universidade de aveiro
theoria poiesis praxis

Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

Rich Cook

ANÁLISE E MODELAÇÃO DE SISTEMAS

3

Atenção!

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... **Não estudes apenas a partir desta fonte**. Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Análise e Modelação de Sistemas, próxima de como foi lecionada, no ano letivo de 2016/2017, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

Até agora temos vindo a programar e a desenvolver códigos para integrar em projetos, ou até mesmo desenvolver técnicas de otimização de parte das funções que vamos construindo. Referimos projeto quando tentamos arranjar toda uma parte do código que criamos para gerar uma nova ferramenta ou quando falamos de um novo **produto**. Como já devemos ter alguma opinião sobre o assunto, a tarefa de programar até tem o seu quê de divertido, mas desenvolver toda uma **qualidade** neste *software* é uma tarefa bastante mais árdua. Entre novas e brilhantes ideias, as necessidades de uma "visão", e um produto em *software* que possamos designar como **funcional** há muitas ações que se reproduzem muito para além da programação [1].

produto

qualidade

funcional

Nestas fases do projeto, para além da programação, referimo-nos, claramente, a uma **análise** e **modelação** exaustivas, nas quais se define realmente como é que se pretende resolver determinado problema, comunicando uma ideia num modelo que se torna fácil de perceber, rever, implementar e que possa evoluir da forma mais correta.

análise, modelação

1. Introdução a um Sistema de Informação

Entramos assim no domínio do **sistema de informação**. Um sistema de informação é um conjunto complexo de técnicas e produtos de *hardware* computacional, *software* e telecomunicações, sobre o qual se coleciona, processa, preserva e se distribui informação. Tal sistema possui também um quadro de manutenção, onde se gera a discussão de decisões, de controlos, de análises e de planeamentos [2].

sistema de informação

Uma relação entre o cliente e o produto final

Um sistema de informação, se existe, é porque de alguma forma o produto final há de ter um **fim** - uma parte atribuída pelo que vulgarmente denominamos de **cliente** (pode ser tanto um indivíduo, como uma empresa). Entre este fim e o cliente que o situa podemos ter um **processo**. Um processo é assim uma sequência de tarefas que se inicia com uma conceção de uma ideia e termina na finalização, no nosso caso em particular, num *software* desenvolvido.

fim, cliente

processo

Em termos mais detalhados, podemos descrever um processo como tendo quatro fases essenciais: uma fase de conceção, uma fase de elaboração, uma fase de construção e uma fase de transição. Em cada uma destas fases de um processo de produção há que exisir iterações suficientes para que todos os trabalhos possam ser dados como concluídos.

Comecemos assim pela fase de **conceção** de uma ideia. Esta é a fase mais pequena de todas, mas é onde se estabelece a justificação para a criação do projeto que se inicia. Aqui devem ser especificados o **domínio** do problema e as suas condições-limite, para a sua execução. Quando referimos domínio tentamos descrever o contexto de aplicação de um projeto: por exemplo, quando idealizamos um apontador eletrónico (dispositivo apontador que possui um *laser* e uns botões para controlar uma apresentação gráfica em computador) devemos estar a pensar num determinado público-alvo e em determinadas situações de aplicação, como aulas de uma universidade. Ao mesmo tempo, também devemos pensar em situações que consideramos limite, como se houver dois recetores de apontadores, um apontador controlar apenas e somente um computador, e nunca os dois aos quais os recetores se conetam.

conceção

domínio

Nesta primeira fase também se devem estudar **casos de uso** (em inglês geralmente denominam-se de *use-cases*) e **requisitos**, começando também por desenhar uma ou mais possíveis **arquiteturas** para os casos, identificando também alguns dos **riscos** associados. Dadas estas partes concluídas deve-se, logo de seguida, preparar uma agenda para o projeto, ajudando assim a estimar o custo do mesmo. Em suma, há que definir uma primeira visão aproximada do sistema que estamos prestes a criar, onde podemos fazer questões como "É exequível?", "Compra-se um serviço ou cria-se um?", "Qual é a estimativa de custo: entre 10 000€ e 100 000€ ou já estamos a falar de 1 000 000€?" ou "Vale a pena continuar o projeto ou paramos aqui?" [1, cap. 4].

casos de uso

requisitos-chave

arquiteturas, riscos

elaboração

Numa segunda fase, na denominada fase de **elaboração**, o projeto começa a ganhar algum corpo, isto é, as ideias e a conceção que foram delineadas previamente na fase anterior devem começar a ser modeladas face às tecnologias e conhecimentos que se tem sobre o que é possível fazer. Refazendo diagramas de casos de uso e modelos do domínio, há que iniciar uma relação mais prática entre os possíveis atores presentes no sistema, os vários requisitos apontados na conceção e nos contratos entre as várias partes.

Nas últimas duas fases (construção e transição) o projeto terá de ser realmente desenvolvido, entre várias iterações ao longo de um processo que pode ser mais moroso, mas não menos eficaz e eficiente.

Processo de negócio e diagramas de atividade

processo de negócio

No fundo, no entretanto da construção de um projeto, vários processos, inclusive de construção, são tomados. De uma forma mais generalizada, é criado um super-processo ao qual atribuímos a relação de todos os que referimos antes. A este super-processo damos o nome de **processo de negócio**. Em suma, um processo de negócio é um conjunto de atividades e tarefas que produzem um determinado serviço ou produto, por um objetivo particular e para um cliente ou mais.

Descrevendo-se, então, uma coleção de atividades que transformam entradas do nosso sistema em produtos finais diferentes, dependendo das nossas aplicações, os sistemas de informação frequentemente automatizam estes processos.

UML

Para descrever estes processos iremos, ao longo de todo este documento, estudar uma linguagem de descrição para modelação de sistemas denominada de **UML**. A UML (sigla para *Unified Modellling Language*) é uma linguagem própria para modelação que possui uma interface gráfica para a sua interpretação, através de um conjunto muito estrito de figuras representativas de várias atividades, ações, sequências, desenvolvimentos, lançamentos, utilizações, ... entre outros. Na verdade, podemos começar por estudar um primeiro tipo particular de diagrama que nos ajudará a perceber a forma como os processos são desenhados. A estes primeiros diagramas daremos o nome de **diagramas de atividade**. Os diagramas de atividade servem então para documentar a lógica de uma operação ou método simples, componente do fluxo de um processo de negócio. Estes têm uma relação muito próxima com os fluxogramas, como os que definimos em disciplinas como Arquitetura de Computadores I (a2s1), com a pequena diferença que estes regem-se pela sua instituição no paradigma de **orientação a objetos** [3].

diagrama de atividade

diagramas de atividade

orientação a objetos

Num diagrama deste tipo o processo representado inicia pela indicação de um ponto a cheio. Note-se que não tem, necessariamente, que haver apenas um ponto de partida. Um determinado processo poderá ter entre um a mais pontos de partida, desde que este número seja finito, caso contrário não seria um processo, mas antes um intermediário entre dois ou mais processos. Tal como um processo se inicia com um ponto a cheio, este também terá de terminar, mais uma vez, num ou em mais pontos ao longo do mesmo. Para representarmos um possível fim de um processo usamos a representação de um ponto a cheio circunscrito num segundo ponto, a vazio. Estas representações poderão ser analisadas nas figuras seguintes, mais em particular na Figura 1.1.

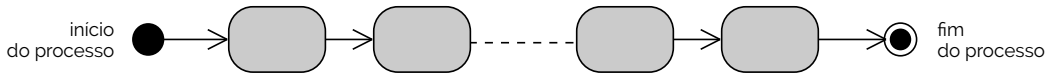


figura 1.1

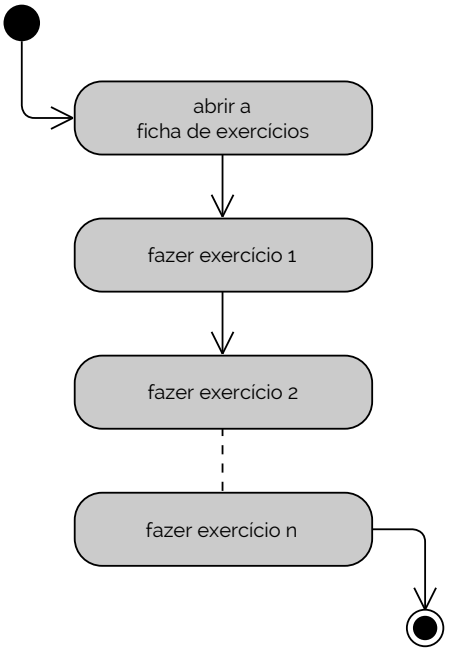
Entre os pontos de início e de fim temos as várias ações do processo, na Figura 1.1 simplificadamente representadas por retângulos arredondados vazios. Os acontecimentos a designar num processo são representados, de facto, por tais figuras, com a exceção de que terão uma designação em si escrita. Mais, tais figuras só deverão aparecer se existir **sincronismo** no processo em causa, isto é, quando o passo seguinte definido neste depende inteiramente da conclusão desta ação. Uma **atividade** é assim uma definição de uma ação a ser executada, como um passo de uma receita de culinária, por exemplo, ou em termos mais próximos com o universo da programação, como uma instrução atómica ou composta

sincronismo

atividade

(desde que elaborada numa abstração) - neste último exemplo, um diagrama de atividades executa um processo de um programa. Vejamos assim, na Figura 1.2, um exemplo de aplicação com algumas atividades para descrever um processo de realizar uma ficha de exercícios (com n exercícios) de uma determinada disciplina.

figura 1.2



A Figura 1.2 mostra um diagrama para a resolução de uma ficha de exercícios de uma forma muito ingénua, isto porque este processo torna-se muito específico de uma (ou um conjunto de) ficha de trabalho, dado que teremos tantas atividades "fazer exercício" quantos exercícios tivermos. Se estivéssemos num contexto de programação, muito provavelmente olharíamos para a Figura 1.2 e rapidamente substituíamos a repetição de "fazer exercício 1", "fazer exercício 2", "fazer exercício 3", ..., "fazer exercício n" por "fazer exercício i", iterando sob um ciclo. Para fazermos um ciclo (como um ciclo *for* das mais convencionais linguagens de programação) existem dois componentes que lhe são essenciais: uma condição de veracidade (lógica) e uma repetição de código.

