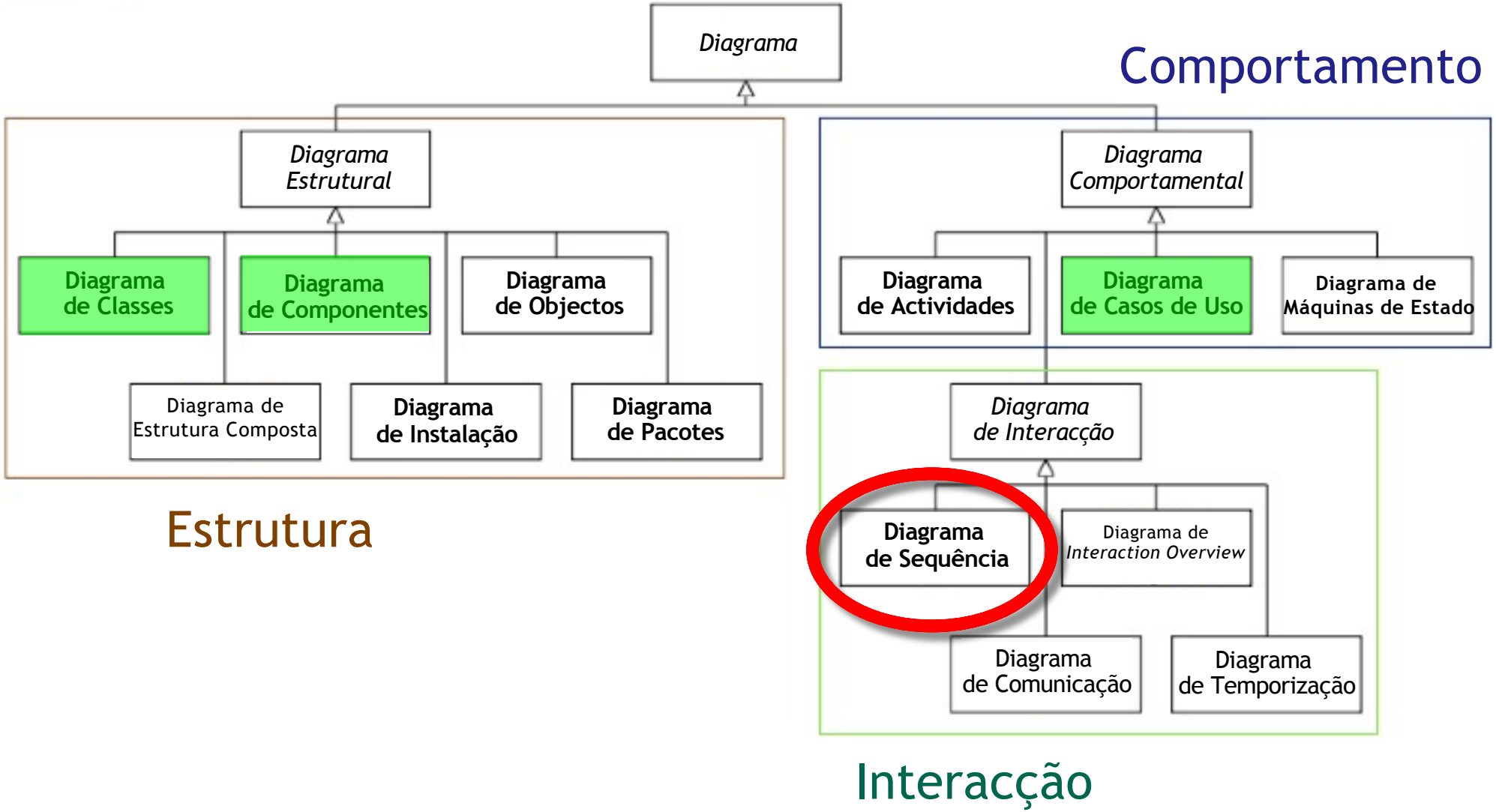


Onde estamos…

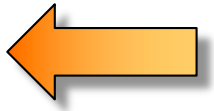


(Kruchten, 1995)

Diagramas da UML 2.x

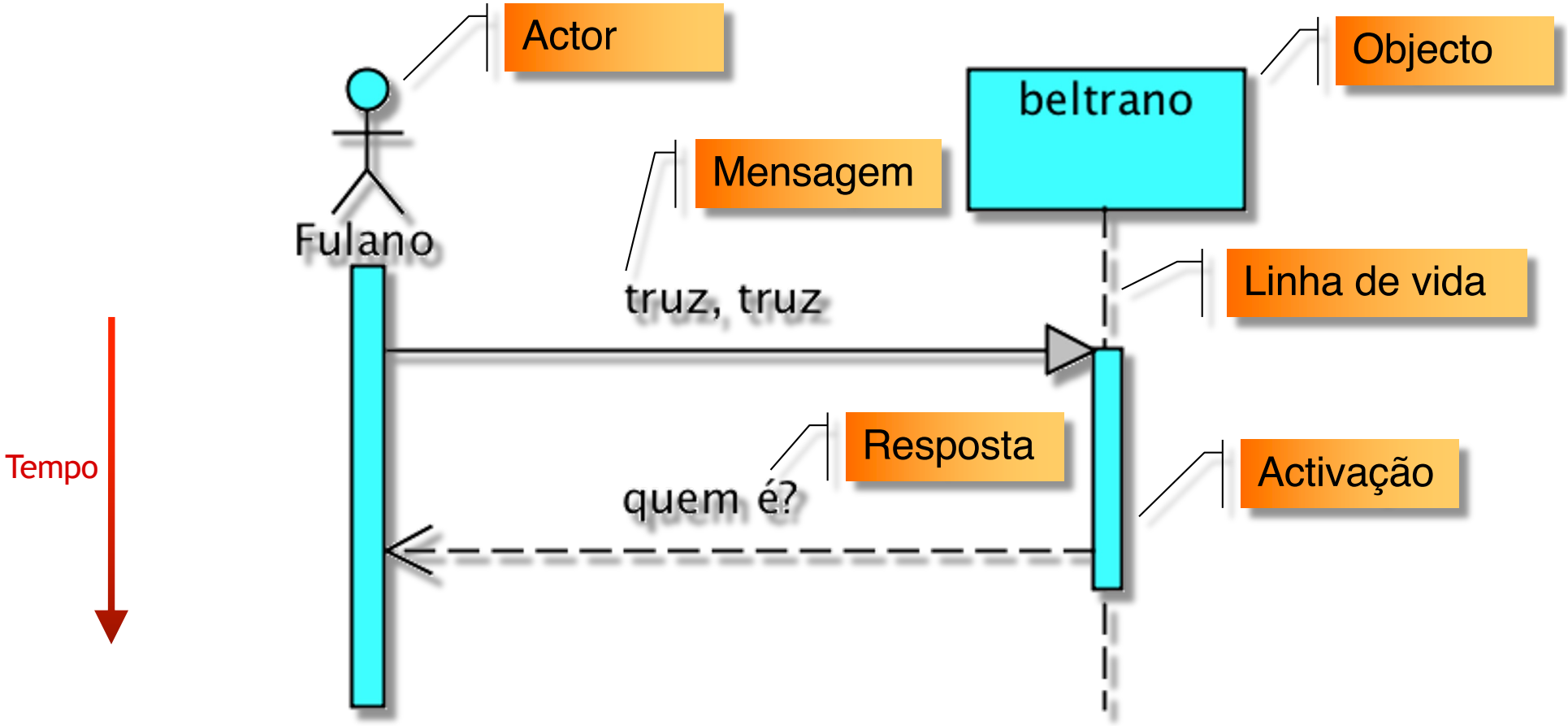


Diagramas de Interacção

- Um tipo de Diagrama Comportamental
- Descrevem como um conjunto de objectos coopera para realizar um dado comportamento
  - modelam as interacções entre os objectos para atingir um objectivo (p.e. realizar um *Use Case*)
  - **Diagramas de sequência** 
    - foco no ordenamento temporal das trocas de mensagens
  - **Diagramas de comunicação**
    - foco na arquitectura
  - **Diagramas de Temporização (*Timing Diagrams*)**
    - foco nos aspectos temporais
  - **Diagramas de *Interaction Overview***
    - visão de alto nível que combina os anteriores

