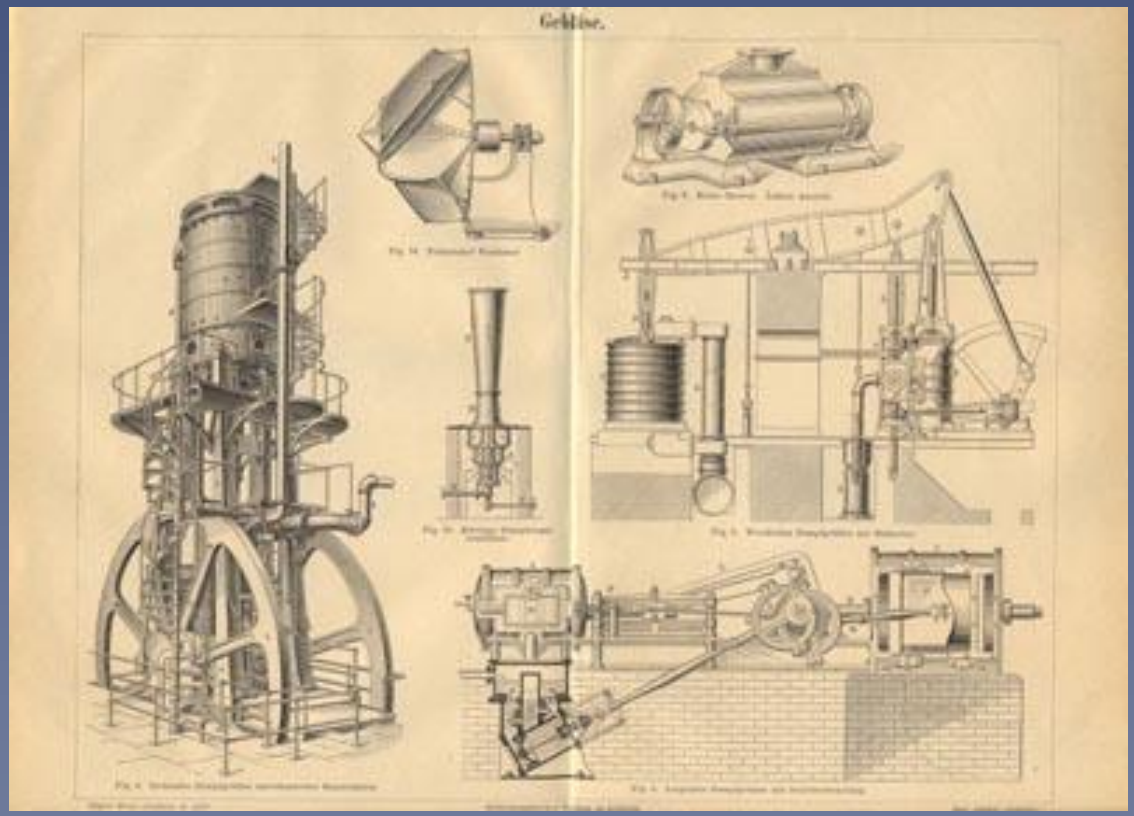
Modelação e Padrões de Desenho

Capítulo 2
Processo de Desenho OO

Fernando Miguel Carvalho
DEETC
Instituto Superior de Engenharia de Lisboa,
Centro de Cálculo
mcarvalho@cc.isel.ipl.pt

Desenvolvimento de Software



Desenvolvimento de Software

Actividades mais comuns no desenvolvimento de *Software*:

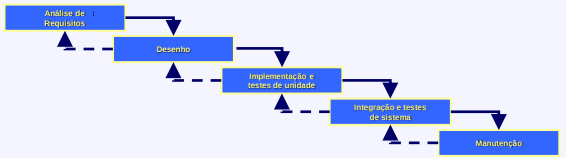
- **Análise** de requisitos (funcionais e não funcionais);
- **Desenho** (texto e diagramas);
- **Implementação** e testes (*Unit Testing*);
- Integração e teste (*System Testing*);
- Distribuição e instalação (*Deploy*);
- Manutenção.

Os processos de desenvolvimento distinguem-se:

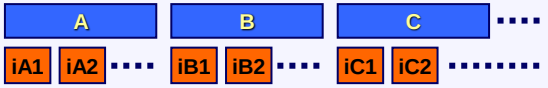
- Número e tipo de actividades;
- A ordem pela qual as actividades são realizadas;
- O *output* de cada uma das actividades.

Tipos de Processos de desenvolvimento mais populares:

- Em cascata (Waterfall);

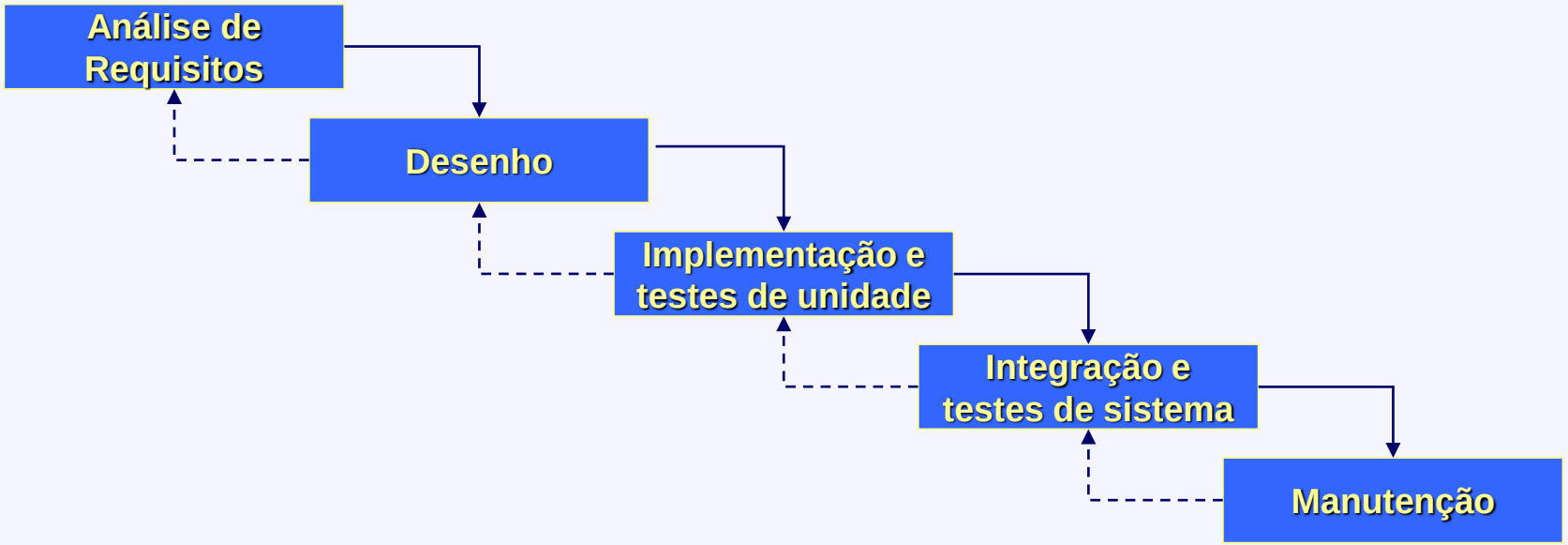


- Iterativo.

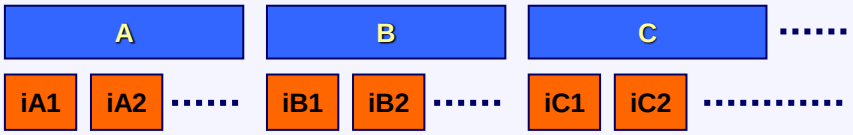


Desenvolvimento de Software: ***Waterfall...***

- **Aprovação dos *deliverables*** de uma fase, antes do início da fase seguinte;
- O **custo de alterações** de requisitos é tanto **maior, quanto mais tarde** forem introduzidas essas alterações.
- O objectivo é **minimizar alterações** após a entrega dos documentos (*deliverables*) de cada fase.



Desenvolvimento de Software: **Iterativo**...



- O sistema é desenvolvido **incrementalmente** e **não como uma peça monolítica**;
- Cada **iteração** trata uma **pequena peça ou incremento** do sistema;
- Cada **iteração** resulta numa **versão** de um **produto** ou **componente,** que faz parte do sistema;
- O processo continua até todo o sistema estar completo.

➔ **Ao contrário do processo waterfall, este assume que existem mudanças (de requisitos) durante o período de desenvolvimento de software.**

Processos iterativos mais populares:

- RUP(*Rational Unified Process*)

  Requisitos de Negócio; Domínio; Casos de utilização; Analise; Desenho; Processos; Implementação; Teste; Deployment.
- XP (*Extreme Programming*)

  Ênfase na produção de código de elevada qualidade desde o início do seu desenvolvimento. A primeira implementação deve produzir um esqueleto mínimo e executável do sistema. Cada iteração deve produzir sempre uma nova versão executável.

Fases do Desenvolvimento de Software

Independentemente do processo e metodologia existem 3 fases que podem ser claramente identificadas no desenvolvimento de software:

- Análise;
- Desenho;
- Implementação.

➔ **Será redutor e não se deve assumir que o desenvolvimento de software é apenas uma progressão linear destas 3 fases.**

Fase de Análise

- Transforma o vago entendimento do **problema** numa descrição precisa das tarefas que o sistema de software deverá desempenhar.
- Descreve o **que tem que ser feito e não como fazer**.

O resultado da fase de análise é uma descrição textual, designada normalmente de **Especificação Funcional**, que tem as seguintes características:

- Definição completa das tarefas a disponibilizar pelo software;
- Legível por quem conhece o domínio do problema e por quem desenvolve o software;
- Revisto pelas diversas partes interessadas e envolvidas;
- Pode ser testado em confronto com a realidade.

Uma forma de descrever o comportamento do sistema é através de cenários de utilização (***Use Cases)***:

- **Descrevem uma sequência de interacções entre o sistema e o utilizador.**

Fase de Análise… ***Use Cases***... Cenário Principal

Deixar mensagem:

1. O cliente marca o número do sistema de voicemail;
2. O sistema de voicemail diz a mensagem:

   `"Introduza o número da caixa de correio seguido de #"`
3. O utilizador introduz o número da extensão do destinatário;
4. O sistema de voicemail valida o numero e responde a mensagem:

   `"Chegou à caixa de correio de xx."`

   `"Por favor, deixe a sua mensagem agora ou introduza a password!"`
5. O utilizador diz a mensagem;
6. O utilizador desliga;
7. O sistema de voicemail coloca a mensagem gravada na caixa de correio do destinatário.

Fase de Análise… ***Use Cases***... Cenários Alternativos

Variação #1:

3. No ponto 3, o utilizador introduz uma extensão inválida;
4. O sistema de voicemail valida o numero e responde a mensagem:
   `"Introduziu um número inválido de caixa de correio."`
5. Continuar no ponto 2.

