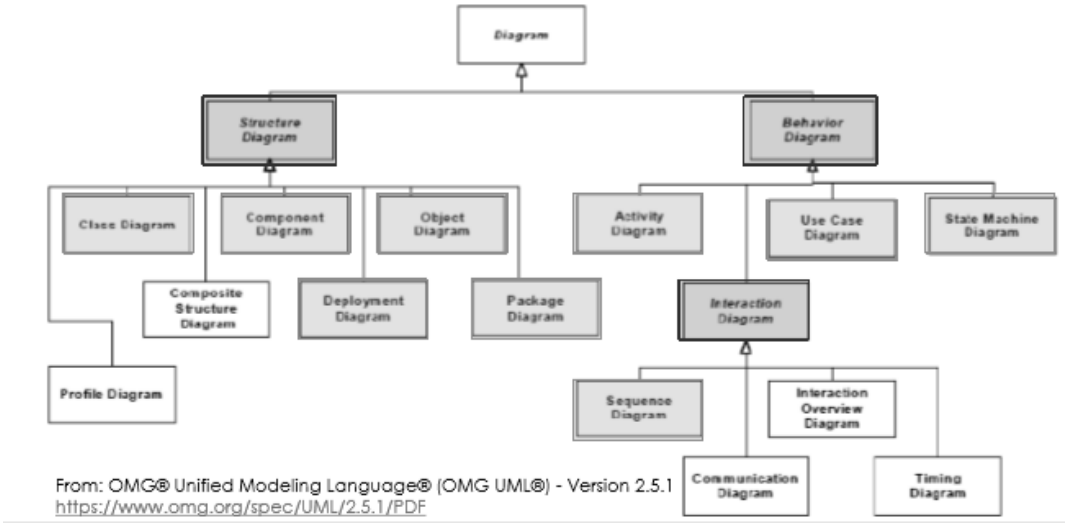
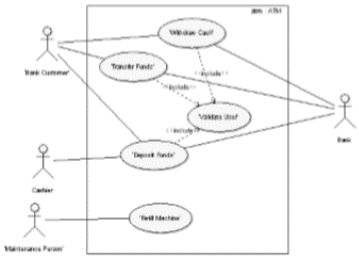


01-Use Cases

UML (Unified Modeling Language) é uma linguagem gráfica para visualizar, especificar, construir e documentar os artefactos de um sistema intensivo de software.

Uma linguagem de modelação que permite especificar, construir, visualizar e documentar sistemas de software



Modelo de Casos de Uso

Descreve como o sistema será usado

Descreve do ponto de vista dos utilizadores as funcionalidades do sistema que deverão ser construídas na solução proposta

Representação gráfica e simplificada da descrição do que o sistema deverá fazer

Modelos usados na análise de requisitos para capturar as necessidades dos utilizadores e as interações com o sistema, e servem com meio de comunicação entre o analista e o utilizador

Permitem:

Apresentar a utilidade do sistema

Especificar quem interage com o sistema (Ator) e com que objetivo (Casos de Uso)

Identificar/capturar as funcionalidades do sistema (Casos de Uso) do ponto de vista dos utilizadores

Compostos por:

Casos de Uso (funcionalidades)

Atores (entidades externas que interagem com o sistema)

Associações entre os Casos de Uso e os Atores

Fronteira do Sistema- Elemento que representa a fronteira do sistema, sendo comum colocar os casos de uso como estando dentro do sistema e atores como estando fora do sistema

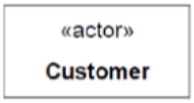
Associações

Interação entre os atores e os casos de uso

Descrevem o envio e/ou receção de mensagens entre os atores e o sistema

Representadas por uma **linha** que liga o ator a um caso de uso

Atores



Pode ser uma pessoa, um sistema (software), hardware etc.

Interação consiste na troca de mensagens entre um ator e o sistema

Encontram-se fora do sistema, não são entidades nem componentes do sistema

Papel que um utilizador desempenha em relação ao sistema, onde poderá interagir com um ou mais caso de uso

Quando várias pessoas desempenham o **mesmo papel** estamos perante o **mesmo ator**

Quando a mesma pessoa desempenha **vários papéis** então estamos perante **vários atores**.

Papéis abstratos e não pessoas concretas

Os atores podem ser **ativos**, quando iniciam eventos, ou **passivos**, quando apenas recebem informação do sistema

Os atores podem generalizar outros atores, onde um ator (costumer) representa um conceito mais geral que o outro (comercial costumer)

Caso de uso

Descrição de um conjunto de sequências de ações

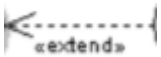
É uma funcionalidade do sistema do ponto de vista dos utilizadores, e fornece uma visão de alto nível do comportamento para alguém ou algo fora do sistema

A notação para os casos de uso são **elipses**

Deve ser detalhado:

- Nome, Descrição, Atores que interagem
- Pré-condições (casos de uso ou cenários necessários que tem de ocorrer antes deste caso de uso)
- Pós-condições (cenários ou situações que tem que acontecer após este caso de uso ocorrer)
- Fluxo de eventos- normal e alternativo (sequência numerada e ordenada dos eventos que ocorrem entre utilizadores e sistema)
- Requisitos especiais - requisitos que não são considerados no caso de uso, mas que devem ser tratados

Dependências



<<extend>> é usado para incluir o comportamento **opcional** de um caso de uso noutro caso de uso, quando um caso de uso é semelhante a outro, mas que faz um pouco mais ou um pouco diferente.

Os atores que têm um relacionamento com o caso de uso que passa a ser estendido, podem executar tanto o caso de uso como as suas extensões.

Estas dependências são representadas por **setas tracejadas com o estereotipo <<extend>>**, ligando o use case de extensão ao use case básico

<<include>> é usado para incluir o comportamento **comum** de um caso de uso incluído num caso de uso básico, a fim de apoiar a reutilização do comportamento comum

Os casos de uso podem conter a funcionalidade de outro caso de uso como parte do seu processamento normal.

Nas dependências do tipo <<include>> poderá não haver ator associado ao caso de uso partilhado. No caso de existir não significa que esse ator execute os casos de uso ligados ao que está partilhado

Estas dependências representadas por **setas tracejadas com o estereotipo <<include>>**, ligando o use case básico ao use case comum

Importante

O nome dos casos de uso deve começar com um **verbo**

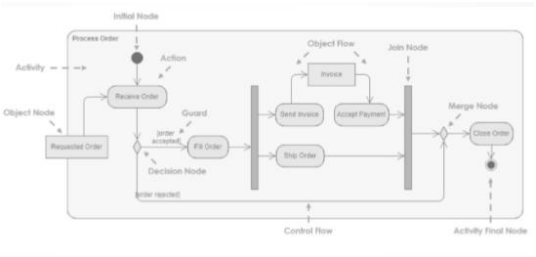
O nome dos casos de uso **não deve ser vago** (por exemplo, “Registar novo utilizador” em vez de “Registar”)

Os atores devem ser nomeados no **singular** (ou seja, "cliente", não "clientes", não “Ana Paula")

O diagrama mostra apenas a funcionalidade do sistema, não qualquer tipo de comportamento ou sequência

<<Include>> e <<extende>> deve ser usado quando estritamente necessário. Caso contrário, estamos a adicionar complexidade desnecessária

O nível de detalhe deve ser equilibrado entre informações insuficientes e um diagrama muito complexo



02-Diagramas de Atividade

Apresenta o fluxo de trabalho desde o ponto inicial até o ponto de chegada, detalhando os muitos caminhos de decisão que existem na progressão de eventos contidos numa atividade.

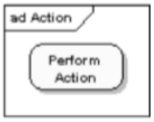
Os diagramas de atividades destinam-se a modelar os processos computacionais e organizacionais (ou seja, fluxos de trabalho), bem como os fluxos de dados que se cruzam com as atividades.

