

---

---

---

---

---



## Fase 1 → Modelo de domínio

Abstração = O processo de remover informação para ficarem apenas os aspetos relevantes

→ Mecanismo para lidar com a complexidade

### Vantagens

- ↳ compreender a realidade
- ↳ comunicar ideias de forma simplificada
- ↳ documentar as decisões tomadas durante o desenvolvimento
- ⇒ Base para análise de requisitos
- estático → não representa fluxo de dados

As **entidades** são apenas candidatas a serem classes

↳ "Substantivos" na descrição

**Relações** → "Verbos" na descrição

↳ relações persistentes

↳ candidatas a existirem na solução

↳ relação "é um (a)"

**UML** → linguagem usada

Representarmos usando diagramas de classe UML

↳ subconjunto da notação

↳ Entidades → representadas por classe

↳ Relacionamentos → representados por associações

### Multiplicidade

\* → zero ou mais objetos

n → n objetos ou mais

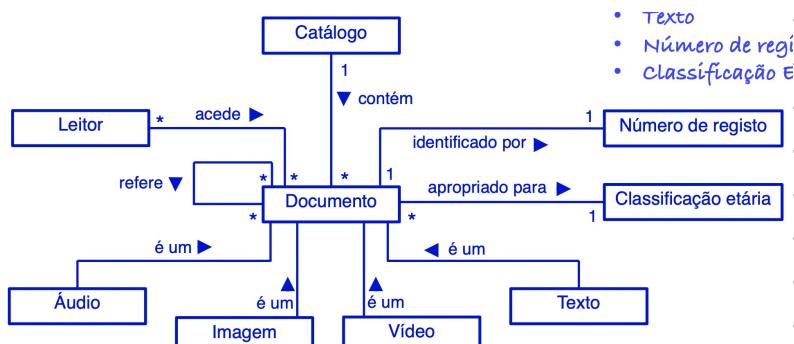
n..m → entre n e m objetos (n..m)

n..\* → n ou mais objetos

### Classes de associação



### Associações n-árias



## Fase 2 → Modelação do Requisitos Funcionais

### ↳ Use Cases

- O que o sistema deve fazer
- Descrevem as interações entre o Sistema e o seu ambiente

Requisitos não funcionais → Como o sistema deve fazê-lo

Validação dos requisitos → com o cliente

- ↳ **Completo** → todos os aspetos relevantes foram considerados
- ↳ **Consistência** → não existem contradições entre requisitos
- ↳ **Ausência de ambiguidade** → nenhum requisito pode ser interpretado de formas diferentes
- ↳ **Correção** → Os requisitos descrevem corretamente o que o cliente pretende e o que a equipa de desenvolvimento se propõe fazer
- ↳ **Realista** → não se deve prometer o que não podemos realizar
- ↳ **Verificável** → termos que poder saber se cumprimos os objetivos
- ↳ **Rastreável** → termos que poder saber porque é que cada requisito foi definido

### Identificação de Use cases

- ↳ **Identificar actores** → quem utiliza o sistema
- ↳ **Identificar use cases** → o que se pode fazer no sistema
- ↳ **Identificar associações** → quem pode fazer o quê

### Notação semi-estruturada

Use case: nome do use case

Descrição: breve descrição do use case

Cenário: cenários que originam o use case

Pré-condição: o que deve ser verdade no sistema para que executar o use case seja válido

Pós-condição: condição de sucesso do use case (o que deve ser verdade depois)

Fluxo normal:

Fluxo de eventos mais comum

Fluxos alternativos:

Especificação das modos alternativos de completar o use case

Fluxos de exceção:

Especificação de situações relevantes em que o use case não tem sucesso

### Processo

- 1- Recolher cenários
- 2- Identificar e especificar os Use Cases
- 3- Identificar Actores
- 4- Identificar relações entre Actores e Use Cases

### Vantagens

- Não há trabalho desnecessário
- O sistema de Informação suporta as fases do cliente
- As fronteiras do sistema ficam bem definidas

Use Case: Levantar €

Descrição: Cliente levanta quantia da máquina

Cenários: O João levanta €50 com cartão

Pré-condição: Sistema tem notas

Pós-condição: Cliente tem quantia desejada e saldo da conta foi actualizado

Fluxo normal:

1. Cliente apresenta cartão e PIN
2. Máquina MB **valida acesso** e pede operação
3. Cliente indica que pretende levantar dada quantia
4. Máquina MB pergunta se quer talão
5. Cliente responde que não
6. Máquina MB devolve cartão, fornece notas e actualiza saldo da conta
7. Cliente retira cartão e notas

Fluxo alternativo (1): [Cliente quer talão] (passo 5)

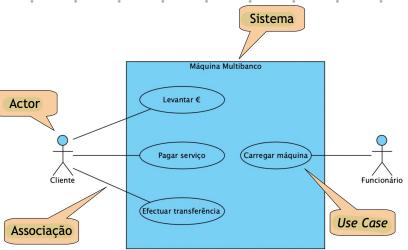
- 5.1. Cliente responde que sim
- 5.2. Máquina MB devolve cartão, notas e talão e actualiza saldo da conta
- 5.3. Cliente retira cartão, notas e talão

Fluxo de exceção (2): [PIN inválido] (passo 2)

- 2.1. Máquina MB avisa sobre PIN inválido e fornece cartão
- 2.2. Cliente retira cartão

## Fase 2 → Modelação dos Requisitos Funcionais

### ↳ Diagrama de Use Case

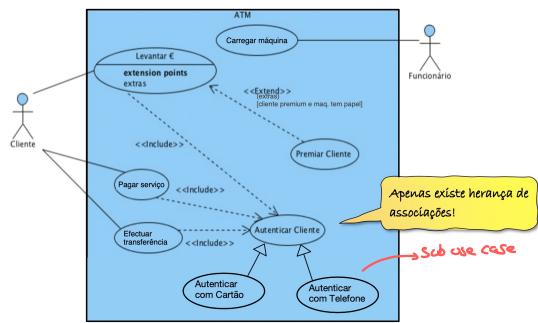


**Sistema** → define as fronteiras da solução a desenvolver

**Actor** → Abstração para uma entidade fora do sistema  
↳ interação com o sistema

**Associação** → comunicação entre o actor e o sistema - através de use cases

**«include» / «extend» / herança**



**«extend»** → utilizar quando pretendemos reutilizar/alterar um Use Case base já existente

→ caso base deve ser um use case bem formado sem extensões

→ extensão não precisa de ser um use case bem formado por si só

**Sub use case** → caso particular do use case base

### Fase 3 → Diagrama de Componentes / Interfaces

#### Fluxo normal:

1. Funcionário indica nome e/ou data de nascimento do cliente
2. Sistema <b>procura clientes</b>
3. Sistema apresenta lista de clientes
4. Funcionário seleciona cliente
5. Sistema <b>procura cliente</b>
6. Sistema apresenta detalhes do cliente
7. Funcionário confirma cliente
8. Sistema <b>procura produtos</b> e apresenta lista
9. Funcionário indica Código de armazenagem e lentes
10. Sistema <b>procura detalhes dos produtos</b>
11. Sistema apresenta detalhes dos produtos
12. Funcionário confirma produtos
13. Sistema <b>regista reserva dos produtos</b>
14. <<include>> imprimir talão

responsabilidades que a lógica de negócios tem que cumprir para satisfazer o use case

#### OculistaLN

procura clientes  
procura cliente  
procura produtos  
procura detalhes dos produtos  
regista reserva dos produtos

→ API

OculistaLNFacade → <b>fat</b> layer
+getClientes(name: ?, dataN: Data): ?
+getCliente(codcli: ?): Cliente
+getProdutos(): ?
+getArmação(cArm: Código): Armação
+getLente(cLen: Código): Lente
+registaReserva(codcli: ?, cArm: Código, cLenD: Código, cLenE: Código)

APIs parciais

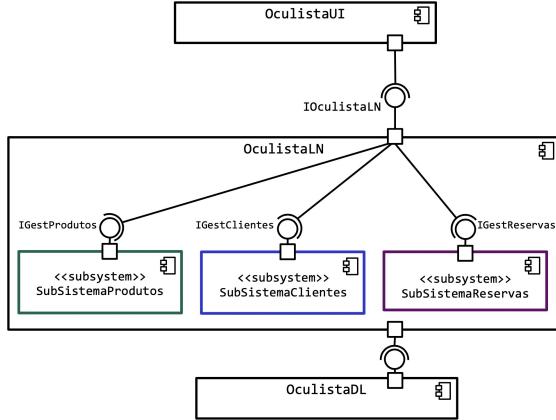
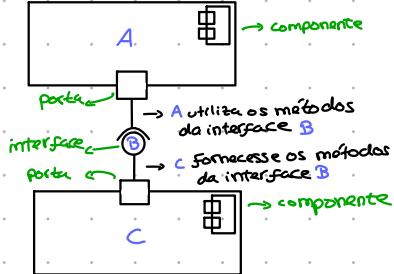
Diagrama de componentes descreve os componentes do sistema e as dependências entre eles