Variação #2:

4. No ponto 4, o utilizador desliga antes de dizer a mensagem;
5. O sistema de voicemail descarta a mensagem vazia.

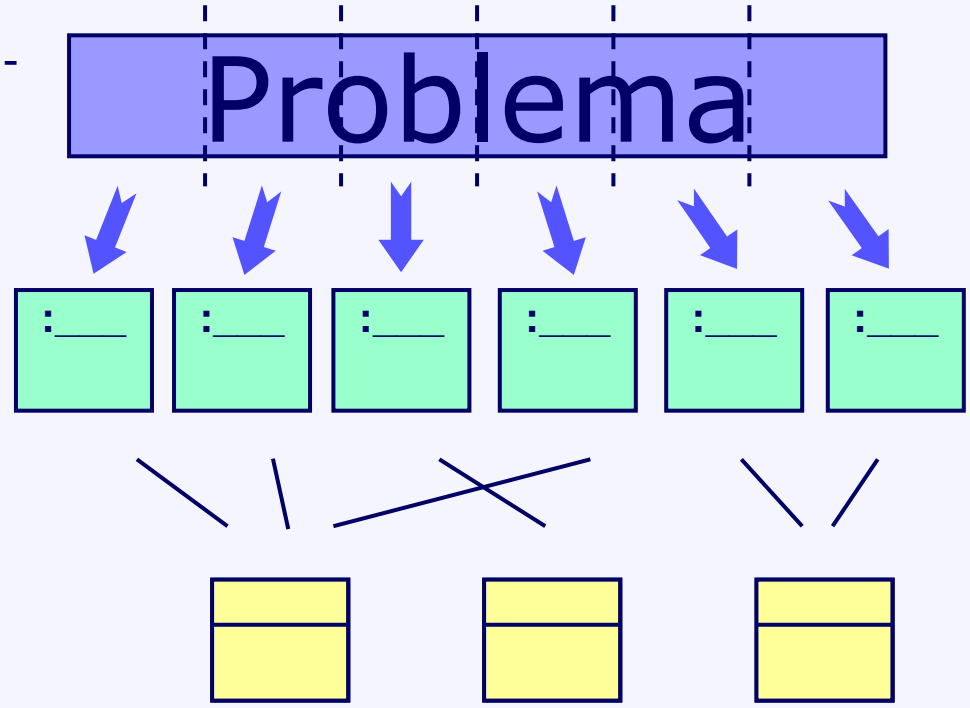
Variação #3:

4. ~~No ponto 4, o utilizador introduz a password da sua mailbox~~ (caixa de correio);
5. ~~O sistema de voicemail valida a password~~ e lê o menu de opções;
6. …

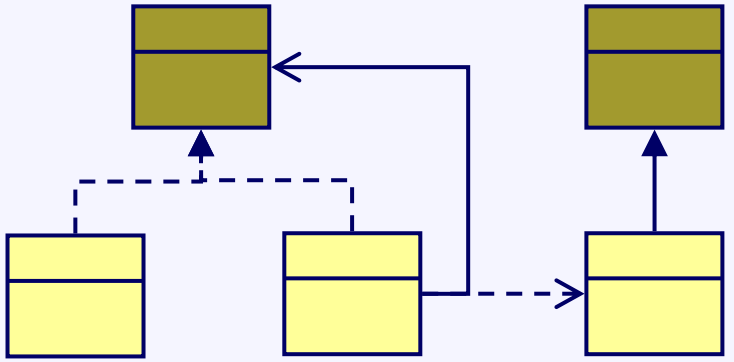
Não é um cenário alternativo, mas sim outro Use Case.

Fase de Desenho

- Uma vez descrito o problema há que dividi-lo e organiza-lo por partes.



- Encontrar os objectos pertinentes que representam conceitos do problema.
- Decompor os objectos em tipos, que definem esses objectos.
  (na granularidade **correcta**)
- Desenhar hierarquias e relações **correctas**.



Fase de Desenho…

Objectivo:

- Identificar as classes, as suas responsabilidades e relações entre estas.

Não fazem parte do desenho:

- Escolha exacta das estruturas de dados.
  por exemplo: utilização de tabelas de dispersão versus árvores binárias.
- Escolha da linguagem de programação.

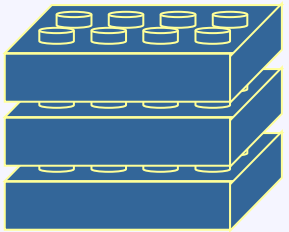
Resultados da fase de desenho:

- Descrição textual das classes e as suas responsabilidades mais importantes;
- Diagramas de classes;
- Diagramas de sequência dos cenários mais importantes;
- Diagramas de estado dos objectos com comportamento altamente dependente do estado.

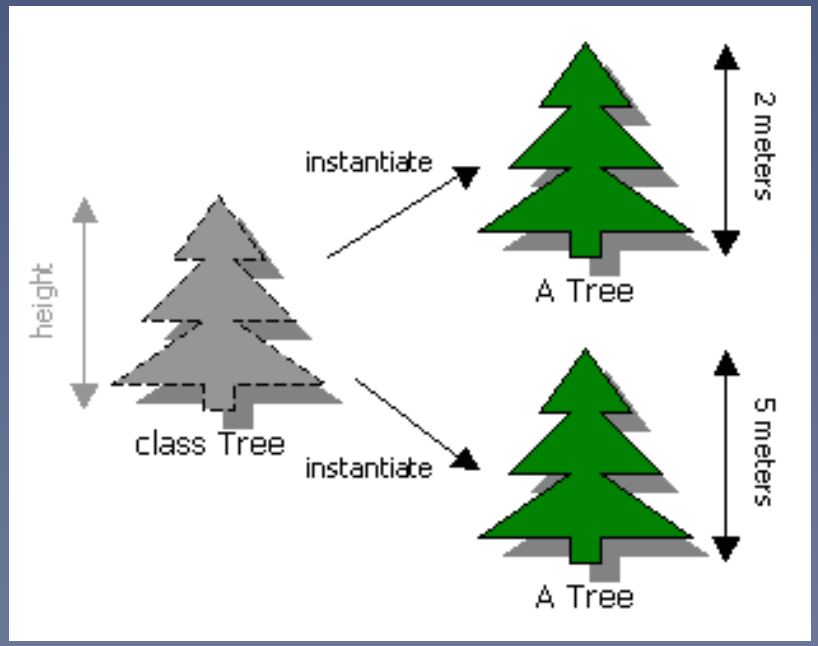
Fase de Implementação

Classes são **codificadas**, **testadas** e **distribuídas** (*deployed*).

- Numa fase preliminar poderá existir um protótipo do sistema global.
- Unidades procedimentais devem passar por **testes unitários**;
- A junção de diferentes módulos é submetida a **testes de integração**;
- O sistema cresce gradualmente através da junção de novos módulos testados unitariamente e por novos testes de integração.



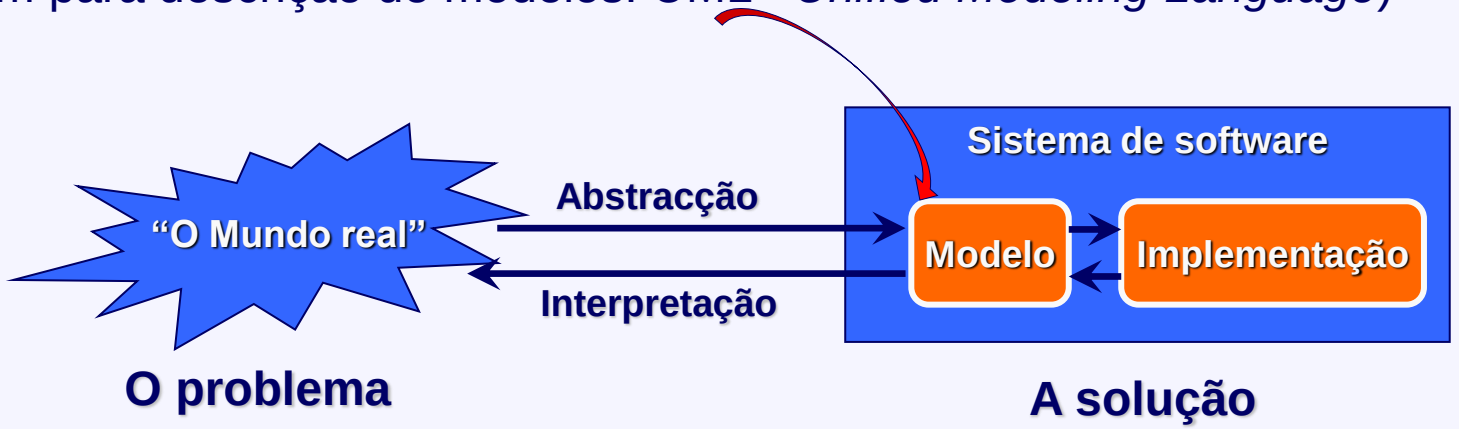
Conceitos OO
e fase de desenho revista



Paradigma *Object-Oriented*

"**modelar o mundo real**".

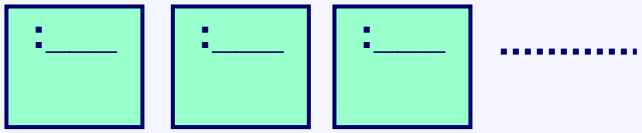
(linguagem para descrição de modelos: UML - *Unified Modeling Language*)



Breve história:

- Os algoritmos (Anos 50 e 60):
  - computação; controlo de fluxo;
- O modelo de dados (anos 70 e 80):
  - a computação foi secundária;
- **Object-Oriented**:
  - **Objectos e classes representam os conceitos do problema.**

Objectos



	Interpretação no Mundo Real	Representação no Modelo
Objecto	Um objecto representa **qualquer coisa** no mundo real, que possa ser distintamente identificado (**identidade**).	Um objecto é caracterizado por ter: • **Identidade** • **Estado** • **Comportamento**

- Cada objecto tem uma **identidade única** (ex: **endereço**).
- A identidade de um objecto distingue-o de todos os outros objectos.
- O **estado** de um objecto é composto pelos **valores de um conjunto de campos**.
  (um objecto **imutável** tem estado constante)
- Um objecto pode transitar entre diversos estados durante o seu ciclo de vida.
- **Dois objectos são iguais se tiverem o mesmo estado**. Isto é, se os valores dos campos correspondentes dos dois objectos forem iguais.
- O **comportamento** de um objecto é definido por um conjunto de **métodos**, que **podem operar sobre o objecto**:
  - para **aceder** ao estado de um objecto (***accessors***);
  - para **modificar** o estado de um objecto (***mutators***).
- **Métodos** são também identificados como **operações**.

Objectos… exemplos

Considere-se a mailbox do voicemail:

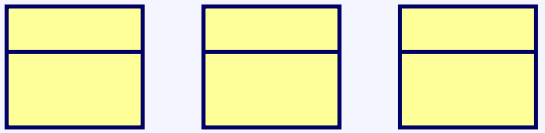
- O objecto mailbox pode estar num **estado**:
  - ***Empty***, logo após a sua criação;
  - ***Full*** quando já não aceitar mais mensagens.