Os diagramas de atividade podem ser utilizados para detalhar situações em que o processamento paralelo pode ocorrer na execução de algumas atividades.

Atividades

Uma atividade é a especificação de uma sequência parametrizada do comportamento.

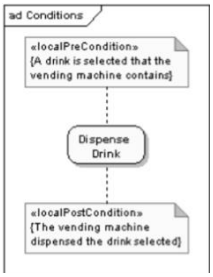
Uma atividade é mostrada como um **retângulo com cantos arredondados**, envolvendo todas as ações, fluxos de controlo e outros elementos que compõem a atividade



Ação

Uma ação representa uma única etapa dentro de uma atividade.

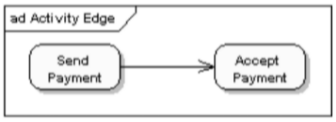
Ações são indicadas por retângulos de cantos arredondados.



Ação com pré-condição ou pós-condição: As restrições podem ser anexadas a uma ação.

Fluxo de Controlo/Direção (Conector)

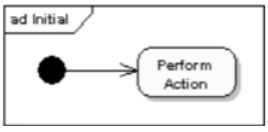
Mostra o fluxo de direção da atividade.



Uma seta de entrada inicia um passo de uma atividade.

Uma vez concluído o passo, o fluxo continua com a seta de saída

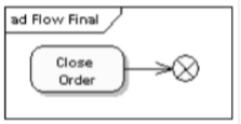
Nó Inicial



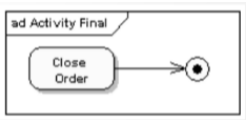
Começo de um **processo** ou **fluxo de trabalho** em um diagrama de atividade.

O nó inicial pode ser utilizado por si só ou com um símbolo de nota que explica o ponto de partida.

Nó Final (Fluxo, Atividade)

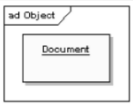


Representa o final de um fluxo de processo específico. Este símbolo não deve representar o fim de todos os fluxos em uma atividade. Nesse caso, use o símbolo de **término**.



Marca o estado final de uma atividade e representa a **conclusão de todos os fluxos** de um processo.

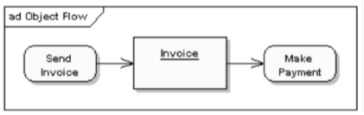
Objeto



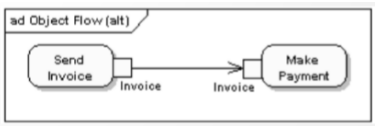
Um objeto é representado por um **retângulo**.

Um repositório de dados é representado como um objeto com a palavra-chave **«datastore»**.

Fluxo de Objetos



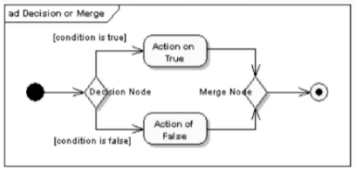
Um fluxo de objetos é um caminho ao longo do qual objetos ou dados podem passar.



Um fluxo de objetos é representado como um conector com uma seta na ponta, denotando a direção em que o objeto está sendo passado.

Um nó de objeto alternativo é representado por um **pin.**

Nó de Decisão

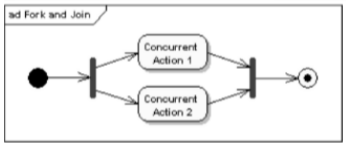


Representa uma decisão e tem pelo menos dois caminhos ramificados e com texto de condição, permitindo aos utilizadores visualizarem opções.

Este símbolo (**forma de diamante**) representa a ramificação ou fusão de diferentes fluxos

Se duas ou mais entradas forem recebidas por um nó de decisão, a ação apontada por sua saída é executada duas ou mais vezes.

Nós de bifurcação ou junção

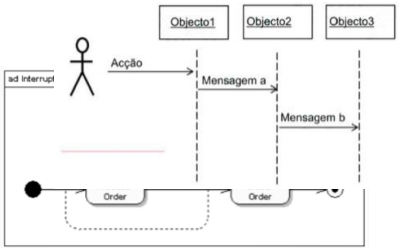


Nó de bifurcação ou junção têm a mesma notação: uma **horizontal** ou **barra vertical** (se está a ser da esquerda para a direita ou de cima para baixo).

Estes Nós indicam o início e o fim de threads de controlo simultâneos.

Um nó de junção é diferente de um nó de decisão, pois a junção **sincroniza dois fluxos internos e produz uma única saída**.

A saída de um nó de junção não pode ser executado até que todas as entradas tenham sido recebidas.



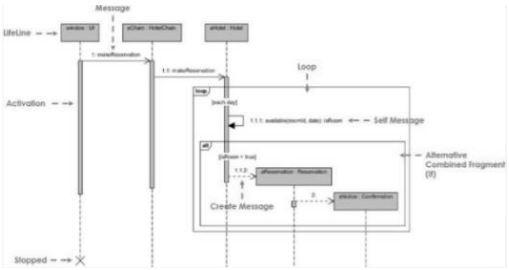
Região de Atividade interrompível

Envolve um grupo de ações que podem ser interrompidas.

No exemplo acima, a ação “Processar pedido" será executada até a conclusão, quando passará o controlo para a ação "Fechar pedido", a menos que seja recebida uma interrupção "Cancelar pedido", que passará o controlo para a ação "Cancelar pedido".

Indica que um sinal está sendo enviado a uma atividade que irá receber.

Demonstra a aceitação de um evento. Após o evento ser recebido, o fluxo que vem desta ação é concluído.



03-Diagramas de Sequência

Objetivos

Modelar a interação entre objetos numa colaboração que realiza uma operação

Modelar interações genéricas (todos os caminhos possíveis através da interação) ou instâncias específicas de uma interação (um caminho através da interação)

Os diagramas de sequência são **diagramas de interação** que detalham como as operações são realizadas, estrutura do sistema envolve: os objetos, as linhas de vida e as mensagens

Capturam:

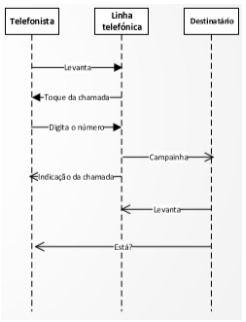
> a interação entre instâncias de objetos que ocorre numa colaboração que realiza um caso de uso ou uma operação (diagramas de instância ou diagramas genéricos)
>
> interações de alto nível entre objetos ativos do sistema- utilizador do sistema e o sistema, entre o sistema e outros sistemas ou entre subsistemas (algumas vezes conhecidos como diagramas de sequência do sistema)

Úteis para descrever uma sequência particular de funcionamento, mas não muitas sequências alternativas.

Há dois tipos de utilização distintas dos diagramas de sequência, consoante a fase do ciclo de desenvolvimento em que nos encontramos:

<u>Documentação dos casos de uso</u>

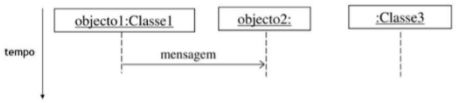
> Interações descritas em termos gerais, sem entrar em pormenores de sincronização
>
> As **setas** limitam-se a indicar eventos, e **não mensagens** no sentido específico entendido em programação
>
> Em geral não se pretende distinguir fluxos de controlo e fluxos de dados



<u>Outras representações de interações entre objetos</u>

> O utilizador (ator) pode ser ilustrado para mostrar a sua interação com o sistema

Elementos básicos



Objetos e Linhas de vida

Cada objeto participante é representado por uma caixa em cima de uma linha vertical a traço interrompido (linha de vida)

Podem aparecer atores (objetos externos ao sistema), normalmente a iniciar interações

O tempo cresce de cima para baixo.