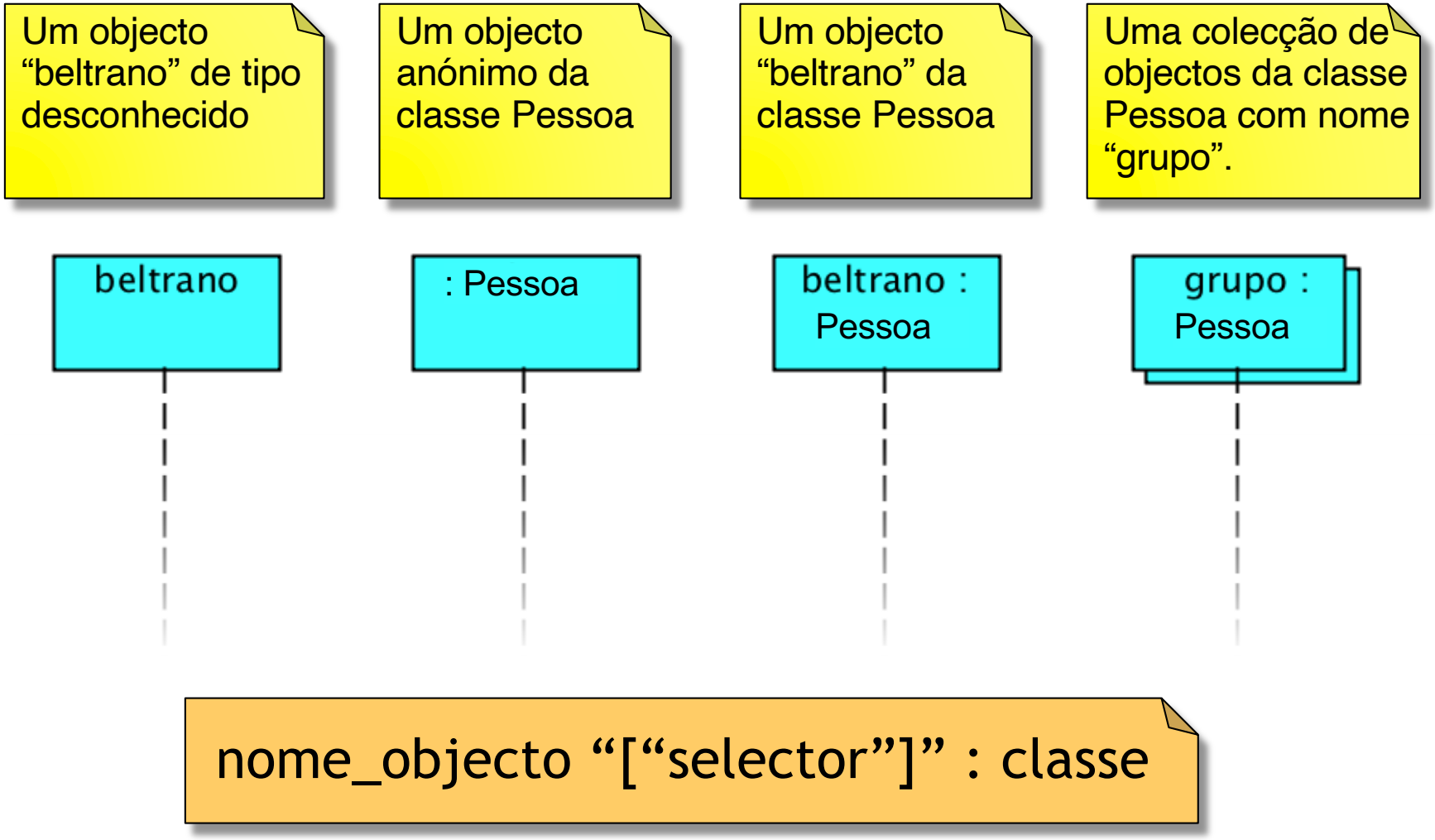
Diagramas de Sequência - notação essencial

- representam as interacções entre objectos através das mensagens que são trocadas entre eles
- a ênfase é colocada na ordenação temporal das mensagens
- permitem analisar a distribuição de "responsabilidade" pelas diferentes entidades (analisar onde está a ser efectuado o processamento)

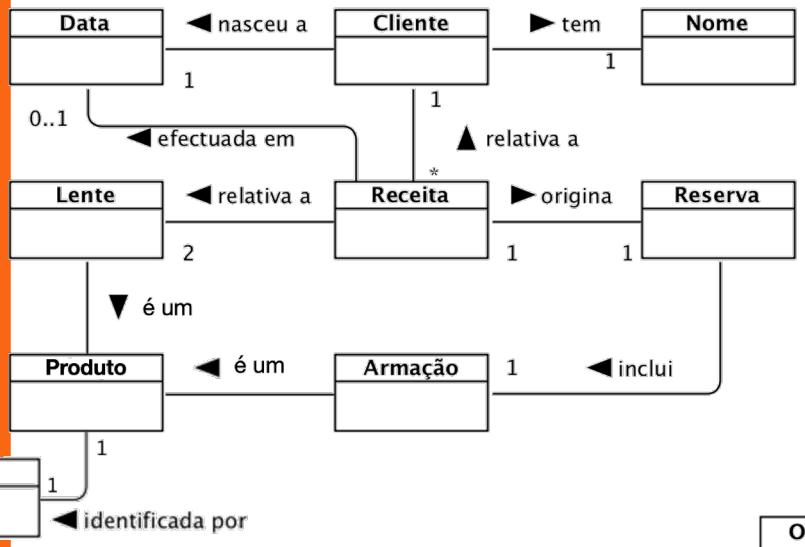


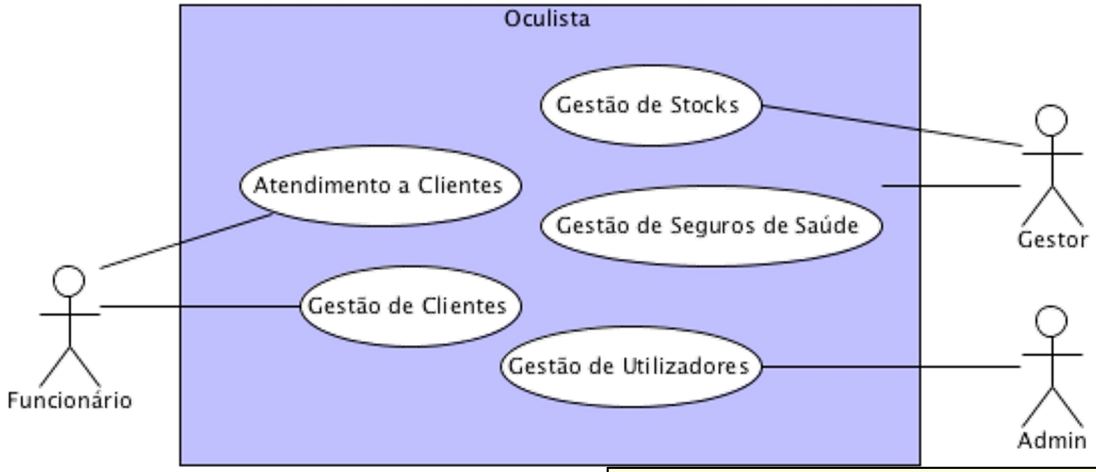
Diagramas de Sequência - notação essencial

Objectos



Um exemplo...



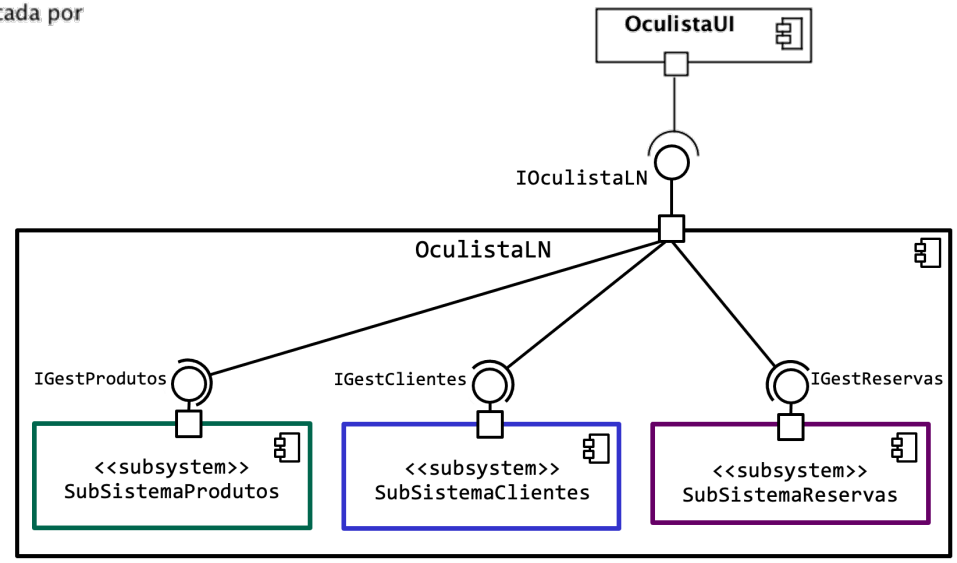


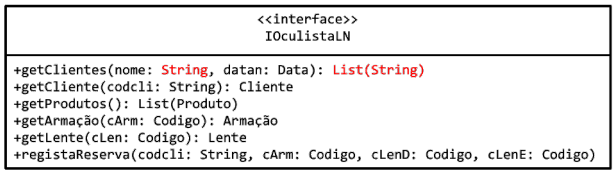
Use Case: Reservar armação e lentes

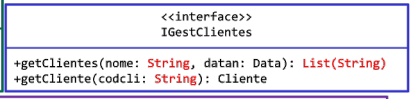
Descrição: Funcionário regista uma reserva de armação e lentes
Pós-condição: Reserva fica registada
Fluxo normal:
1. Funcionário indica nome e/ou data de nascimento do cliente
2. Sistema procura clientes
3. Sistema apresenta lista de clientes
4. Funcionário selecciona cliente
5. Sistema procura cliente
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

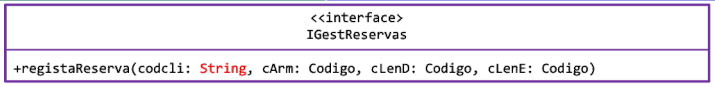
Fluxo alternativo: [lista de clients tem tamanho 1] (passo 3)
3.1. Sistema apresenta detalhes do único cliente da lista
3.2. regressa a 7