Componente → pedaço de software reutilizável, bem encapsulado e "facilmente" substituível

↳ blocos que combinados constroem o sistema

Candidatos a componentes → grandes blocos do sistema

↳ itens com funcionalidades usadas recorrentemente em diferentes sistemas



Relação de concretização → um componente pode concretizar uma ou mais interfaces

↳ As interfaces são fornecidas ou exportadas

Relação de dependência → um componente pode usar uma ou mais interfaces

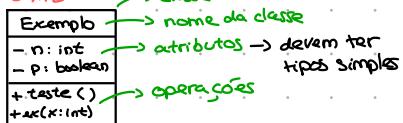
↳ Essas interfaces são requeridas ou importadas

## Fase 4 → Diagramas de Classe ↳ Modelação Estrutural

### objectivos

Facilitar a rectificação e facilitar a manutenção

### UML



### Níveis de modelação

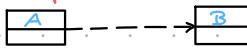
- conceptual → modelo de domínio
- especificação → Definição das interfaces (API's)
- implementação → Definição concreta das classes a implementar

### Visibilidade dos atributos e das operações

- privado → acessível ao objecto a que pertence
- # protegido → " " a instâncias das sub-classes
- pacote/package → " " de classes do mesmo package
- + público → acessível a todos os objectos do sistema

### Relações entre classes

#### → Dependência



A depende de B

→ Dim. o nº de dependências → objetivo

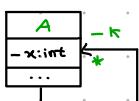
#### → Associação

```
public class A {  
    private int x;  
    private B k;  
}
```



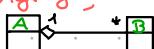
#### → Associação reflexiva

```
class A {  
    private int x;  
    private Collection<A> k;  
}
```



Definem uma relação entre objectos da mesma classe

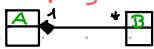
#### → Agregação



Os B fazem parte da estrutura interna de A

Mas, os B têm existência própria

#### → Composição



Os B (de A) só existem no contexto do A

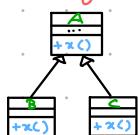
Os B não têm existência para além da existência de A

→ Associações qualificadas



```
public class A {  
    private Map<int, B> xj  
}
```

- **polimorfismo** → 2 sub-classes podem fornecer métodos diferentes para implementar a mesma operação da super- classe
- **Overriding** → sub- classe pode o método associado a uma operação declarada pela super- classe

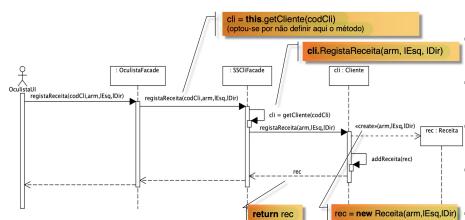


→ Variável de classe (static no java) → exemplo

→ abstract → escrever { abstract }

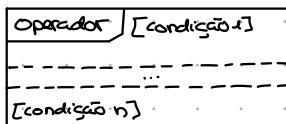
Fase 5 → Diagramas de Sequência  
↳ Modelação Comportamental

- Descreve como um conjunto de objetos coopera para realizar um dado comportamento
- Representam as interações entre objetos através de mensagens que são trocadas entre eles
- Ordenação temporal das mensagens
- permite analisar a distribuição de "responsabilidades" pelas diferentes entidades



→ fragmentos combinados

→ permitem expressar fluxos condicionais e estruturar os modelos



## Operadores:

alt - define fragmentos alternativos (mutuamente exclusivos)

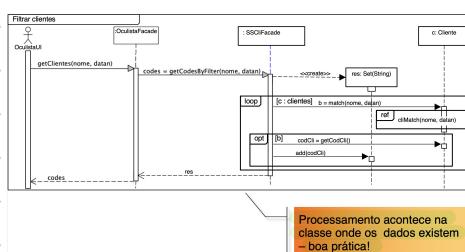
bop / bop(n) - fragmento é repetido enquanto a guarda for verdadeira / n vezes