Mensagens

Comunicação entre objetos (emissor e recetor) que veicula informação na expectativa de provocar uma resposta (ação ou atividade).

Uma mensagem é representada por uma seta horizontal, do emissor para o recetor, com o nome e possíveis argumentos

Uma ação de um objeto capaz de provocar uma resposta noutro objeto pode ser modelada como uma mensagem do primeiro para o segundo objeto

Tipos de mensagens

Síncrona (o emissor fica parado à espera de resposta): Corresponde tipicamente a chamada de operação/procedimento no recetor

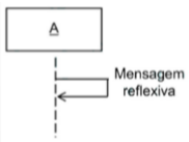
Retorno de mensagem síncrona: desnecessário indicar quando há barras de ativação

Assíncrona: o emissor não fica parado à espera de resposta, corresponde tipicamente a envio de sinal entre dois objetos concorrentes

Simples ou indiferenciada (não se decide se é síncrona, de retorno ou assíncrona): usadas normalmente na modelação de interações na fronteira do sistema (entre atores e o sistema representado por um ou mais objetos

A seta pode ser representada numa posição oblíqua para exprimir atrasos de transmissão, que não sejam desprezáveis relativamente à dinâmica do conjunto.

Um objeto pode também enviar uma mensagem a si próprio. Trata-se de uma mensagem reflexiva (pode não ser uma verdadeira mensagem, mas o início de uma atividade de mais baixo nível que tem lugar no interior do objeto)



Mensagens condicionais, iteradas e com retorno

O valor de retorno de uma mensagem síncrona pode ser indicado na chamada, com atribuição **:=**, ou na mensagem de retorno

Exemplo: ret := msg(args); Nome "ret" será usado em mensagens e condições a seguir, também se escreve "ret" na mensagem de retorno

Uma mensagem condicional é indicada por uma condição de guarda entre parêntesis retos **[ ]**

Exemplo: [x<0] invert(x,color); A mensagem só é enviada se a condição se verificar, condições permitem mostrar várias sequências alternativas num único diagrama

Uma mensagem iterada é indicada com asterisco *****, seguido ou não de uma fórmula de iteração

Exemplo: *[i:=1..n] update(i)

Troca de mensagens entre objetos

Setas sólidas que vão do objeto dado para o objeto pedido

Sintaxe: return := message(parameter:parameterType):returnType

Onde, **return** é o nome do valor de retorno

message é o nome da mensagem

parameter é o nome de um parâmetro da mensagem

parameterType é o nome do tipo desse parâmetro

returnType é o tipo do valor de retorno

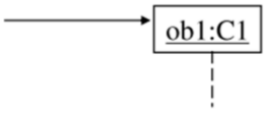
Estereótipos para mensagens

Fronteira (boundary): Classes de interface com o mundo externo (sistemas externos)

Control: Coordenam o comportamento do caso de uso definindo uma interface entre classes fronteira e entidade

Entity: Classes que armazenam informações manipuladas pelo sistema

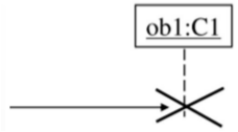
Criação e Destruição de objetos



Criação de objeto é representada por mensagem dirigida à própria caixa que representa o objeto (em vez de ser dirigida à linha de vida)

Pode ter estereótipo **«create»**

Destruição de objeto é representada por um X no fim da linha de vida do objeto.



Pode ocorrer na receção de mensagem ou no retorno de chamada, o objeto pode auto destruir-se

Pode ter estereótipo **«destroy»**

Barra de ativação

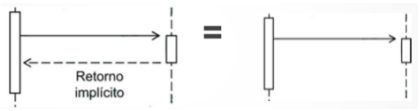
Uma barra de ativação mostra o período de tempo durante o qual um objeto está a executar uma ação/função.

> Inclui situação em que está á espera de retorno de uma chamada síncrona.
>
> Não inclui situação em que um processo está adormecido à espera de receber uma mensagem assíncrona que o "acorde".

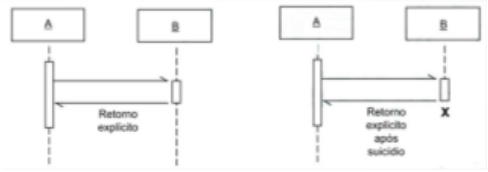
Em termos de processos, significa que o objeto tem um processo ou thread ativo associado.

A sua indicação é **opcional**.

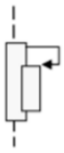
Retorno de **chamada síncrona** é implícito no fim da barra de ativação, não é necessário representá-lo.



No caso dos **envios assíncronos**, o retorno deve ser representado, quando existe.



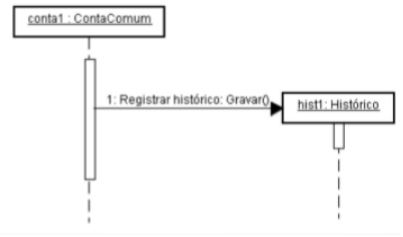
Chamadas recursivas provocam barras empilhadas



Instanciação de um objeto

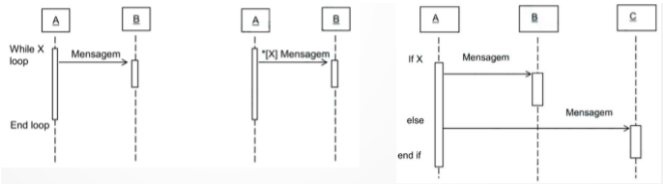
> A seta atinge o retângulo que representa o objeto
>
> O objeto passa a existir a partir daquele momento

A mensagem representa a chamada do método construtor



Outros elementos

Pseudo-códigos



A inclusão de pseudo-código na parte esquerda do diagrama permite representar ciclos e saltos, pelo que se pode recorrer aos diagramas de interação para representar a forma geral de uma interação, para além da simples descrição de um cenário particular

Repetições

O diagrama de sequência permite que repetições sejam realizadas durante o fluxo

Para isso são utilizados quadros (frames) do tipo “loop”

Decisões

O diagrama de sequência permite que decisões sejam tomadas durante o fluxo

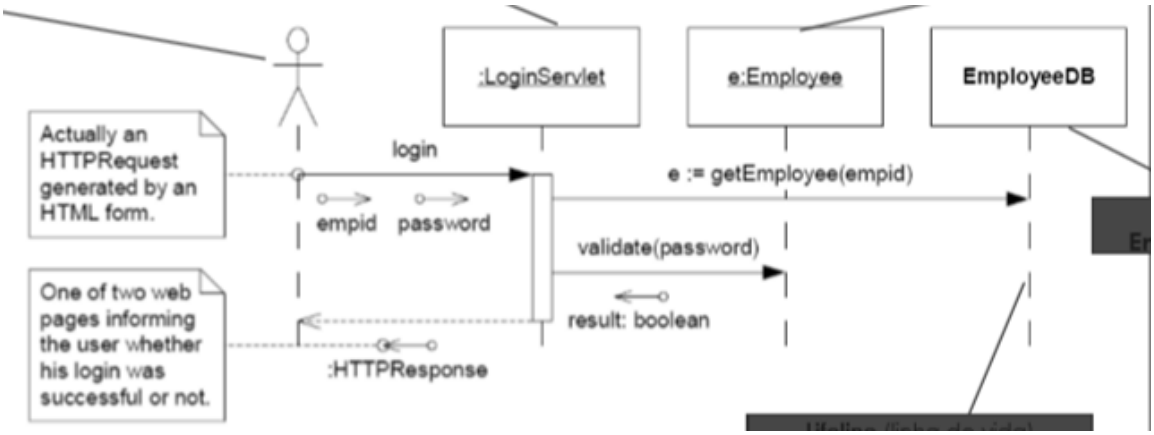
Para isso são utilizados quadros (frames) do tipo “alt” ou “opt” com condições de guarda

Além dos quadros do tipo “loop”, ”opt” e “alt”, existem outros tipos, entre eles:

“par”: Contém vários seguimentos e todos são executados em paralelo

“region”: Determina uma região crítica, que deve ter somente uma thread em execução em um dado momento

10-UML EM JAVA: <u>Diagrama de Sequência</u>



Objeto anonimo- representa a origem e o fim da sequencia de mensagens representada. É opcional

:LoginServlet- Objeto anónimo da classe LoginServlet

e: Employee- Objeto **e** da classe Employee

EmployeeDB: classe EmployeeDB

Linha de vida, o tempo corre de cima para baixo.