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

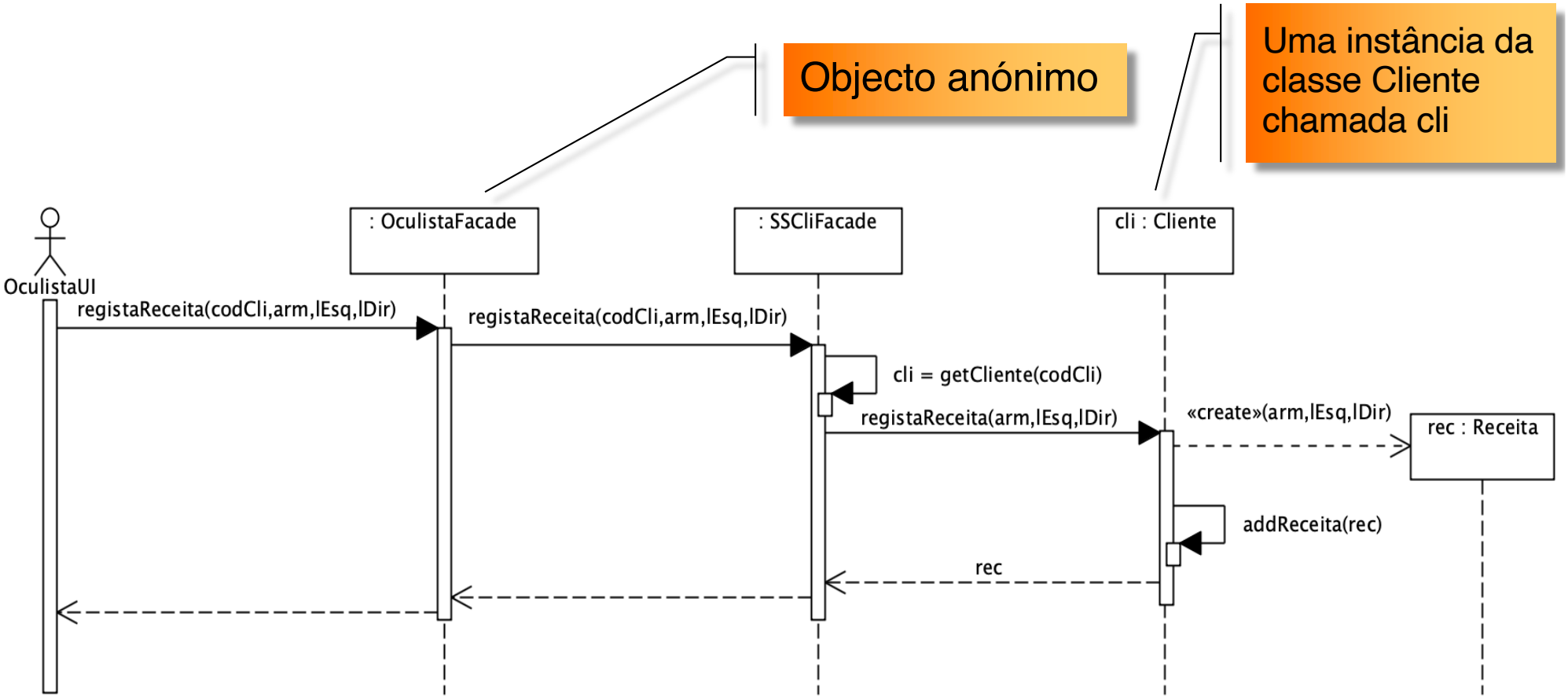






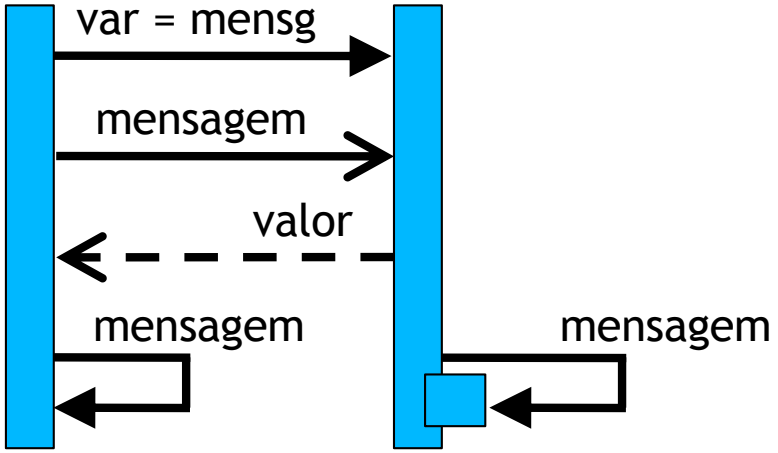


Diagramas de Sequência - notação essencial

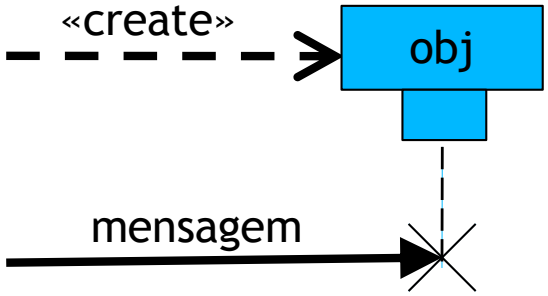


Mensagens

- invocação síncrona
- invocação assíncrona
- return/resultado
- self messages



- criar objectos

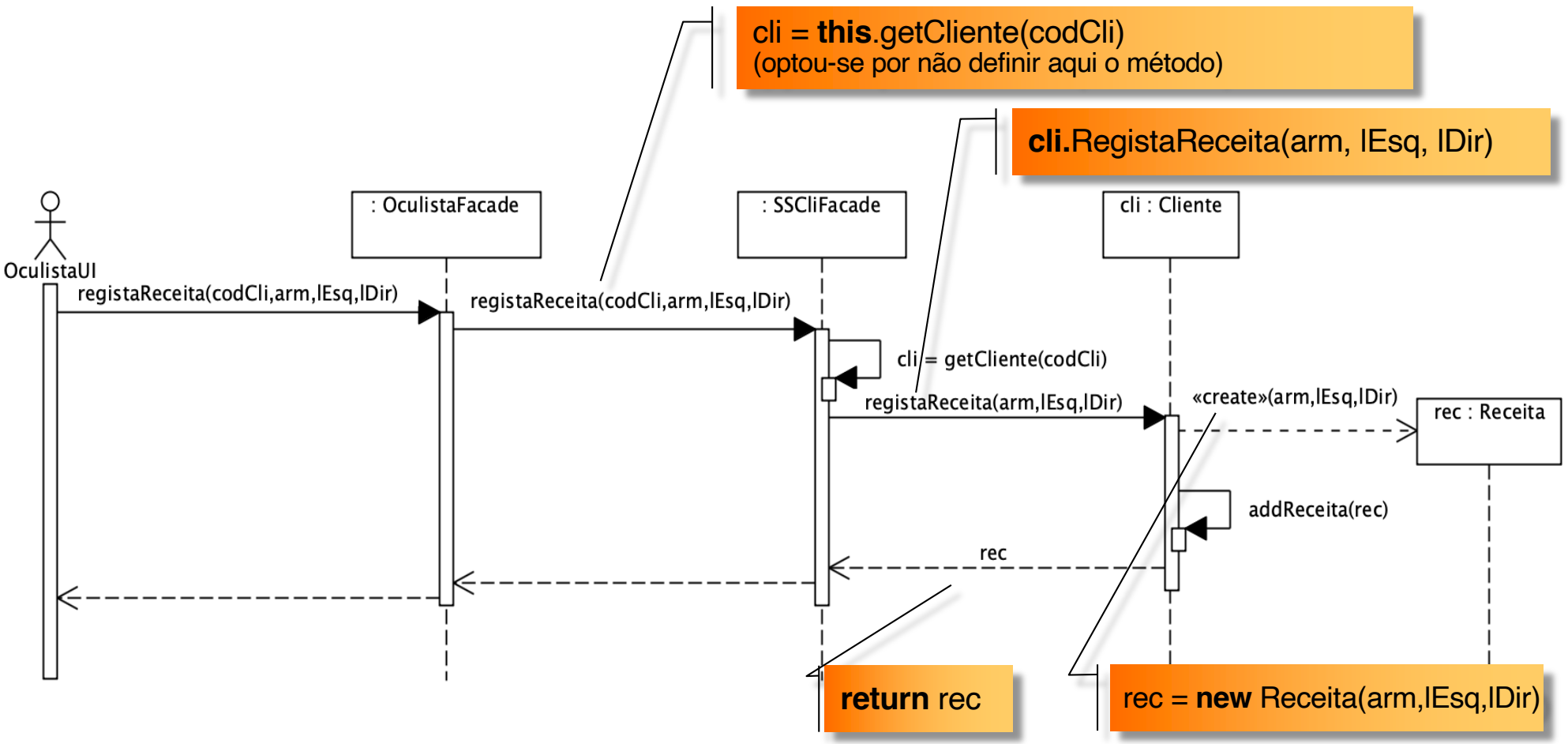


- destruir objectos

- lost/found messages

[atributo '='] nome_da_operação_sinal [argumentos] [':' tipo_resultado]

