

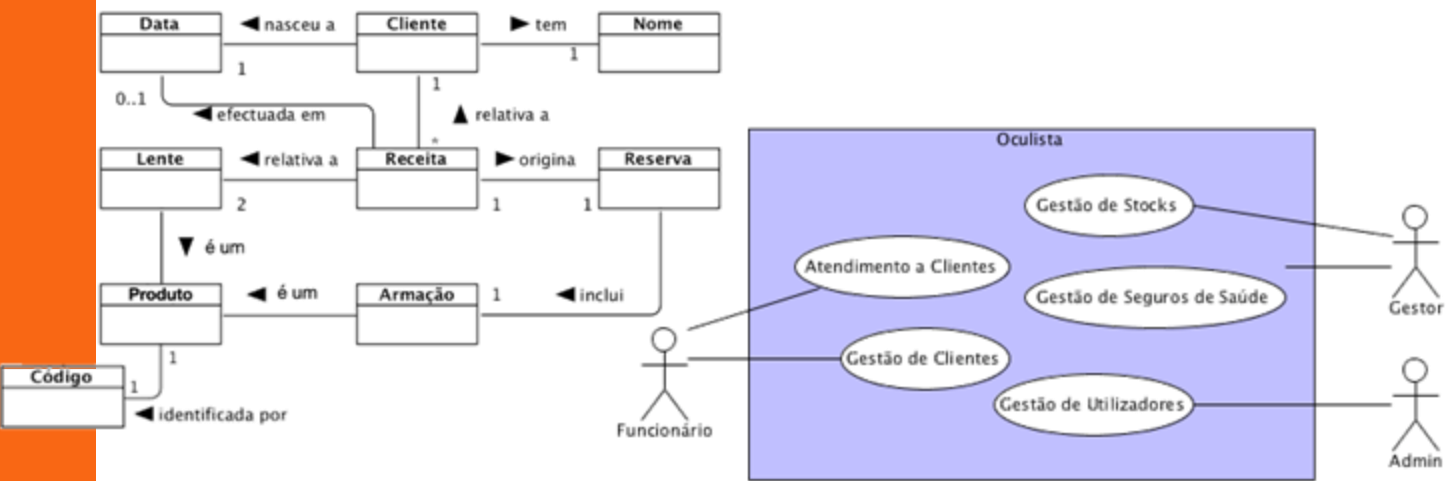


# Desenvolvimento de Sistemas Software

## Modelação Comportamental (Diagramas de Sequência)



# Em resumo...



**Use Case:** Reservar armação e lentes

**Descrição:** Funcionário regista a uma reserva de armação e lentes

**Pós-condição:** Reserva fica registada

**Fluxo normal:**

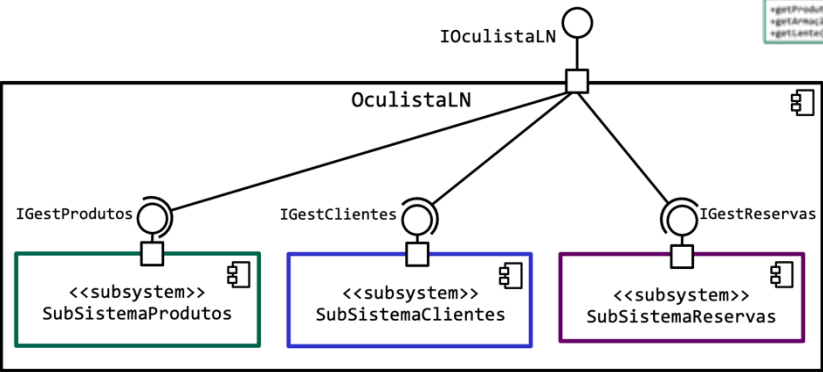
1. Funcionário indica nome e/ou data de nascimento do cliente
2. Sistema procura clientes
3. Sistema apresenta lista de clientes
4. Funcionário selecciona cliente
5. Sistema procura cliente
6. Sistema apresenta detalhes do cliente
7. Funcionário confirma cliente
8. Sistema procura produtos e apresenta 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 reservas 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

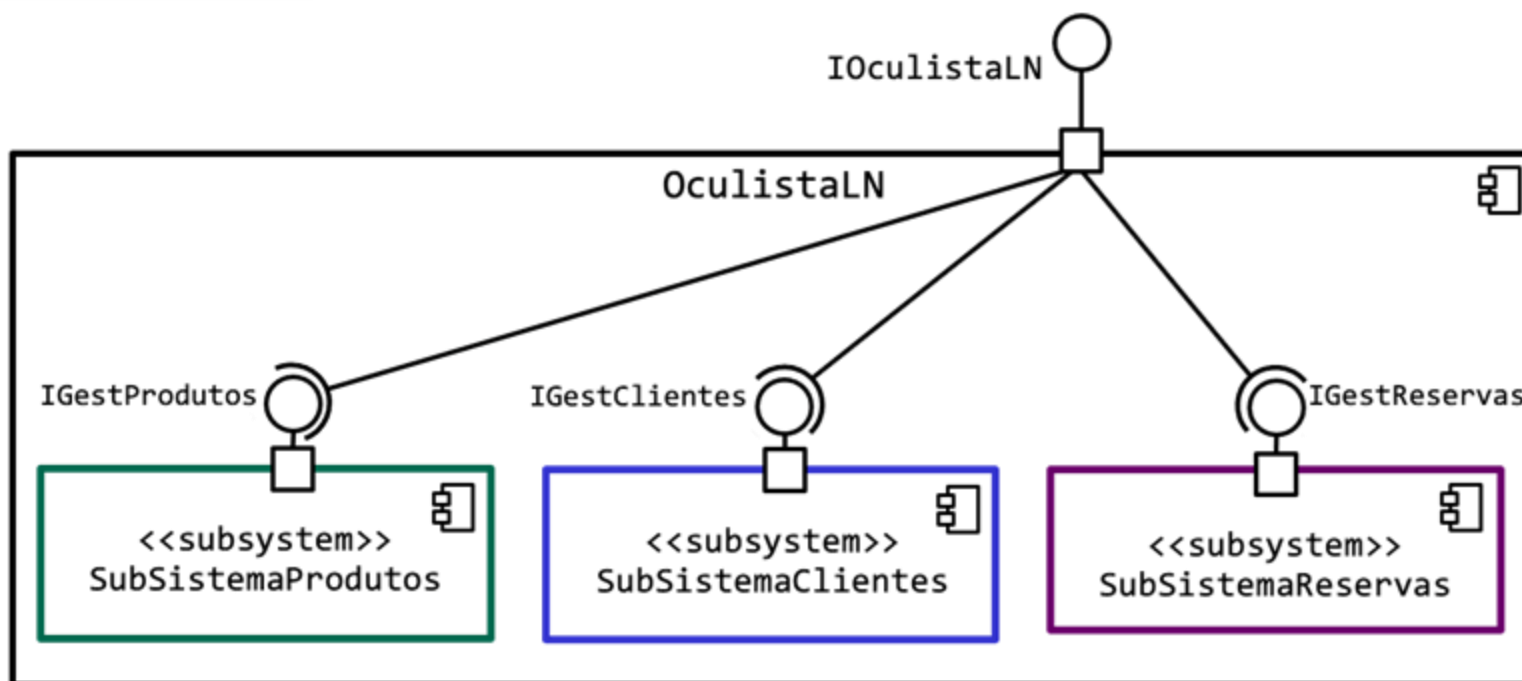
**Fluxo de excepção:** [cliente não quer produto] (passo 12)

- 11.1. Funcionário rejeita produtos
- 11.2. Sistema termina processo





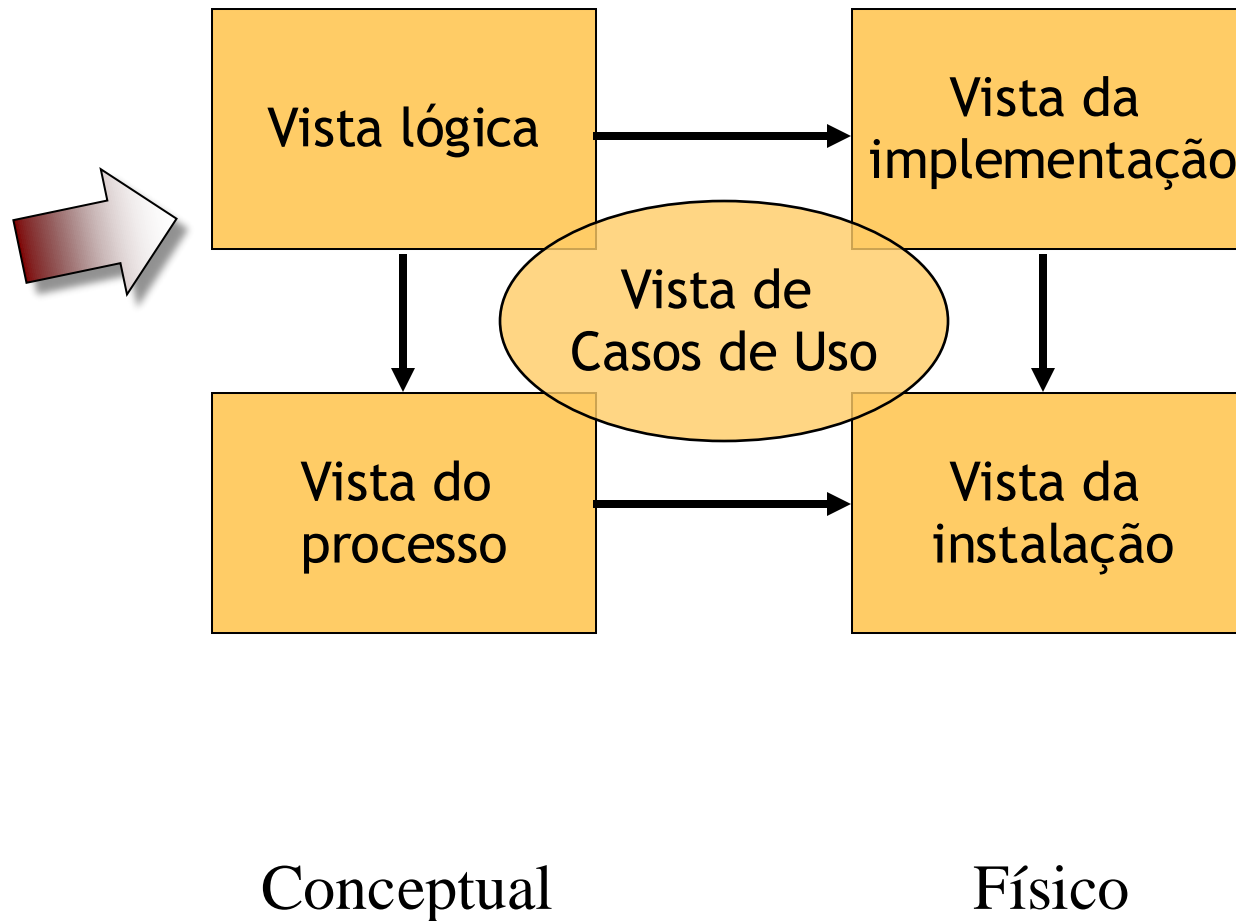
## Em resumo...