Mensagens:

Os argumentos podem aparecer entre parentesis

Mensagem entre linhas de vida; Cada mensagem tem uma etiqueta (nome)

Como EmployeeDB é uma classe, getEmployee() pertence à classe e não a uma instância, logo… static

(public class EmployeeDB{

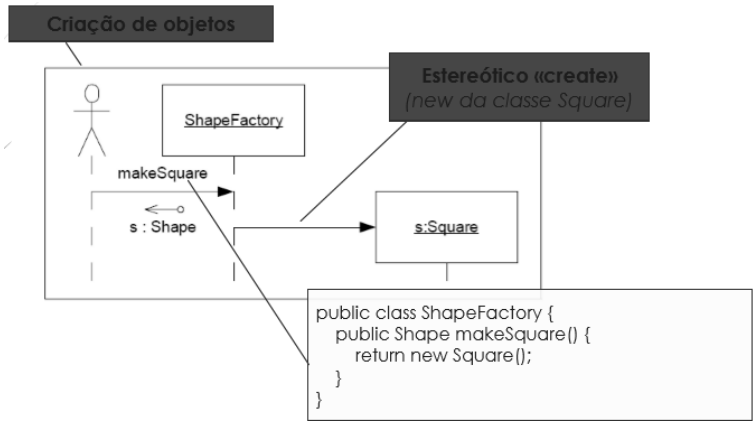
public static Employee getEmployee(String empid){

...

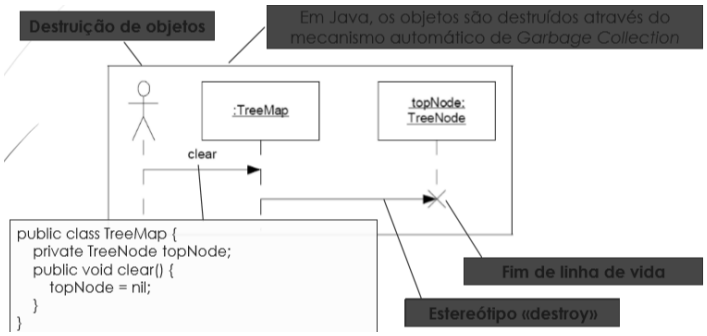
}

...)

Criação Objeto:



Destruição Objeto:



04-Método 4SRS – Four Step Rule Set

Transformar Requisitos em Arquiteturas Lógicas

A atividade mais complexa durante o desenvolvimento de sistemas de software, processo menos formalizado

As decisões devem ser tomadas com muito cuidado, pois geralmente causam um grande impacto na qualidade do sistema resultante

1ºPasso: Definir o **modelo funcional que captura as funcionalidades do siste**ma oferecidas aos seus utilizadores

Os casos de uso são uma das técnicas mais adequadas para esse fim, uma vez que são simples e fáceis de ler

O baixo número de conceitos(casos de uso, atores e relações) é uma característica fundamental para envolver, dentro do processo de captura de requisitos, clientes e utilizadores não técnicos

2ºPasso: **descrever** seu comportamento dos **casos de uso**

Neste caso de estudo os casos de uso serão descritos através de **texto informal**, mas podem ser descritos de outras maneiras: etapas numeradas com pré e pós-condições e diagramas de atividades)

Método 4SRS

Método composto por quatro passos, que converte requisitos funcionais representados através de diagramas de Casos de Uso em **diagramas de Objetos ou Componentes** que representam a arquitetura lógica do sistema .

Step 1: creation

Transformar cada caso de uso em 3 elementos arquiteturais básicos (objeto ou componente)

elementos representativos de uma **interface**, de uma unidade **de armazenamento de dados** e uma unidade de **controlo**

Os elementos arquiteturais básicos podem ser do tipo "objeto"(UML1.x) ou do tipo "componente" (UML2.x)

Passo totalmente "automático", uma vez que não há necessidade de qualquer tipo de decisão ou justificação para o contexto específico de cada caso de uso.

A partir deste passo, existem apenas elementos básicos como entidades de design.

A criação dos elementos arquitetónicos é executada sem considerar o contexto do sistema.

Step 2: elimination

Com base na descrição de cada caso de uso, deve-se decidir qual dos 3 elementos arquitetónicos devemos manter para representar totalmente, em termos computacionais, o caso de uso.

Este passo suporta a eliminação de redundância no levantamento de requisitos do utilizador, bem como a descoberta de requisitos ausentes

É o passo mais importante do método 4SRS, pois as entidades definitivas ao nível do sistema são decididas aqui

Micro-step 2i: use case classification

O engenheiro de software vai classificar cada caso de uso

Quantas combinações de elementos arquiteturais de um determinado caso de uso existem?

> O 4SRS assume que cada caso de uso gera um máximo de três elementos arquitetónicos: interface (i), controlo (c) e dados (d)
>
> Assim, podemos obter 8 combinações ou padrões diferentes (Ø, i, c, d, ic, di, cd, icd), se ignorarmos as ligações entre os elementos arquitetónicos

Ajudar na transformação de cada caso de uso em elementos arquitetónicos, vai fornecer dicas sobre quais as categorias de elementos arquitetónicos a usar e como associar esses elementos

Micro-step 2ii: local elimination

Responder, para cada elemento arquitetural criado no passo 1, se faz sentido a sua existência no domínio do problema. (T-eliminar; F-manter)

Micro-step 2iii: object naming

Os elementos arquitetónicos que não foram eliminados devem receber um nome adequado que reflita o caso de uso do qual se originou e a função específica do elemento arquitetónico, levando em consideração seu componente principal

Micro-step 2iv: object description

Descrever cada elemento arquitetural, que recebeu um nome, para que os requisitos de sistema que eles representam sejam incluídos no modelo arquitetural

As descrições devem ser baseadas nas descrições originais dos casos de uso

Não é recomendável introduzir requisitos, que não sejam baseados nos requisitos do utilizador capturados por meio de descrições de casos de uso

Exceções a esta recomendação são permitidas se forem classificadas como requisitos não funcionais (NFR) ou como decisões de conceção (DD)

Micro-step 2v: object representation

É o micro-passo mais crítico, uma vez que suporta a eliminação de redundância no levantamento de requisitos do utilizador, bem como a descoberta de requisitos ausentes

Constitui uma etapa de validação interna garantindo a coerência semântica do modelo da arquitetura e descobrindo anomalias no modelo de casos de uso

Micro-step 2vi: global elimination

É "automático", uma vez que é completamente baseado nos resultados do micro-passo anterior

Os elementos arquitetónicos representados por outros devem ser eliminados

É chamado de "eliminação global"- gera um modelo arquitetónico coerente e canônico

Kill or Alive

Micro-step 2vii: object renaming

Renomear os elementos arquitetónicos que não foram eliminados no micro-passo anterior

Os novos nomes devem refletir a plenitude dos requisitos do sistema representados

Step 3: packaging and aggregation

Os elementos arquitetónicos que sobreviveram ao 2º passo devem ser agregados sempre que houver acordo mútuo para uma representação unificada desses elementos arquitetónicos

Empacotamento- técnica imatura, introduz uma coesão semântica muito leve entre os elementos arquitetónicos. O empacotamento pode ser usado com flexibilidade para permitir a obtenção temporária de modelos arquitetónicos mais abrangentes e compreensíveis

Agregação- forte coesão semântica entre os elementos arquitetónicos. O nível de coesão mais difícil de reverter nas próximas fases do projeto. A agregação deve ser usada apenas quando for assumido explicitamente que o conjunto de elementos arquitetónicos considerados é afetado por uma decisão consciente do projeto.