- **Operações** sobre o objecto mailbox (definidas por métodos) afectam o seu estado:
  - A mailbox pode adicionar uma nova mensagem à sua colecção;
  - A mailbox pode retornar uma mensagem.

- O **comportamento** da mailbox é afectado consoante o seu **estado**:
  - ***Empty***: pode dar uma mensagem especial ao utilizador:
    "não tem novas mensagens";
  - ***Full***: rejeita novas mensagens.

- Dois objectos mailbox no mesmo estado, com as mesmas mensagens, terão **identidades** diferentes que distingam os dois objectos.

Classes



	Interpretação no Mundo Real	Representação no Modelo
Classe	Uma classe representa um conjunto de objectos com **características e comportamento similares**. Estes objectos são designados instâncias da classe.	A classe define a **estrutura** dos estados e **comportamentos** partilhados por todas as instâncias.

- Uma classe define um ***template* para criação ou instanciação** das suas instâncias, isto é, **objectos**.
- **O objecto é uma instância de uma classe**.
- Os termos **instância** e **objecto** têm o mesmo significado.
- Em linguagens como o C++ e o Java, **as características dos objectos são especificadas pelas suas classes**, em vez de em cada objecto individualmente.
- Uma classe define:
  - nome e tipos dos **campos de instância**;
  - nome, tipo retorno, argumentos e implementação dos **métodos de instância**.
- Os valores dos campos de instância (estado do objecto) não são fixados pela definição da classe. Cada instância da classe pode ter o seu próprio estado. **Diferentes instâncias poderão ter diferentes estados**.
- A implementação dos métodos é definida pela classe e é igual para todas as instâncias da mesma classe.

  Ou seja, a **implementação é imutável para instâncias da mesma classe**.

Classes… exemplos

A classe **`Mailbox`**:

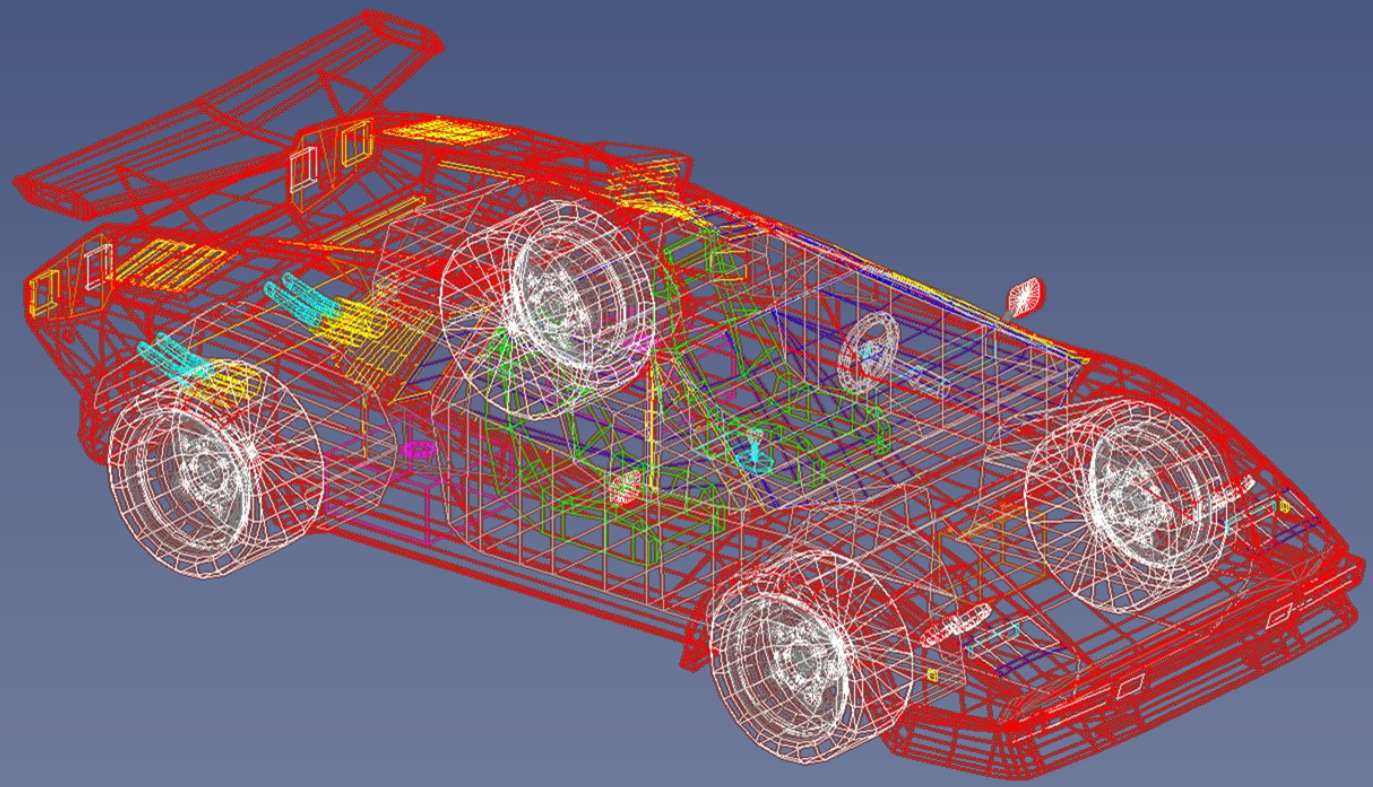
- Define o conjunto de operações disponíveis em todas as suas instâncias: deixar mensagem, devolver mensagem, gravar saudação, etc);

- Define o tipo de informação que pode ser guardado num objecto mailbox:
  - Colecção de mensagens;
  - Índice da mensagem corrente;
  - etc.

- Cada objecto é confinado à definição da sua classe.

- O objecto mailbox é uma instância da classe **`Mailbox`** e uma mensagem uma instancia da classe **`Message`**.

Conceitos OO
e fase de desenho revista



Last modified: 14/03/2023

Classes… Fase de Desenho Revista

Na fase de desenho há que:

1. Identificar as classes que definem os objectos do problema;
2. Identificar as **responsabilidades** dessas classe;
3. Encontrar **relações** e **hierarquias** correctas entre classes.

Este processo é normalmente iterativo

Resultados da fase de desenho:

- Descrição textual das classes e as suas responsabilidades mais importantes;
- Diagramas de classes;
- Diagramas de sequência dos cenários mais importantes;
- Diagramas de estado dos objectos com comportamento altamente dependente do estado.

Classes… Fase de Desenho…1.

1. Identificar Classes:

- Uma ideia simples é que a identificação de classes passa por encontrar os substantivos pertinentes na especificação funcional.

  Ex: Mailbox, Message, User, etc

➔ **Estes são as entidades tangíveis do problemas.**

Outras menos evidentes:

- Eventos;
- Utilizadores e perfis (*roles*);
- Sistema - tipicamente responsável pela inicialização e *shutdown*;
- Interfaces de sistema modelam interfaces com o SO (ex: consola, teclado, janelas, etc), e externos como bases de dados, entre outros.
- Foundation Classes: tipos “básicos” do sistema como `Date`, `Collection`,...;

Classes… Fase de Desenho... 2.

2. Identificar Responsabilidades:

- Enquanto que os substantivos da especificação funcional representam as classes, os **verbos representam as responsabilidades**.

  Exemplo: Mensagens são gravadas, reproduzidas ou apagadas.

Uma responsabilidade pertence exactamente a um única classe.

➔ **Encontrar a classe correcta nem sempre é evidente!**

Exemplo: "Adiciona a mensagem à mailbox"

Será que a responsabilidade de adicionar é da Mensagem?

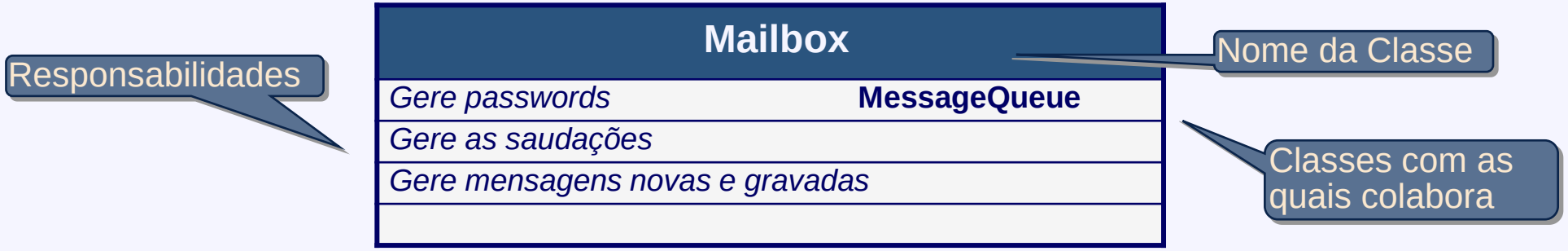
➔ Isso obrigaria a classe mensagem a ter conhecimento da mailbox, o que limitaria a flexibilidade de alterações à classe mailbox. Isto é, sempre que a classe `Mailbox` fosse modificada a classe `Message` teria que ter esse conhecimento.

Classes… Fase de Desenho... 2…

2. Identificar Responsabilidades...

"CRC Cards": técnica proposta por *Beck* e *Cunningham* em 1989:

- Sigla para: **Class Responsibility Collaborator**;
- Catálogo de classes que descrevem as suas responsabilidades de alto nível e colaborações com outras classes;
- Têm pouco detalhe tecnológico;
- Recomendado na discussão de grupo entre vários programadores.



➔ As responsabilidades são de alto nível e não devem ser ligadas a métodos.

Classes… Fase de Desenho... 2…

Continuando a navegar pelo *Use Case* "**Deixar Mensagem**":

…

3. O utilizador introduz o número da extensão do destinatário;
4. O sistema de voicemail **valida o numero da extensão** e responde a mensagem:

➔ "Alguém" dentro do sistema de voicemail precisa localizar a mailbox que tem o número de extensão dado pelo utilizador!

Esta tarefa não é da responsabilidade de **`Message`**, nem **`Mailbox`**.

MailSystem	
gere mailboxes	**Mailbox**

➔ Nesta fase, ainda não é relevante como é que **`MailSystem`** localiza a **`Mailbox`** correcta!

Classes… Fase de Desenho… 3.

3. Encontrar relações e hierarquias.

As três relações mais comuns entre classes são:

- **Dependência** (“usa”);
- **Associação** (“tem”);
- **Herança** (“é”).

➔ **A modelação torna-se ainda mais relevante nesta etapa.**

Modelar Estruturas e Relações

Modelar Estruturas e suas Relações

O **diagrama de classes UML** serve para modelar as **estruturas estáticas** de um sistema de software OO e os vários tipos de relações entre elas. Consiste:

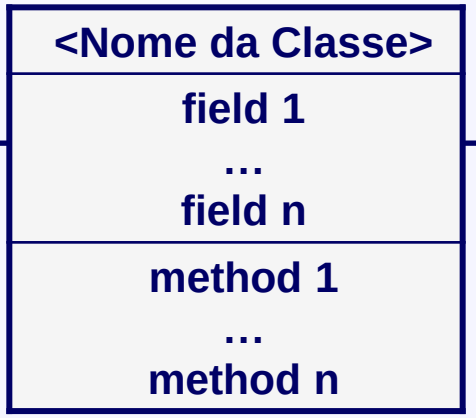
- num conjunto de nós que representam **classes** e **interfaces**;
- um conjunto de ligações que representam relações entre os tipos.