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



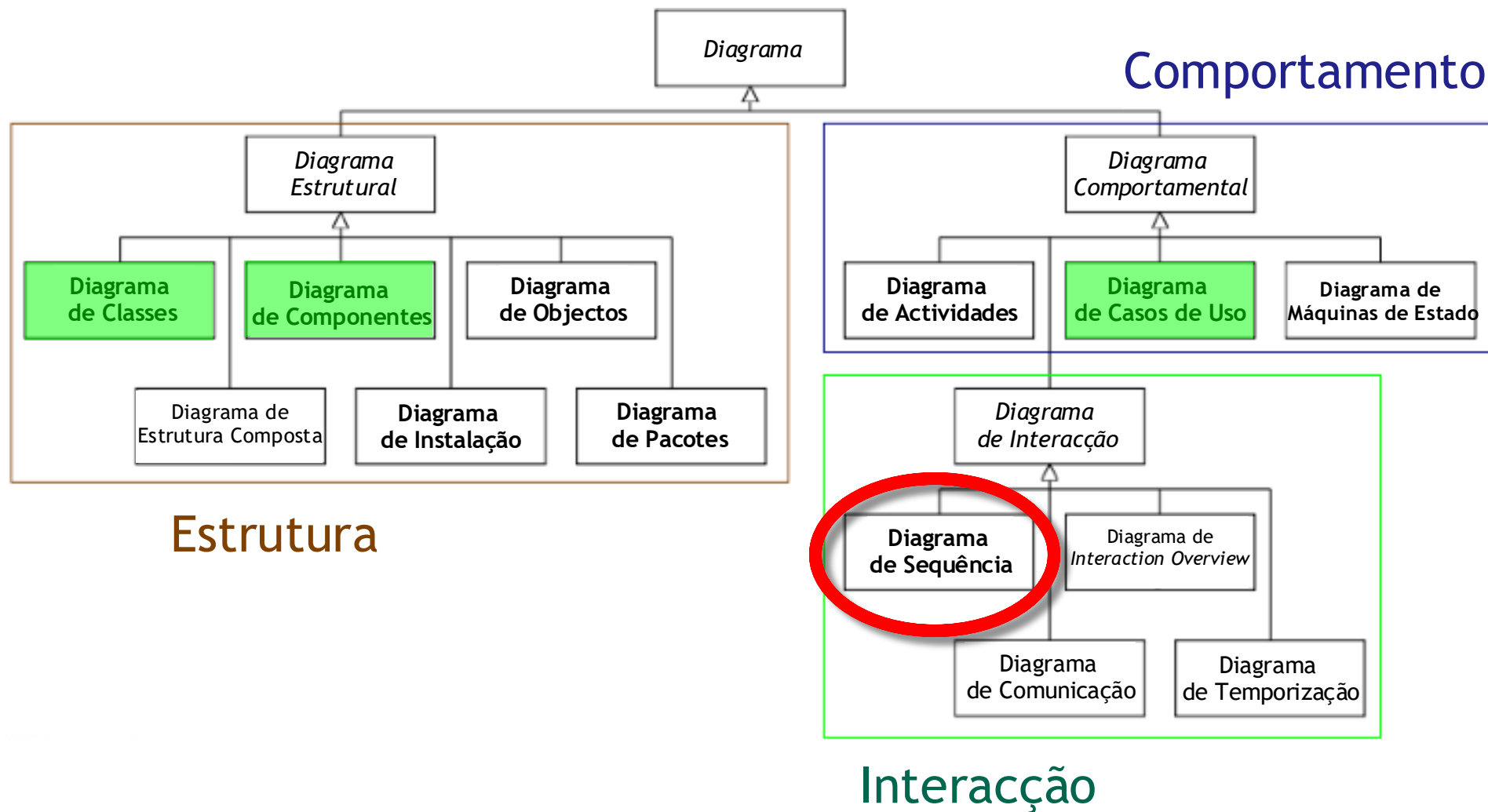
## Onde estamos...



(Kruchten, 1995)