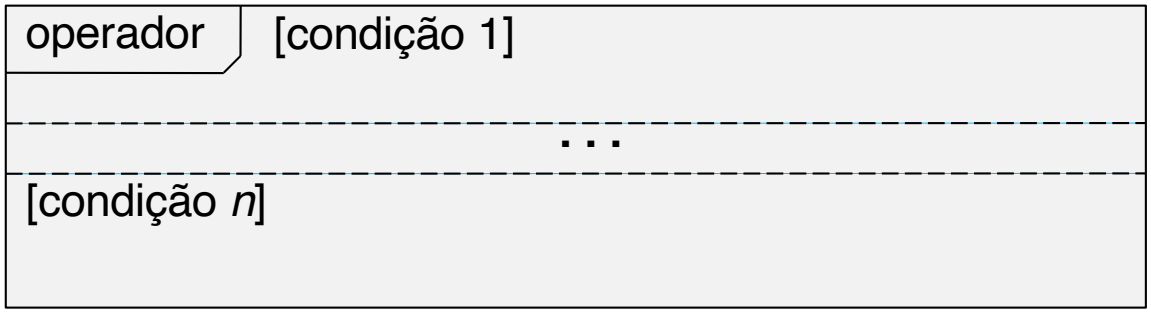
Diagramas de Sequência - notação essencial



- Todas as invocações são síncronas.
- Dois dos métodos não estão aqui definidos (+ construtor).
- **Atenção!** O objecto que envia a mensagem tem que "conhecer" o objecto a quem a envia.

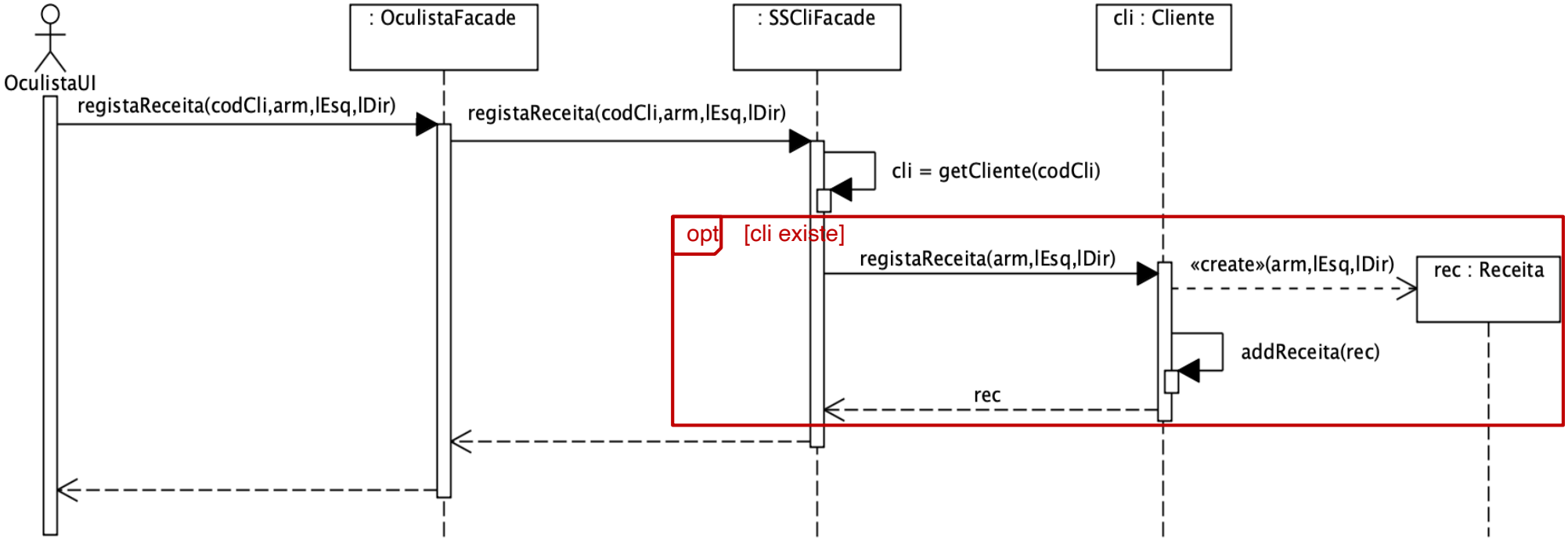
Diagramas de Sequência - fragmentos combinados

- Um fragmento combinado agrupa conjuntos de mensagens
- Permitem expressar fluxos condicionais e estruturar os modelos



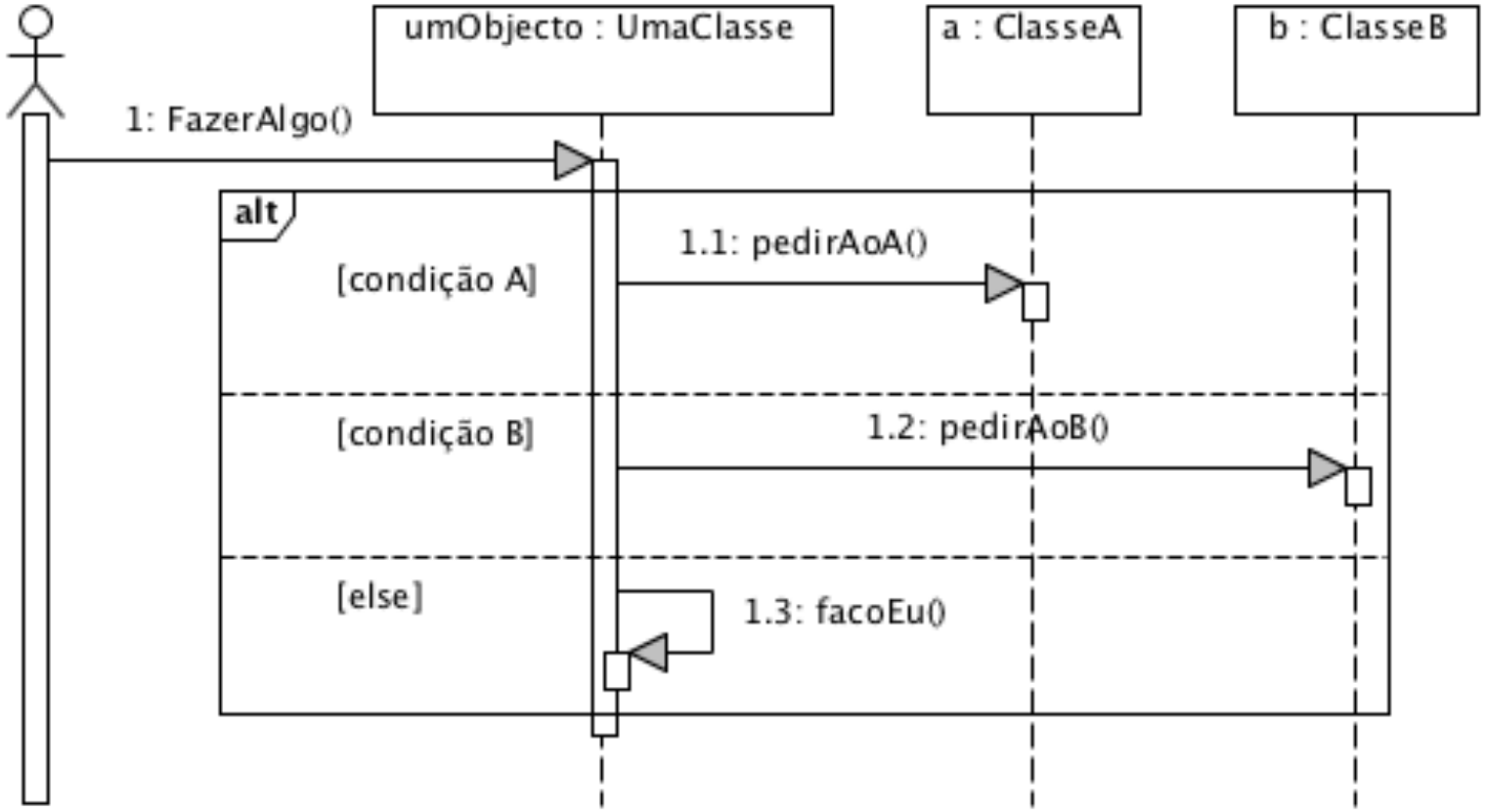
- Operadores mais comuns
  - **alt** – define fragmentos alternativos (mutuamente exclusivos)
  - **loop** / **loop(*n*)** – fragmento é repetido enquanto a guarda for verdadeira / ***n*** vezes
  - **opt** – fragmento opcional (ocorre se a guarda for verdadeira)
  - **break** – termina o fluxo
  - **ref** – referência a outro diagrama

Operador *opt*



Registo só é efectuado se cliente existe.

