

Fase 1 → Modelo de domínio → 2. análise

Abstracção = 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

Representamos 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



1. planeamento

Fase 2 → Modelação do requisitos funcionais

↳ Use Cases → 2. análise

→ O que o sistema deve fazer

→ Descrevem as interacções entre o Sistema e o seu ambiente

Requisitos não funcionais → Como o sistema deve fazê-lo

Validação dos requisitos → com o cliente

↳ **completude** → 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** → temos que poder saber se comprimimos os objetivos

↳ **rastreável** → temos que poder saber porque é que cada requisito foi definido

Identificação de Use cases

↳ **identificar atores** → quem utiliza o sistema

↳ **identificar use cases** → o que se pode fazer no sistema

↳ **identificar associações** → quem pode fazer o que

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 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 dos modos alternativos de completar o use case

Fluxos de exceção :

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

Vantagens

→ Não há trabalho desnecessário

→ O sistema de informação suporta as funcionalidades do cliente

→ As fronteiras do sistema ficam bem definidas

Use Case: Levantar €

Descrição: Cliente levanta quantia da máquina

Cenários: O João levanta 500 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 saldo
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 saldo] (passo 5)

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

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 do requisitos funcionais

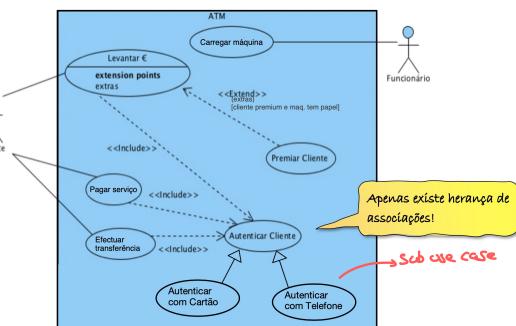
↳ Diagrama de Use Case → 2. análise

Sistema → define as fronteiras da solução a desenvolver

Autor → Abstracção para uma entidade fora do sistema
↳ interacção com o sistema

Associação → comunicação entre o autor 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 b em formado por si só

Sub use case → caso particular do use case base

Fase 3 → Diagrama de Componentes/Interações → 3. concepção

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 seleciona 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 armazém 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

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



→ API

OculistaLNFacade → faz ligação

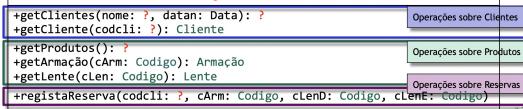


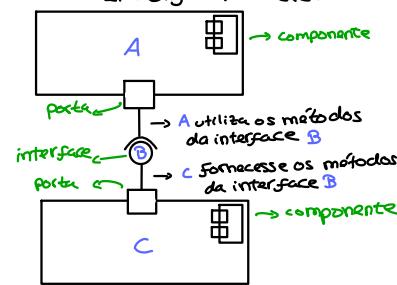
Diagrama de componentes descreve os componentes do sistema e as dependências entre eles

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

↳ blocos que combinados construem o sistema

Candidatos a componentes → grandes blocos do sistema

↳ itens com funcionalidades usados recorrentemente em diferentes sistemas



Relação de concretização

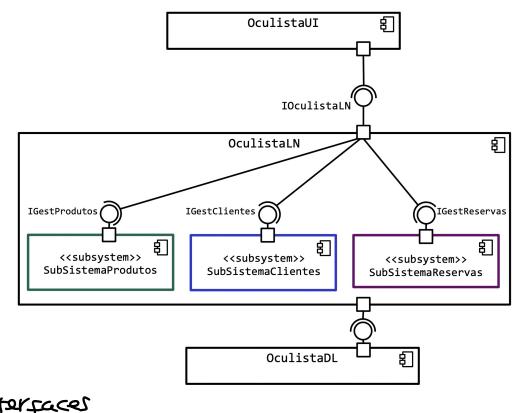
→ um componente pode concretizar uma ou mais interfaces

↳ 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 → 3. concepção

↳ Modelação Estrutural

→ 3. concepção

objectivos

Facilitar a rectificação e facilitar a manutenção

UML

Exemplo → classe
- n: int → nome da classe
- p: boolean → atributos → devem ter tipos simples
+ teste() → operações
+ ex(x: int)

Níveis de modelação

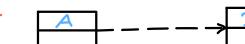
- conceptual → modelo de domínio
- especificações → Definição das interações (API's)
- implementação → Definição concreta das classes a implementar