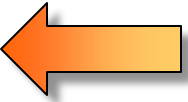


# Diagramas da UML 2.x





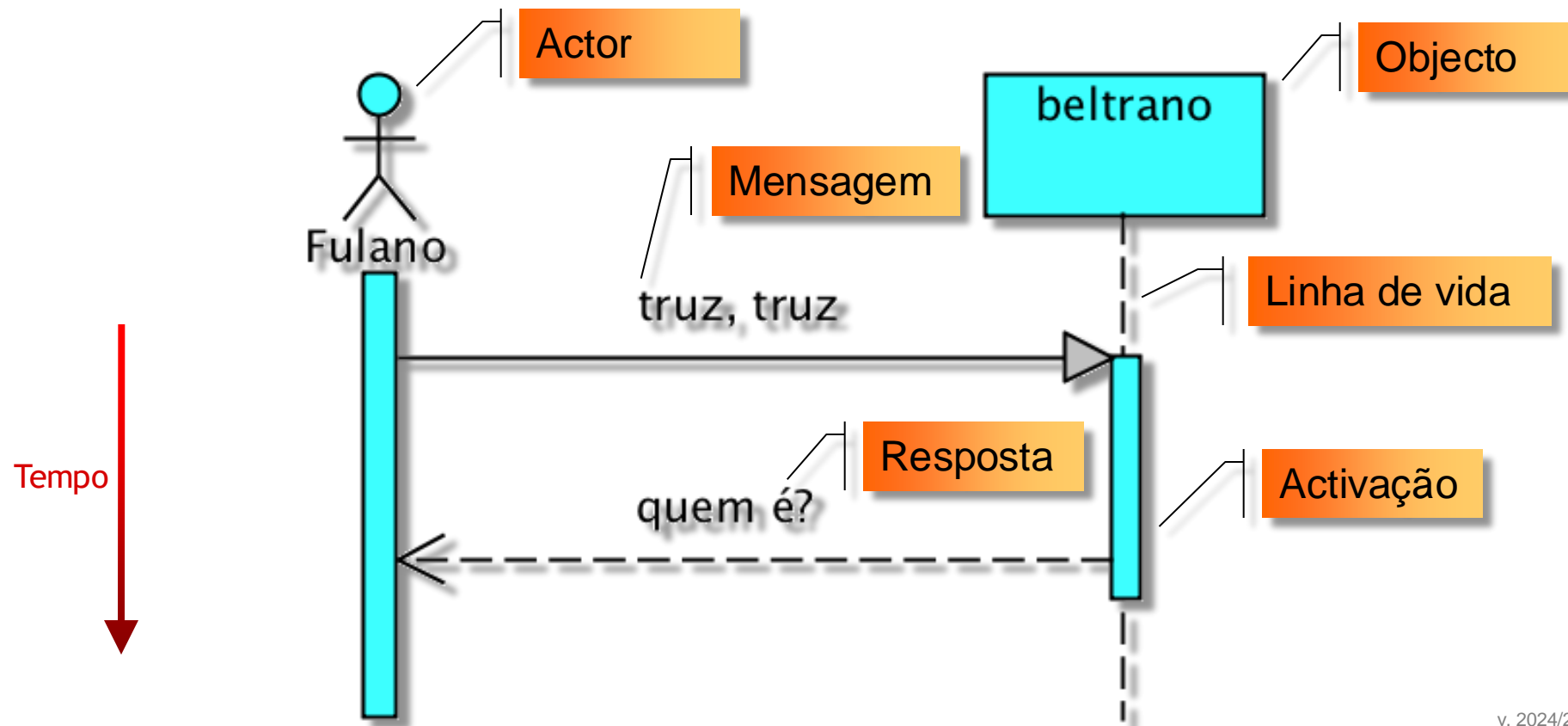
# Diagramas de Interação

- 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

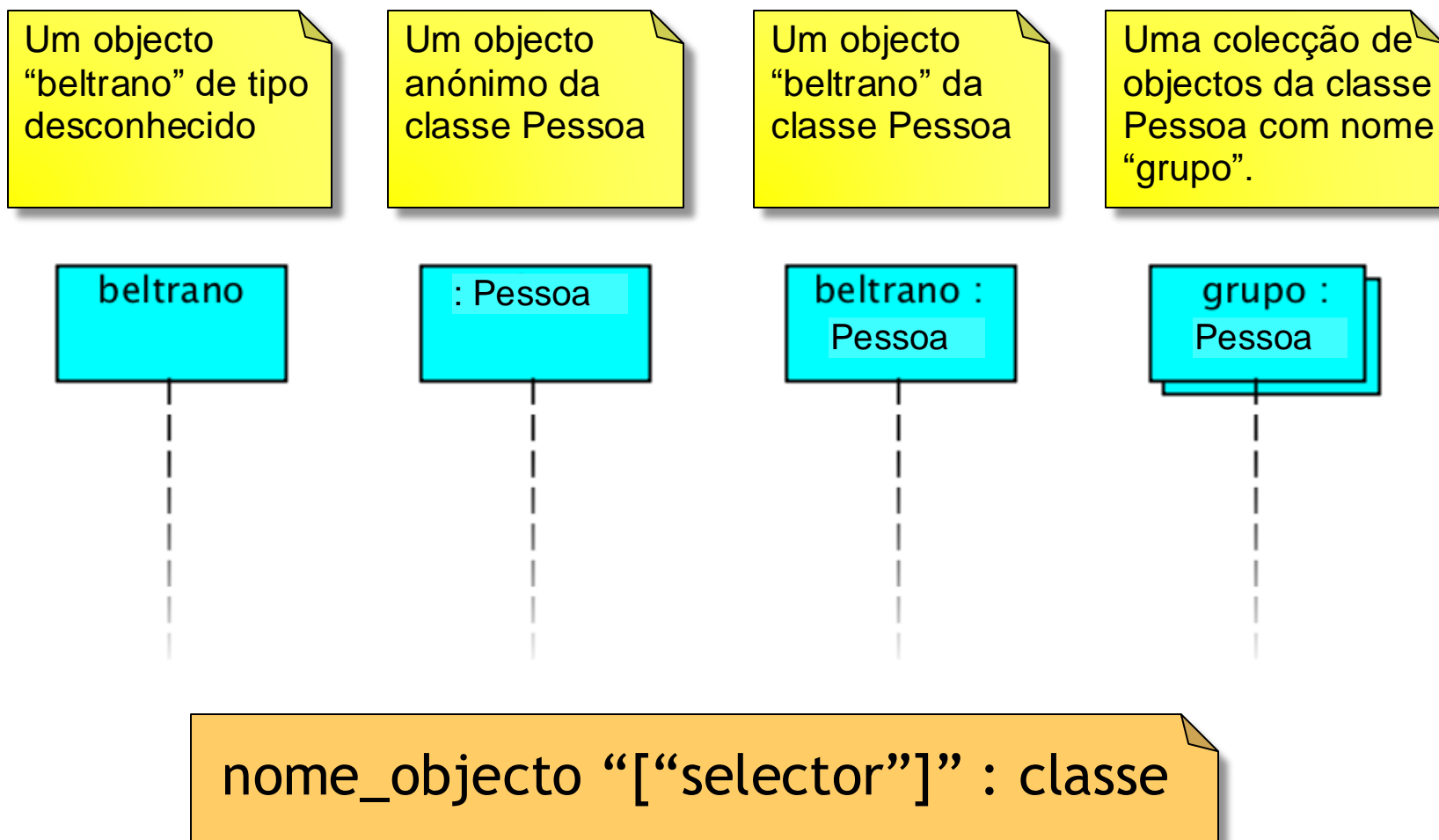
- representam as interações entre objectos através das mensagens que são trocadas entre eles
- a ênfase é colocada na ordenação temporal das mensagens
- permitem analisar a distribuição de “responsabilidade” pelas diferentes entidades (analisar onde está a ser efectuado o processamento)





# Diagramas de Sequência - notação essencial

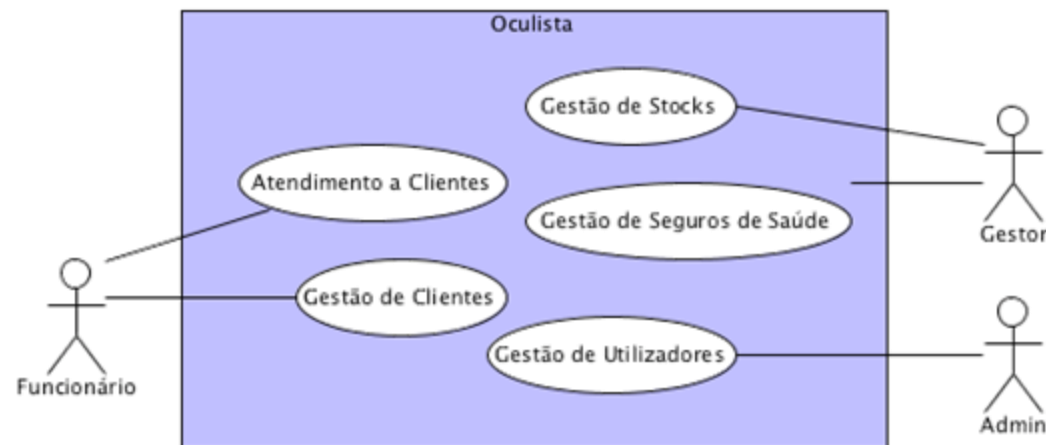
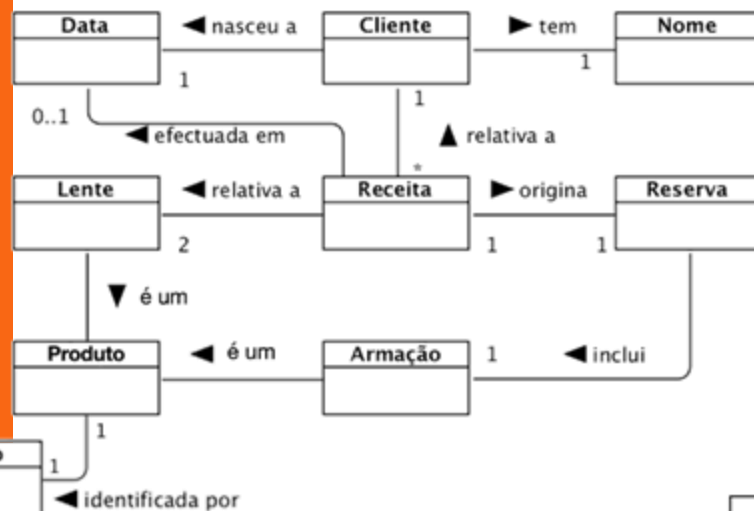
## Objectos







# Um exemplo...



## Use Case: Reservar armação e lentes

**Descrição:** Funcionário regista uma reserva de armação e lentes

**Pós-condição:** Reserva fica registada

### Fluxo normal:

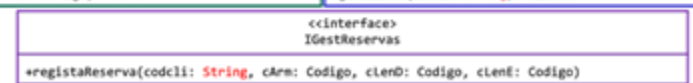
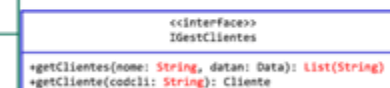
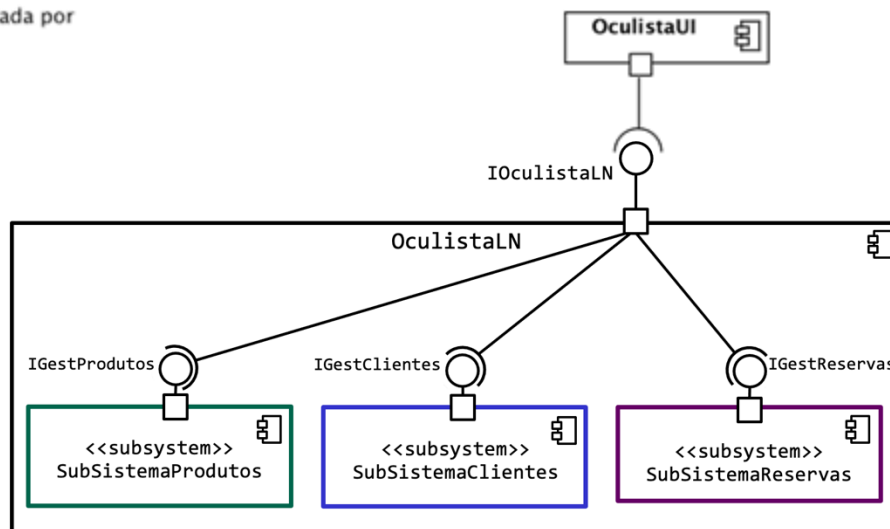
1. Funcionário indica nome e/ou data de nascimento do cliente
2. Sistema procura clientes
3. Sistema apresenta lista de clientes
4. Funcionário selecciona cliente
5. Sistema procura cliente
6. Sistema apresenta detalhes do cliente
7. Funcionário confirma cliente
8. Sistema procura produtos e apresenta 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 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