**opt** - fragmento opcional (ocorre se a guarda for verdadeira)

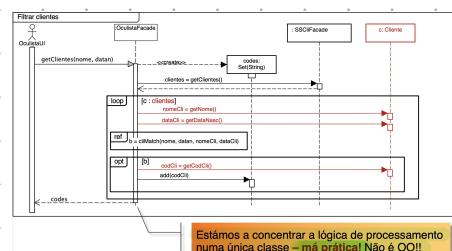
par  $\approx$  threads  $\approx$  concurrent program with parallel blocks

bract - terminating a flower

**ref** - referencia a outro diagrama



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



Estamos a concentrar a lógica de processamento numa única classe – **má prática!** Não é OO!!

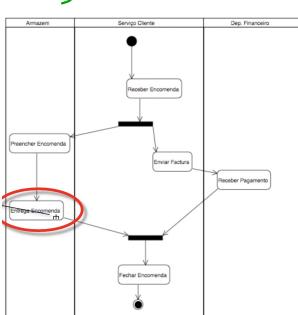
Fase 6 → Diagrama de Atividade  
↳ Modelação Comportamental



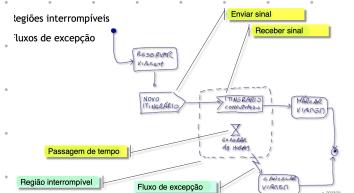
## Etiquetas



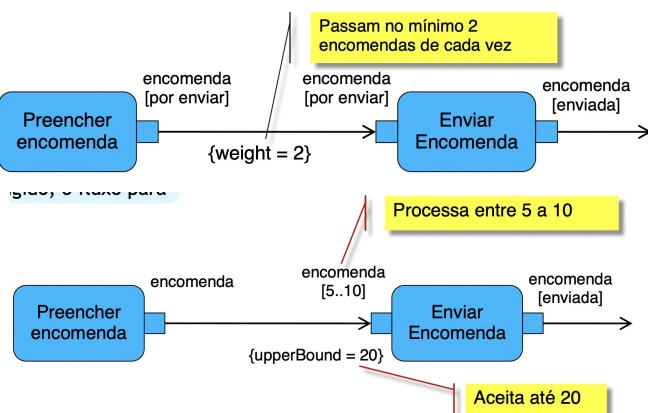
## Particões



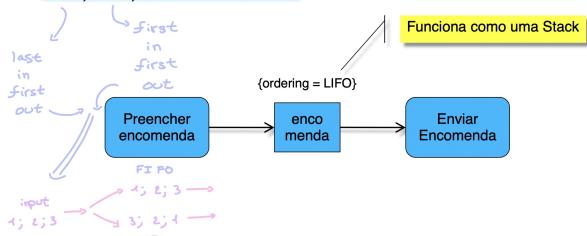
## Sinais



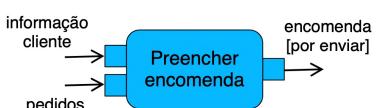
Fluxos com objetos, pesos, multiplicidades, upperBound, e dem de processamento



- LIFO, FIFO, definida no modelo

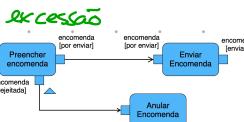


Os nós podem excretar quando todos os fluxos de entrada estão satisfeitos

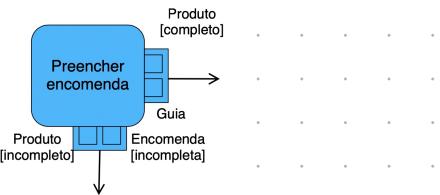


Os nodos produzem tokens em todos os fluxos de saída





### Conjunto de parâmetros



### Fase 7 → Diagramas de Package

#### ↳ Modelação Estrutural II

→ agrupamentos de classes/componentes/use cases/etc.

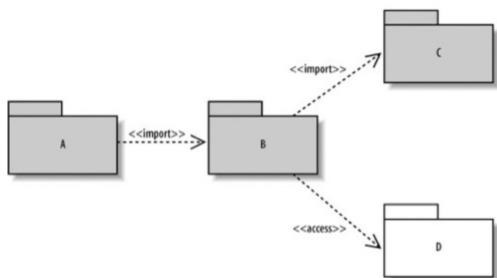


«import» → o package origem importa o conteúdo público do package destino

«access» → o package origem accede a elementos públicos do package destino

«merge» → o package origem é fundido com o package destino para gerar um novo

- 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 accedido por B (não é importado).