As **relações** podem ser modeladas como:

- **Dependência**;
- **Associação**, incluindo agregação e composição;
- **Herança**, incluindo extensão e implementação.

Modelar Estruturas e Relações

Notação UML para classes



- Cada campo além do nome pode ainda indicar:
  - Acessibilidade, tipo e valor inicial.

- Os campos podem ser apresentados na seguinte sintaxe:

```
[<acessibilidade>][<tipo>] <nome> [= <valor inicial>]
```

- Os campos podem ser ainda apresentados com a seguinte sintaxe alternativa, de acordo com o standard UML:

```
[<acessibilidade>] <nome>[: <tipo>][= <valor inicial>]
```

- Os métodos podem também ser apresentados nas duas sintaxes:

```
[<acessibilidade>][<tipo>] <nome> ([parametro1, ...])
```

ou:

```
[<acessibilidade>] <nome> ([parametro1, ...])[: <tipo>]
```

Notação para acessibilidade

UML sintaxe para acessibilidade:

Acessibilidade	Sintaxe Java	Sintaxe UML
public	`public`	**+**
protected	`protected`	**#**
package		**~**
private	`private`	**–**

	Atributo			
Acessibilidade	**`public`**	**`protected`**		**`private`**
Na própria classe	√	√	√	√
No mesmo package	√	√	√	X
Subclasse noutro package	√	√	X	X
Não subclasse noutro package	√	X	X	X

Exemplos

Exemplos de declaração de campos:

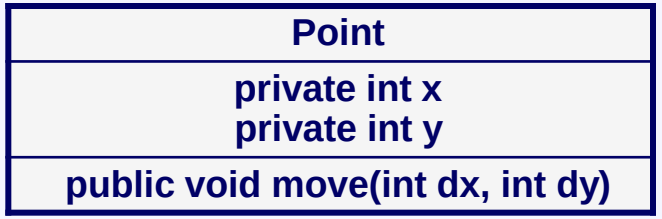
```
Date birthday
birthday: Date

public int duration = 100
+duration: int = 100
```

Exemplos de declaração de métodos:

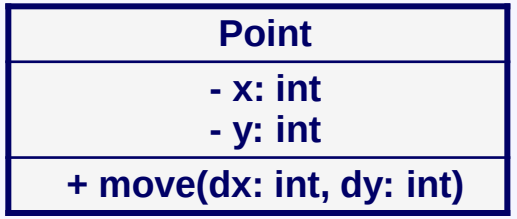
```
void move(int dx, int dy)
~void move(dx:int, dy:int)

public int getSize()
+getSize(): int
```

Exemplos de diagramas de classes

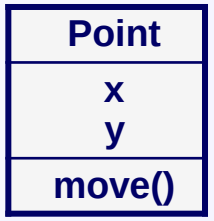
Sintaxe Java:



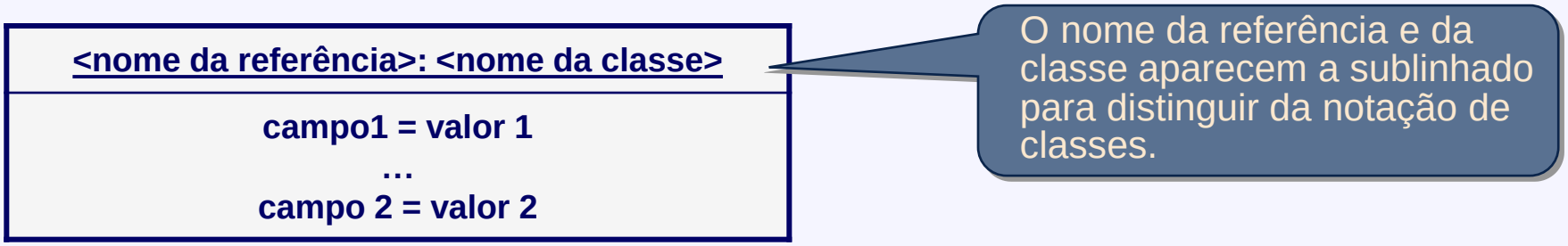
Sintaxe UML:



Formas abreviadas:

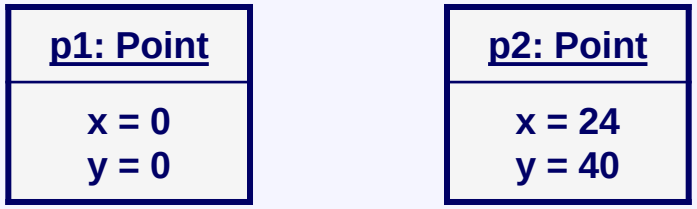


Notação UML para Objectos



- Opcionalmente poderá ser omitida uma das partes: referência ou nome da classe.

Por exemplo, instâncias da classe `Point` com os estados (0,0) e (24,40) podem ter a seguinte representação:

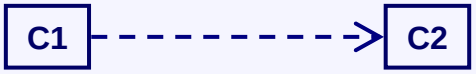


Modelar Estruturas e Relações

Dependência

- Relação entre entidades onde:
  - uma operação de uma entidade depende da presença de outra entidade;
  - alterações numa entidade podem afectar o comportamento da outra entidade.

- Uma situação usual de dependência é a relação "**usar**":
  - A classe C1 depende da classe C2, se C1 usar C2 em lugares como parâmetros, variáveis locais ou tipo de retorno de métodos.



Exemplo:

- A classe `Message` não necessita de usar a classe `Mailbox`.
  Nem precisa de saber que é armazenada numa classe `Mailbox`.
  Contudo a classe `Mailbox` usa a classe `Message`.

Regra:

- Minimizar o número de relações de dependência entre classes, isto é minimizar o acoplamento entre classes.
- Um nível baixo de acoplamento entre classes aumenta a flexibilidade facilitando futuras alterações.

Associação

- Representa uma **relação binária genérica entre duas classes**.
- Pode ser implementada de diferentes formas, mas é usual que uma, ou ambas as classes, na relação de associação tenham referências (campos) **directas** ou **indirectas** para a outra classe.

  Exemplo:

```
class Student{Course course;…}
class Student{List<Course> courses;…}
```

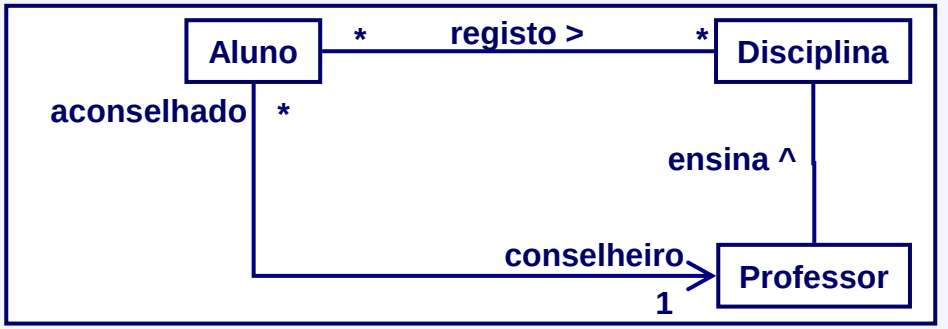
referência directa

referência indirecta

Notação UML: uma linha contínua a ligar as duas classes.

Opcional:

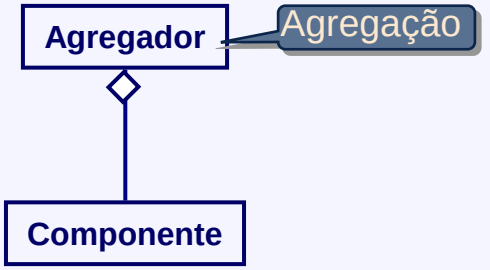
- **nome e direcção**. Exemplo:
  *um aluno **regista-se** em disciplinas*
  *um professor **ensina** alunos*



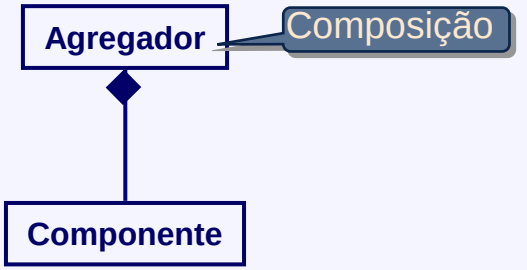
- **multiplicidade e referências**. Exemplo:
  *um aluno tem uma faculdade como **conselheiro***
  *um professor tem alunos como seus **aconselhados***
  uma disciplina tem alunos

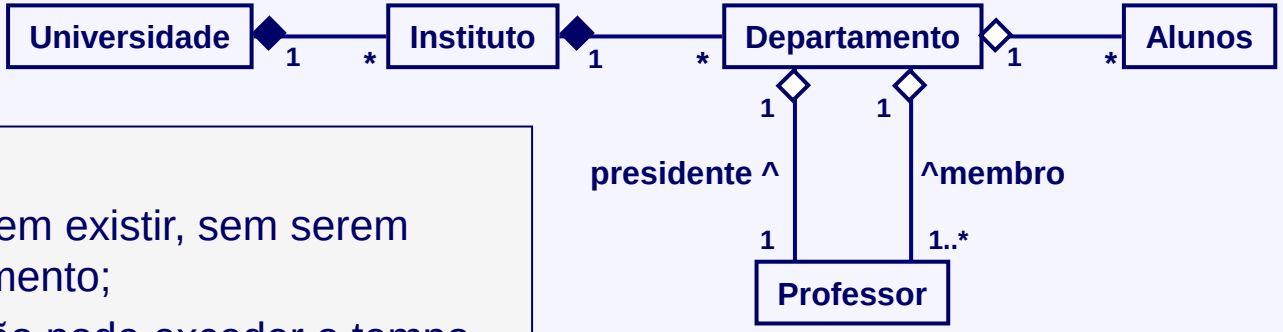
Associação… Agregação e Composição

- **Agregação** é uma forma especial de associação.
- Define uma relação ***ter-um*** (**agregador**)(“***has-a***”), e uma relação ***parte-de*** (**componente**) (“***part-whole***”) um todo.
- Relação estrutural que distingue o **todo** (**classe agregadora**) das suas **partes** (**classe componente**).



- **Composição** é uma forma mais forte de agregação.
- Em composição o **tempo de vida do componente** é completamente **dependente do tempo de vida do agregador**.