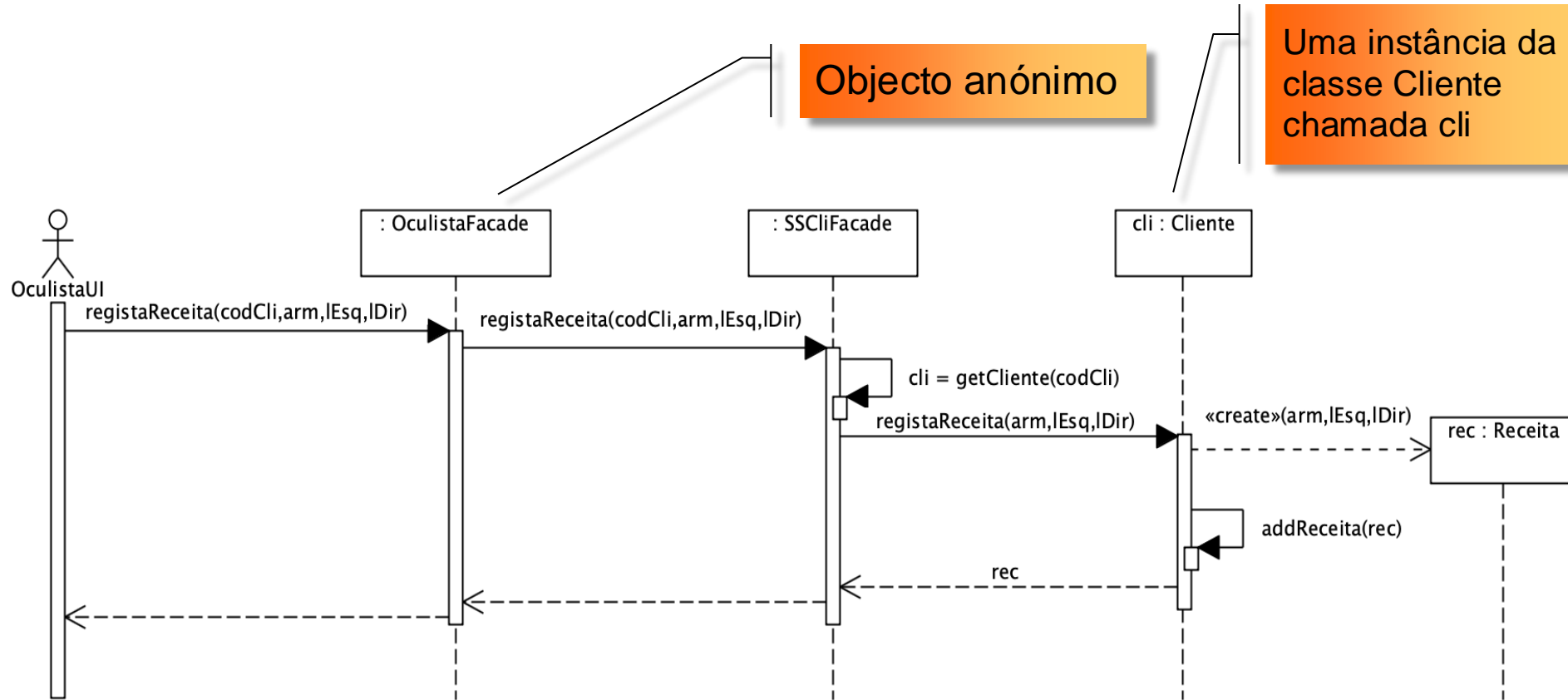




# Diagramas de Sequência - notação essencial

Uma instância da classe Cliente chamada cli

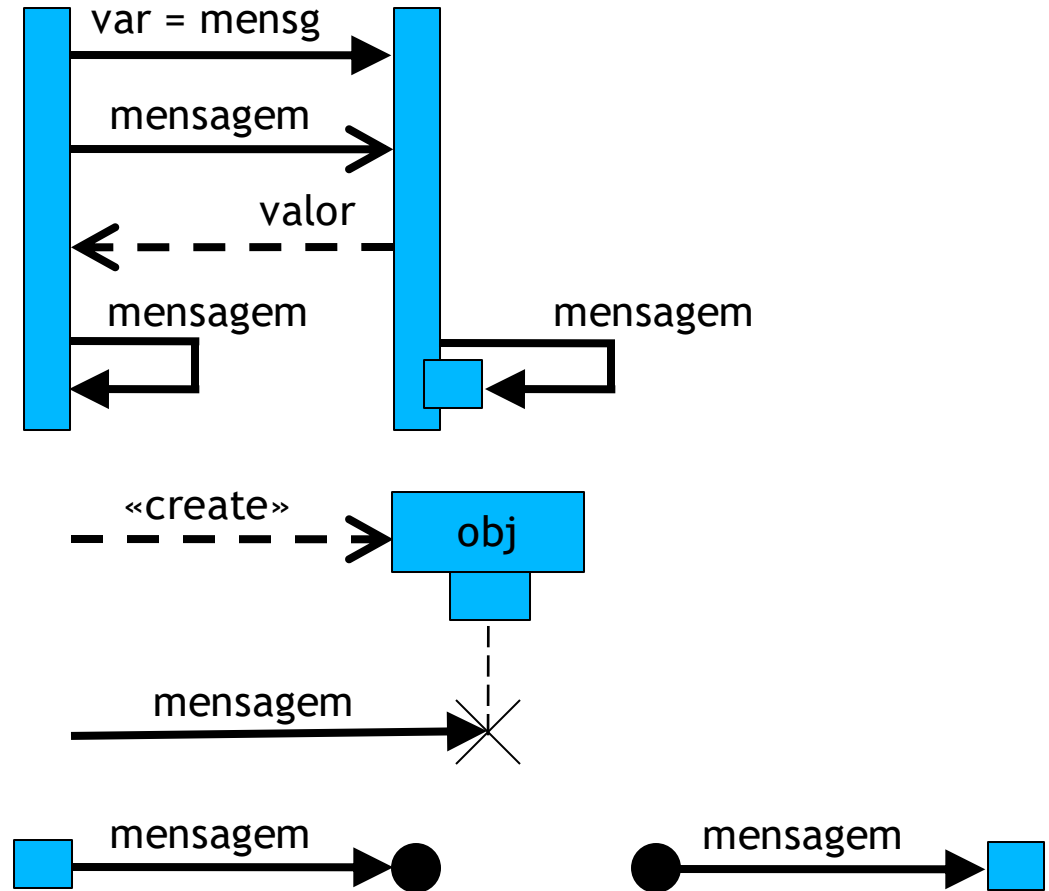
Objecto anónimo





# Mensagens

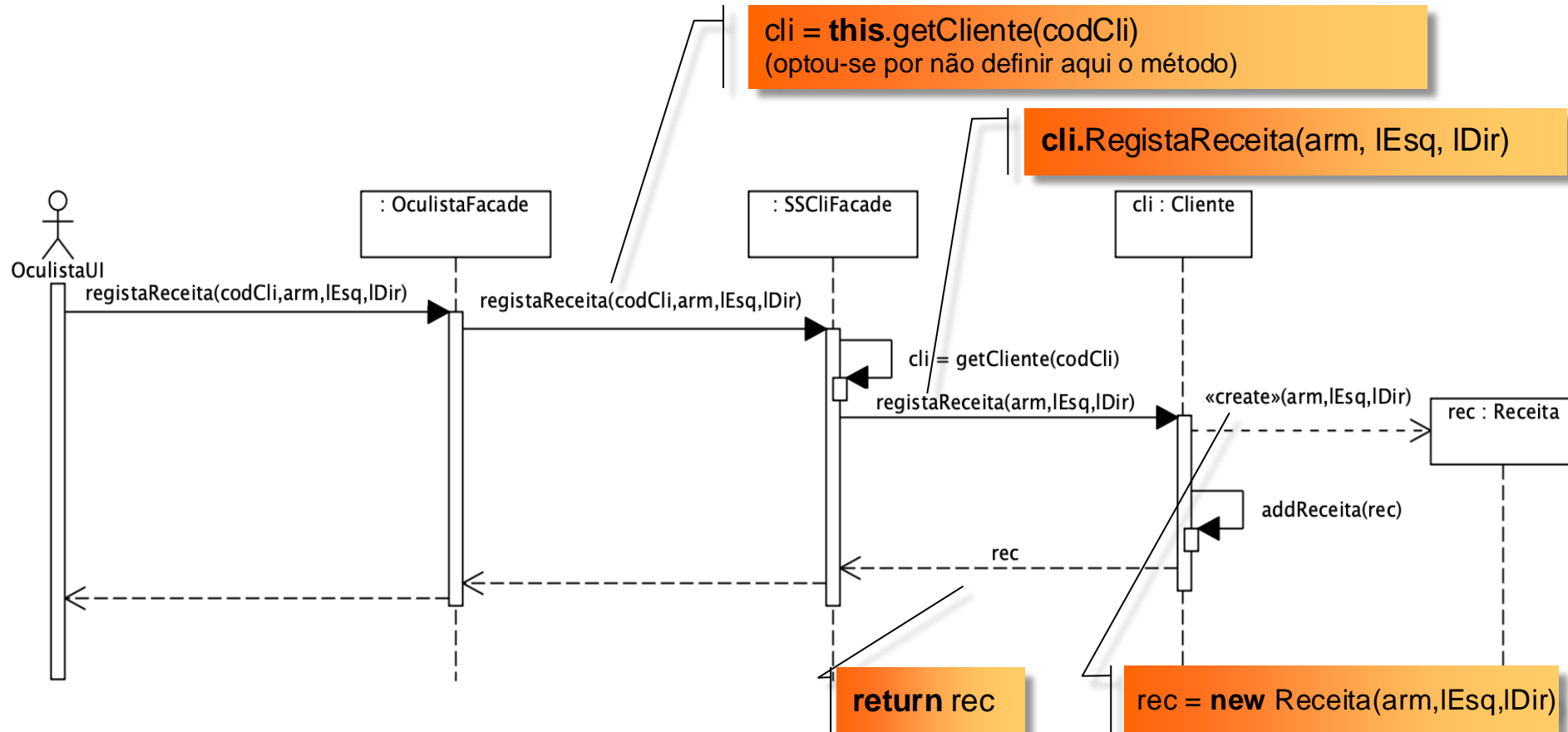
- invocação síncrona
- invocação assíncrona
- return/resultado
- self messages
- criar objectos
- destruir objectos
- lost/found messages



[ atributo '=' ] nome\_da\_operação\_sinal [ argumentos ] [ ':' tipo\_resultado ]



# Diagramas de Sequência - notação essencial

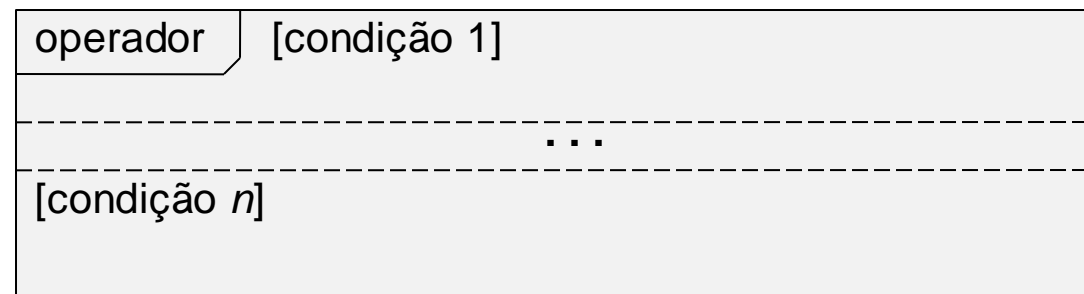


- Todas as invocações são síncronas.
- Dois dos métodos não estão aqui definidos (+ construtor).
- **Atenção!** O objecto que envia a mensagem tem que “conhecer” o objecto a quem a envia.