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 classe

→ Dependência



A depende de B

→ Dim. o nº de dependências → objetivo

→ Associação

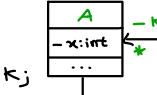
public class A {
private int x;
private B k;
}
A → B
-x: int
+s(b: B)



Indica que objectos de uma classe estão ligados a objectos de outra

→ 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

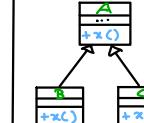


public class A {
private Map<int, B> x;
}

}

→ polimorfismo → 2 sub-classes podem fornecer métodos diferentes para implementar a mesma operação da super-classe

→ Overriding → sub-classe pode implementar associado a uma operação declarada pela super-classe



→ Variável de classe (static no Java) → exemplo
→ abstract → escrever { abstract }

Fase 5 → Diagramas de Sequência → 3. concepção

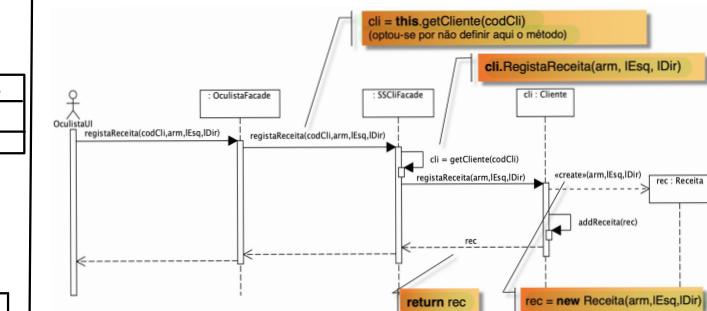
↳ Modelação Comportamental

→ Descreve como um conjunto de objectos coopera para realizar um dado comportamento

→ Representam as interações entre objectos através de mensagens que são trocadas entre eles

→ Ordenação temporal das mensagens

→ permite analisar a distribuição de "responsabilidades" pelas diferentes entidades



Operadores:

switch case
alt - define fragmentos alternativos (mutuamente exclusivos)

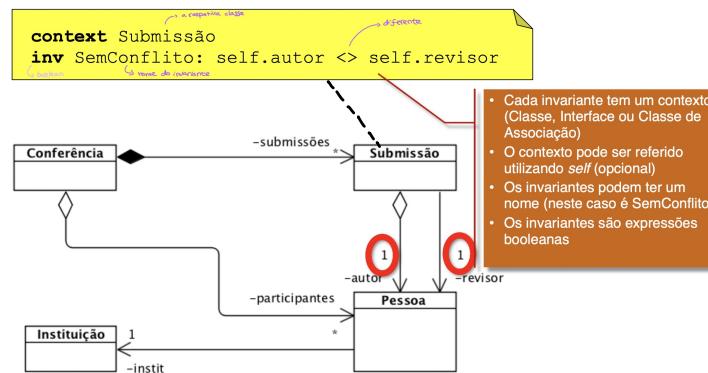
sw while
loop / loop(n) - fragmento é repetido enquanto a guarda for verdadeira/n vezes

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

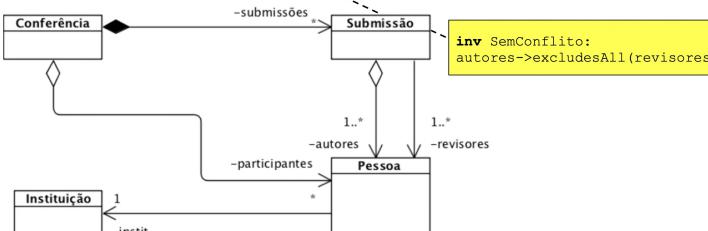
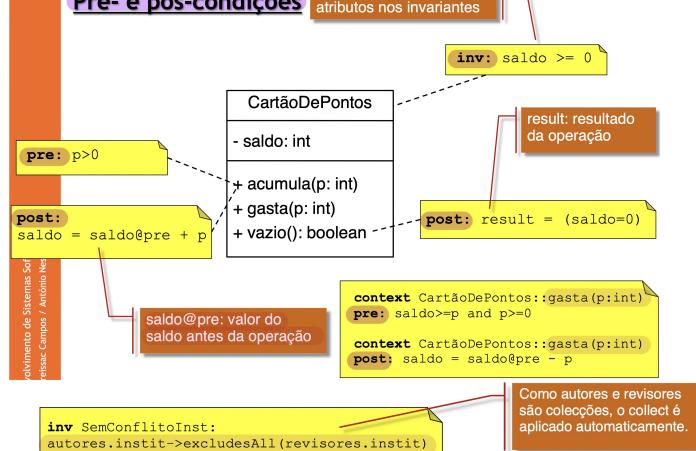
par - fragmentos ocorrem em paralelo

break - termina o fluxo

ref - referência a outro diagrama



Pré- e pós-condições



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))