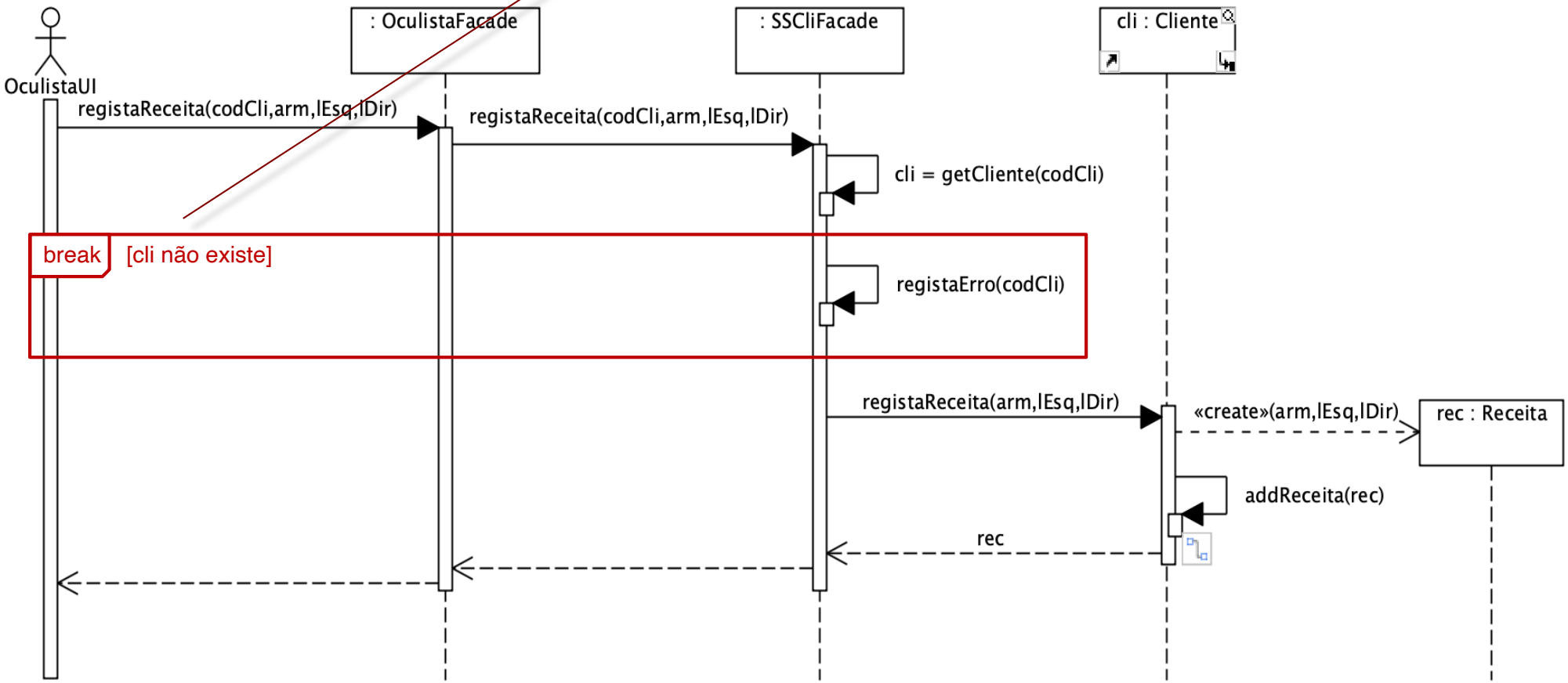
Operador *alt*



- Os fluxos possíveis são mutuamente exclusivos, pelo que apenas um deles será seguido.
- Se mais que uma condição se verificar, não está definido qual acontece.

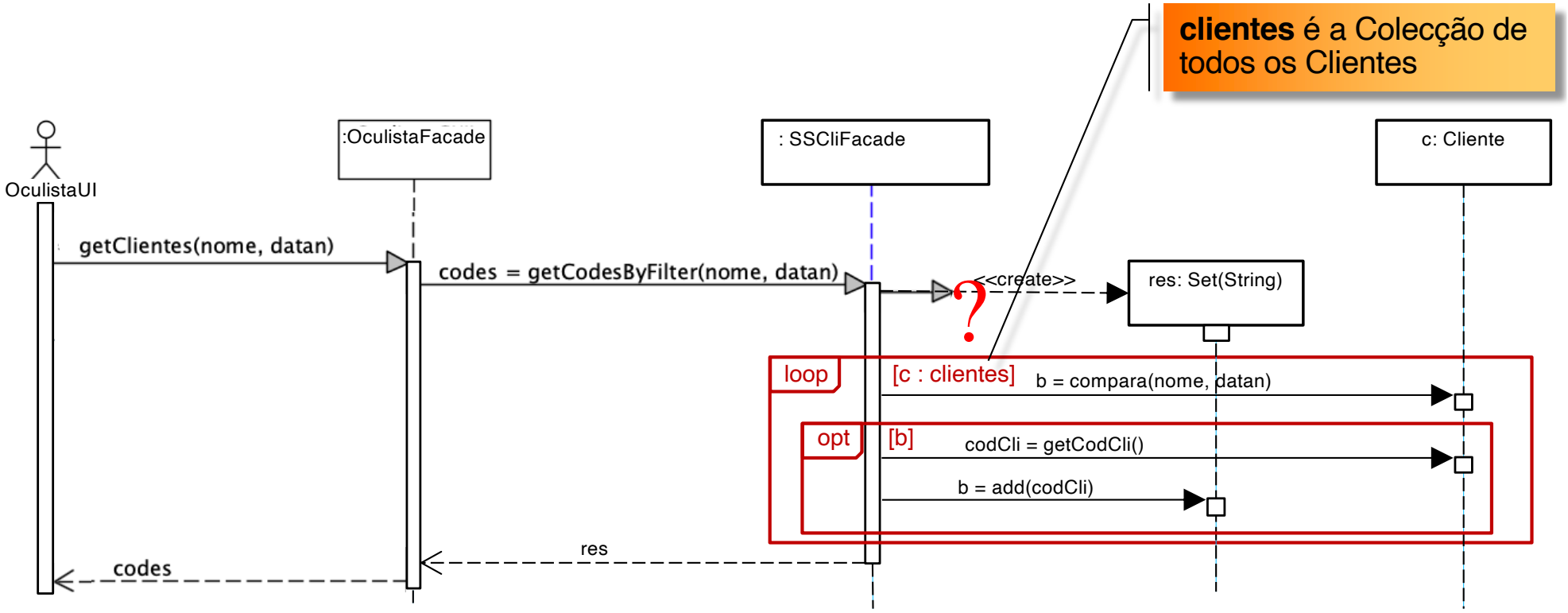
Operador *break*

Se cliente não existe, regista erro e termina.



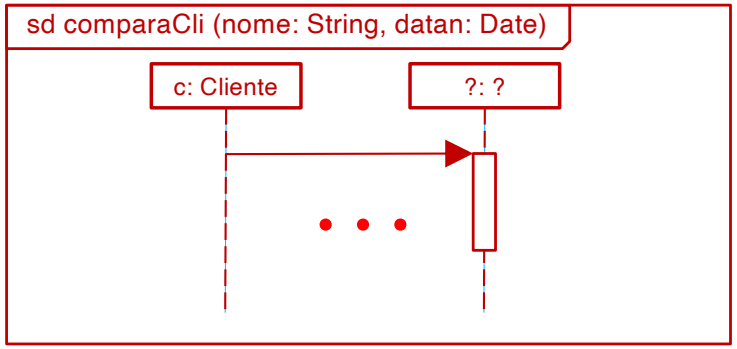
Registo só é efectuado se cliente existe.

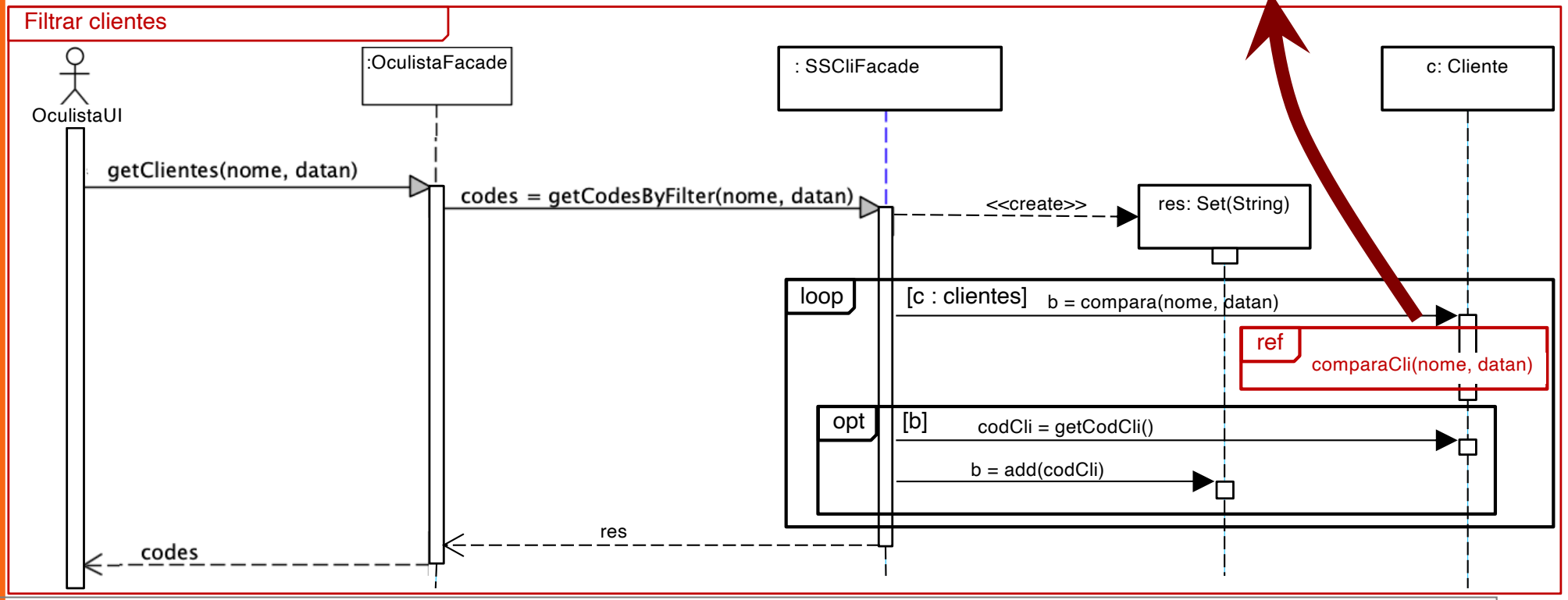
Operador *loop*



codes é o conjunto dos códigos dos clientes que satisfazem o critério.