Step 4: associativo

Os elementos arquitetónicos agregados obtidos devem ser "ligados" para especificar as associações entre os elementos arquitetónicos existentes

As inter-relações entre os casos de uso "arquitetónicos" podem ser simplificadas com êxito, a fim de obter um número reduzido de requisitos relevantes e pertinentes no nível do sistema com semântica arquitetónica

As descrições dos casos de uso fornecem indicações sobre as associações a serem introduzidos no modelo arquitetural.

Este passo final suporta a introdução de associações no modelo de arquitetura, baseadas nas informações existentes no modelo de caso de uso e geradas no micro-passo 2i

Arquitetura Lógica e o 4SRS

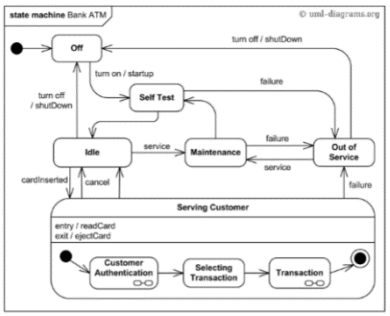
O principal objetivo de uma arquitetura lógica é servir de base para a conceção de um sistema, abrange a descrição dos componentes lógicos do sistema e também as interações entre eles

O modelo arquitetural resultante do 4SRS contém os elementos arquitetónicos e as interações entre eles (associações)

O modelo resultante do método 4SRS pode ser de grande valor para um arquiteto de sistema, porque fornece claramente "sugestões" para os componentes lógicos de um sistema e as interações entre eles

O modelo de arquitetura gerado pelo 4SRS mostra como propriedades significativas de um sistema são distribuídas pelas partes que o constituem

05- Diagramas de Máquinas de Estado



Máquina de Estados define um conjunto de conceitos que podem ser usados para modelar comportamento dinâmico (behavior state machine) asseados em eventos discretos

Expressam as sequências autorizadas de ocorrências de eventos.

Objetivo: especificar as ações/transições (instantâneas) e atividades/estados (duradouras) realizadas em resposta a eventos ou durante a permanência em estados.

Podem ser usadas para especificar

- O comportamento de uma classe
- Um comportamento isolado
- Uma característica comportamental de um método

Notação

Estado inicial

- Descreve o estado inicial do maquina de estados

Estado Final

Podem existir vários estados finais, correspondendo a diferentes condições de finalização

Pode não existir nenhum, no caso do sistema nunca parar

Notação- Estado

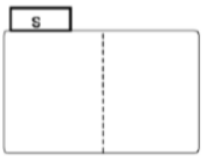
Condição ou situação na vida de um objeto (ou sistema) que está a ser modelado durante a qual:

o objeto satisfaz alguma condição

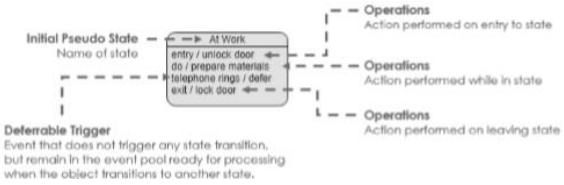
realiza alguma atividade

espera por algum evento

Estado composto por regiões



Estado com compartimentos



Notação- Eventos

É uma ocorrência significativa que tem uma localização no tempo (do evento) e no espaço

Um evento pode ter como resposta uma transição (mudança de estado) e/ou uma ação

Os eventos podem ser de vários tipos:

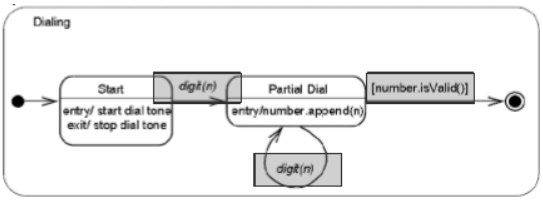
Sinais - eventos simbólicos sinalizados explicitamente, exemplo: kill()

Chamadas - invocação de operações, exemplo: insert(record)

Eventos temporais - passagem de tempo ou ocorrência de um data/hora, exemplo: after(15 seconds), when(12:00), timeout()

Eventos de mudança - uma condição tornar-se verdadeira, exemplo: number.isValid()

Os eventos podem ter **parâmetros**, exemplo: digit(n)

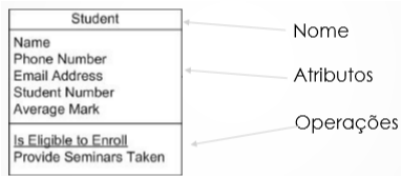


Notação- Transições

Relação entre dois estados: um objeto no 1º estado realizará uma certa ação (opcional) e passará ao 2º estado quando um evento especificado ocorrer se uma condição especificada (opcional) for satisfeita

Duas transições podem sair do mesmo estado e estas devem ter eventos diferentes ou condições mutuamente exclusivas

Podem existir transições automáticas que são transições sem eventos, em que se assume de forma implícita que o evento é o fim da atividade associada ao estado de origem. Caso tenha uma condição, fica à espera que a condição seja satisfeita.



06-Driagrama de Classes

O diagrama de classes descreve a estrutura estática do sistema.

A estrutura do sistema envolve: As classes; As associações entre classes; As dependências entre classes; Interfaces do sistema

Uma classe descreve um conjunto de objetos que partilham os mesmos atributos, operações, associações e semântica.

Um objeto de uma classe é uma instância da classe

A extensão de uma classe é o conjunto de instâncias da classe

Podem representar:

Coisas concretas: Pessoa, Turma, Carro, Imóvel, Fatura, Livro

Papéis que coisas concretas assumem: Aluno, Professor, Piloto

Eventos: Curso, Aula, Acidente

Começa por identificar os tipos de objetos (classes) presentes num sistema

Tipos de objetos são mais estáveis do que as funções, logo a decomposição orientada por objetos leva a arquiteturas mais estáveis

Em UML, uma classe é representada por um retângulo com o nome da classe

Habitualmente escreve-se o nome da classe no singular (nome de uma instância), com a 1ª letra em maiúscula

Atributos

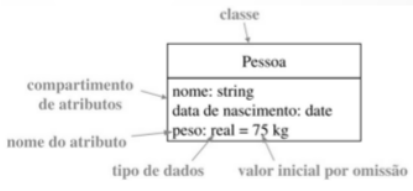
Estado de um objeto é dado por valores de atributos (e por ligações que tem com outros objetos)

Todos os objetos de uma classe são caracterizados pelos mesmos atributos (ou variáveis de instância)

O mesmo atributo pode ter valores diferentes de objeto para objeto

Uma classe não deve ter dois atributos com o mesmo nome

Atributos são definidos ao nível da classe, enquanto que os valores dos atributos são definidos ao nível do objeto



Exemplos: Uma pessoa (classe) tem os atributos nome, data de nascimento e peso

Ze Carlos (objeto) é uma pessoa com nome “José Carlos”, data de nascimento “29/01/1965” e peso “80kg”

Operações

Uma operação é algo que se pode pedir para fazer a um objeto de uma classe

Objetos da mesma classe têm as mesmas operações

Operações são definidos ao nível da classe, enquanto que a invocação de uma operação é definida ao nível do objeto

Princípio do encapsulamento: acesso e alteração do estado interno do objeto (valores de atributos e ligações) controlado por operações