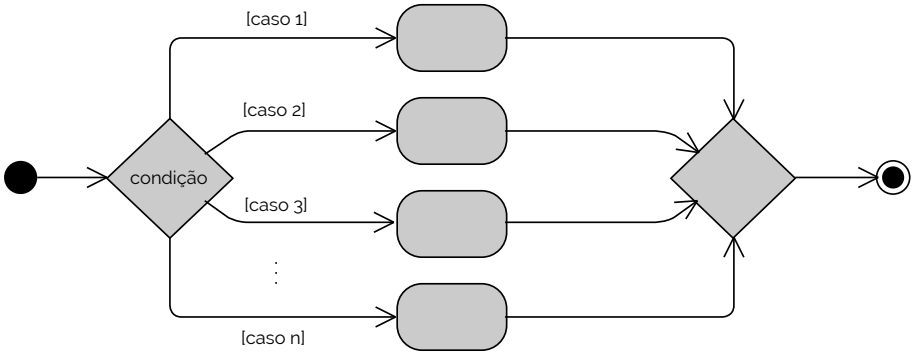
Em termos de linguagem UML é-nos permitido inserir tanto uma condição como uma repetição de atividades. Uma **condição** é apresentada por um nó denominado de **nó de decisão**, isto é, um losango (o qual contêm uma expressão lógica) que dependendo do resultado de veracidade (poderá ser mais do que dois), executa uma atividade diferente. Note-se que o intuito principal deste nó é criar alternativas de execução de vários processos, podendo mais tarde, inclusive, dividir um só progresso de processo em vários, dependendo do número de alternativas fornecidas pelos nós de decisão. Na Figura 1.3 temos uma exemplficação de utilização de um nó de decisão, criando diversas alternativas de execução, todas elas fundindo-se num segundo nó, denominado **nó de fusão**, o qual receberá, de novo, todos os fios de execução do processo, unificando-os.

condição

nó de decisão

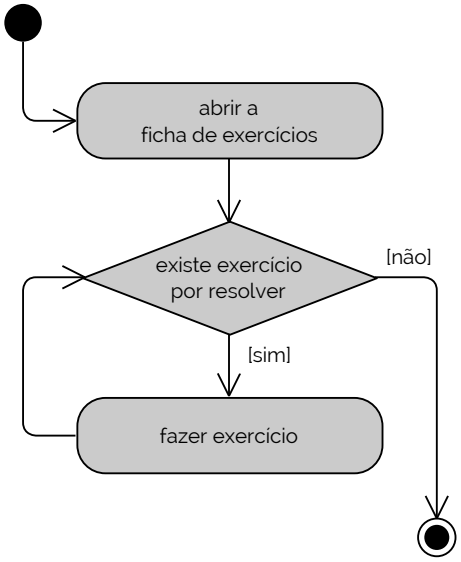
nó de fusão

figura 1.3



Tendo já conhecimento de como funciona o nó de decisão e o nó de fusão podemos tentar melhorar o nosso exemplo da Figura 1.2 para o da Figura 1.4[1].

figura 1.4



Como podemos ver, na Figura 1.4 não precisámos de incluir um nó de fusão, isto porque, no fim de contas, só criámos uma alternativa de execução: ou existe pelo menos um exercício para resolver, ou não existe nenhum e podemos concluir.

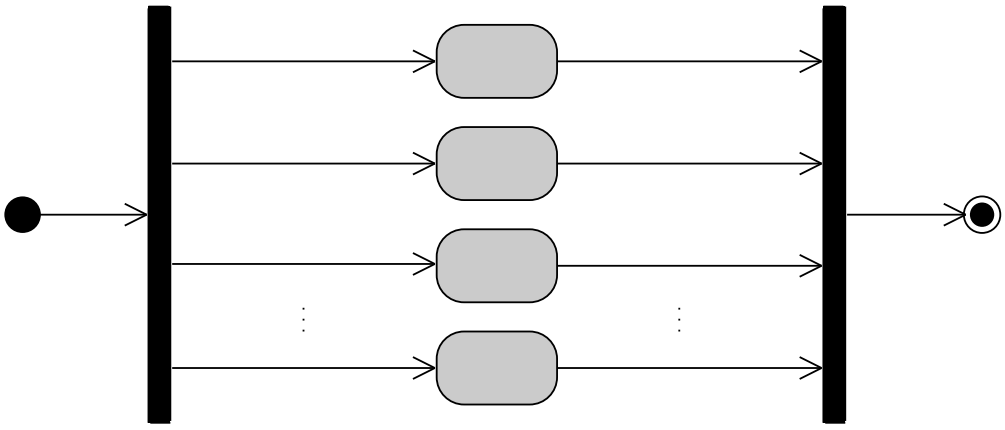
paralelismo

Agora podemos aumentar o nosso exemplo para um exemplo de **paralelismo**: consideremos agora que, enquanto que fazemos a nossa ficha de exercícios estamos também a rever a matéria da mesma disciplina. Como é que podemos transpor essa situação para um diagrama de atividade? Ora, para isso podemos criar uma **nó de bifurcação** sobre o qual executamos tanto o processo de abrir uma ficha de exercícios e resolvê-la, como rever a matéria da disciplina. No final temos apenas de dar por concluída a bifurcação, através de um **nó de junção**. Note-se que ao longo de uma definição de paralelismo não poderão haver inícios ou fins de processos, dado que haverá sempre uma ação que estará a correr em simultâneo, indicando que o processo ainda não terminou (ou que não pode começar, tendo já começado). Na Figura 1.5 podemos ver uma exemplificação de um nó de bifurcação, terminado com um nó de junção.

nó de bifurcação

nó de junção

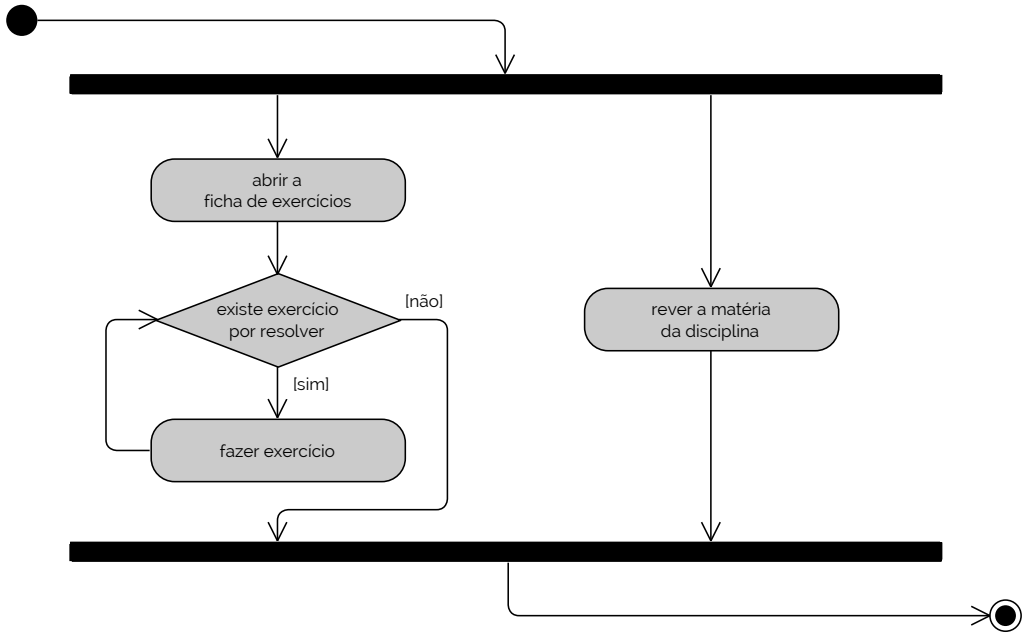
figura 1.5



Aplicando ao nosso caso da resolução de uma ficha de trabalho podemos então representar que, mal se inicie o processo, a pessoa que execute este terá de abrir a ficha e resolver os exercícios um por um e, em simultâneo, rever a matéria da disciplina. Veja-se assim a Figura 1.6.