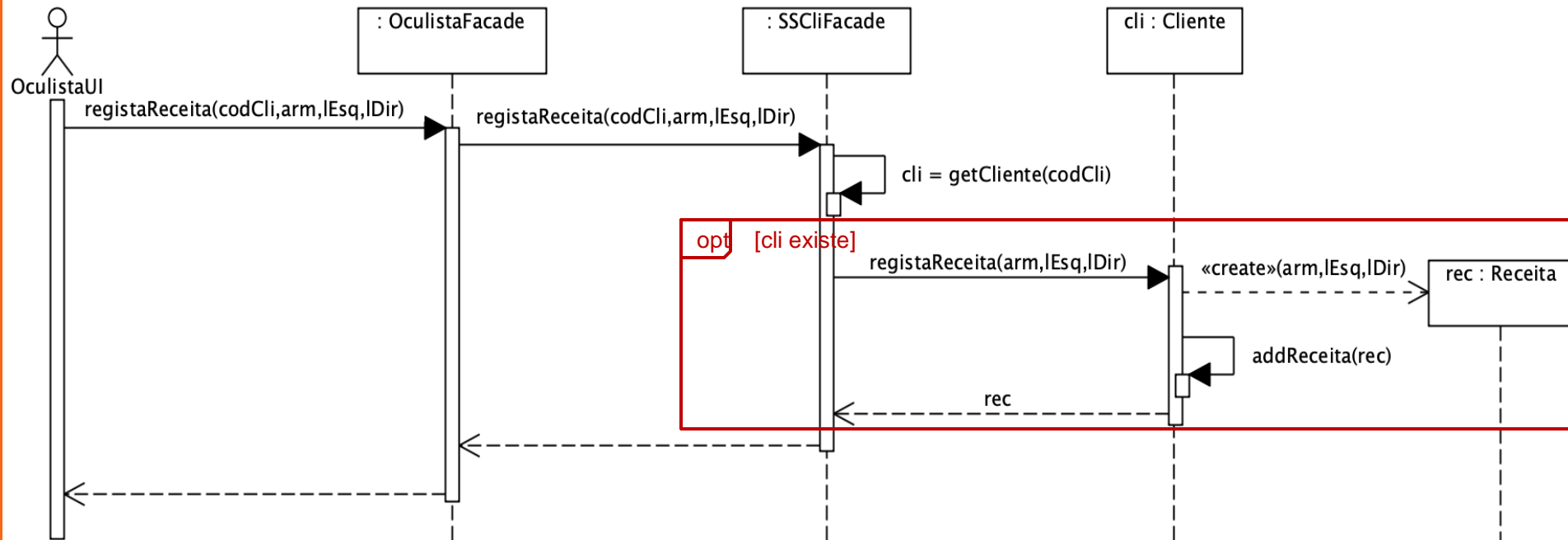
# Diagramas de Sequência - fragmentos combinados

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



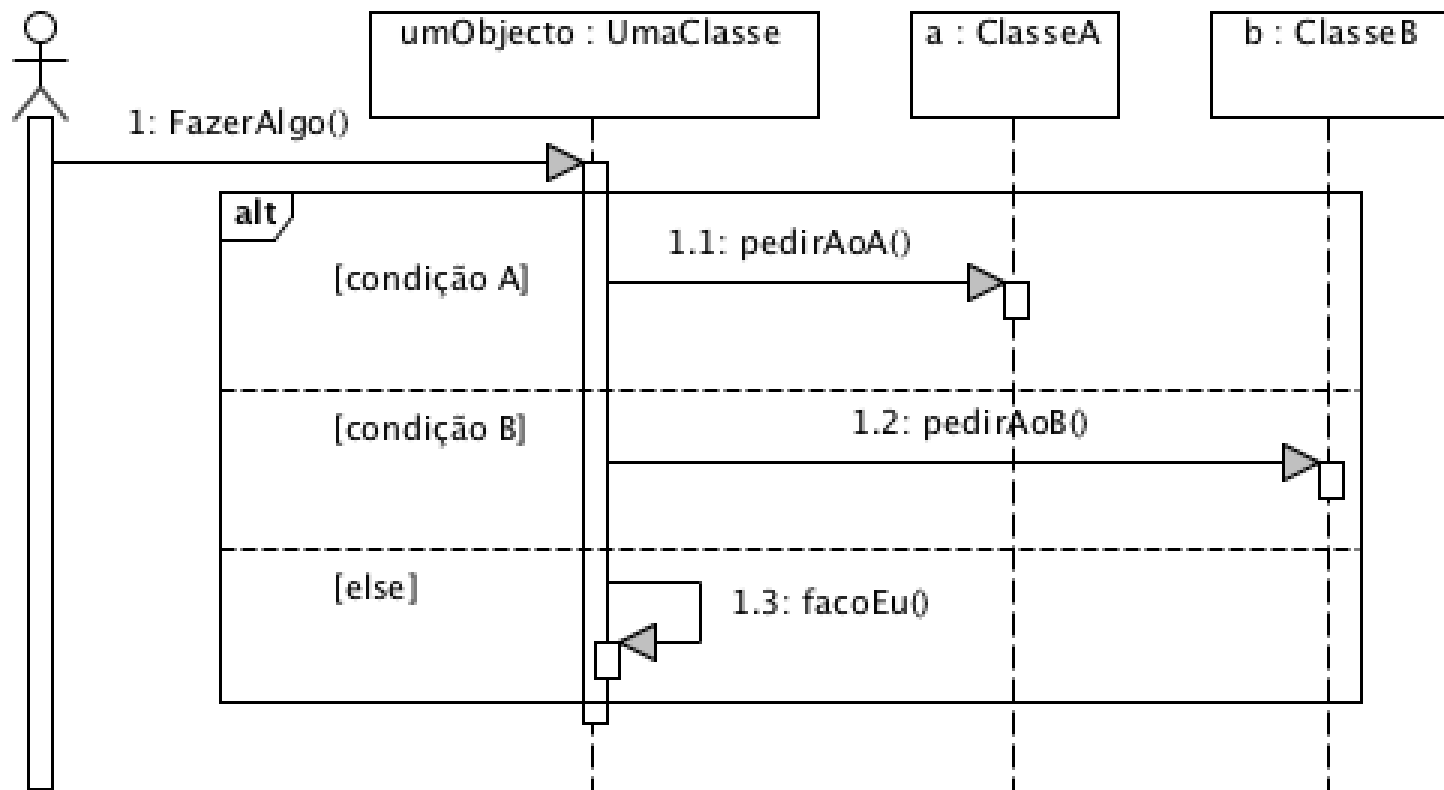
- Operadores mais comuns
  - **alt** - define fragmentos alternativos (mutuamente exclusivos)
  - **loop** / **loop(*n*)** - fragmento é repetido enquanto a guarda for verdadeira / ***n*** vezes
  - **opt** - fragmento opcional (ocorre se a guarda for verdadeira)
  - **break** - termina o fluxo
  - **ref**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

# Operador *opt*



Registro só é efectuado se cliente existe.

## Operador *alt*



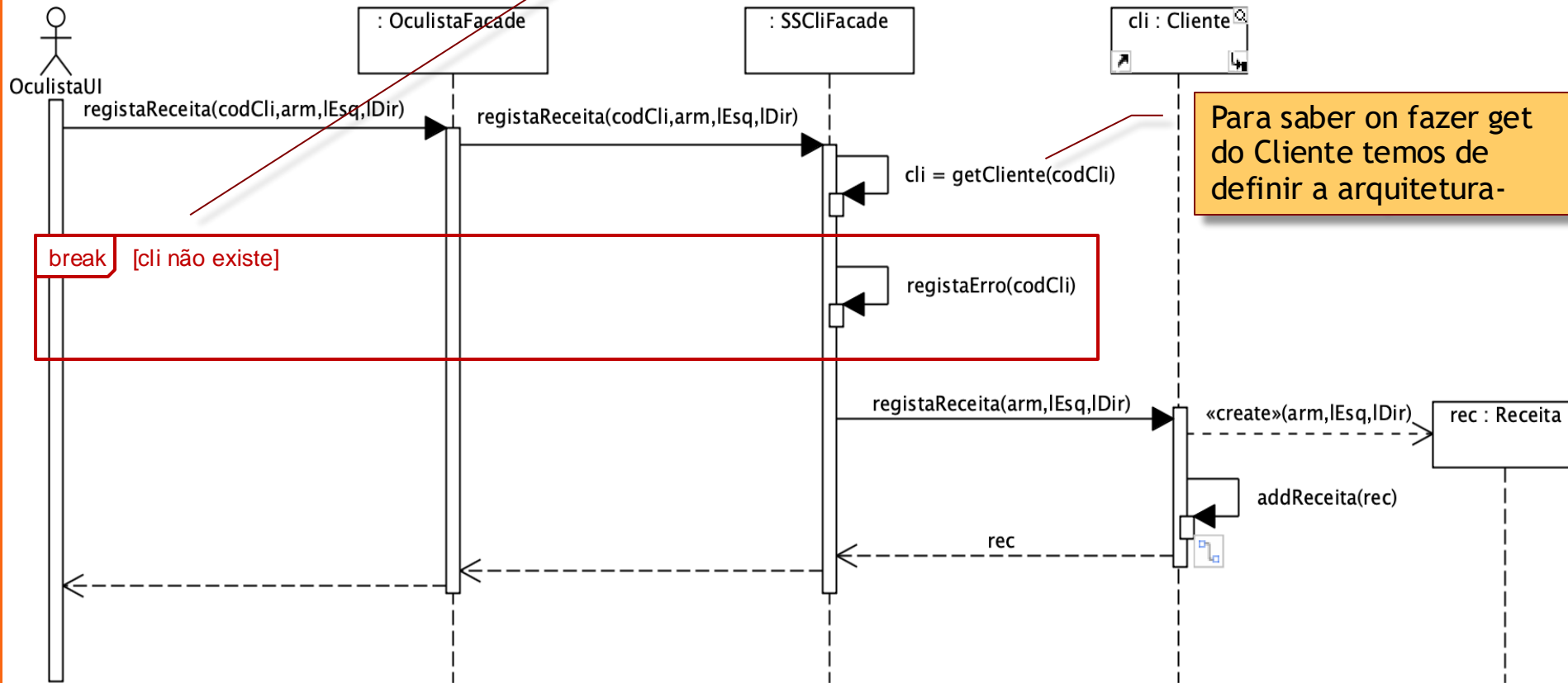
- Os fluxos possíveis são mutuamente exclusivos, pelo que apenas um deles será seguido.
- Se mais que uma condição se verificar, não está definido qual acontece.