Cada package terá o seu Diagrama de classes

### Fase 8 → Object Constraint Language (OCL)

Restrições em operações e associações

Invariáveis de classe e tipos

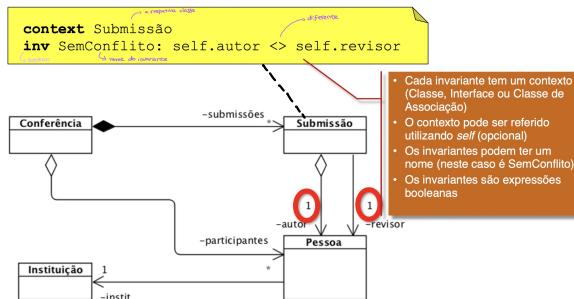
↳ Uma restrição que deve ser verdadeira num objeto durante todo o seu tempo de vida.

Pré-condição dos métodos

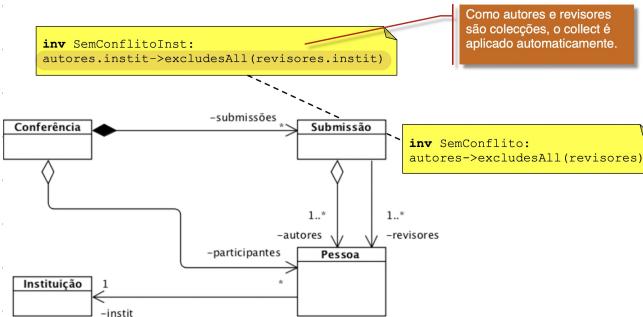
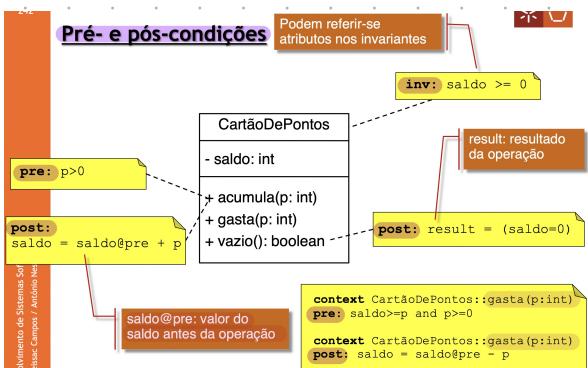
↳ Restrições que especificam as condições de aplicabilidade de uma operação

Pós-condição dos métodos

↳ Restrições que especificam o resultado de uma operação



- Cada invariante tem um contexto (Classe, Interface ou Classe de Associação)
- O contexto pode ser referido utilizando `self` (opcional)
- Os invariáveis podem ter um nome (neste caso é `SemConflito`)
- Os invariáveis são expressões booleanas



size(): Integer  
 isEmpty(): Boolean  
 notEmpty(): Boolean  
 includes(object: T): Boolean  
 excludes(object: T): Boolean  
 count(object: T): Integer  
 includesAll(c2: Collection(T)): Boolean  
 excludesAll(c2: Collection(T)): Boolean  
 sum(): T  
 product(c2: Collection(T2)): Set(Tuple(first:T, second:T2))

Iterator expression  
 select(iterator | body): Collection(T)  
 reject(iterator | body): Collection(T)  
 collect(iterator | body): Collection(T2)  
 collectNested(iterator | body): CollectionWithDuplicates(T2)  
 sortedBy(iterator | body): OrderedCollection(T)

iterate(iterator: T; accum: T2 = init | body) : T2  
 exists(iterators | body): Boolean  
 forAll(iterators | body): Boolean  
 one(iterator | body): Boolean  
 isUnique(iterator | body): Boolean  
 any(iterator | body): T

### Vantagens

- Melhor documentação
  - ↳ adicionaram informação
  - ↳ documentar modelo
- Maior precisão
  - ↳ semântica formal
  - ↳ diminuir ambiguidade
- Melhor comunicação
  - ↳ comunicar sem ambiguidade
  - ↳ mas perde-se representação gráfica