1 Por uma questão de convenção, neste documento, serão sempre colocados os pontos de arranque no canto superior esquerdo do diagrama (ao nível do arranque) e os pontos de conclusão no canto inferior direito (ao nível da conclusão), analogamente à leitura ou escrita de um texto, que se lê ou escreve da esquerda para a direita, de cima para baixo.

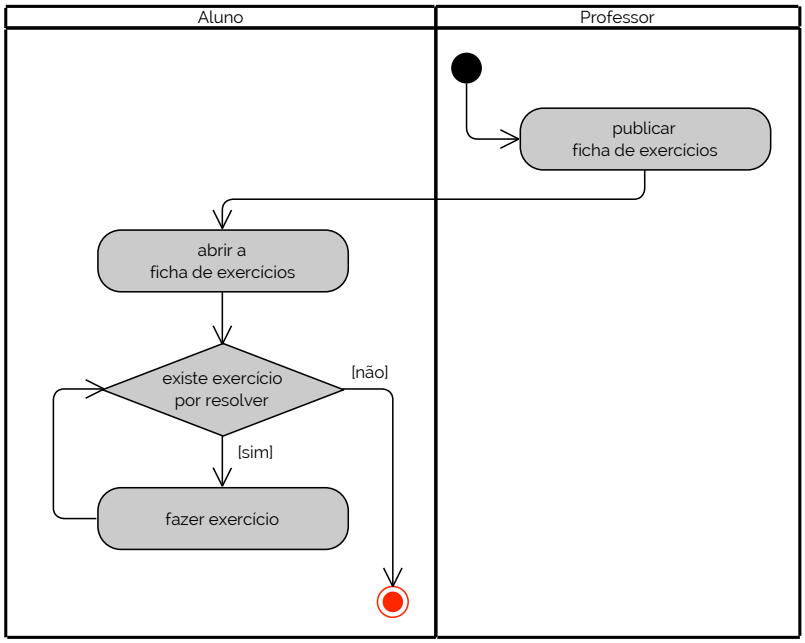
figura 1.6



Voltando ao exemplo mais simples sem o paralelismo, da Figura 1.4, podemos agora pensar onde é que veio a folha de exercícios e querer, assim sendo, incluir o Professor que a cria, no nosso processo. Como é que o podemos fazer? Será que adicionar uma ação chega? Ora, se inseríssemos essa informação como uma ação estaríamos a dizer que o aluno publicaria a ficha de trabalho, o que não é verdade. Isto acontece porque neste momento, embora não o estejamos a explicitar, estamos a referir apenas um ator na nossa atividade - o Aluno. Para inserirmos um novo ator temos de criar uma **partição**. Esta partição é análoga a uma tabela onde cada linha (ou coluna) representa um ator e cada coluna (ou linha) contém as várias ações desse ator. Vejamos assim um exemplo na Figura 1.7.

partição

figura 1.7



Na Figura 1.7 podemos então ver uma partição para um ator denominado de Aluno e uma outra partição denominada de Professor. Representa-se assim que o professor executa a publicação de uma ficha de exercícios, sobre a qual o aluno abre e resolve os exercícios, terminando o processo quando não houver mais exercícios por resolver. No entanto, o processo não está bem projetado, dado que quem começa o mesmo não o termina.

Um processo que começa com um ator A deverá sempre terminar com o mesmo ator A, mesmo que hajam várias alternativas para o seu término. Isto acontece porque o processo ocorre por reponsabilidade de quem o começa, daí haver a necessidade de conduzir o seu término para a mesma pessoa. Na Figura 1.8 podemos ver uma possível solução, onde ainda incluímos o Professor como ator do nosso processo. Se não houver forma possível de fazer com que, neste caso, o Professor não consiga terminar o processo, então isso significa que este ator não pode ser incluído neste.

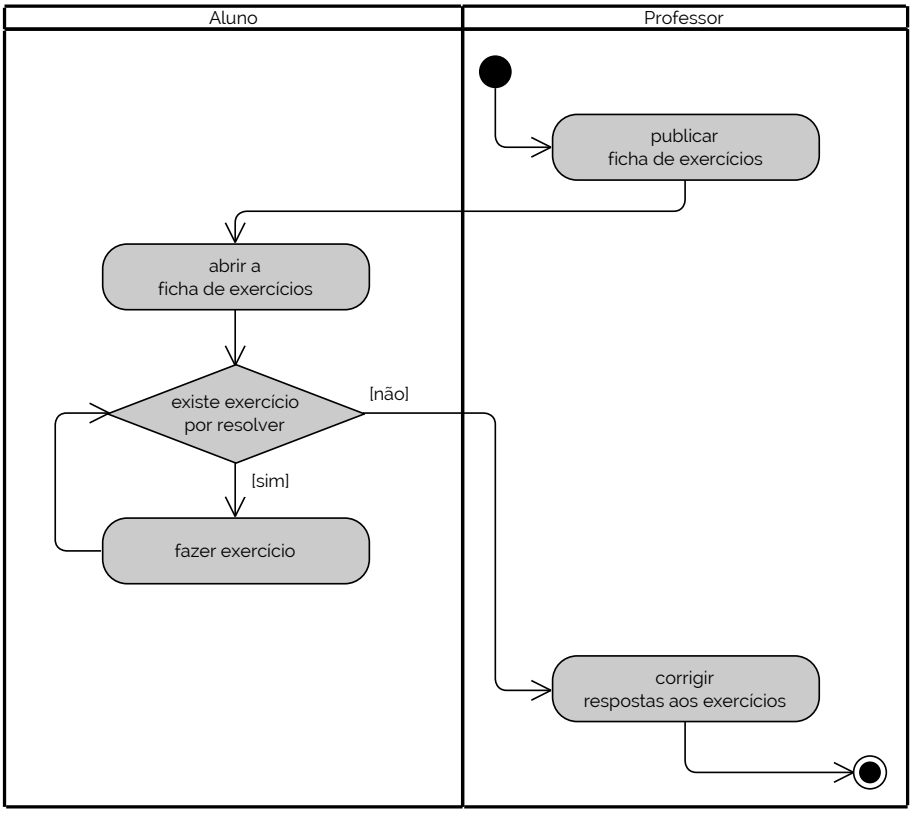


figura 1.8

Atrás vimos que a representação de ações em retângulos arredondados é feita aquando de uma comunicação síncrona, dado que o passo seguinte depende inteiramente da conclusão desta. E no caso de ocorrer assincronia? Por **assincronismo** pretende-se referir que para uma ação ser executada não importa que haja uma resposta ao processo imediatamente a seguir a uma ação anterior e imediatamente antes de uma ação seguinte. Quando isto acontece é importante saber designar dois papeis: quem envia uma mensagem e quem recebe e aceita uma mensagem. Para tal criamos dois blocos - o **bloco de envio** e o **bloco de aceitação** - os quais devem ser usados em situações de comunicações assíncronas. Por exemplo, numa comunicação entre um cliente e um servidor, desde que um não fique à espera do outro, as mensagens são assíncronas, enviando o cliente uma informação para o servidor (envio), este vai trabalhando e só quando puder é que envia uma mensagem de verificação de que o conteúdo chegou (aceitação). Veja-se a Figura 1.9.

assincronismo

bloco de envio

bloco de aceitação

Na Figura 1.9 temos então o envio e resposta, assíncronas, entre um cliente e um servidor. Inicialmente o cliente envia uma mensagem (por exemplo um pacote) e o servidor, só quando receber a mensagem do primeiro, é que avança, por aceitação, para o envio de uma verificação de receção da mensagem inicial. Se esta troca fosse síncrona, então não poderíamos afirmar "só quando receber a mensagem do primeiro", isto é, mal o cliente enviasse a mensagem, quer o servidor tenha recebido ou não esta, enviaria sempre uma verificação (possivelmente verificação de uma mensagem que não exisitiria). Isto é claro no passo seguinte, em que, do lado do cliente, após o envio de uma verficação, está modelado para terminar a ligação, quer o servidor tenha, ou não, recebido uma mensagem e, por conseguinte, enviado uma verficação de chegada.