Exemplo:

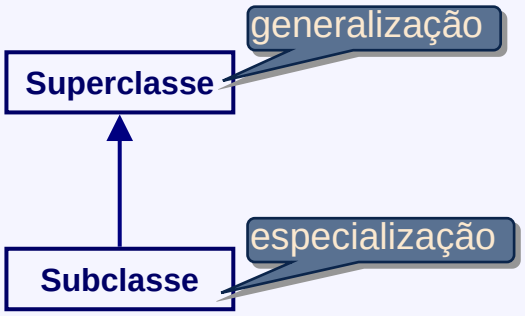
- Um professor e um estudante podem existir, sem serem parte de um determinado departamento;
- O tempo de vida de um instituto não pode exceder o tempo da sua universidade, nem um departamento pode exceder o tempo do seu instituto.

Herança – *is-a(n) relationship*

Diferentes tipos de heranças:

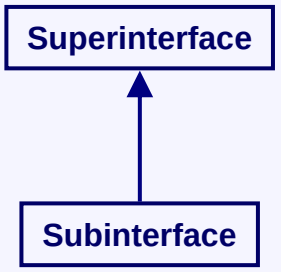
- Relação de **extensão** entre classes

  Quando uma classe **C2 estende a classe C1**, a classe C2 é conhecida como **subclasse** de C1 e a classe C1 é conhecida como a **super classe** (ou **classe base**) de C2.



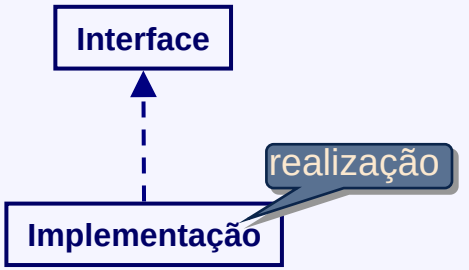
- Relação de **extensão** entre interfaces

  Quando uma interface **I2 estende uma interface I1**, a interface I2 é conhecida como **subinterface** de I1 e a interface I1 é conhecida como a super interface (ou interface base) de I2.

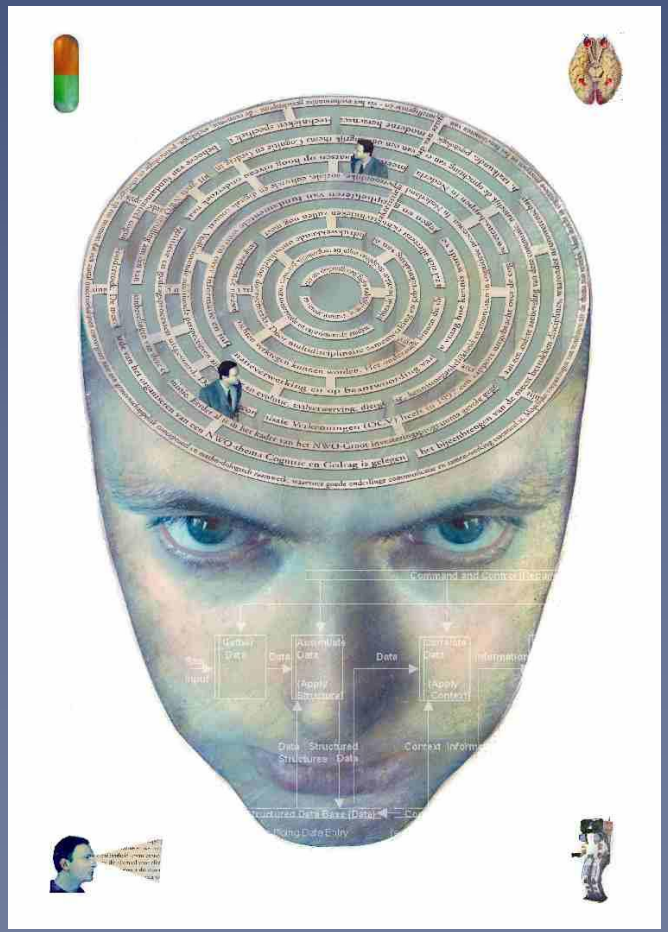


- Relação de **implementação** entre uma classe e uma interface

  Quando a classe **C2 implementa uma interface I1**, a classe C2 é conhecida como **a implementação** de I1 e a interface I1 é conhecida como a **interface** de C2.



Modelar Comportamento Dinâmico



Princípios

Transmissão de mensagens:

- **Objectos comunicam entre si através da transmissão de mensagens**.
- Uma mensagem representa um comando enviado a um objecto (identificado como **recipiente** ou **receptor**).
- O envio de uma mensagem a um objecto reflecte-se na execução de uma acção através da invocação de um método.
- Uma mensagem é composta pela identificação do:
  - objecto **receptor**;
  - **método** a ser invocado;
  - **argumentos** do método (opcional).
- **Transmissão de mensagens** ⇔ **Invocação de métodos**.

Exemplo: p1.move(10, 20)

- Receptor: `p1`
- Método: `move()`
- Argumentos: `(10,20)`

Diagrama de Sequência…

Especifica a interacção entre objectos, evidenciando o tempo e ordem de invocação dos métodos.

- O eixo dos y's representam o **tempo na direcção descendente**;
- No eixos dos x's são dispostos os **objectos que participam na interacção**, representados por colunas;
- Usualmente, o **objecto que inicia a interacção** é colocado na coluna mais à esquerda, com os objectos subordinados à sua direita;

- O objecto representado por cada coluna é colocado no topo e uma linha vertical tracejada mostra o seu **tempo de vida**.
- As **barras rectangulares** sobre o tempo de vida indicam a **duração de execução de um método** do objecto.
- As **linhas horizontais** representam a **criação de objectos** ou **invocação de métodos**.

  Opcionalmente a resposta do método.

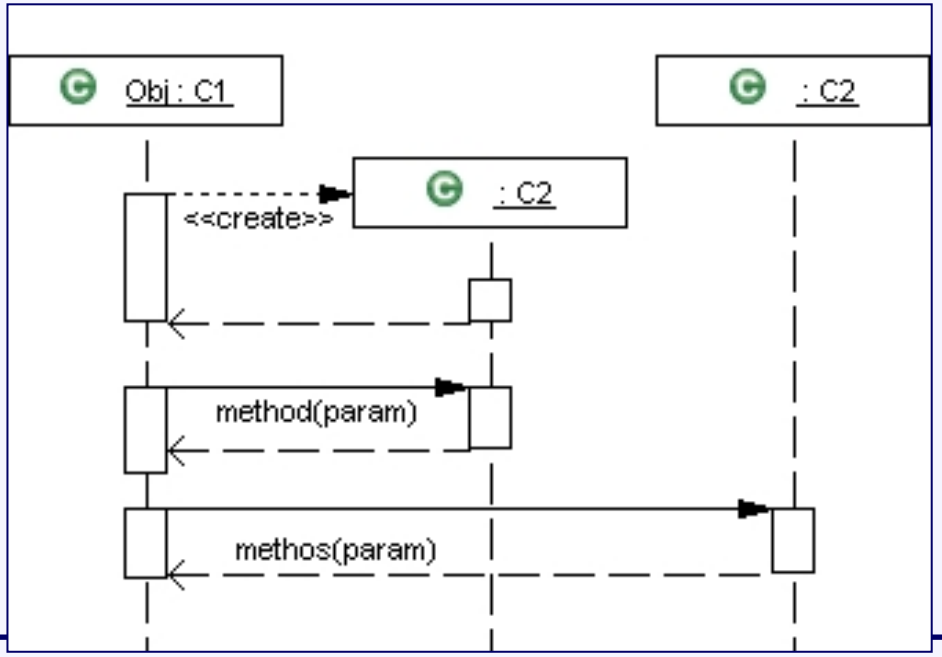
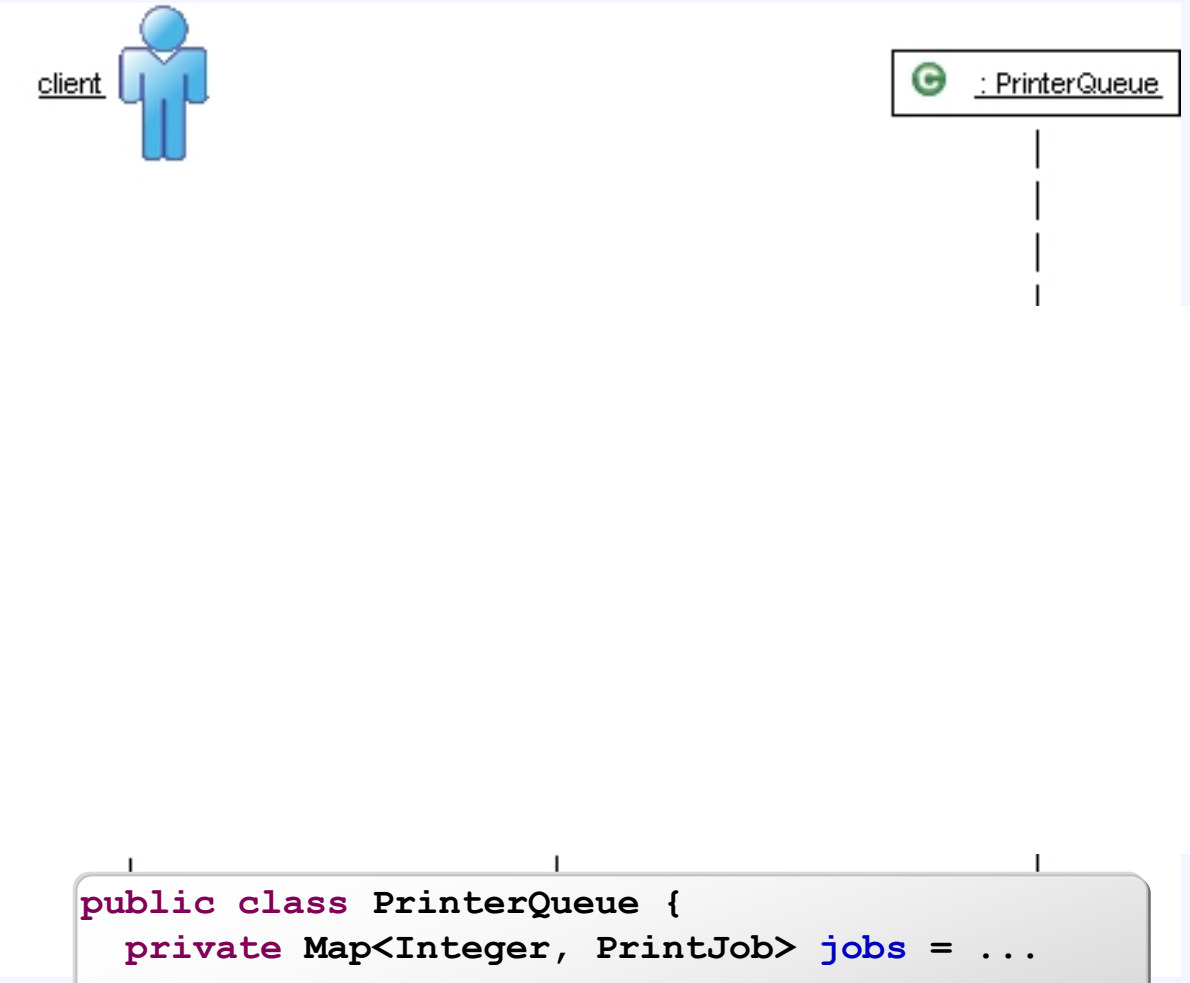


Diagrama de Sequência… Exemplo

1. O cliente cria uma instância de **`PrintJob`**.
2. O cliente invoca o método **`submit`**() de **`PrintJob`**, passando um documento como parâmetro.
3. A instância de **`PrintJob`** referida por **`pj`**, adiciona-se a si própria à instância de **`PrintQueue`**.
4. A instância de **`PrintQueue`** invoca o método **`assignJobNr`**() da instância de **`PrintJob`** referida por pj.
5. O método **`assignJobNr`**() retorna.
6. O método **`add`**() retorna.
7. O método **`submit`**() retorna.
8. Mais tarde, quando for processado a instância de **`PrintJob`** referida por **`pj`**, será invocado o seu método **`print`**().
9. O método **`print`**() retorna.