Operador *ref*



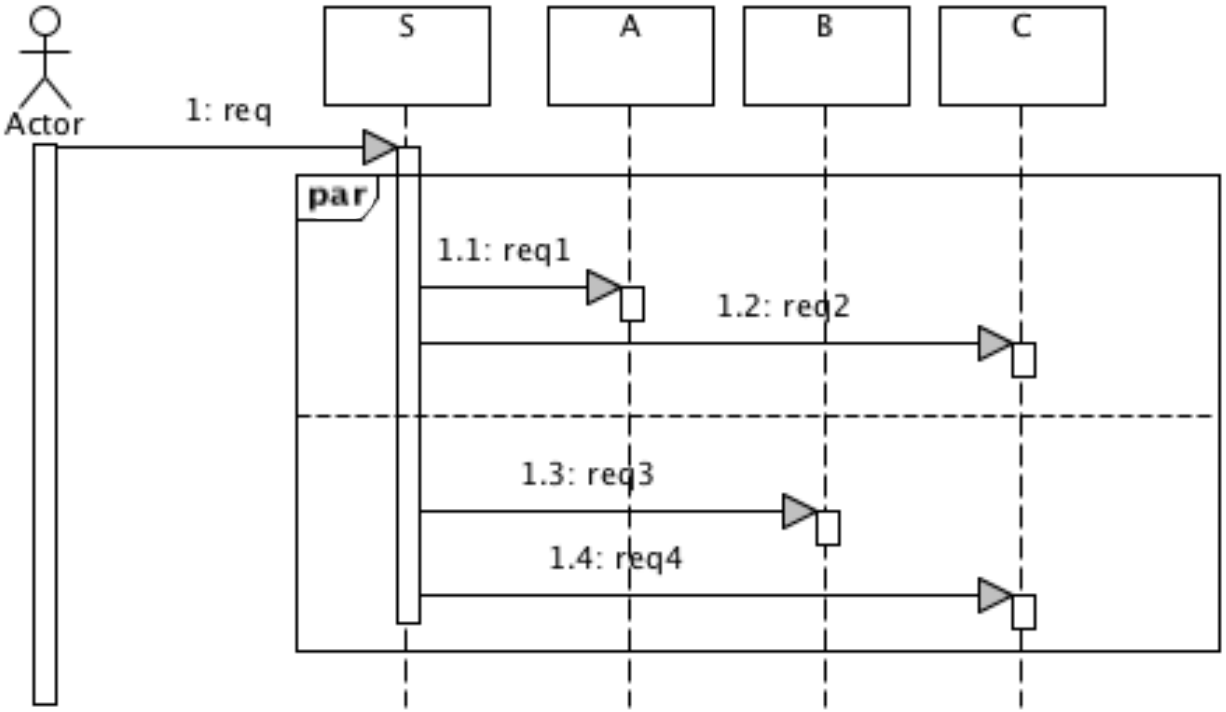


- Todos os diagrama devem ter um nome
- Um SD pode reutilizar outros SD referenciando-os num fragmento com o operador ***ref*** – permite estruturar os modelos

Outros operadores

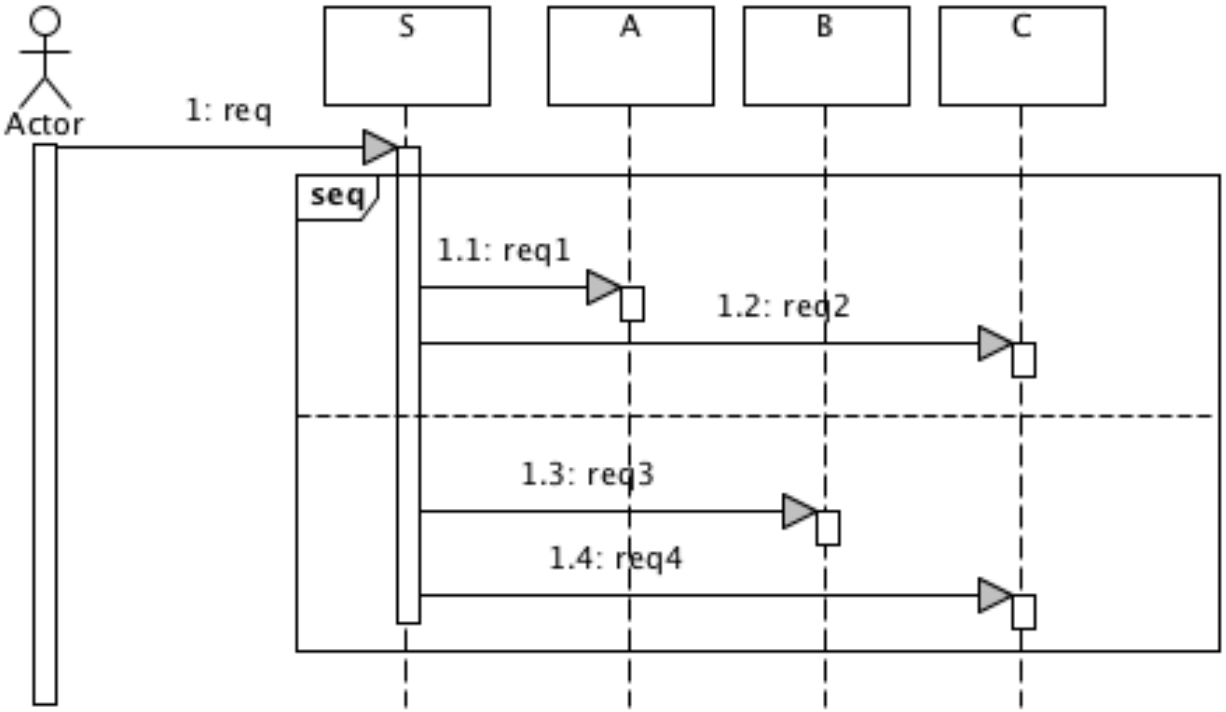
- **critical** – o operando executa de forma atómica
- **par** – os operandos executam em paralelo
- **seq** (sequenciação fraca) – todos os operandos executam em paralelo, mas eventos enviados a uma mesma linha de vida acontecem na mesma sequência dos operandos
- **strict** – os operandos executam em sequência
- **neg -** negação, o operando mostra uma interacção inválida
- **assert** – mostra o único comportamento válido naquele ponto
- **ignore** – indica mensagens intencionalmente omitidas da interacção
  - ignore {m1, m2, …} – m1, m2 podem acontecer mas não são mostradas
- **consider** – indica mensagens intencionalmente incluídas na interacção (dual de ignore)
  - consider {m1, m2, …} – outras mensagens, para além de m1, m2, podem acontecer mas não são mostradas

Operador *par*



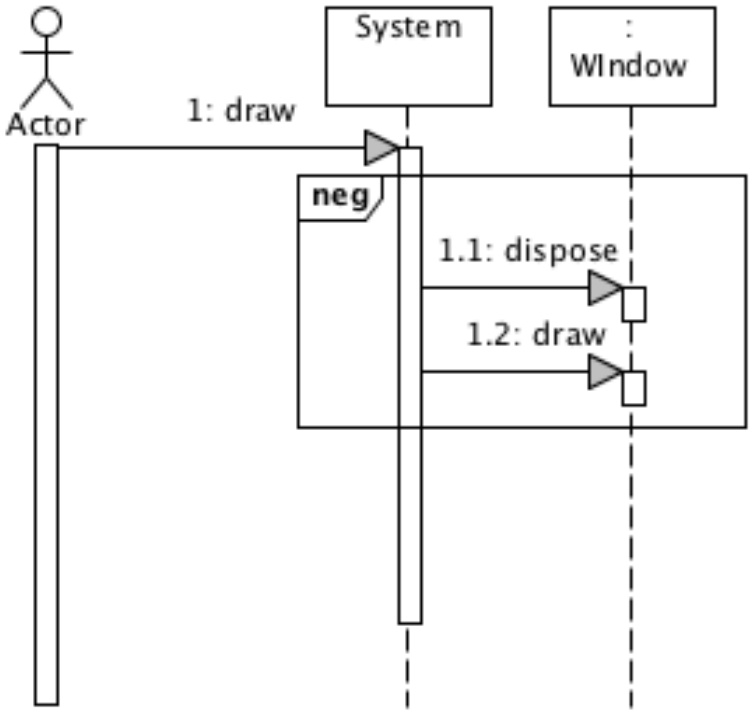
Eventos *req1* e *req2* podem acontecer em paralelo com eventos *req3* e *req4*. Nenhuma ordem é imposta.