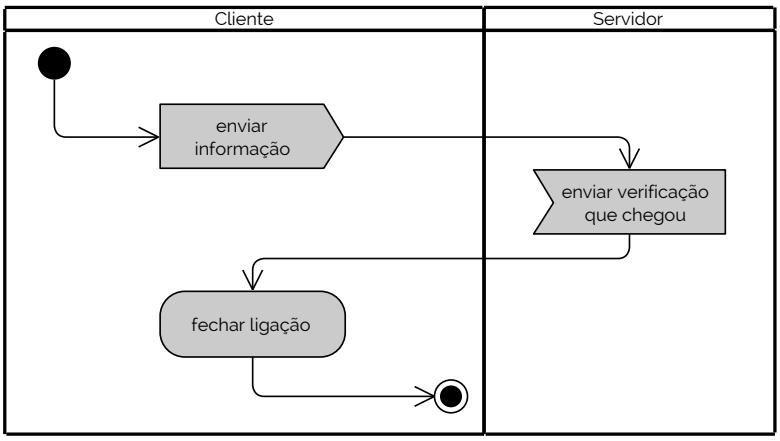


figura 1.9

Finalmente só nos falta referir um pequeno aspeto em termos de diagramas de atividade UML, que é a noção temporal, passível de ser implementada num esquema desta classe através de um **nó ampulheta**, onde o processo é estagnado por uma condição temporal. Na Figura 1.10 temos uma aplicação deste nó.

nó ampulheta

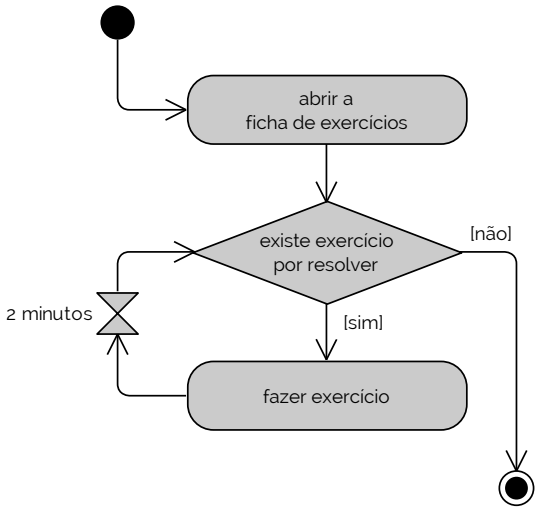


figura 1.10

Não que seja uma medida ótima para trabalho, mas na Figura 1.10 temos então, novamente, o processo descritivo da resolução de uma ficha de trabalho, onde incluímos agora que entre cada exercício, após a sua resolução, há um compasso de espera de 2 minutos.

Requisitos de um sistema/processo

Já definimos um processo, mas para o concretizar é necessário ter um conjunto de aspetos que condicionam o seu funcionamento. Por estes aspetos referimo-nos a objetos ou atores que especificam, mais ao pormenor, o como criar um sistema funcional. Definimos assim **requisitos** como algo necessário para os processos que criamos.

Para iniciar uma listagem de requisitos não podemos simplesmente criar um conjunto de parâmetros essenciais para determinados processos do sistema, mas antes, ao longo de todo o desenvolvimento faseado deste, ir registando novas necessidades ou reutilizando algumas já previamente detalhadas. Isto deverá ser assim, dado que a primeira abordagem que negámos fará com que grande parte dos requisitos saiam do domínio do problema especificado. Na segunda abordagem também não corrigimos essa dispersão a 100%, mas já conseguimos fazer esforços para que isso não aconteça, e por conseguinte nos concentremos mais no problema que foi definido inicialmente. Isto também levanta outras questões - qual é a perda de criar requisitos que não são necessários para o nosso sistema?

Ora, se tal acontecer podemos estar a incorrer numa de duas situações, ambas danosas: a primeira situação será os requisitos serem supérfluos, isto é, muito vulgares, sem necessidade ou sentido, pelo menos específico, o que poderá tornar a arquitetura do sistema muito simples, de facto, mas terá uma curva de aprendizagem muito grande, o que por si levará a um desenvolvimento mais lento do projeto; a segunda situação será a falta de requisitos o que, por outro lado, levará a uma maior dificuldade de integração e desenvolvimento do projeto (e criará uma situação que é difícil de corrigir mais tarde, dado que é nesta fase que estamos a construir algo análogo às fundações de uma casa - precisamos de saber, aproximadamente, o que é que terá de suportar mais tarde) -, o que também levanta o problema da escabilidade, impossibilitando-a.

Os requisitos assim deverão ter uma descrição sucinta onde mostrem como é que deverão ser implementados, como é que o sistema se deverá comportar, quais são os atributos, entre outros, de forma a poderem ser também restrições para o desenvolvimento do sistema. Como **atributos de requisitos** podemos pensar em origens e destinos destes, prioridades, número da iteração, projeto ou versão, entre outros.

atributos de requisitos

Existem também vários **tipos de requisitos**, os quais servem para definir a sua utilidade. E aqui chegamos à definição de um primeiro acrónimo que nos será bastante útil, criado pela Hewlett-Packard em 1989 [4] - o **FURPS+**. Este acrónimo em si significa *Functionality, Usability, Reliability, Performance and Supportability* (em português poder-se-ia traduzir para Funcionalidade, Usabilidade, Fiabilidade, Desempenho e Suporte). Note-se ainda que o acrónimo possui um '+' o que significa que a semântica é passível de ser estendida para outros atributos [5]. Dentro destes vários atributos podemos ligar as suas propriedades para várias partes contribuintes de um sistema, definindo os tipos. Por exemplo, falar de usabilidade não é exclusivo de um utilizador, mas também poderá ser de um sistema, para que haja documentação necessária para que vários componentes possam interagir entre si, por exemplo. Os tipos poderão ser assim um dos vários definidos abaixo, na Figura 1.11 [6].

tipos de requisitos

FURPS+

termo	descrição
requisito de negócio	um objetivo da organização que cria um produto ou de um cliente que o procura
requisito de interface externa	descrição de uma ligação entre um sistema de software e um utilizador, outro sistema ou um dispositivo físico
requisito funcional	descrição de um comportamento do sistema em determinadas condições
requisito não-funcional	descrição de uma propriedade ou caraterística que o sistema deverá exibir ou uma restrição que deverá respeitar
requisito de sistema	requisito de alto-nível para um produto que contenha múltiplos subsistemas, os quais poderão ser todos software, hardware ou outros.
requisito de utilizador	objetivo ou tarefa que uma classe específica de utilizadores deverá ser capaz de desempenhar ou um atributo específico.

figura 1.11

Em termos dos atributos quando falamos em **funcionalidade** devemos estar a querer referir as capacidades lógicas que um sistema tem disponíveis para fornecer a um cliente. Quando pretendemos avaliar não-funcionalidades apontamos, automaticamente, para a parte URPS+ do acrónimo original. Assim sendo, é intuito da **usabilidade** entregar capacidades tanto estéticas como documentais até ao cliente final, tal como consistência nas suas interfaces de comunicação. Por **fiabilidade** pretendem-se também referir caraterísticas como a disponibilidade (o tempo que um dado sistema consegue permanecer ativo), precisão de cálculo e capacidade de recuperação de falhas. A letra 'P' representa o **desempenho**, o qual avalia caraterísticas como o *throughput*, tempo de resposta, tempo de recuperação, tempo de início de ligação e de encerramento de sessão. Finalmente, o **suporte** aponta para as caraterísticas que permitem o teste, a adaptação, a compatibilidade, a manutenção, a configuração, a escalabilidade, ..., entre outros [5].

funcionalidade

usabilidade

fiabilidade

desempenho

suporte

Partindo da elaboração dos atributos FURPS+ podemos agora definir com maior precisão os requisitos, mais uma vez, introduzindo um novo acrónimo, desta vez, **SMART**. SMART significa *Specific, Measurable, Achievable, Relevant, Time-bound* (em português podendo ser traduzido sobre Específico, Mesurável, Realizável, Relevante, Temporalmente limitado - tempo finito de execução). São estes os critérios SMART aos quais os nossos requisitos se devem reger. Para descrever cada requisito de um sistema geralmente cria-se um documento denominado de **SRS** (sigla para *Software Requirements Specification*) o qual especifica os critérios de funcionamento de um sistema de software. Este documento terá de conter um vasto conjunto de casos de utilidade que descreverão um segundo conjunto de interações que os utilizadores terão com o sistema. Mais, também conterão os atributos não-funcionais, os quais irão impor restrições ao desenho e à implementação.

SMART

SRS

Embora já existam várias outras normas para a descrição de um requisito, como a norma da IEEE [7], neste documento iremos usar uma norma mais simples, de forma a que nos possamos preocupar com o detalhe das suas especificações e, ao mesmo tempo, não tenhamos uma lista extensiva de especificidades.