Nas classes que representam objetos do mundo real é mais comum definir responsabilidades em vez de operações

Atributos e Operações Estáticas

Atributo estático(static): tem um único valor para todas as instâncias (objetos) da classe, valor está definido ao nível da classe e não ao nível das instâncias, **nome sublinhado**

Operação estática(static): não é invocada para um objeto específico da classe, **nome sublinhado**

Visibilidade de atributos e operações

Visibilidade:

+ (public): visível dentro da própria classe e para qualquer outra classe

- (private): visível só por operações da própria classe

(protected): visível por operações da própria classe e descendentes (subclasses)

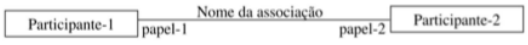
Princípio do encapsulamento

Esconder todos os detalhes de implementação que não interessam aos utilizadores da classe

permite alterar representação do estado sem afetar clientes

permite validar alterações de estado

Notação para a multiplicidade



Associações binárias

Associações representam o **relacionamento estático entre duas classes**, juntamente com a sua **multiplicidade**

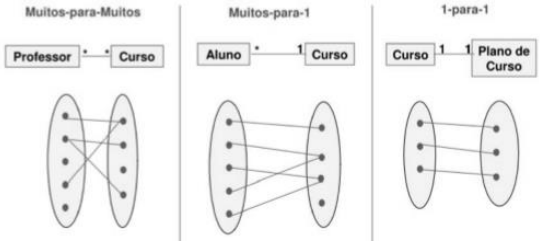
Uma associação é uma relação entre objetos das classes participantes

Uma ligação é uma instância duma associação

Pode haver mais do que uma associação (com nomes diferentes) entre o mesmo par de classes

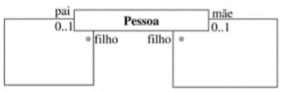
Papéis nos extremos da associação podem ter indicação de visibilidade (pública, privada…)

Relação entre classes



1	- exactamente um
0..1	- zero ou um (zero a 1)
*	- zero ou mais
0..*	- zero ou mais
1..*	- um ou mais
1, 3..5	- um ou três a 5

<u>Associação reflexiva:</u> Pode-se associar uma classe com ela própria (em papéis diferentes)



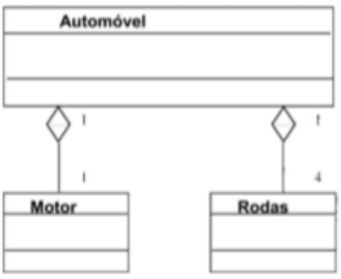
<u>Associação Bidirecionais (normais)</u>

cada uma das classes do relacionamento refere-se uma à outra chamando o método da outra.

<u>Associação Unidirecionais</u>

Mostra que o objeto de origem pode chamar métodos da classe de destino.

<u>Agregação</u>



Não está pintado é um **relacionamento do tipo "tem um"**

Um todo é constituído pelas suas partes

Ex: Se motor é destruído o automóvel permanece permanecem

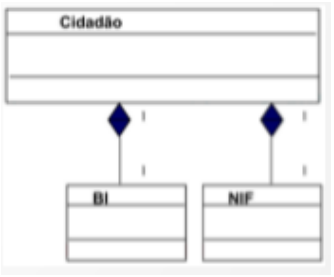
Hierarquias de objeto, relacionamente de associação

Os valores dos atributos de uma classe propagam-se para os valores dos atributos de outra classe

Uma ação sobre uma classe implica uma ação sobre outra classe

Os objetos de uma classe estão subordinados aos objetos de outra classe

É representado através de um diamante com com o símbolo do diamante apontando para o todo.



Composição

Pintado é um **relacionamento do tipo "pertence a"**

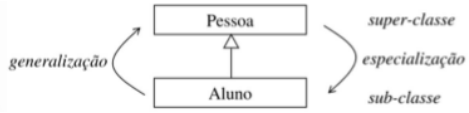
Forma mais forte de agregação aplicável quando:

Existe um forte grau de pertença das partes ao todo

Espera-se que as partes só vivam enquanto viver o todo: qualquer apagamento ou destruição do todo leva ao apagamento ou destruição das partes

O objeto parte **só** pode pertencer a um (único) todo (a multiplicidade do lado do todo não excede 1)

Generalização



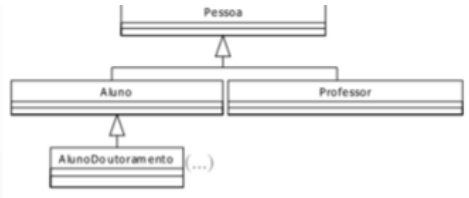
Relação semântica "is a" ("é um" / "é uma"): um aluno é uma pessoa

Relação de inclusão nas extensões das classes:

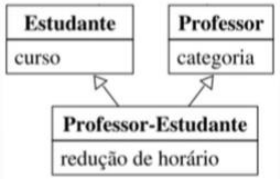
A **subclasse** herda as propriedades (atributos, operações e relações) da **superclasse**, podendo acrescentar outras

Essas classes mais específicas (subclasses) especializam certos aspetos da classe mais geral (classe base ou superclasse).

Em cada classe da hierarquia colocam-se as propriedades que são comuns a todas as suas subclasses; evita-se redundância, promove-se reutilização, herança!



Herança Múltipla: ocorre numa subclasse com múltiplas super-classes, suportada por algumas linguagens de programação OO



Classes e operações abstratas

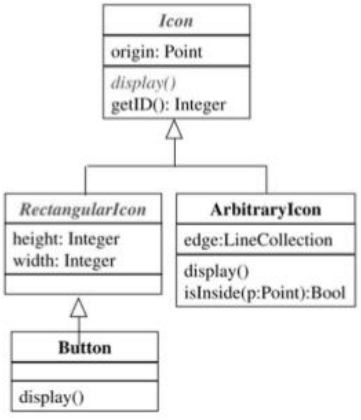
Classe abstrata: classe que não pode ter instâncias diretas

pode ter instâncias indiretas pelas subclasses concretas

Operação abstrata: operação com implementação a definir nas subclasses

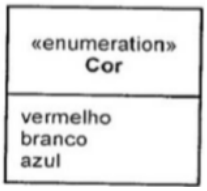
uma classe com operações abstratas tem de ser abstrata

Notação: nome *em **itálico*** ou propriedade *{**abstract**}*



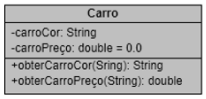
Enumerações

São utilizadas para mostrar um conjunto fixo de valores que não possuem quaisquer propriedades além do seu valor simbólico



09-UML EM JAVA: Diagrama de Classes

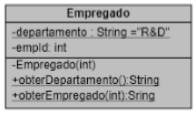
Ex. A classe Carro possui várias variáveis privadas “carroCor” e “carroPreço”




```
public class Carro
{
    private String carroCor;
    private double carroPreço = 0.0;

    public String obterCarroCor(String modelo)
    {
        return carroCor;
    }
    public double obterCarroPreço(String modelo)
    {
        return carroPreço;
    }
}
```

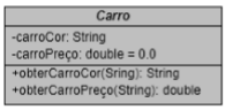
Visibilidade: **~** package- Visível só dentro da própria classe e das classes dentro do mesmo pacote

Ex. Static




```
public class Empregado {
    private static String departamento = "R&D";
    private int empId;
    private Empregado(int empregadoId) {
        this.empId = empregadoId;
    }
    public static String obterEmpregado(int emplId) {
        if (emplId == 1) {
            return "Ana Paula";
        } else {
            return "Empregado não encontrado";
        }
    }
    public static String obterDepartamento() {
        return departamento;
    }
}
```