# Operador *break*

Se cliente não existe,  
registra erro e termina.

Para saber on fazer get  
do Cliente temos de  
definir a arquitetura-

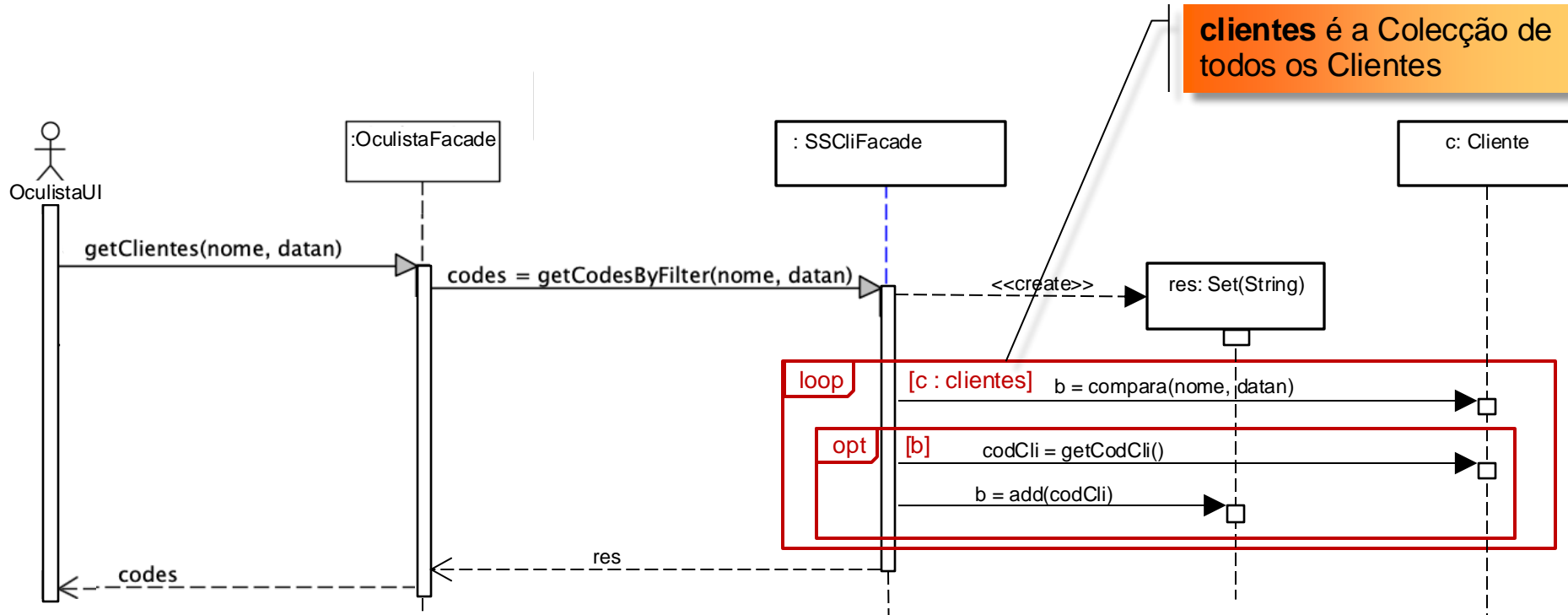


Registo só é efectuado se cliente existe.





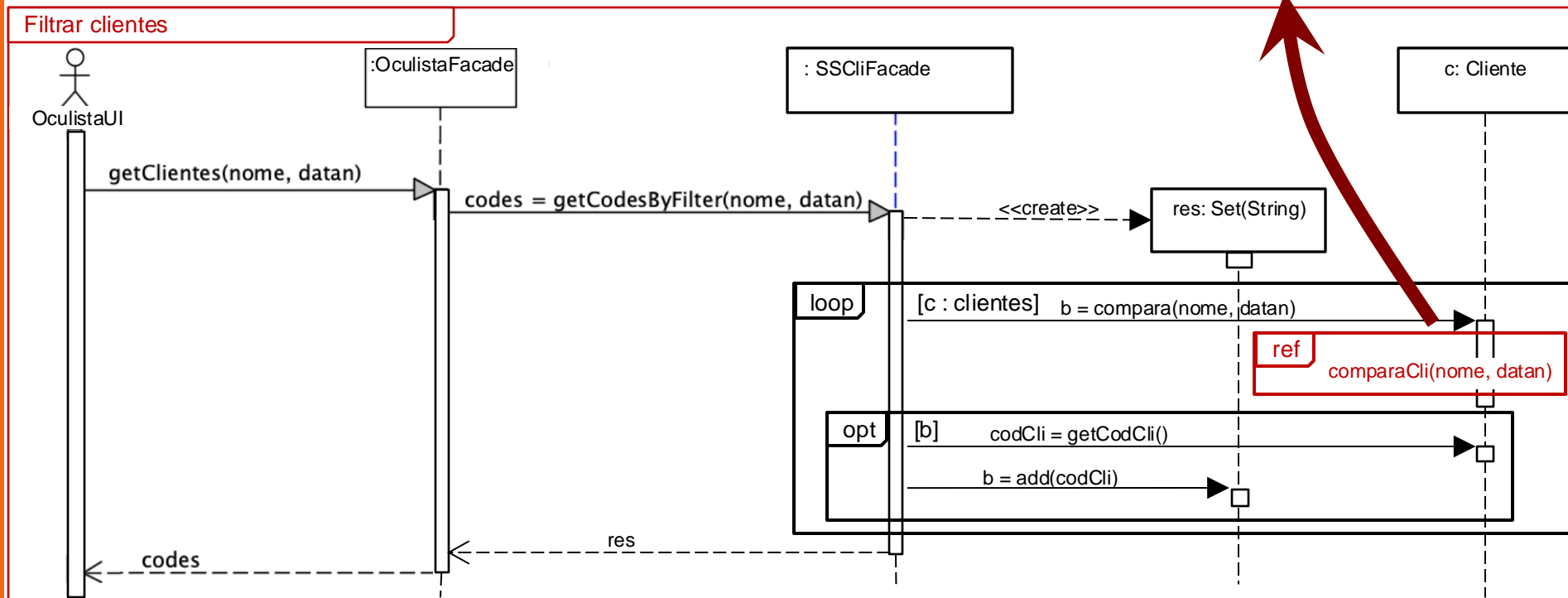
# Operador loop



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



# Operador *ref*



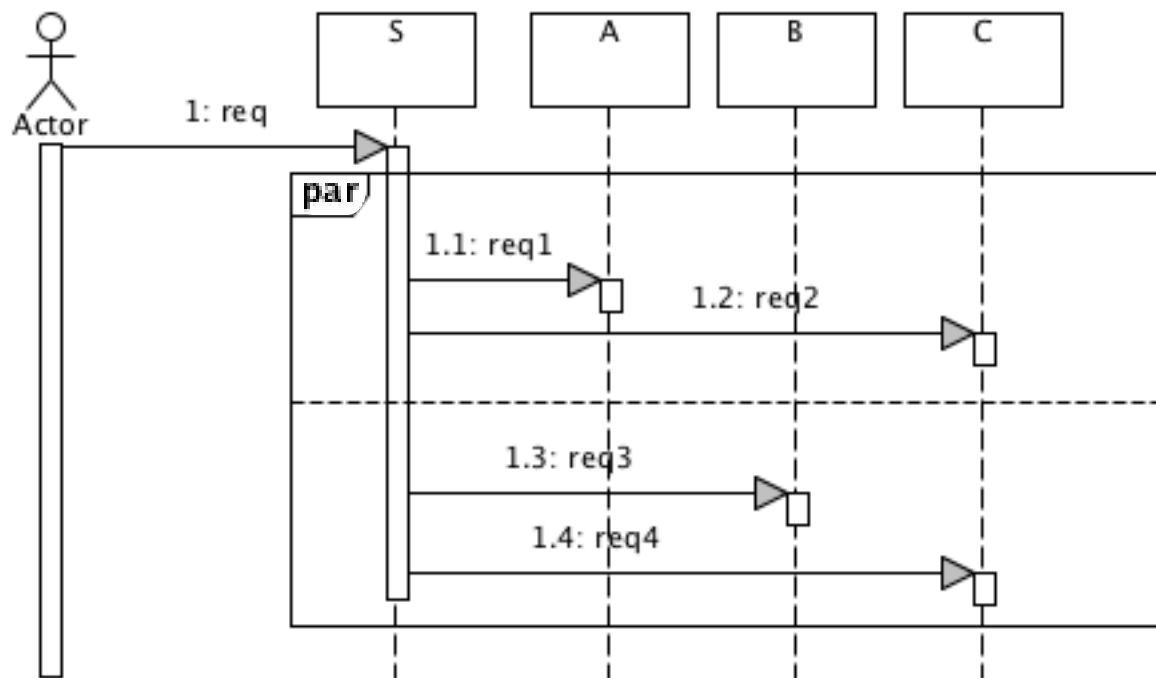
- Todos os diagrama devem ter um nome
- Um SD pode reutilizar outros SD referenciando-os num fragmento com o operador ***ref*** – permite estruturar os modelos