Consideremos assim um exemplo de aplicação, onde queremos, numa plataforma Moodle, que um professor crie um novo trabalho para os alunos entregarem. Antes de conseguirmos explicitar os vários requisitos para que tal ação possa acontecer, há que primeiro conhecer o sistema. A melhor forma (e a mais sumária, também) de o fazer é usando **diagramas de casos de utilização** (também denominados de diagramas de *use-case*). Estes diagramas são mesmo muito simples, exibindo apenas um contexto para o sistema a ser projetado, ou seja, definindo os atores intervenientes num conjunto de ações, que poderão suscitar novas utilizações. Este é mais um dos diagramas da linguagem UML - o mais generalizado que existe.

diagramas de casos de utilização

Os diagramas de casos de utilização, tal como os próprios sistemas que nele são representados, não existem sem **atores**. O conceito de ator é tal que se designa um interveniente nas ações de um produto final. Por exemplo, um ator A pode participar no caso de utilização x onde o ator B também possa estar envolvido no esquema - veja-se a Figura 1.12.

diagrama de use-case

atores

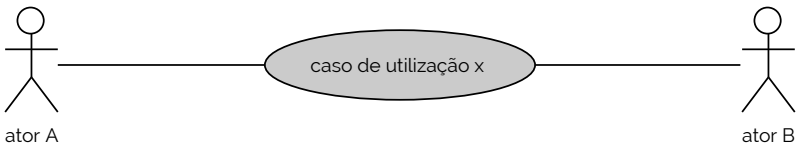


figura 1.12

Mas um ator não tem, necessariamente, que ser interveniente em apenas um dos casos de utilização dispostos no sistema. De facto, se é ator do sistema, terá de ter, pelo menos, um caso a si associado, mas não apenas um, podendo ter múltiplos. Se designarmos mais do que um caso, neste diagrama não podemos referir qualquer efeito de sequência, pelo que representar um caso x primeiro que outro y não significa que x seja feito primeiro que y, ou que y seja feito primeiro que x. Este diagrama, não comportando qualquer informação acerca de sequência, poderá, no entanto, referir casos que provêm da realização de outros, usando a relação **include**. Isto significa que, se o caso x inclui o caso y, então x incorpora y, por outras palavras, dependendo do comportamento de x este delega y. Em termos de representação, como podemos ver na Figura 1.13, usamos uma linha a traço interrompido, com uma seta direcionada para o caso que foi incluído, entre x e y.

include

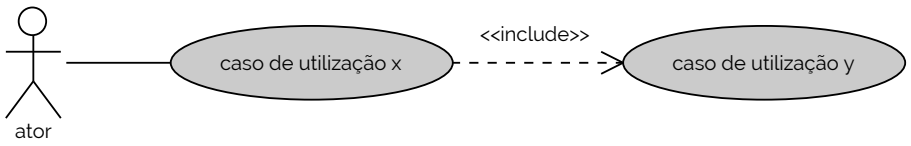


figura 1.13

Tal como podemos incluir novos casos de utilização noutros, também podemos **estender** casos. Por exemplo, se um caso de utilização y puder incorporar um caso x, então podemos afirmar que y pode ser estendido por x, como podemos ver na Figura 1.14. Note-se que, contrariamente ao caso de inclusão, a seta aqui encontra-se no sentido contrário, dado que a extensão provém de y.

estender

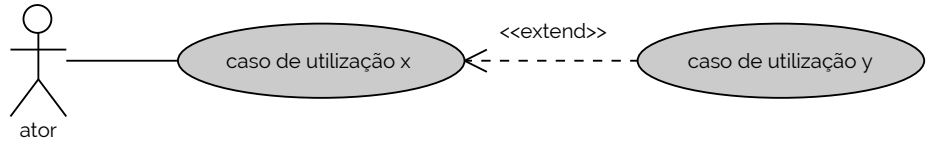


figura 1.14

Como já referimos anteriormente, a linguagem UML tem apoios no paradigma de orientação a objetos. Assim sendo, uma das grandes mais-valias deste paradigma também é válido nos diagramas de casos de utilização - referimo-nos à **generalização**. Se um caso de utilização y partilha semântica e procedimentos com um caso x, então podemos gerar a generalização, representando uma seta a vazio no sentido de y para x, como podemos ver na Figura 1.15.

generalização

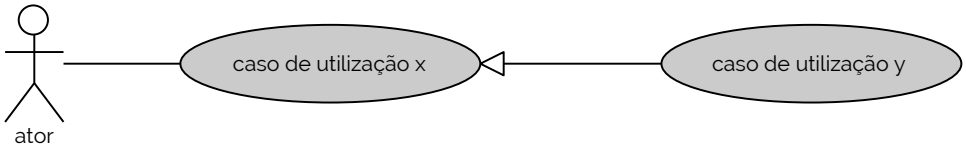


figura 1.15

Vejamos agora, na Figura 1.16, tendo já conhecimento do funcionamento dos diagramas de caso de utilização, o diagrama para o sistema Moodle (aqui simplificado).

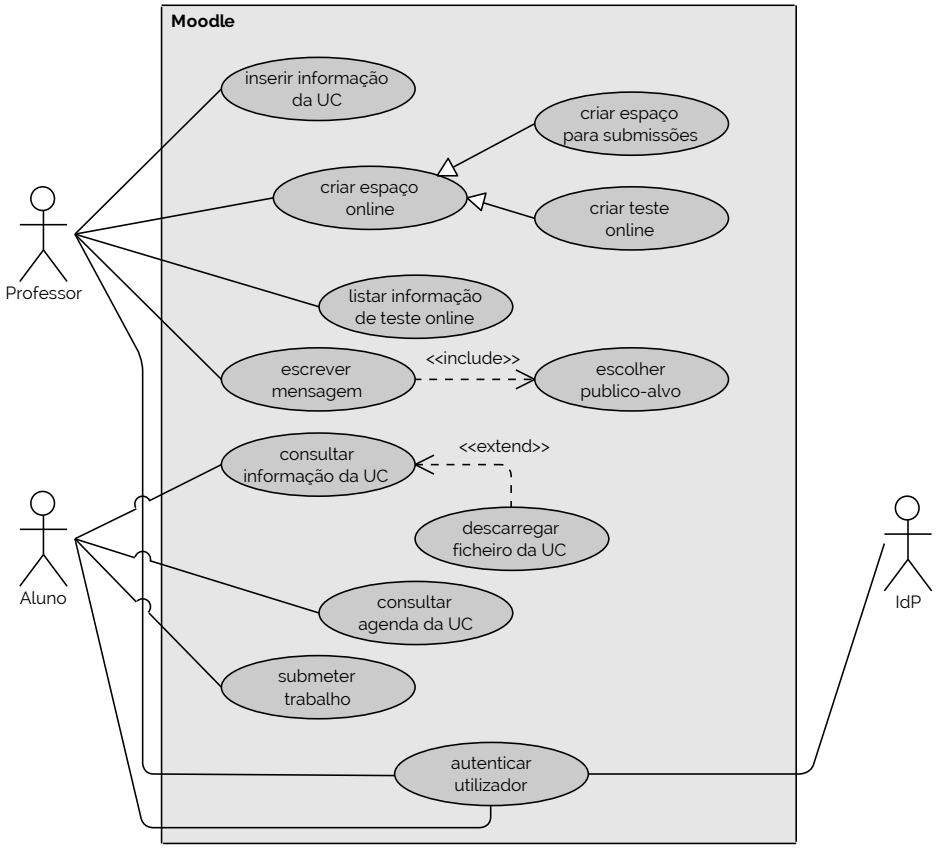


figura 1.16

Na Figura 1.16 podemos então ver um diagrama de casos de utilização para o sistema Moodle que, embora simplificado, possui cada um dos tipos de conexão que referimos anteriormente. O primeiro aspeto que podemos focar é o facto de haver um novo elemento

descrito - todos os casos de utilização, contrariamente aos atores, estão inseridos numa só área, esta a que damos o nome de **sistema**. Se pensarmos bem, os atores não são parte do sistema, mas antes seus intervenientes. Esta é a razão pela qual estes não estão dentro da área do sistema com a nomenclatura "Moodle".

sistema

Começando por descrever o diagrama da Figura 1.16, temos que um aluno poderá consultar a informação de uma Unidade Curricular (UC), consultar a agenda de uma UC, submeter um trabalho ou até mesmo autenticar-se no sistema. Dentro destes vários casos de utilização temos duas situações já previstas: primeiro, o caso "consultar a informação de uma UC" permite também, através de uma relação de extensão, que se descarregue um ficheiro da UC. O aluno não é obrigado a descarregar o ficheiro para o consultar, mas poderá fazê-lo, se assim o pretender. Continuando, por muito que o caso "consultar agenda de uma UC" esteja referido depois de "consultar informação de uma UC" não significa que o aluno tenha que fazer uma depois da outra, por esta ordem, como já tinhamos referido.

Olhando agora em mais pormenor para o caso de utilização "autenticar utilizador", podemos reparar que nele são intervenientes o aluno, o IdP (serviço de autenticação) e o professor. Isto não significa que as três entidades surjam em simultâneo nos processos de tal caso de utilização, mas apenas indica que este caso de utilização, se ocorrer, poderá interagir com um ou mais destes atores.

O ator professor é aquele que, dentro dos presentes, terá mais privilégios de administração. Os seus casos de utilização passam por inserir informação de uma UC, criar um espaço online, listar informações de uma avaliação ou até mesmo por escrever uma mensagem. O caso de utilização "criar espaço online" possui uma relação de generalização com mais dois casos, entre os quais, "criar espaço para submissões" e "criar teste online". Esta generalização é válida porque ambos os casos-filho de "criar espaço online" partilham atributos ou procedimentos com este (o caso-pai). Basicamente, no domínio do Moodle, o professor poderá criar um espaço para um determinado fim, só depois elaborando este, designando, se pretende criar um espaço para um teste ou para uma submissão. Com este ator também temos um segundo caso de utilização que usa uma das conexões que estudámos anteriormente - a inclusão. No caso de utilização "escrever uma mensagem" temos que quando este é realizado um novo caso de utilização deverá ser tido em conta, neste caso, "escolher público-alvo". Note-se que este último caso não pode acontecer sem que o primeiro tenha acontecido (pelo menos a ele não estão associados quaisquer atores).

Se repararmos melhor no diagrama podemos ver que os casos de extensão ou de inclusão não possuem atores a si designados - isto acontece porque estes casos já estão associados a outros casos que ligam atores.