### Fase 3 → Máquinas de Estado

↳ Modelação comportamental

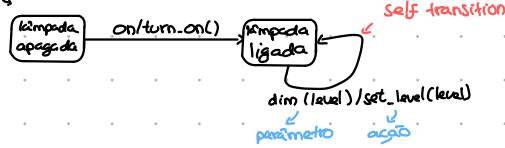
→ modelar o comportamento de um dado objeto/sistema

estado → Lâmpada apagada

estado inicial → ●

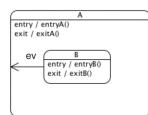
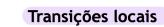
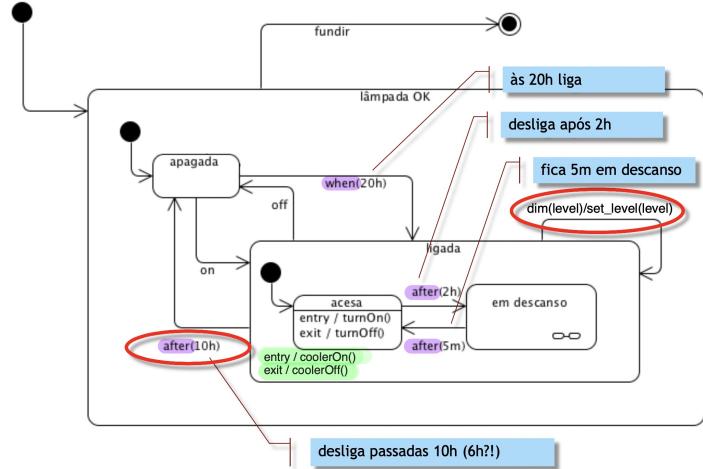
estado final → ○

transições - evento [guarda] / ação

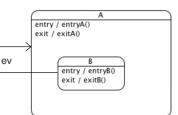


## Atividades internas

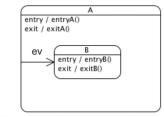
- entry / ação** "ação" executada auto. quando o obj. entra no estado
- do / ação** "ação" é continuadamente executada, enquanto o obj. estiver no estado
- exit / ação** "ação" é executada auto. quando o obj. sai do estado
- evento / ação** "ação" é automaticamente executada se "evento" acontecer
- evento / deser** "evento" é desarmado até o estado atual ser abandonado



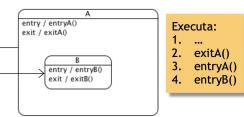
(sub-estado para super-estado)



(sub-estado para super-estado)

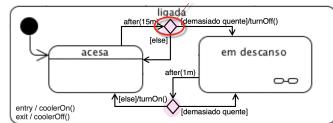


(super-estado para sub-estado)



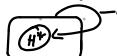
(super-estado para sub-estados)

## Pseudo -estado

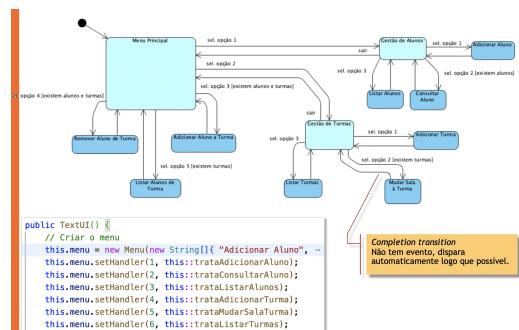


- (H) shallow history }  $\downarrow$  permits modular  
(H\*) deep history }  $\downarrow$  interruptor

4. Atividade da máquina retira no estoado  
em que se encontrava quando da última sarda



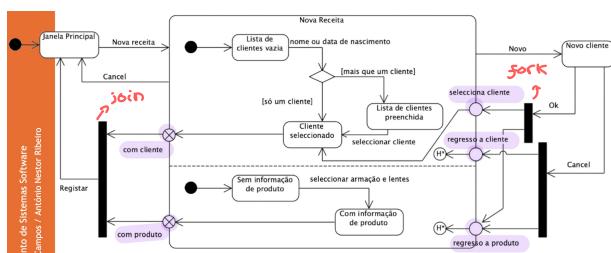
## Modelação do controlo de diálogo da interface



em evento, dispara automaticamente logo que possível

## Estados de Concorrência

↳ Diagramas das regiões são executados de forma concorrente



## Pseudoestado do terminação