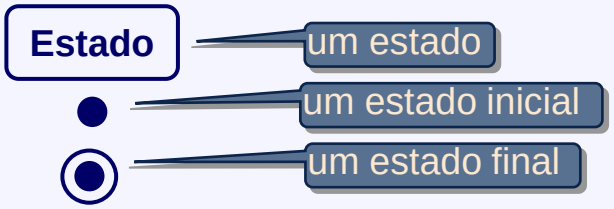



```
public class PrinterQueue {
  private Map<Integer, PrintJob> jobs = ...

  public void add(PrintJob job){
    jobs.put(jobNr++, job);
    job.assignJobNr(jobNr);
  }
  public void doWork(){... job.print() ...}
```

Diagrama de Estados

Especifica o controlo do fluxo através do conceito de **estado** e **transições**:

- Diagramas de estados são uma generalização da tradicional **máquina de estados finita**.
- **Estado** = situação no ciclo de vida de um objecto em que é:
  - satisfeita uma dada condição (**guarda**);
  - desencadeada determinadas **acções**;
  - aguardada a ocorrência de certos **eventos**.
- **Transição** = **relação entre dois estados** que indica que um objecto no primeiro estado (**estado fonte**) desencadeia determinadas acções e entra no segundo estado (**estado destino**), quando ocorre um certo evento ou são satisfeitas determinadas condições.
- O ciclo de vida de um objecto começa no **estado inicial** e termina no **estado final**.



Estado
entrada: acções
saída: acções

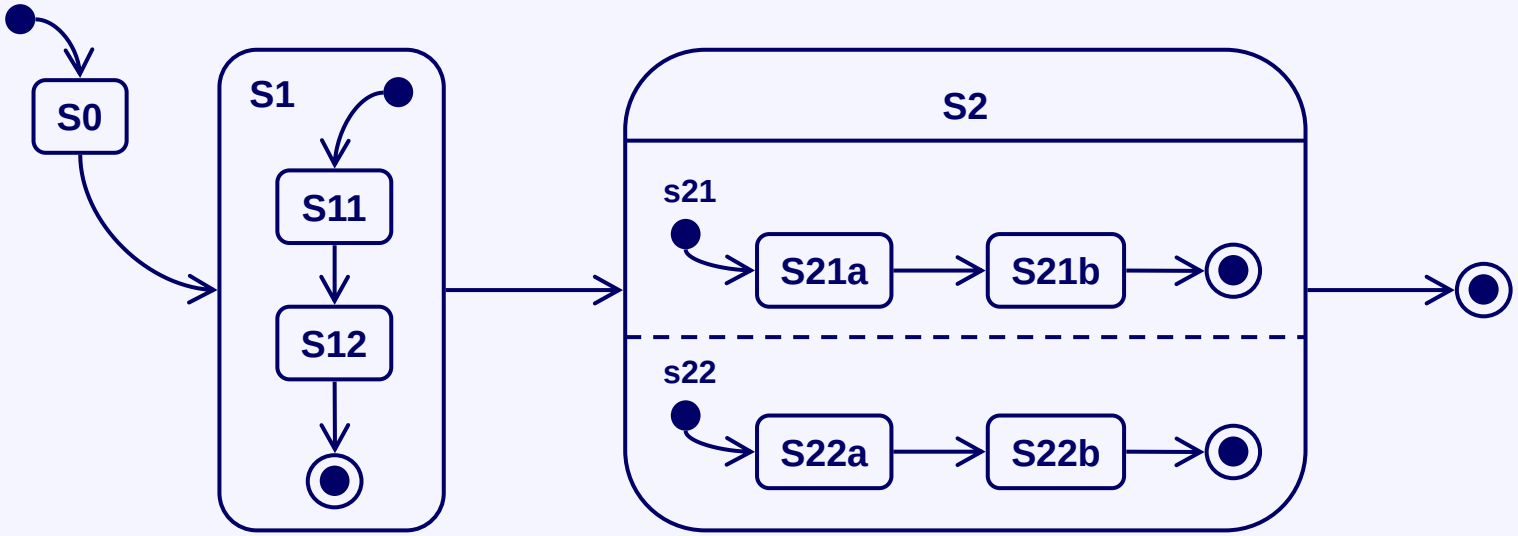
um estado pode ainda ter uma lista de acções a serem executadas quando o objecto entra ou sai desse estado.

dá-se a transição quando:
- **ocorre um dos eventos** especificados na lista de eventos;
- **todas as guardas** (condições são avaliadas a `true`),

quando se dá a transição:
- **são executadas as acções** especificadas.

Diagrama de Estados… aninhado (*nested)*

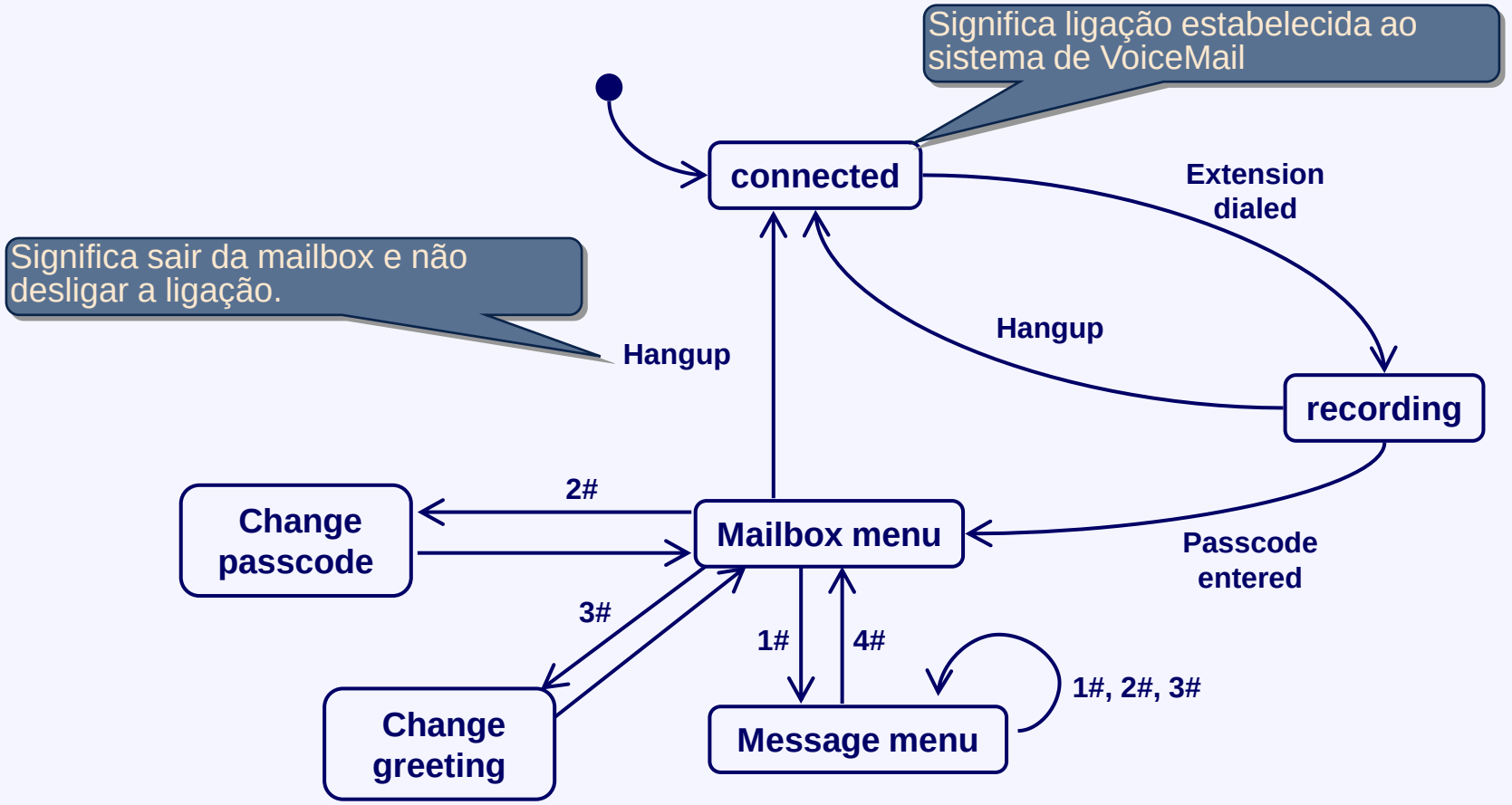
Diagramas de estados aninhados são também conhecidos como **estados compostos** ou **super estados**.