Ex. *Abstract*




```
public abstract class Carro
{
    private String carroCor;
    private double carroPreço = 0.0;

    public  String obterCarroCor(String modelo)
    {
        return carroCor;
    }
    public double obterCarroPreço(String modelo)
    {
        return carroPreço;
    }
}
```

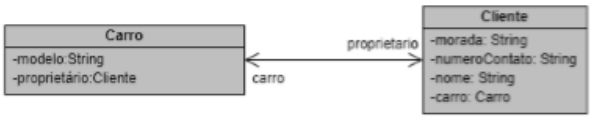
Ex. Associação

Unidirecional: um exemplo possível pode ser a variável da instância da classe de origem que faz referência à classe de destino


```java
public class Cliente {
private String nome;
private String morada;
private String numeroContato;
}
```

```java
public class Carro {
    private String modelo;
    private Cliente proprietario;
}
```

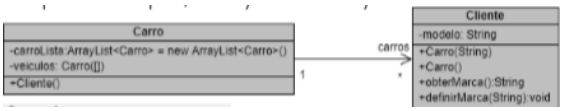
Bidirecional: a associação é descrita como instância da variável da classe Carro chamada dentro da classe Cliente e vice-versa; o carro e o proprietário referem-se ao role e são representados pelo nome da instância da variável no código.




```java
public class Carro {
    private String modelo;
    private Cliente proprietario;
}
```

```java
public class Cliente {
private String nome;
private String morada;
private String numeroContato;
private Carro carro;
}
```

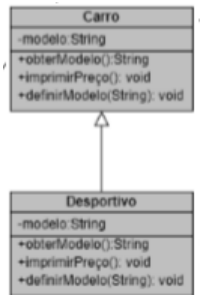
Ex. Multiplicidade- podemos usar por exemplo, ArrayList e Array




```java
public class Carro {
    private String marca;
    public Carro(String marcas){
        this.marca = marcas;
    }
    public Carro() {
    }
    public String obterMarca() {
        return marca;
    }
    public void defineMarca(String marca) {
        this.marca = marca;
    }
}
```

```java
import java.util.ArrayList;

public class Cliente {
    private Carro[] veiculos;
    ArrayList<Carro> carroLista = new ArrayList<Carro>();
    public Cliente(){
        veiculos = new Carro[2];
        veiculos[0] = new Carro("Audi");
        veiculos[1] = new Carro("Mercedes");
        carroLista.add(new Carro("BMW"));
        carroLista.add(new Carro("Chevy"));
    }
```

Ex. Generalização- está relacionado com a palavra-chave "extends".




```java
public class Carro {
    private String modelo;
    public void imprimirPreco() {
    }
    public String obterModelo() {
        return modelo;
    }
    public void definirModelo(String modelo) {
        this.modelo = modelo;
    }
}
```

```java
public class Desportivo extends Carro {
    private String modelo;
    public void imprimirPreco() {
        System.out.println("Preço Carro Desportivo");
    }
    public String obterModelo() {
        return modelo;
    }
    public void definirModelo(String modelo) {
        this.modelo = modelo;
    }
}
```

Ex. Agregação e Composição

Agregação-


```
public class Estudante(
}
```

```
public class Escola {
private Estudante estudante;
}
```

Composição-


```
public class Funcionario
{
}
```

```
public class Empresa {
    private Funcionario[] funcionario;
}
```

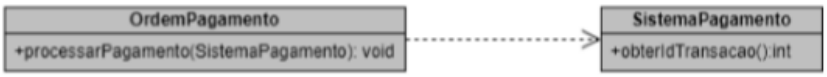
Ex. Dependências

Um relacionamento que mostra que uma classe depende de outra classe para a sua existência ou implementação.

O relacionamento de dependência é mostrado como uma **linha pontilhada com uma seta da classe de origem para a classe dependente**.

Em Java, podemos considerar o relacionamento de dependência se:

- a classe de origem tiver uma referência direta à classe dependente;
- ou se a classe de origem tiver métodos pelos quais os objetos dependentes são passados como parâmetro;
- ou refere-se à operação estática da classe dependente ou a classe de origem possui uma variável local referente à classe dependente, etc.




```
public class SistemaPagamento {
}
```

```
public class OrdemPagamento {
    public void processarPagamento(SistemaPagamento ps){
    }
}
```

07-Diagrama de Objetos

Um diagrama de objetos representa uma instância específica de um diagrama de classes num determinado momento, estrutura do sistema envolve: instâncias de classes, atributos das classes instanciadas e instâncias das relações das classes.

Os diagramas de objetos usam um subconjunto dos elementos de um diagrama de classes para enfatizar o relacionamento entre instâncias de classes em algum momento.

Objetos são as entidades do mundo real cujo comportamento é definido pelas classes.

Não podemos definir um objeto sem sua classe.

A **diferença** entre o diagrama de classes e o de objetos é que:

> O diagrama de classes representa principalmente a visão geral de um sistema que também é chamado de **visão abstrata**.
>
> O diagrama de objetos descreve a instância de uma classe, visualiza a funcionalidade **específica** de um sistema.

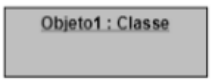
O objetivo é capturar a visão estática de um sistema em um momento específico.

O objetivo do diagrama de objetos pode ser resumido como:

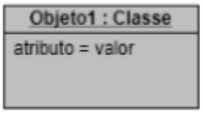
> Engenharia avançada e reversa
>
> Relacionamentos de objetos de um sistema
>
> Visão estática de uma interação
>
> Compreender o comportamento dos objetos e o seu relacionamento a partir de uma perspetiva prática

Durante a fase de análise de um projeto, podemos criar um diagrama de classes para descrever a estrutura de um sistema e, em seguida, criar um conjunto de diagramas de objetos como casos de teste para verificar a precisão e a integridade do diagrama de classes

Notação



Os objetos são representados por um retângulo, que inclui o nome do objeto e a respetiva classe sublinhados, divididos por dois pontos



Semelhante às classes, os atributos dos objetos são colocados num compartimento separado. No entanto, ao contrário das classes, os atributos do objeto devem ter **valores atribuídos**

Os links são instâncias das relações das classes

Os objetos:

- Caracterizado por um conjunto de atributos
- Os atributos são características (propriedades)presentes em todos os objetos de uma mesma classe
- O conjunto dos valores de cada atributo num determinado momento representa o estado de um objeto (não confundir atributos com variáveis locais)
- Os objetos são criados a partir de uma classe
- Cada objeto é uma instância da sua classe
- Criação de um objeto é igual à instanciação de uma classe
- Os objetos podem ser construídos e destruídos

Para capturar um sistema específico, o número de diagramas de classes é limitado. No entanto, se considerarmos os diagramas de objetos, podemos ter um número ilimitado de instâncias, que são únicas por natureza.

Do que foi referido atrás, fica claro que um único diagrama de objetos não pode capturar todas as instâncias necessárias ou, em vez disso, não pode especificar todos os objetos de um sistema.

Portanto, devemos:

1. Analisar o sistema e decidir quais as instâncias que têm dados e associação importantes.

2. Considerar apenas as instâncias, que cobrirão a funcionalidade.

3. Otimizar, pois o número de instâncias é ilimitado.

Antes de iniciar a construção do diagrama de objetos:

O diagrama de objetos deve ter um nome significativo para indicar sua finalidade.

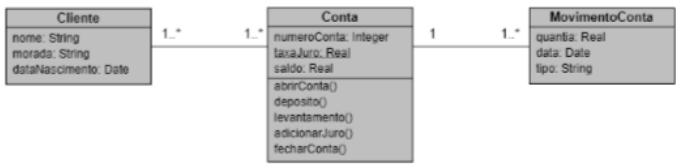
Os elementos mais importantes devem ser identificados.

A associação entre objetos deve ser esclarecida.