# Outros operadores

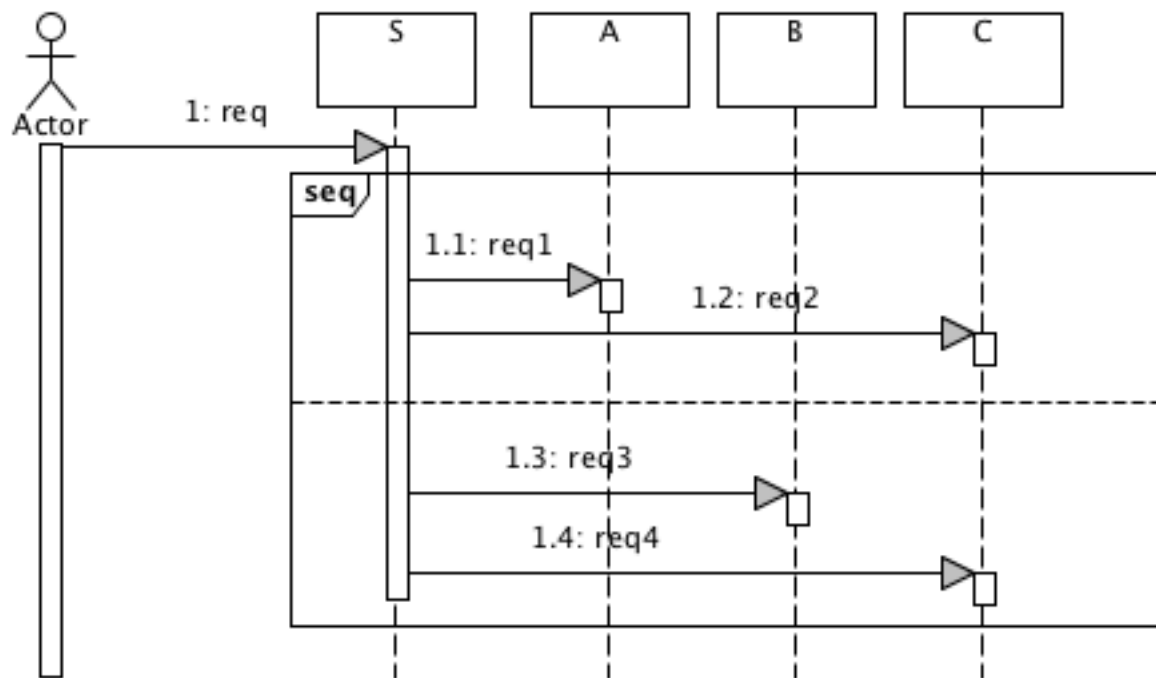
- **critical** - o operando executa de forma atômica
- **par** - os operandos executam em paralelo
- **seq** (sequenciação fraca) - todos os operandos executam em paralelo, mas eventos enviados a uma mesma linha de vida acontecem na mesma sequência dos operandos
- **strict** - os operandos executam em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

# Operador *par*



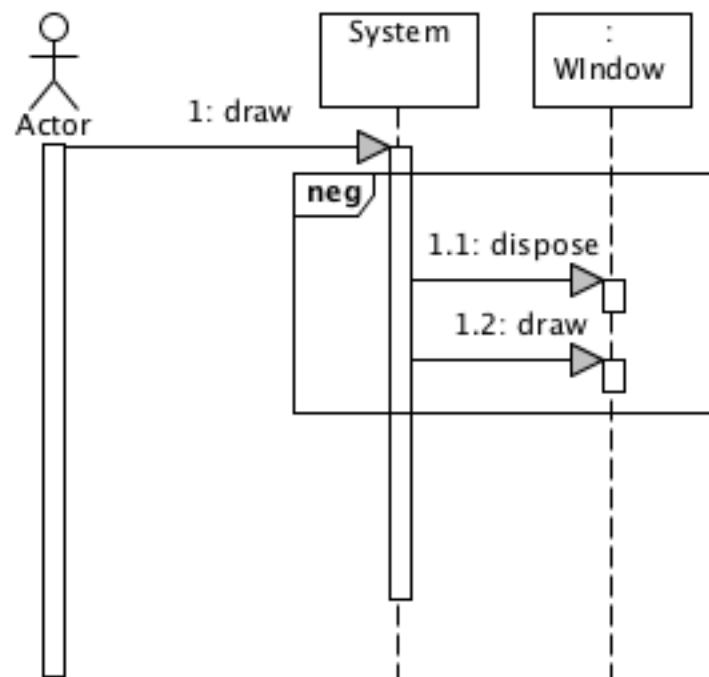
**Eventos *req1* e *req2* podem acontecer em paralelo com eventos *req3* e *req4*. Nenhuma ordem é imposta.**

## Operador seq



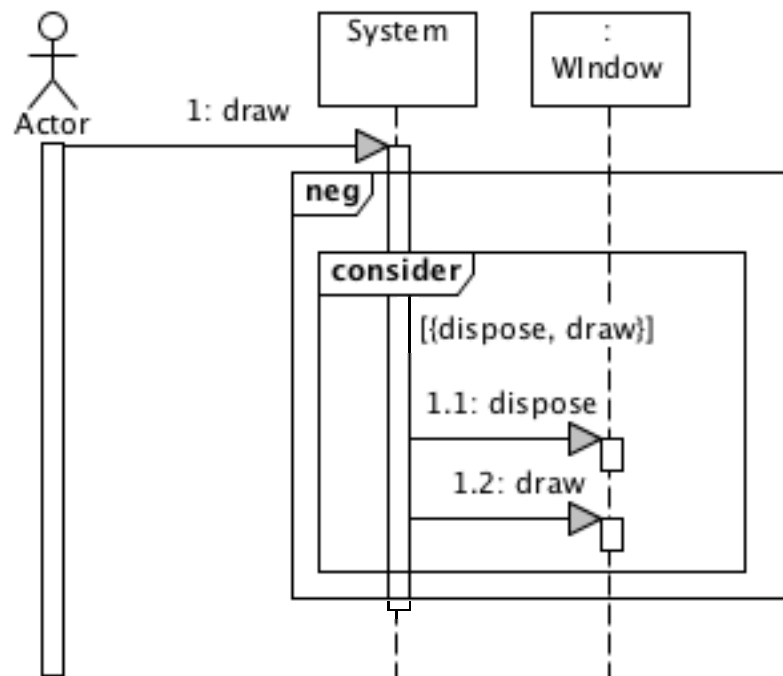
**Eventos *req1* e *req3* podem acontecer em paralelo. Evento *req2* acontece antes de evento *req4* (porque ambos vão para C).**

## Operador *neg*



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

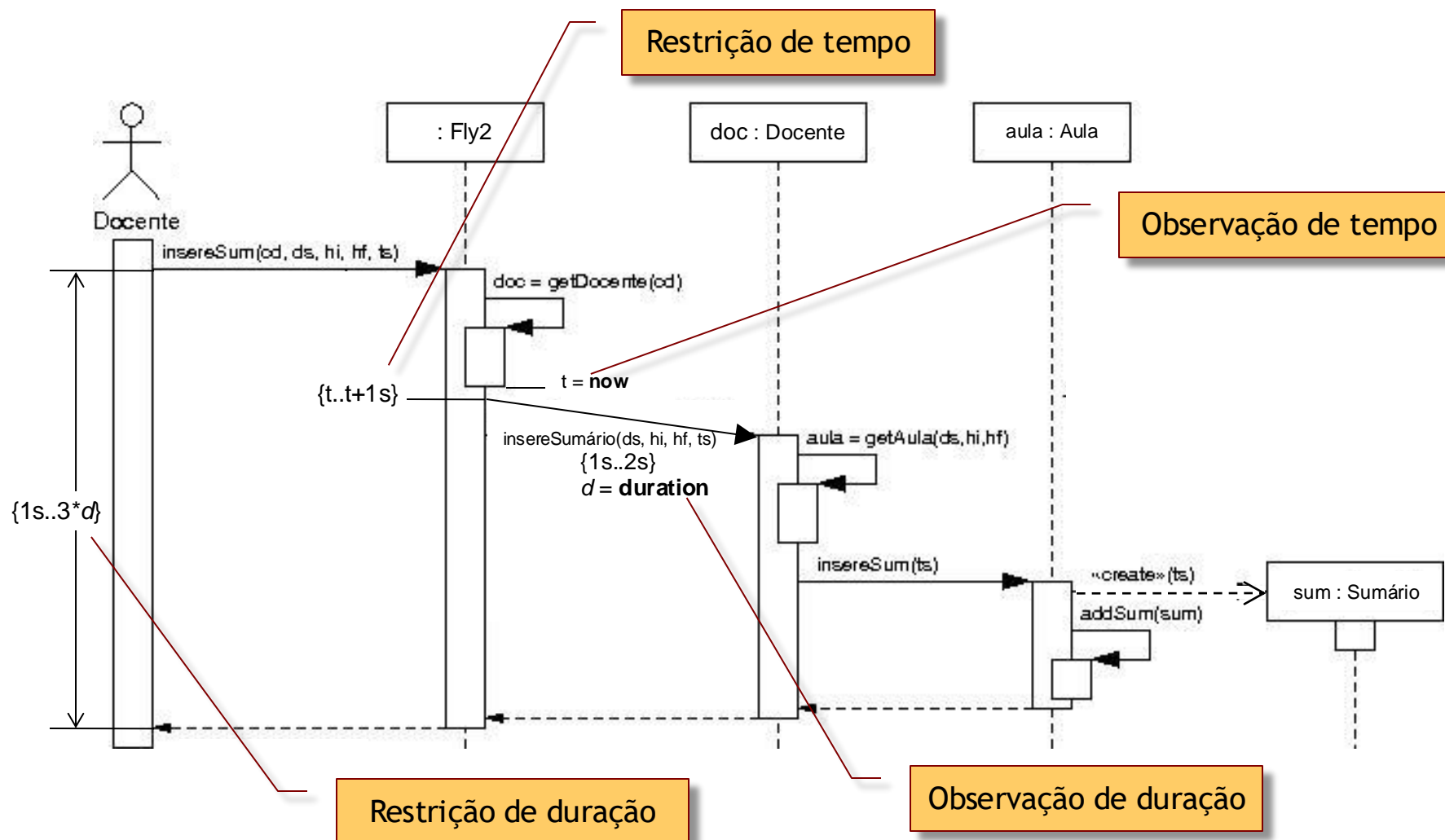
# Operador *consider*



Porque podem existir outros eventos pelo meio...