Agora que já designámos o nosso sistema, por muito ingénuo que tenha sido, já podemos espcificar melhor os requisitos para uma determinada tarefa, como o professor criar um espaço para submissões. Numa primeira fase é importante que se consiga cortar o nível mais alto de abstração e conseguir descrever o que significa "criar um espaço para submissões". Para tal criamos uma descrição ao género da realizada na Figura 1.17 [8].

Caso de Utilização	criar um espaço para submissão
descrição	O Professor cria novos trabalhos diretamente na página da disciplina, escolhendo um dos tipos disponível e configurando o período e modo de entrega. O trabalho fica disponível para os alunos e aceitam-se entregas no período indicado.

figura 1.17

Mas o nível de detalhe para a especificação de requisitos da Figura 1.17 não é, de todo o mais adequado. Assim sendo, há que aumentar o nível, passando para uma maior descrição conforme se pode ver na Figura 1.18 [8], onde se apresenta o fluxo da interação entre os atores e o sistema, de uma forma ainda generalizada. Neste passo também podemos apresentar alguns aspetos que ficaram em aberto, isto é, situações às quais ainda não se elaborou qualquer resposta, geralmente apontando consequências que, embora tenham sido identificadas, ainda não foram deliberadas em termos de soluções para as evitar ou para as melhorar.

Caso de Utilização	criar um espaço para submissão
descrição	O Professor cria novos trabalhos diretamente na página da disciplina, escolhendo um dos tipos disponível e configurando o período e modo de entrega. O trabalho fica disponível para os alunos e aceitam-se entregas no período indicado.
fluxo típico	**1.** Autenticar-se no sistema **2.** Selecionar a página da disciplina pretendida **3.** Ativar o modo de edição **4.** Adicionar o Trabalho diretamente na página de entrada **5.** Configurar os parâmetros da entrega **6.** Confirmar as alterações
fluxos alternativos	**FA1:** Sistema central de autenticação indisponível. **FA2:** Em vez de criar um novo trabalho de raiz, o Professor pretende reusar a configuração de um já existente.
aspetos em aberto	- Um curso de semestres passados pode aceitar a criação de novos trabalhos? - Se o browser não aceitar HTML5 qual é a etsratégia alternativa?

figura 1.18

Ainda assim, ao passo do detalhe aqui fixado, não é possível contudo especificar muito acerca dos requisitos. Com a Figura 1.18 podemos verificar o que é que se pretende efetuar, no entanto, em termos de atributos para os requisitos, não conseguimos avaliar, de forma dedutiva ou direta, quais são as necessidades deste sistema. Assim, podemos especificar ainda mais, ao nível máximo de detalhe, os requisitos do sistema, fundindo as técnicas SMART com os atributos FURPS+, como podemos ver na Figura 1.19 [8].

Na Figura 1.19 o detalhe aumentou, em comparação com a Figura 1.18, designando o fluxo típico mais em pormenor, quase ao estilo de um manual de utilização, apontando os fluxos alternativos aos passos de execução concretos, do fluxo típico, e na especificação de pré-condições, para que o caso de utilização possa ocorrer. Mais, adicionou-se, essencialmente, uma nova secção onde se explicitam os requisitos mais especiais, isto é, que se encaixam dentro da analogia FURPS+.

Critérios para o desenho de projetos e diagramas de sequência

Para a criação de um projeto é importante que, no fim, as várias peças desenhadas possam trabalhar todas em conjunto para criar um produto único, coerente e coeso com os princípios usados para a criação do projeto. Mas para que isto possa acontecer é necessário que desde o início bons princípios sejam aplicados. Esses princípios serão abordados nesta secção.

acoplamento

Um primeiro bom princípio a seguir é o de **acoplamento** (em inglês *coupling*). Dizer que é importante aplicar um bom acoplamento nas várias partes de um projeto é afirmar que um projeto, no fim, não terá pontas soltas, isto é, que nenhuma das partes foi esquecida ao longo de um processo de desenvolvimento. Esta ideia pode provir da força que uma classe tem perante a sua ligação com outras, numa perspetiva mais orientada aos objetos. É análogo a um guindaste que ergue um conjunto de paletes com ferramentas e tijolos para um piso elevado de um edifício em construção: as paletes são retangulares e dependendo da forma como é segurado por cabos até ao guindaste, as ferramentas e os tijolos poderão, ou não, cair no chão. Isto acontece porque as cordas que ligam o guindaste à palete poderão ser anexas a cada uma das esquinas desta (4 cordas) ou simplesmente ao centro (1 corda). No entanto a diferença do acoplamento pode até nem estar no número de cordas que fixam a palete, mas antes no tipo de corda - uma corda de aço terá mais resistência que uma corda de pano, por exemplo. Até mesmo o próprio guindaste poderá influenciar no resultado.

Dado isto até poderíamos pensar em artilhar, a partir de agora, todos os nossos trabalhos com o maior nível de acoplamento que consigamos aplicar. No entanto, nem sempre é bom haver muito acoplamento. Consideremos um componente x que se encontra instalado num dado projeto. Este componente x até poderia ser útil noutras situações, co-

figura 1.19

Caso de Utilização	criar um espaço para submissão
versão	Iteração 1, v2016-1-apontamentos
descrição	O Professor cria novos trabalhos diretamente na página da disciplina, escolhendo um dos tipos disponível e configurando o período e modo de entrega. O trabalho fica disponível para os alunos e aceitam-se entregas no período indicado.
pré-condições	A Unidade Curricular (UC) existe no sistema, com edição para o semestre ativo e o Professor está incluído na respetiva equipa docente.
sequência típica	**1. Autenticar-se no sistema** Inicia quanto o Professor acede à sua página "my Moodle" para criar um novo trabalho. O sistema verifica a sessão ativa do utilizador. Se necessário, o sistema redireciona para página de autenticação do IdP central. O IdP retorna o contexto da sessão com o perfil do utilizador. **2. Selecionar Unidade Curricular (UC)** O sistema lista as UC ativas daquele utilizador na página de entrada, de forma destacada. O Professor seleciona a UC pretendida. O sistema apresenta a página de entrada da disciplina com paineis com opções para administrar a página. Se a UC estiver configurada no modo semanal, o sistema deve posicionar na semana atual, por omissão. **3. Ativar modo de edição** No painel de administração, o Professor escolhe o modo de edição. O sistema atualiza a página, mostrando opções de edição junto de todos os elementos da página, usando pequenos botões/símbolos. **4. Adicionar uma atividade** O Professor localiza o sítio da página onde pretende inserir o trabalho e escolhe a opção/símbolo de criar atividade colocada nessa zona. O sistema apresenta um quadro para escolher o tipo de atividade e o Professor seleciona a opção Trabalho. O sistema apresenta a página para definição do novo trabalho. **5. Parametrizar o trabalho.** O Professor fornece um título e descrição obrigatórios. O sistema propõe valores por omissão para o período em que o trabalho fica disponível (uma semana, a partir da data atual). O Professor define o período de entrega; o modo de grupo (individual ou grupos); e o tipo de submissão (ficheiros ou texto). **6. Confirmar edição.** O Professor confirma a configuração do trabalho. O sistema destaca problemas com campos obrigatórios, marcando o fundo com uma cor de aviso. O sistema mostra a página principal da UC, posicionada no sítio onde foi criado o trabalho.
sequências alternativas	**Passo 1: Professor não está nesta UC** O sistema verifica que o Professor não pertence à UC e mostra uma mensagem de erro. A navegação retorna a página de entrada do utilizador. **Passo 4: Importar de outra UC** O Professor pode optar por criar o novo trabalho com as definições de um trabalho definido noutra UC. **Passo 3/4: Duplicar trabalho existente** O Professor pode optar por duplicar um trabalho que já existia. No passo 3, escolhe a opção de duplicar. O sistema cria um novo trabalho, com as mesmas configurações e abre a página de edição do trabalho. **Passo 5: Carregar ficheiro com descrição.** O Professor pode anexar ficheiros na descrição do trabalho, escolhendo do sistema de ficheiros.
requisitos especiais	**[Usabilidade]** O campos de texto livre devem suportar texto com hipermedia, inserido com o apoio de um widget com opções para formatar o texto e colocar hiperligações. **[Usabilidade]** A escolha de ficheiros do sistema de ficheiros deve, em alternativa, suportar drag-and-drop para a página. **[Desempenho]** A autenticação com o IdP tem de responder em menos de 2 segs.
aspetos em aberto	- Um curso de semestres passados pode aceitar a criação de novos trabalhos? - Se o browser não aceitar HTML5 qual é a etsratégia alternativa?

mo num projeto que um amigo estaria a produzir, no entanto, como o acoplamento desse componente com o nosso projeto é tão grande, é muito difícil usá-lo fora do contexto deste, isto é, o elevado acoplamento inibe a reutilização de componentes. Mais, como ele está desenhado em função do nosso projeto, por muito que até se pudesse usar fora dele, seria difícil usá-lo fora do contexto.