Operador *seq*



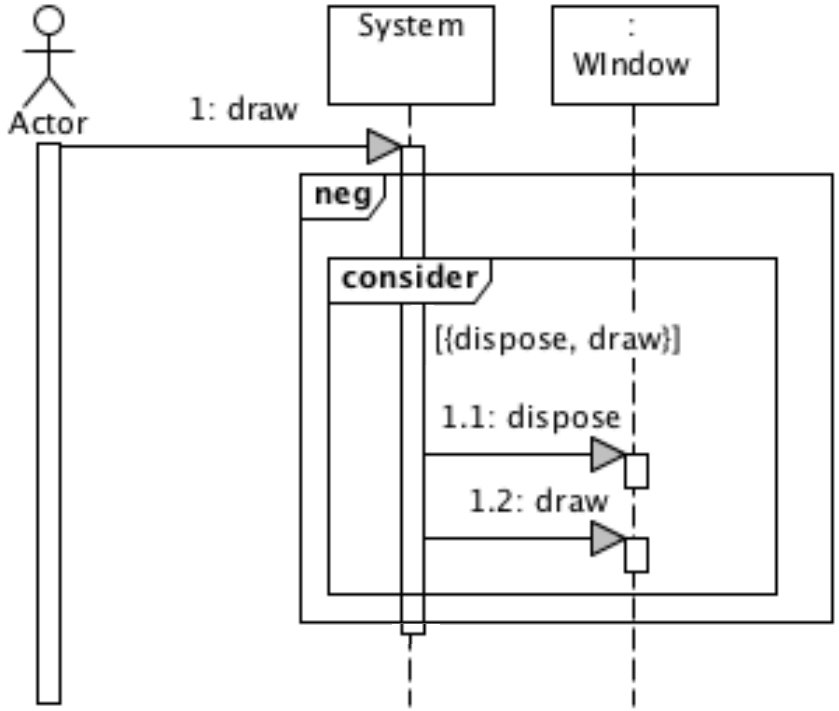
Eventos *req1* e *req3* podem acontecer em paralelo. Evento *req2* acontece antes de evento *req4* (porque ambos vão para C).

Operador *neg*



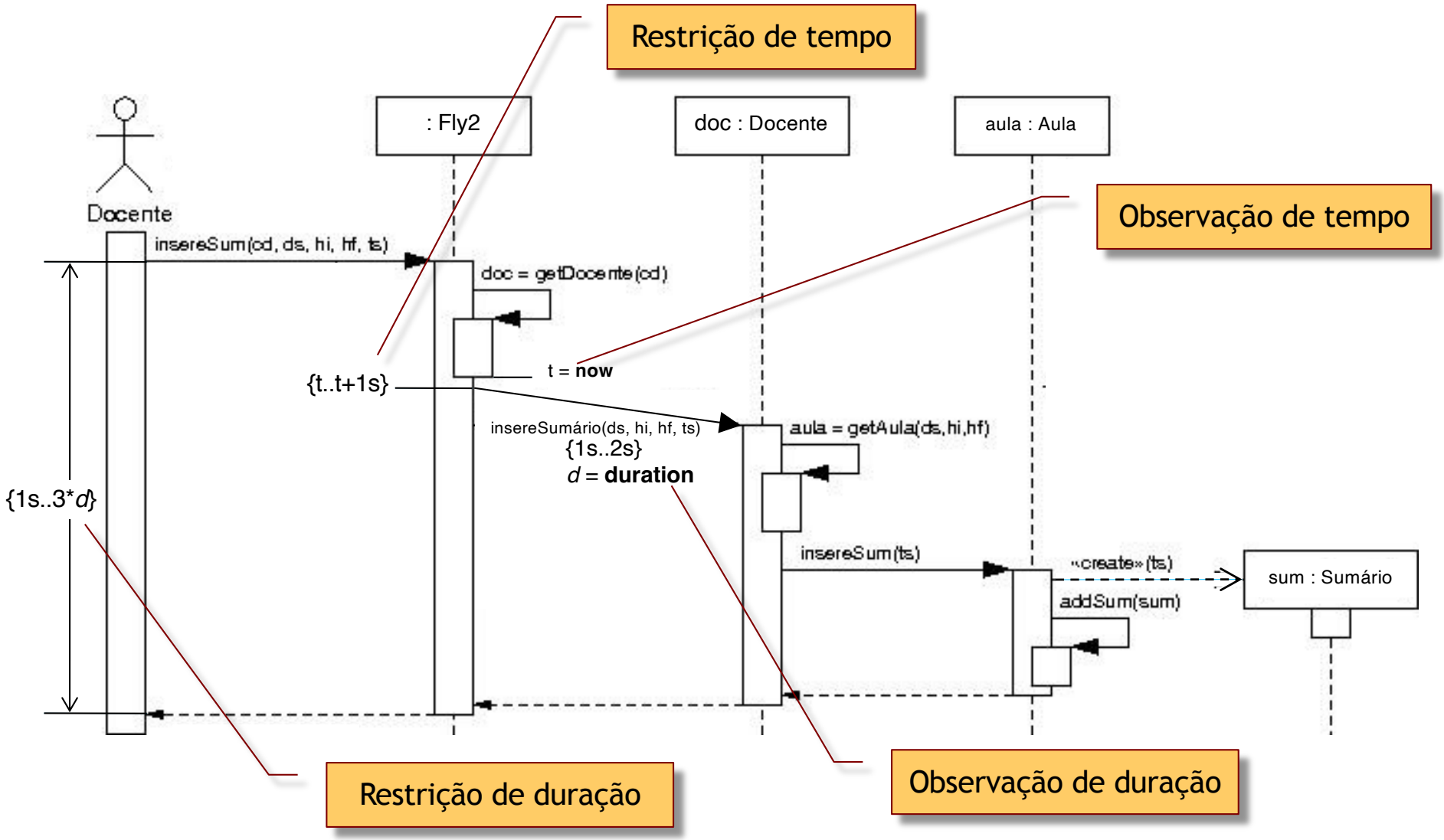
Não é válido desenhar numa janela depois de ela ter sido removida.

Operador *consider*

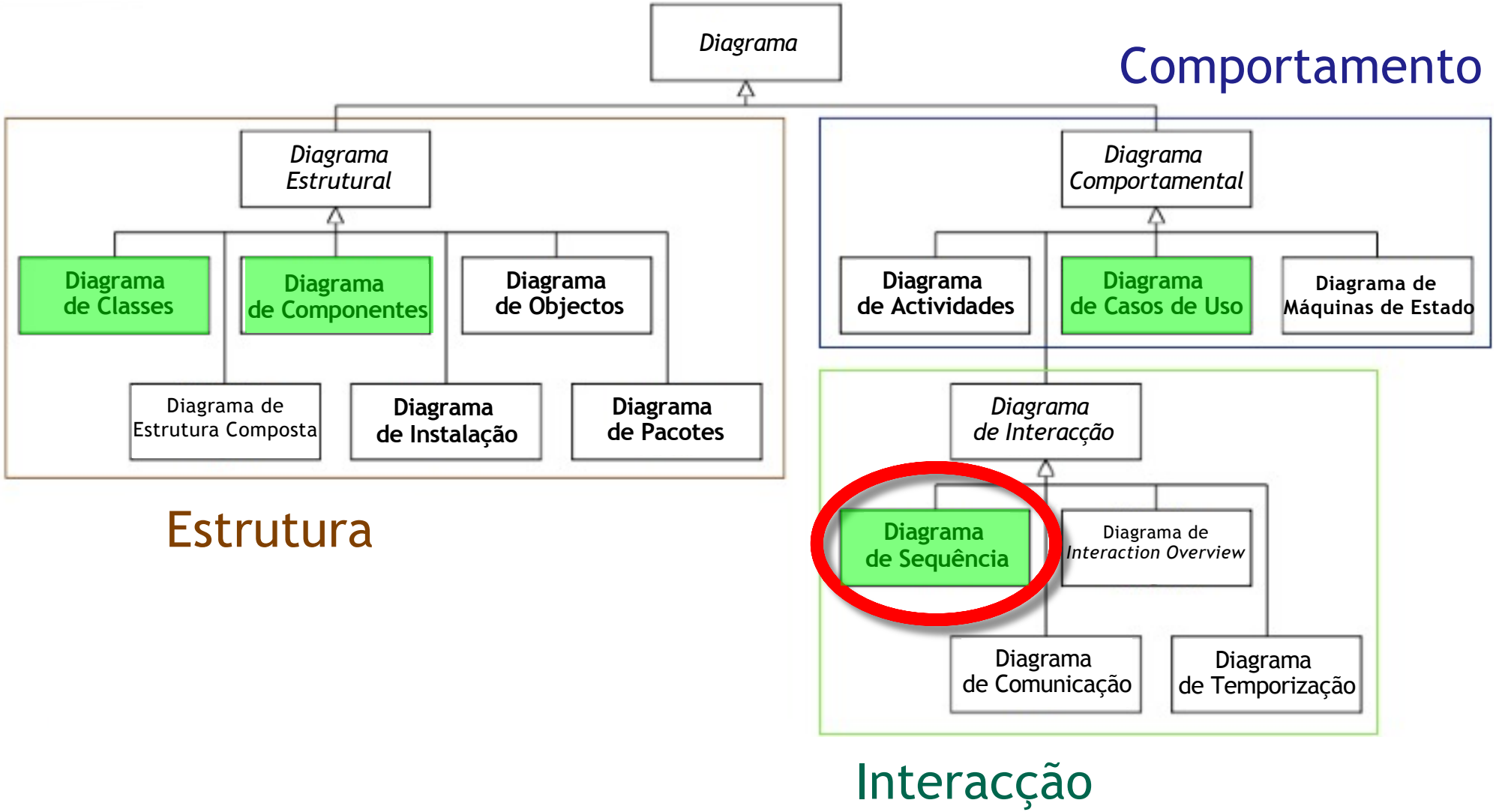


Porque podem existir outros eventos pelo meio...