Valores de diferentes elementos precisam ser capturados para incluir no diagrama de objetos.

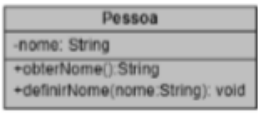
Adicionar notas apropriadas nos pontos em que é necessária mais clareza.

1. Diagrama de Classes



2. Diagrama de Objetos

11- UML em JAVA: Diagramas de Objetos

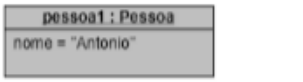



```
public class Pessoa
{
    private String nome;

    public Pessoa(){};

    public Pessoa (String nome){
        this.nome = nome;
    };

    public String obterNome()
    {
        return this.nome;
    }

    public void definirNome(String nome){
        this.nome = nome;
    }
}
```

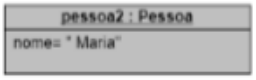
Definimos a classe Pessoa (UML e Java)

Vamos considerar duas pessoas diferentes, Antonio e Maria

corresponde a duas instâncias diferentes



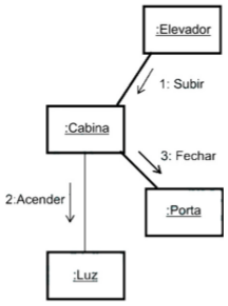

```
Pessoa pessoa1;
pessoa1 = new Pessoa();
pessoa1.definirNome("Antonio");
```




```
Pessoa pessoa2;
pessoa2 = new Pessoa("Maria");
```

08- Diagramas de Comunicação e Diagramas de Deployement

Diagramas de Comunicação



Mostram as interações entre objetos, insistindo sobre a estrutura espacial estática que permite que os objetos de um grupo colaborem entre si

São grafos com objetos (instâncias de classes) e ligações (instâncias de associações) através das quais fluem mensagens numeradas

O tempo não é representado de maneira explícita… Por isso, as sucessivas mensagens têm de ser numeradas para exprimir a ordem de envio

A notação permite fazer figurar atores nos diagramas de comunicação para exprimir o desencadear de interações provocadas por elementos externos ao sistema

Tipos de Ligações

Como todas as mensagens têm de passar por ligações, para além das ligações que representam instâncias de associações, pode ser necessário indicar ligações mais dinâmicas

Assim, em qualquer extremo de uma ligação, pode-se indicar o tipo de ligação através de um estereótipo:

«association» - instância de associação (tipo por omissão)

«parameter» - parâmetro de operação do objeto que faz a chamada «local» - variável local de operação do objeto que faz a chamada

«global» - variável global (usada pelo objeto no outro extremo da ligação)

«self» - auto-ligação (para enviar mensagens para o próprio)

Dinâmica de objetos e ligações

Objectos e ligações podem ser criados e/ou destruídos durante a execução duma interação

Notação: junto de um objeto ou ligação indicar

{new} - criado durante a interação

{destroyed} - destruído durante a interação

{transient} - criado e destruído durante a interação

Um objeto pode mudar de estado (valores de atributos e ligações) durante a execução duma interação

Notação: replicar o objeto, ligando as réplicas com mensagem «become»

Mensagens

Ligações funcionam para as mensagens como canais de comunicação

O fluxo (passagem) de uma mensagem é indicado por uma seta, do emissor para o recetor, acompanhada de uma string com:

Número de sequência da mensagem seguido do separador ":"

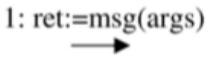
Nome da mensagem e argumentos entre parêntesis

Há 3 tipos de fluxos de controlo, correspondentes a diferentes tipos de mensagens e sistemas de numeração: plano, encaixado e assíncrono

Resultado (retorno): constituído por uma lista de valores devolvidos pela mensagem, esses valores podem ser utilizados como parâmetros das outras mensagens que compõem a interação

Este campo não existe quando não há valores devolvidos; O formato do campo é livre

Nome da mensagem: corresponde frequentemente a uma operação definida na classe do objeto destinatário da mensagem



Argumentos: Trata-se da lista de parâmetros da mensagem, podem conter valores devolvidos por mensagens enviadas anteriormente, assim como expressões de navegação construídas a partir do objeto fonte

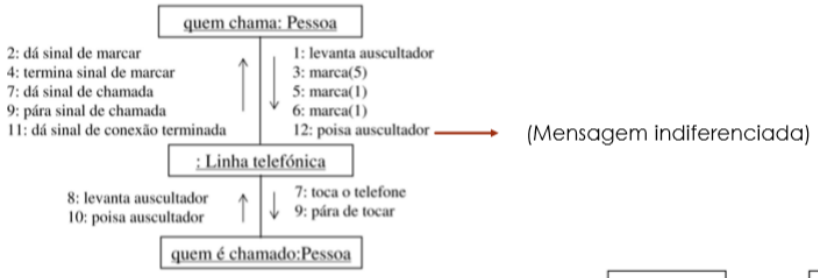
A notação propõe também a representação gráfica dos argumentos sob a forma de setas que originam a partir de pequenos círculos

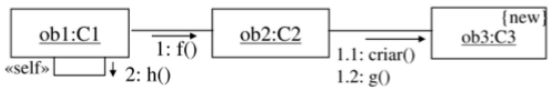
Os argumentos e o nome da mensagem identificam de maneira única a ação que deve ser desencadeada no objeto destinatário

Fluxo de Controlo plano (flat)

Caso em que há uma sequência simples de mensagens indiferenciadas, numeradas 1, 2, etc.

Exemplo: comunicação telefónica





(Mensagem síncrona)

Fluxo de Controlo encaixado (nested)

Caso em que há subsequências de mensagens numeradas de forma hierárquica (com ponto)

Pode iniciar uma subsequência de mensagens

Aplicada normalmente com chamadas de procedimentos ordinárias

Também aplicável entre objetos ativos concorrentes, quando um deles envia um sinal e espera que uma subsequência de comportamento se complete no outro

Diagramas de Deployment ou Instalação

O diagrama de deployment é útil em projetos onde há muita interdependência entre partes de hardware e software

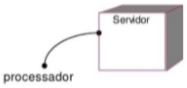
Um diagrama de deployment consiste na organização do conjunto de elementos de um sistema para a sua execução

Descreve os componentes de hardware e software e sua interação com outros elementos de suporte ao processamento

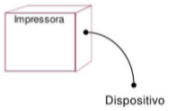
Um diagrama de deployment consiste num conjunto de nós ligados por associações de comunicação

Os nós podem conter instâncias de componentes (de execução), o que significa que um componente é instalado e executado num nó.

Elementos Básicos

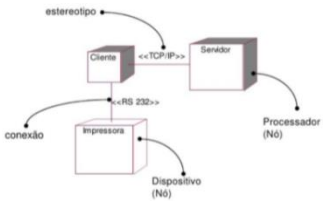


Processador: é qualquer máquina que possua capacidade de processamento. Os servidores, postos de trabalho, etc



Dispositivo (Device): é qualquer máquina com finalidade ou finalidade limitada. Os dispositivos são os itens como impressora, roters, raids, storages, scanners, leitor de código de barras, etc.

Connection (Conexão): A conexão é o vínculo entre processadores e dispositivos. Geralmente representam as conexões de rede físicas (rede local ou distribuída)



Nota: Os processadores e os dispositivos podem ser chamados de nó. Um nó é um elemento físico que existe em tempo de execução e representa um recurso computacional.

- A listagem de artefactos dentro de um nó mostra que ele está instalado nesse nó do sistema que está em execução
- Os nós contêm artefactos, que são manifestações físicas de software, normalmente ficheiros.
- Esses ficheiros podem ser executáveis (como ficheiros.exe, binários, DLLs, ficheiros JAR, aplicações em linguagem assembly ou scripts), ou base de dados, ficheiros de configuração, ficheiros HTML ,etc.