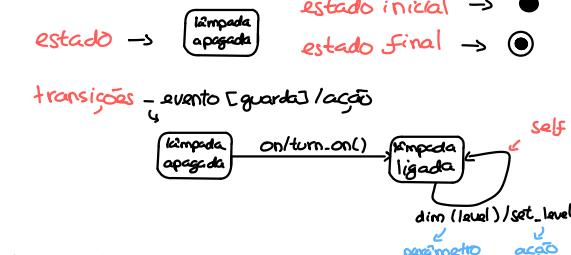
Iterator expression
select(iterator | body): Collection(T)
reject(iterator | body): Collection(T)
exists(iterator | body): Boolean
forAll(iterator | body): Boolean
collect(iterator | body): Collection(T2)
collectNested(iterator | body): CollectionWithDuplicates(T2)
one(iterator | body): Boolean
isUnique(iterator | body): Boolean
any(iterator | body): T

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

Fase 9 → Máquinas de Estado → 3. concepção

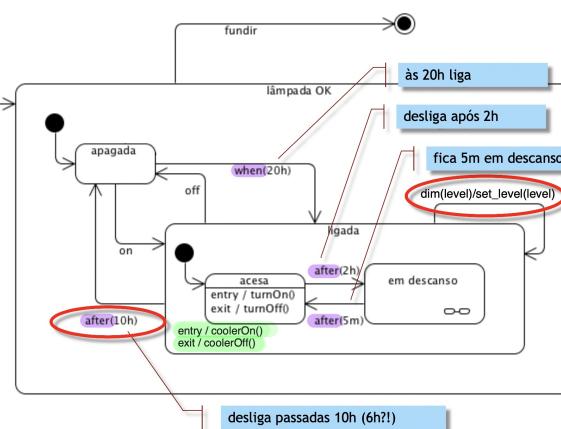
↳ Modelação comportamental

→ modelar o comportamento de um dado objeto/sistema

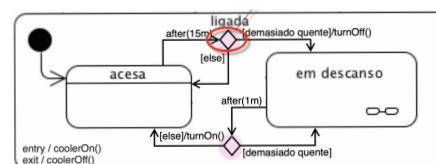


Atividades internas

- 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 desferido até o estado atual ser abandonado



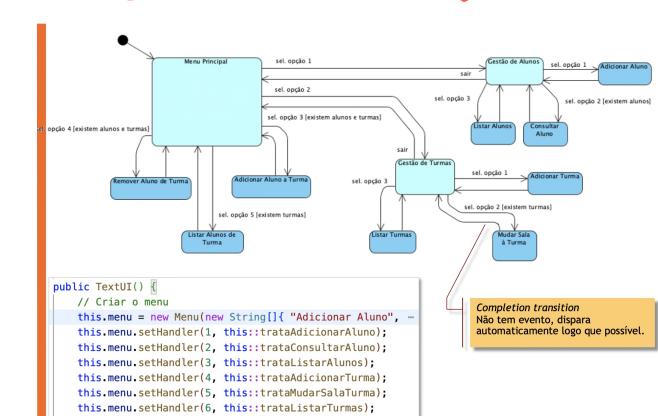
Pseudo-estado



- ④ shallow history } permite modelar
④ deep history } interrupções

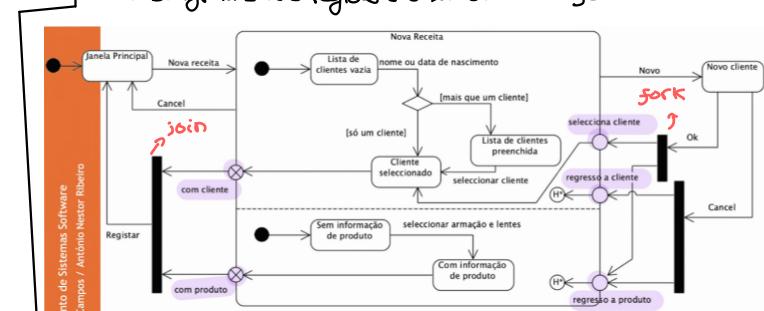
↳ atividade da máquina retorna no estado em que se encontrava quando da última saída

Modelação do controlo de diálogo da interface



Estados da concorrência

↳ Diagramas das regiões são executados de forma concorrente



Pseudoestado do terminação