Restrições de tempo / duração



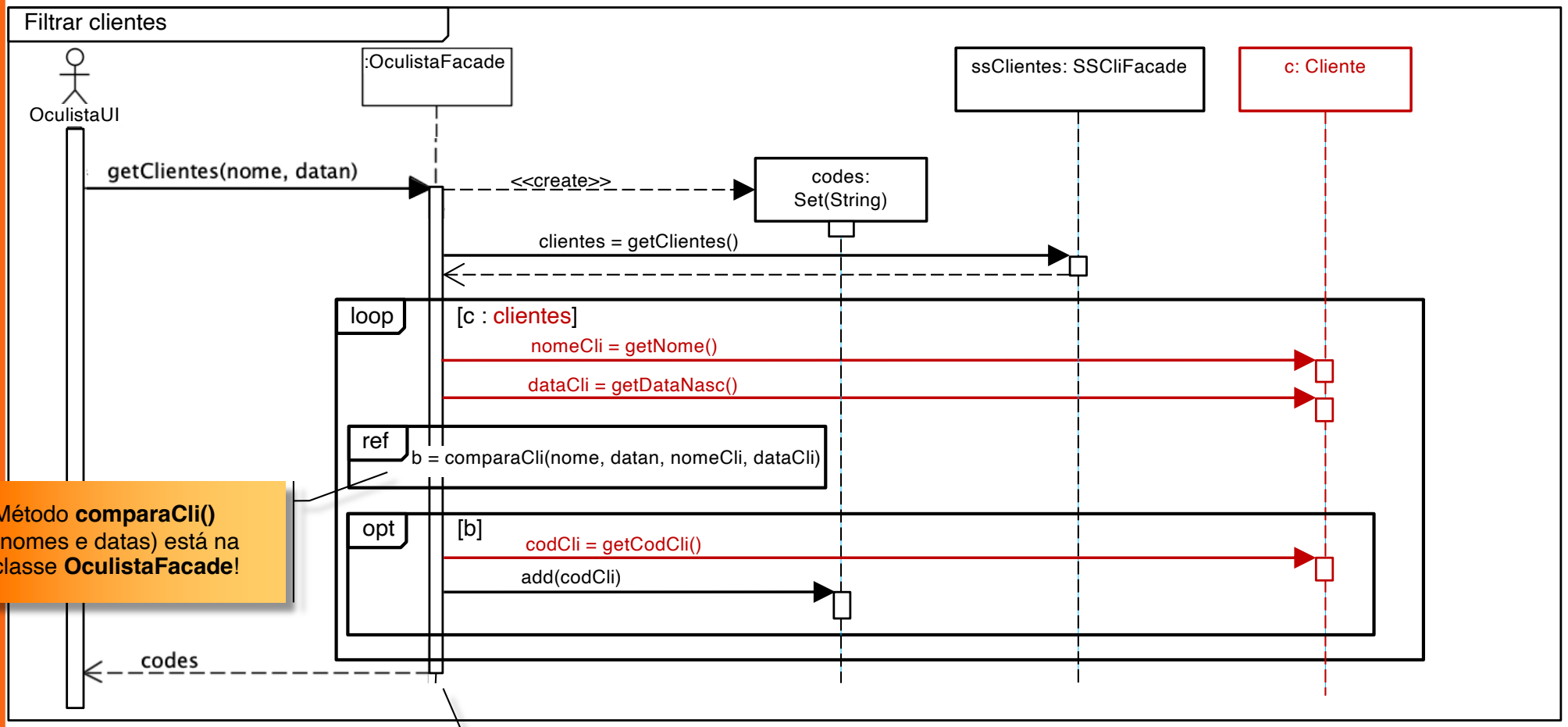
Diagramas da UML 2.x



Princípio SLAP 👋

- Single Level of Abstraction Principle
  - “todo o código dentro de um método deve estar no mesmo nível de abstração” (nível de abstração = grau de detalhe do código)
  - visa tornar o código mais legível e compreensível
- Manter o mesmo nível de abstração
  - Extrair código que representa um nível de abstração mais baixo para outros métodos; invocar esses métodos no método principal
  - Método principal fica mais claro e simples
  - Métodos auxiliares ficam mais coesos e reutilizáveis
- Contribui para a manutenção, testabilidade e modularidade do sistema

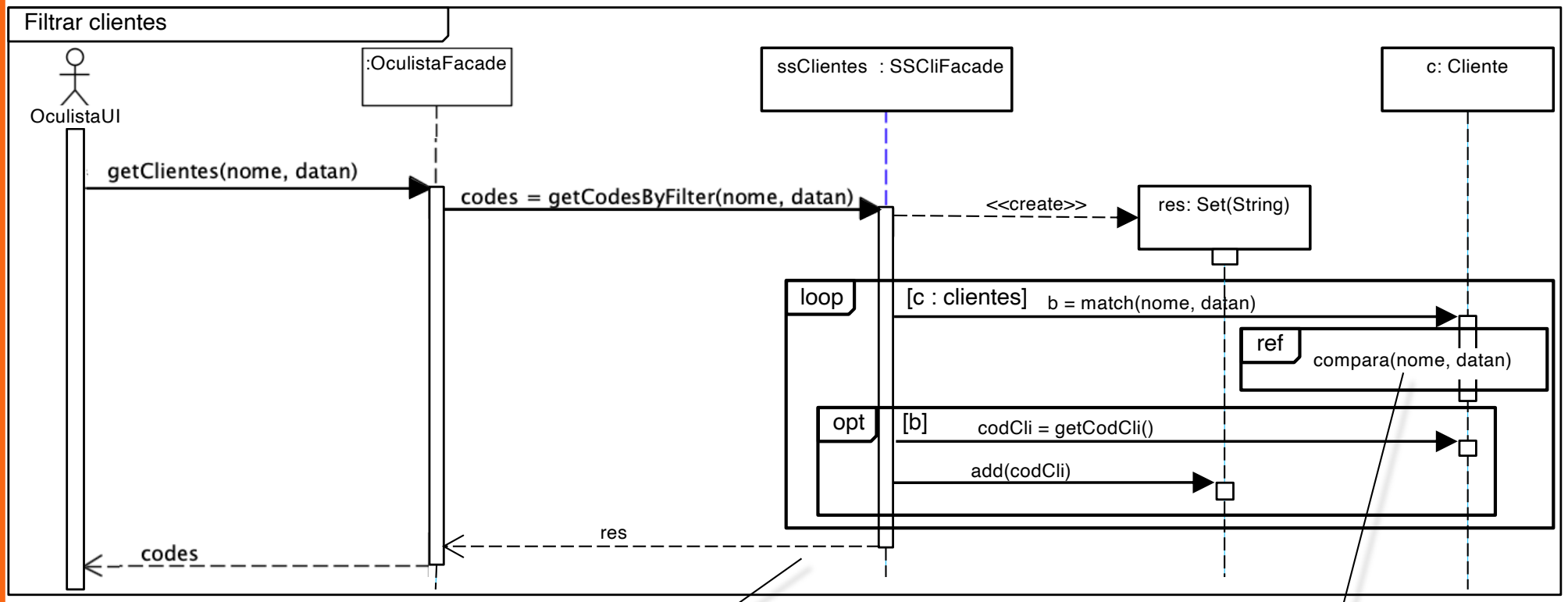
Distribuição de responsabilidades



Método **comparaCli()** (nomes e datas) está na classe **OculistaFacade**!

Estamos a concentrar a lógica de processamento numa única classe – **má prática**! Não é OO!! (algoritmo mais complexo e mais dependências!)

Distribuição de responsabilidades



Processamento acontece na classe onde os dados existem – boa prática!

Método **compara()** pasa a estar na classe **Cliente**! (onde o nome e data de nascimento do cliente estão)

- Devemos fazer um diagrama de sequência…
  a) para cada Use Case
  b) para cada passo dos Use Case
  c) depois de definida a arquitectura, para cada operação das classes
  d) para cada operação da lógica de negócio identificada nos Use Case

- Escolha apenas uma opção!

Singleton Pattern

- Garantir que só é criada uma instância da classe….

```
public class Singleton {
    private static Singleton instancia = null;

    public static Singleton getInstance() {
        if (Singleton.instancia==null)
            Singleton.instancia = new Singleton();
        return Singleton.instancia;
    }

    private Singleton() {}
}
```