coesão

Outro critério importante a ser aplicado é a **coesão**. Quando referimos que um determinado componente é coeso com o projeto significa que o componente consegue obedecer às necessidades que o projeto tem, tal como às suas responsabilidades: se um componente foi desenhado para produzir x e, ao funcionar produzir x, podemos dizer que este é coeso, contrariamente ao produzir outra coisa diferente de x. Falando em responsabilidades significa também que um componente só terá uma funcionalidade. Numa analogia com um canivete suiço é importante que um componente apenas tenha uma faca, sendo coeso com o seu objetivo final. Ter um largo conjunto de ferramentas, com o mesmo objetivo de cortar, é tornar o mesmo componente não-coeso. A coesão permite também que um determinado componente seja reutilizado, mantido e seja fácil de compreender.

diagrama de sequência

estória

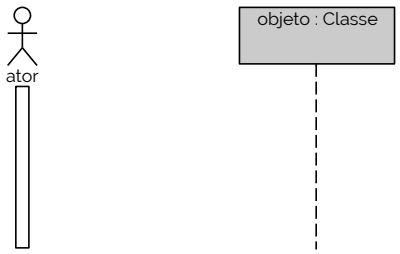
épico

A coesão e o acoplamento são dois critérios que são fáceis de avaliar tendo em conta um novo estilo de diagrama da linguagem UML - o **diagrama de sequência**. Os diagramas de sequência são usados para analisar cenários de utilização - descrição de uma forma de utilização do sistema que pode ser considerada potencial, tendo em conta um caso de utilização e a sua respetiva **estória** (descrição mais afincada de um caso de utilização, com enquadramento sequencial e cujo conjunto é um **épico**) -, para verificar o funcionamento lógico de um método ou para verificar o funcionamento e interação entre sistemas.

diagrama de sequência

Neste tipo de diagramas (diagramas de sequência) voltamos a explicitar o papel dos atores, criando linhas temporais para cada um destes [9]. Contrariamente ao que se passava nos diagramas de casos de utilização, aqui o mais importante é saber designar a sequência de utilização dos casos. Assim sendo é importante saber detalhar a forma como dois ou mais atores interagem nos nossos sistemas. Nos casos em que não pretendemos referir o papel de um ator, mas antes de uma partícula do nosso sistema, representamos o objeto que possui uma linha temporal, através de um retângulo com uma legenda do tipo `objeto : Classe`, como podemos ver na Figura 1.20.

figura 1.20



porta

O início de um diagrama de sequência poderá ser proveniente de uma mensagem exterior ou poderá ser interno. No caso de ser uma mensagem externa denominamos uma **porta**, isto é, uma transmissão de mensagem entre o nosso diagrama e algo que não esteja comtemplado no diagrama (e vice-versa). Para tal, em termos de representação, devemos ter o cuidado de começar por inserir uma seta, sem origem, no início do nosso diagrama temporal e, no fim, incluir uma seta, sem destino, na direção contrária à primeira seta.

Para a introdução deste diagrama iremos tomar em consideração o exemplo em que um aluno vai para a universidade num dia de avaliação e sai quando terminar o exame. Assim, precisamos de contemplar no nosso diagrama pelo menos dois atores: um Aluno e um Professor. Quando um aluno chega à universidade dirige-se até à porta do exame onde espera até à hora de início deste. Quando o exame começar o Professor irá chamar pelo nome do Aluno, permitindo que este procure um lugar na sala para se sentar. Após o Aluno se sentar, tal como todos os outros seus colegas, o Professor fará circular o enunciado do exame para que os alunos o possam resolver, dando início ao tempo de

exame. O exame, que demora 1h30min terminará com o Professor a dar a ordem de recolha e com o Aluno a entregar o exame ao Professor.

Para começar, há então que representar a chamada do Professor pelos alunos. Nestes diagramas podemos então executar o método "chamar um aluno" do Professor para o Aluno usando uma seta a cheio da linha temporal do Professor para a linha do Aluno. Como nós queremos que o Aluno, quando ouvir o Professor, responda "estou aqui", há que retornar um valor para o método invocado, o que se pode representar através de uma linha a traço interrompido com uma seta no sentido contrário à anterior, como podemos ver na Figura 1.21.

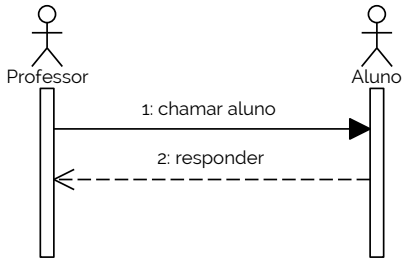


figura 1.21

De seguida o Aluno irá procurar um lugar para se sentar na sala do exame. Para representarmos esta situação temos de exibir uma situação recursiva em que o Aluno procure sempre lugar até obter um, como podemos ver na Figura 1.22.

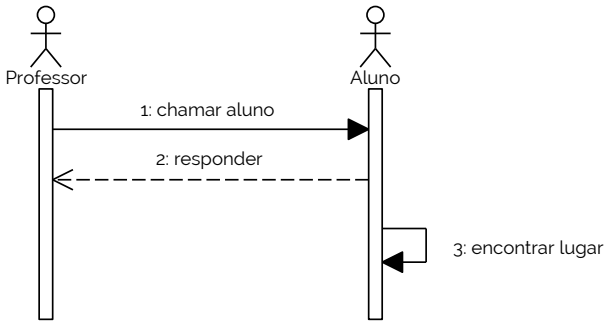


figura 1.22

Da mesma forma que um aluno procurou um lugar, também irá querer, após receber o exame, fazer todos os exercícios até que não haja mais nenhum para fazer, dentro de um espaço temporal de 1h30min. Para representarmos o tempo devemos usar uma dupla seta desde o momento em que uma determinada atividade inicia até que esta acaba e/ou a designação do tempo entre chavetas, nas proximidades do evento. Na Figura 1.23 também temos a representação de quando o Aluno entrega o exame ao Professor.

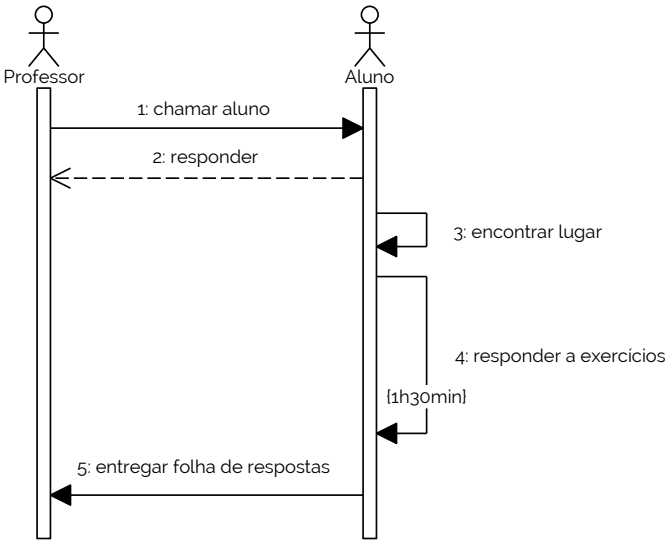


figura 1.23

Embora o diagrama da Figura 1.23 seja correto para uma comunicação bastante simples entre um Professor e um Aluno, durante o exame, o Aluno poderá querer questionar o professor para esclarecer uma dúvida à qual o Professor poderá querer responder ou não. Esta situação poderá ser explicitada no nosso diagrama. Para o fazer apenas temos de demarcar uma **região alternativa** onde seccionamos o número de áreas (operandos) necessárias para distinguir fluxos diferentes. Se for realizado um ciclo devemos, ao invés de uma região alternativa, demarcar uma **região de loop**, a qual não poderá contar com vários operandos. Este novo diagrama pode ser visto na Figura 1.24.

região alternativa

região de loop

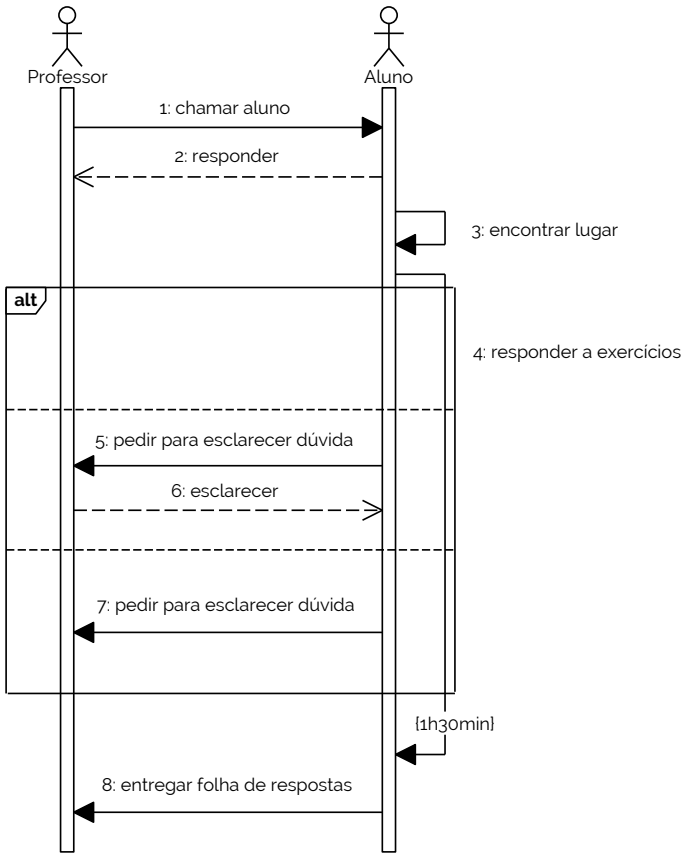


figura 1.24

Em termos de coesão é importante frisar, agora que já conhecemos os diagramas de sequência, que tudo depende das responsabilidades dos módulos. Vejamos a Figura 1.25, onde temos um exemplo de execução onde o grau de coesão é muito baixo - é muito baixo porque um mesmo módulo tem duas responsabilidades que são distintas e sem relação entre si.