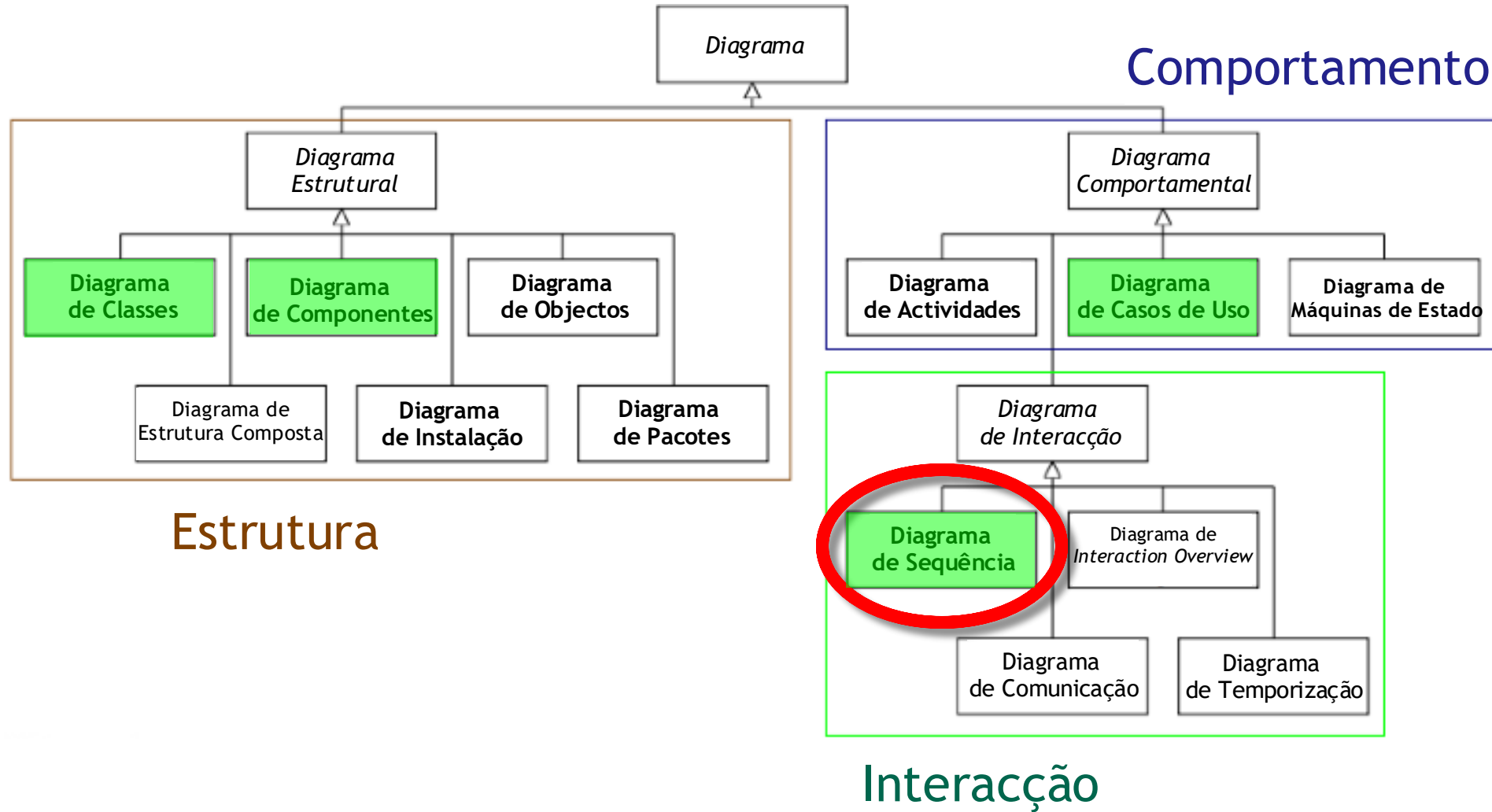
# Restrições de tempo / duração







# Diagramas da UML 2.x



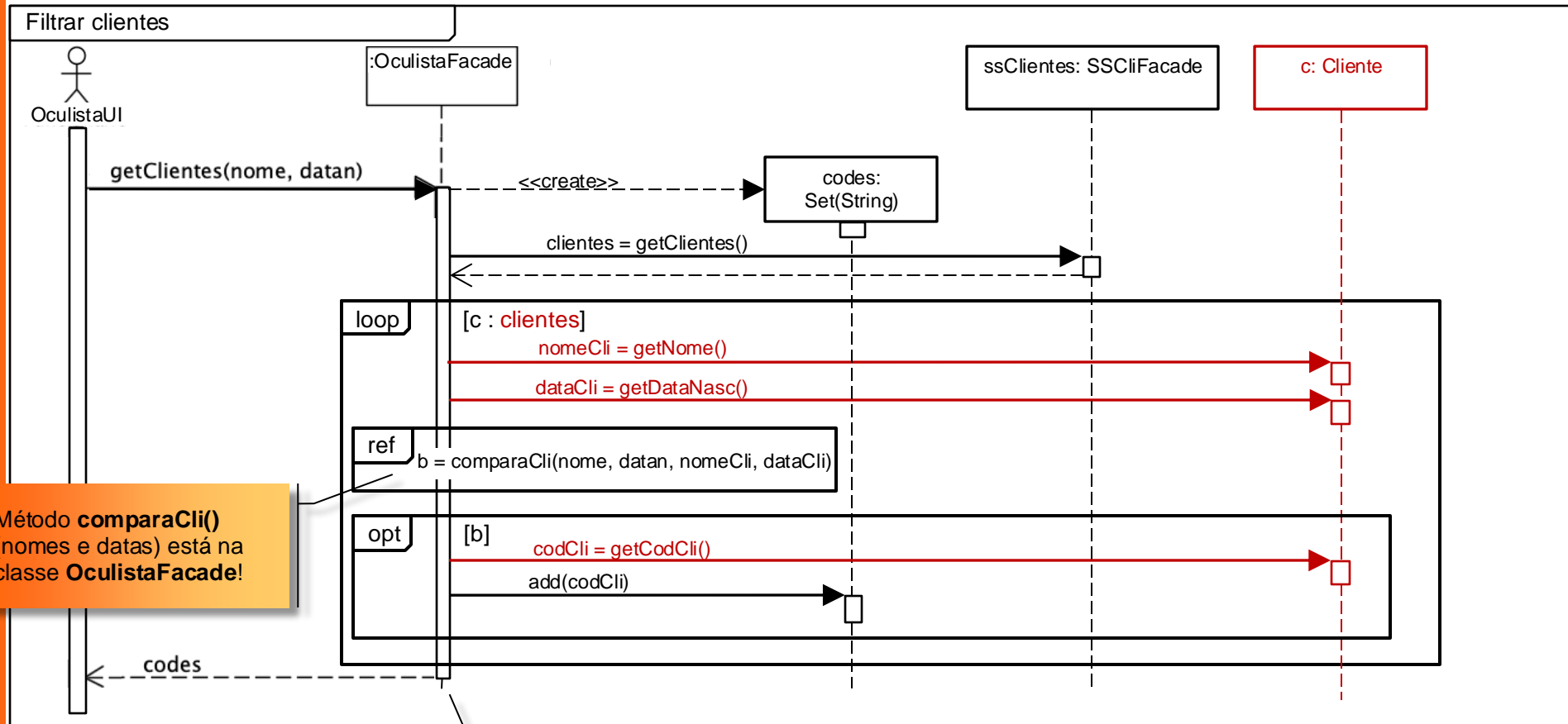


# Princípio SLAP 🙌

- 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

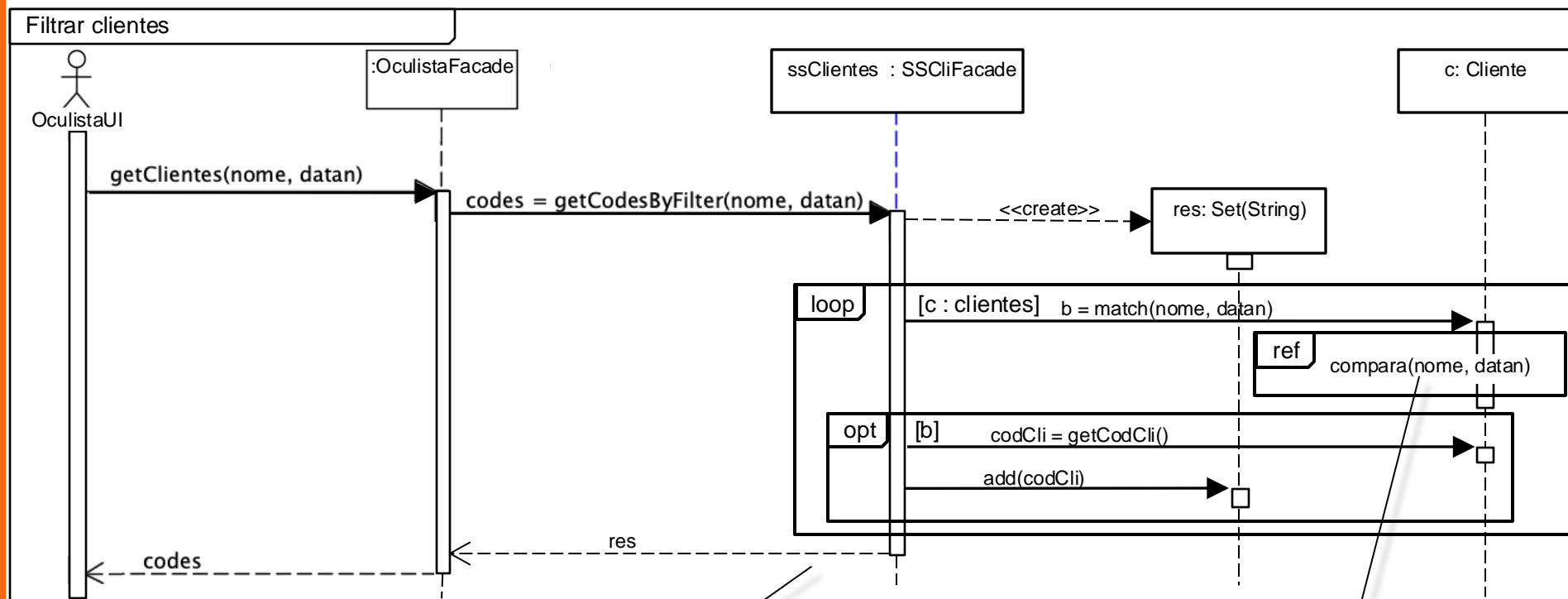


Método **comparaCli()** (nomes e datas) está na classe **OculistaFacade**!

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