estado composto sequencial

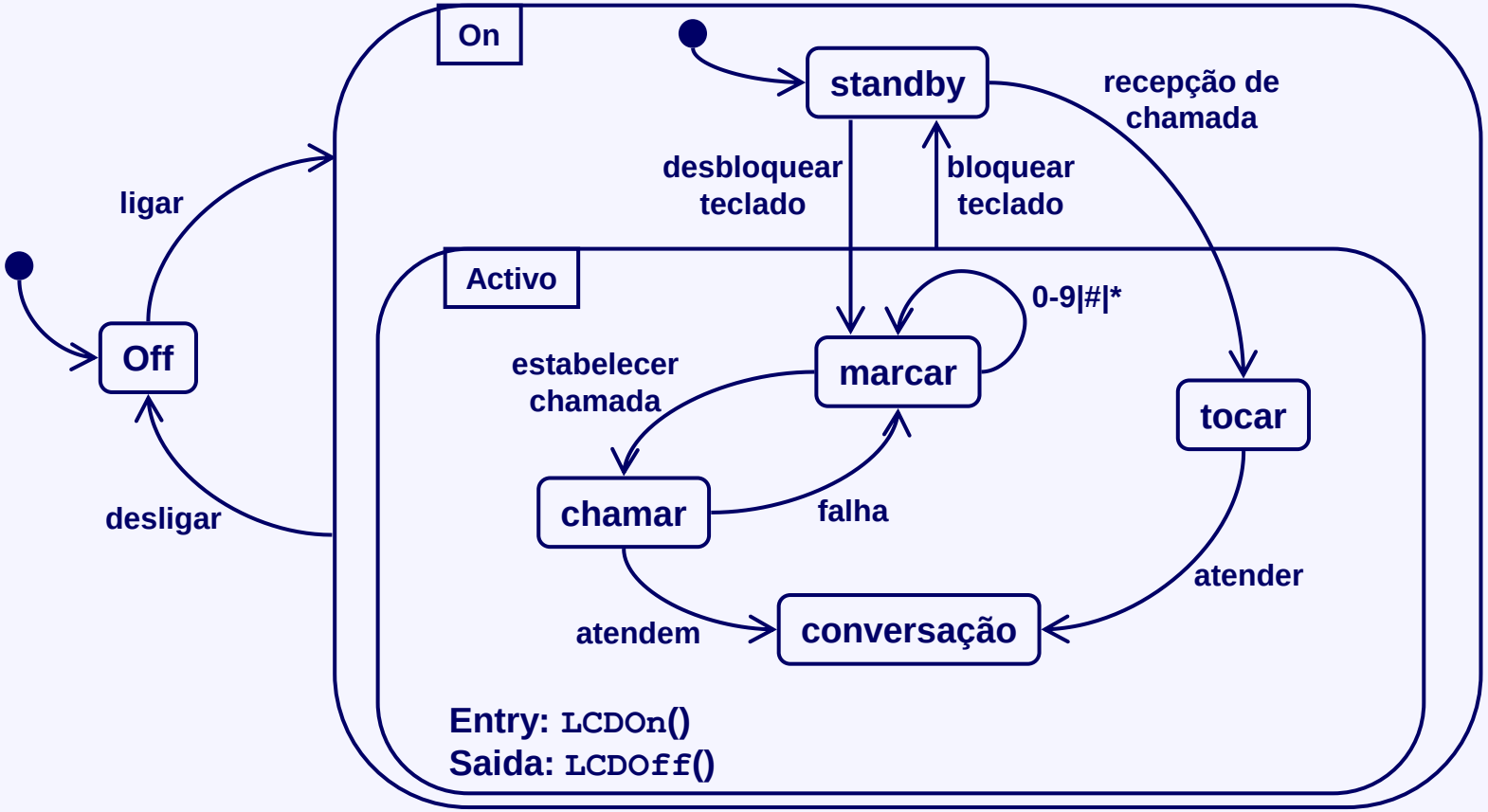
estado composto concorrente:
- quando uma transição entra num **estado composto concorrente**, entra simultaneamente nos dois estados iniciais dos **dois subestados concorrentes**.
- em cada momento o objecto está simultaneamente num dos estados dos diagramas aninhados, de cada subestado concorrente.

Exemplo 1


Significa ligação estabelecida ao sistema de VoiceMail
connected
Extension dialed
Significa sair da mailbox e não desligar a ligação.
Hangup
Hangup
recording
2#
Change passcode
Mailbox menu
Passcode entered
3#
1#
4#
1#, 2#, 3#
Change greeting
Message menu

Exemplo 2…

Exemplo de diagrama de estados para um telemóvel.



Caso de Estudo do sistema de voicemail

Identificar responsabilidades e colaborações

As mensagens são geridas pela **`Mailbox`** que mantém 2 filas: mensagens gravadas e novas mensagens.

Mailbox	
Gere passwords	**MessageQueue**
Gere as saudações	
Gere mensagens novas e gravadas	

MessageQueue	
Adiciona e remove mensagens	**Message**
numa ordem FIFO	

Message
Conteúdo da mensagem

Como é que são geridas as mailboxes?

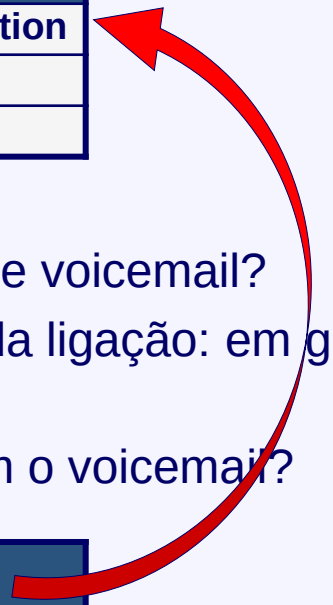
MailSystem	
manage mailboxes	**Mailbox**

Identificar responsabilidades e colaborações…

Como é processado o *input* e *output* do utilizador?

Existirá uma classe **`Telephone`** para simular o equipamento móvel:

Telephone	
Processa input do utilizador do teclado, microfone e desligar.	**Connection**
Reproduz mensagens.	



- Pode o telefone interactuar directamente com o sistema de voicemail?

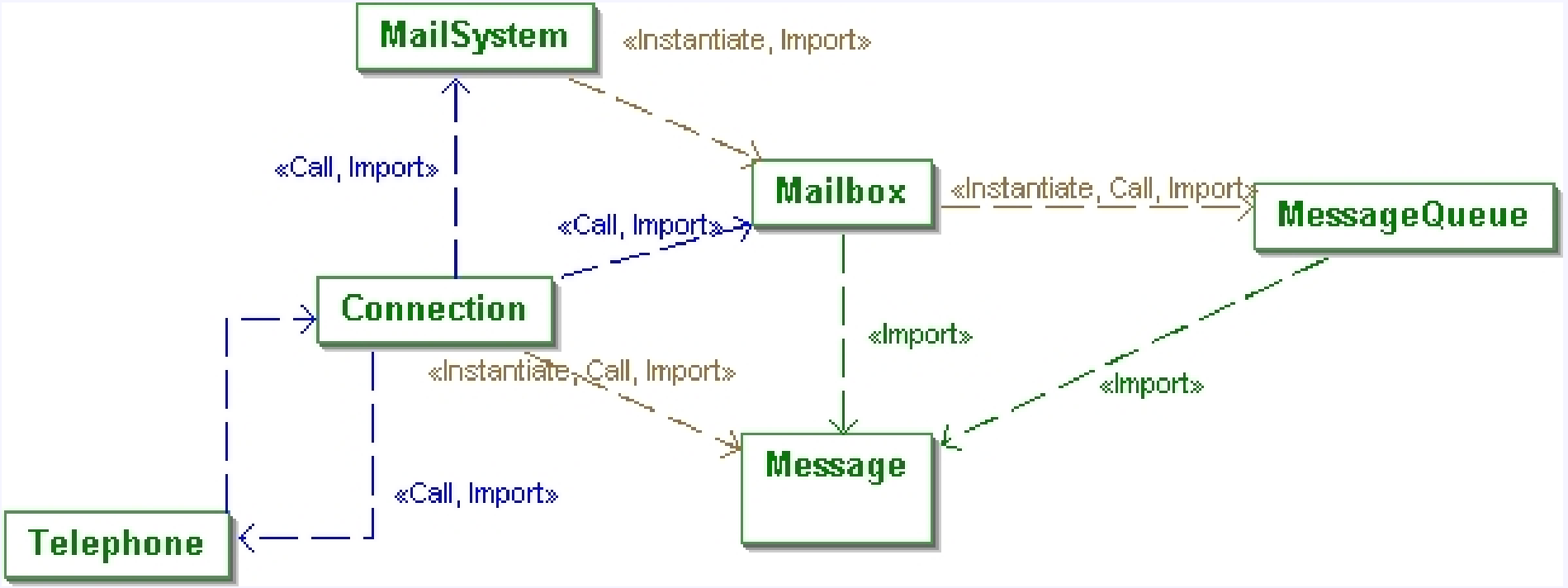
  A resposta aos inputs do utilizador dependem do estado da ligação: em gravação, reprodução de mensagem, menu de opções, etc.

➔ E se existirem 2 telefones a interactuar em simultâneo com o voicemail?

Connection	
Recebe o input do telefone	**Telephone**
Processa os comandos do utilizador	**MailSystem**
Mantém o estado da ligação	
Grava mensagem	**Mailbox**
	Message

Relações e hierarquias

Relações de dependência:



Relações e hierarquias…

Relações de agregação:

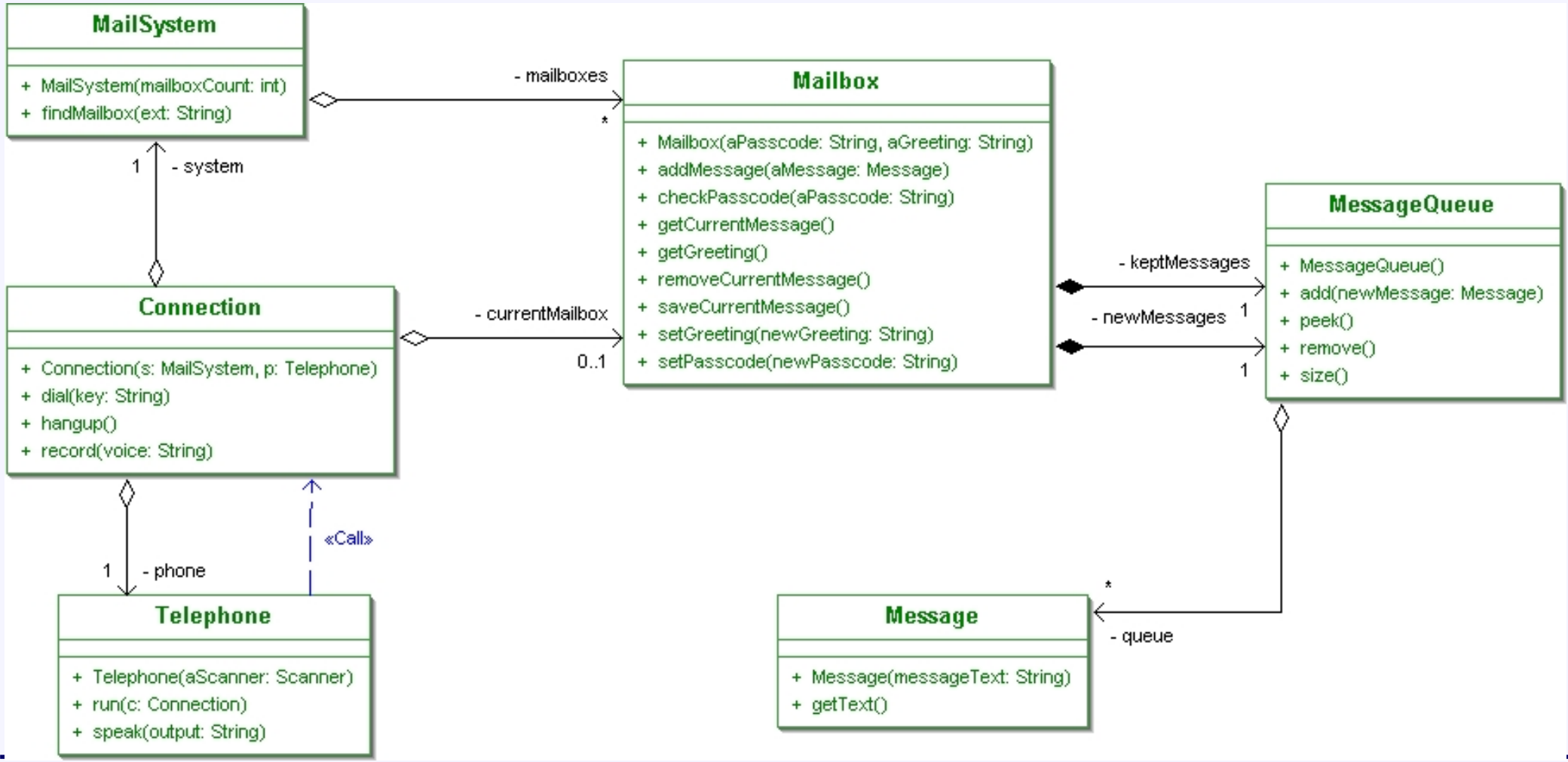
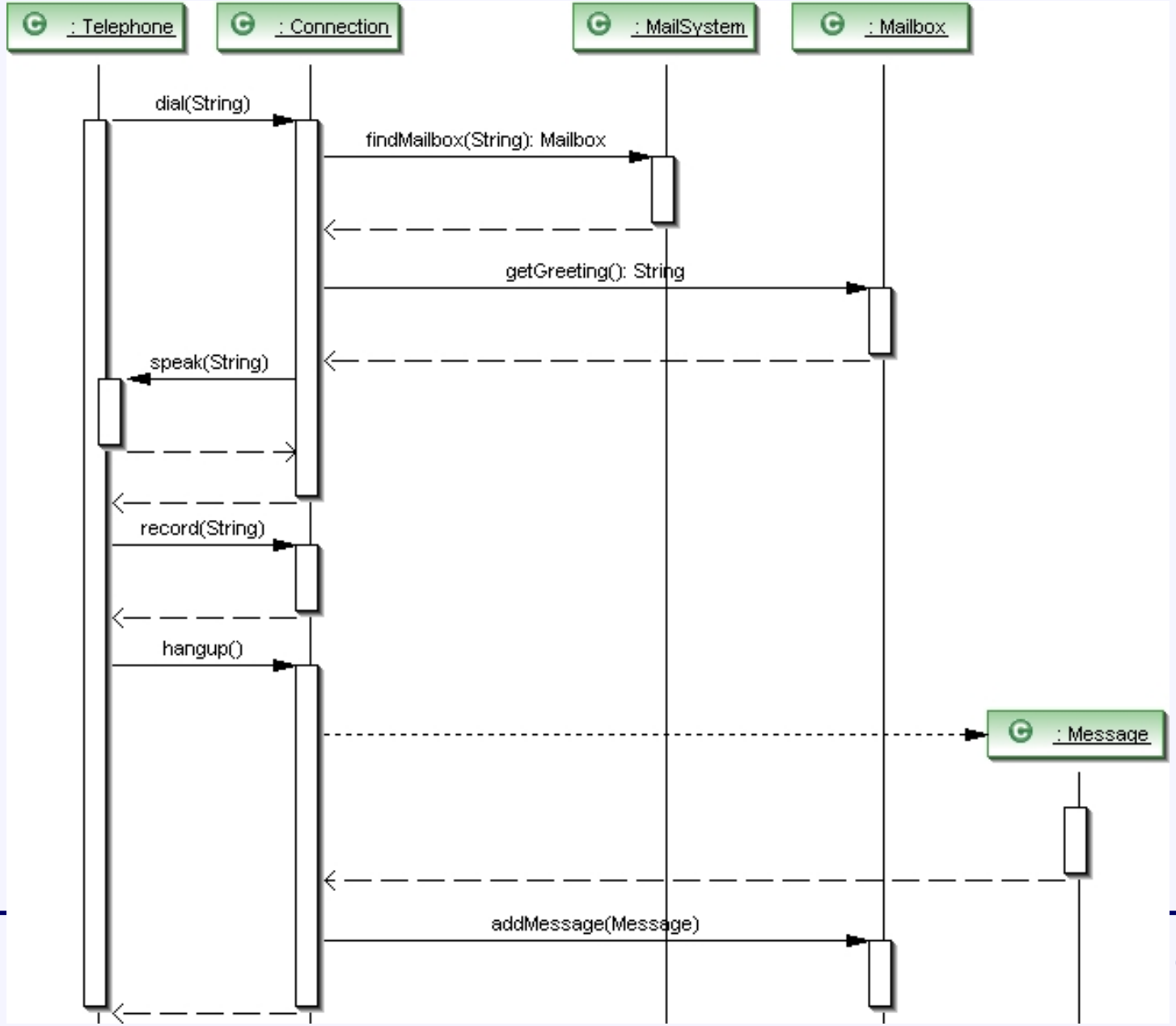


Diagrama de Sequência de “Deixar mensagem”



ANEXO - Use Cases: conceitos e relações

Conceitos

Descreve o comportamento funcional do sistema na forma de **interacções** entre o sistema e as entidades externas, conhecidas como **actores do sistema**.

- **Actor**: representa o **papel** desempenhado por um **conjunto de entidades** que interagem com o sistema (pode ser um utilizador do sistema ou outro sistema).
- **Objectivo**: descrever **o que** o sistema faz e **não como**.
- Cada *use case* tem um **nome** e um **conjunto de cenários** (pode ainda ter pré condições).

Cenário: descrição informal do **conjunto de interacções** entre os actores e o sistema:

- **cenário principal**: descreve o fluxo normal de eventos e seu resultado;

  opcionalmente tem:
- **cenários alternativos**: fluxo de eventos alternativo ao principal.
- **cenários excepcionais**: quando ocorrem erros ou excepções no fluxo normal.

Descrição usual de um cenário:

- um parágrafo que descreve o fluxo de eventos;
- uma tabela de duas colunas que descrevem os eventos de entrada gerados por actores e as resposta produzidas pelo sistema.

Exemplo… **Use Case: Compra**

Pré condição: cliente previamente registado.

Cenário principal do Use Case Compra:

Eventos do actor Cliente	Eventos do sistema e respostas
Log on	• Apresenta uma mensagem de boas vinda e pede um identificador e *password*.
Inserção da identificação e *password*	• Autentica do cliente; • Autenticação do cliente com sucesso.
Repete até estar concluído: • procura títulos; • selecciona um título.	• Apresenta informação sobre os títulos; • Adiciona um título ao carrinho de compras.
Compras feitas e *check out*	• Apresenta o conteúdo do carrinho de compras e endereço de entrega e facturação.
Confirma encomenda e meio de pagamento	• Valida o meio de pagamento; • Validação com sucesso; • processamento da encomenda, emissão de um recibo electrónico e notificação de logística.
Log off	• Apresenta uma mensagem de agradecimento e volte sempre.

- **Cenário alternativo**: o cliente grava o carrinho de compras e termina sem *check out*.
- **Cenário excepcional**: a autenticação do cliente falha

  ➔ repetir o procedimento de *log on*.
- **Cenário excepcional**: a validação do meio de pagamento falha

  ➔ pedir ao cliente um novo meio de pagamento.

Relações

Podem ser representados os seguintes tipos de relações em use cases:

- **extensão ou generalização entre actores**

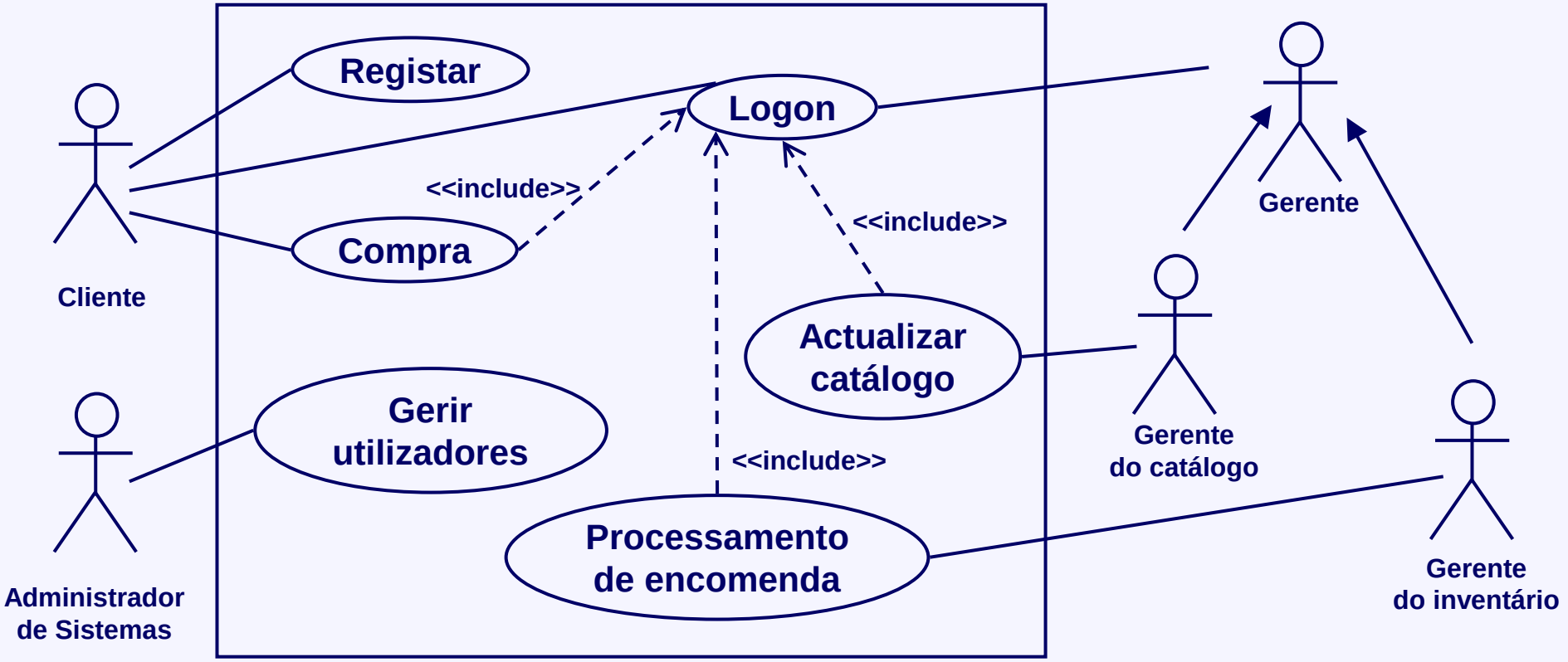
  Se um actor a2 estende do actor a1, significa que a2 representa um subconjunto das entidades representadas por a1.

- **associação entre actores e use cases**

  Um actor está associado a um use case se for participante nas interacções descritas num dos cenários do use case.

- **dependência entre use cases: *include* ou *extend***
  - c1 inclui c2 significa que o use case c2 faz parte do use case c1
  - c2 estende c1 significa que o use case c2 é uma especialização do use case c1.

Exemplo


Registar
Logon
<<include>>
Compra
<<include>>
Actualizar
catálogo
Gerir
utilizadores
<<include>>
Processamento
de encomenda
Cliente
Administrador
de Sistemas
Gerente
Gerente
do catálogo
Gerente
do inventário