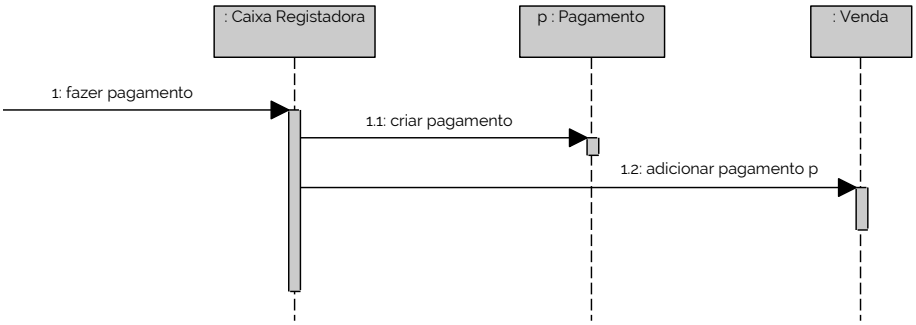


figura 1.25

Por outro lado, se conseguirmos separar as várias reponsabilidades por cada módulo existente no nosso sistema, obtemos algo semelhante como à Figura 1.26.

Para remediarmos a dificuldade de ajuste de coesão e de acoplamento e, em simultâneo, termos medidas mais próprias para arrancarmos um projeto da melhor forma,

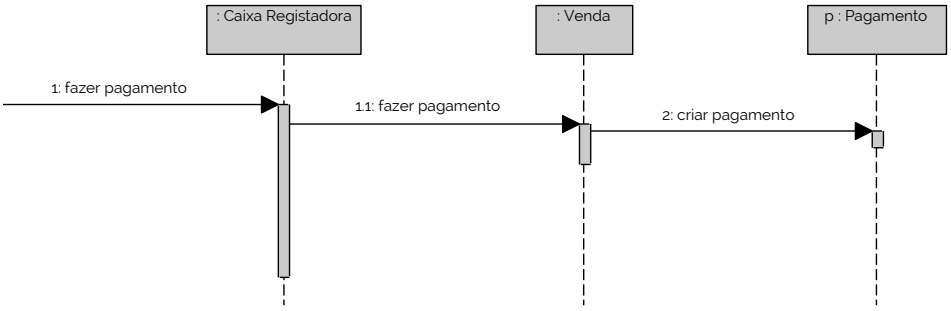


figura 1.26

existem dois modelos que nos poderão ajudar - o SOLID e o GRASP. O acrónimo **SOLID** significa *Single responsability, Open-closed, Liskov substitution, Interface segregation, Dependency inversion* (em português Responsabilidade única, Aberto-fechado, Substituição de Liskov, Segregação de interface e Inversão de dependências). Quando falamos em responsabilidade única significa que a coesão deverá ser alta, isto é, mesmo que um módulo possa suster mais que uma responsabilidade, este deverá ter uma e uma só. Em termos de Aberto-fechado, com este critério pretende-se afirmar que um módulo deverá estar aberto para extensões, mas fechado para modificações. Por substituição de Liskov pretende-se argumentar que, se A for um subtipo de B, então objetos do tipo B podem todos ser substituídos por objetos do tipo A - este conceito provém do polimorfismo do paradigma de orientação a objetos. Quando referimos segregação de interface queremos dizer que um módulo deverá preferir receber um cliente numa interface específica para o mesmo, contrariamente a numa porta com propósito geral. Por fim, para concluir o SOLID, haver uma inversão de dependências significa que não deverá ser intuito de quem desenha depender de concretizações de algo, mas antes de abstrações.

SOLID

⊙ Barbara Liskov

Estudar o SOLID leva-nos então para o estudo de um novo acrónimo, o GRASP. **GRASP** provém de *General Responsability Assignment Software Patterns* (em português Padrões de Software para Cumprimento de Responsabilidades Gerais). Nesta filosofia é importante que as várias partes sejam percetíveis para si e para os outros, de forma a que possam responder a diferentes responsabilidades, focando-se em apenas uma de caso para caso. Para que isso possa acontecer é importante que os modelos consigam perceber o que são, para o que servem e com que intuito é que são usados - quais são as interações que um modelo tem perante outros? Depois, há que tentar saber identificar a responsabilidade do módulo no contexto de aplicação.

GRASP

Responsabilidades de módulos e diagrama de classes

Para estudarmos as várias responsabilidades entre os vários módulos de uma determinada implementação de um sistema precisamos, para sermos mais corretos e concisos, de investigar de forma mais detalhada. Para isso precisamos de usar os nossos conhecimentos de paradigma de orientação aos objetos e investigar o sistema classe a classe, tomando partido das várias relações (e tipos de relações) entre as várias classes.

De forma a podermos analisar as várias relações entre as classes podemos usar um novo diagrama da linguagem UML - o **diagrama de classes**. O universo da linguagem UML está dividido, na sua versão 2.0, em dois grandes grupos - os diagramas comportamentais e os diagramas estruturais. Os diagramas que temos visto até agora (diagrama de atividades, de casos de utilização e de sequência) são os mais usados dentro dos comportamentais - têm esta designação porque exibem, com detalhe, as relações e o sentido das relações entre as várias partes de um processo. Agora passamos para um nível diferente, onde vemos como é que os sistemas estão construídos e onde é que os processos ocorrem - entramos nos diagramas estruturais.

diagrama de classes

No diagrama de classes o elemento mais atómico de todos, e onde iremos basear o nosso estudo, é a **classe**. Uma classe pode ser vista como um tipo de dados/objetos, que é definida através de um conjunto de atributos e um conjunto de métodos. Para represen-

diagrama de classes

classe

tarmos uma classe em UML usamos um retângulo tri-partido, onde numa primeira parte designamos um nome para a mesma, depois designamos os atributos e, no fim, os métodos [10]. Note-se que os atributos **privados** são identificados por um carater '-' no início dos mesmos, enquanto que os métodos **públicos** são identificados por um '+'. Veja-se na Figura 1.27, onde se exibe um exemplo de classe.

privados

públicos

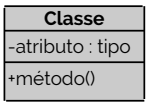


figura 1.27

Note-se, não obstante, que na definição dos atributos, como podemos ver na Figura 1.27, têm a explicitação do seu tipo, isto é, se for um atributo, por exemplo, `Nome`, referente a uma classe `Pessoa`, este será designado como `nome : String`.

Uma classe poderá ter várias relações com outras classes. Uma primeira relação que podemos estudar é a mais simples de todas a **associação**. As associações podem ser bidirecionais ou unidirecionais. Uma associação é uma relação entre duas classes que têm uma designação própria do domínio do problema. Por exemplo, se tivermos duas classes `Aluno` e `Trabalho`, podemos ter uma associação referente ao `fazer`: o Aluno faz Trabalhos e os Trabalhos são feitos por Alunos. Note-se que, para além da associação que estamos a referir, também estamos a falar em termos de número, isto é, estamos a aplicar **multiplicidade** a esta relação. A multiplicidade pode ser a mais variada possível desde que aplicável no domínio em concreto, sendo que o carater '*' significa 0 ou mais. Vejamos a Figura 1.28.

associação

multiplicidade

figura 1.28

Na descrição da associação da Figura 1.28 temos também uma direção aplicada. Esta direção não significa que o Trabalho não se relacione com o Aluno, mas existe apenas para simplificar a leitura do diagrama - é uma ajuda para a leitura. Referimos este caso porque é fácil confundir com uma associação unidirecional, como podemos ver na Figura 1.29, onde entre duas classes só uma é que sabe que a relação existe.

figura 1.29

Em termos de diferenças entre a Figura 1.29 e a Figura 1.28 temos que na Figura 1.29 a associação só é vista na perspetiva do Aluno, e não do Trabalho. Esta solução até pode não fazer sentido - tudo depende do domínio do problema em questão.

Existem outros tipos de associação, entre as quais as agregações, composições e as generalizações. Dizemos que uma classe A agrega uma classe B se e só se B é parte de A. Consideremos as classes Carro e Motor. Sendo um Motor parte de um Carro, então podemos dizer que Carro tem Motor. Esta relação, denominada de **agregação**, é representada por um losango na extremidade que possui a classe, como podemos ver na Figura 1.30.

agregação

figura 1.30

Existe um caso particular de agregação, denominado de **composição**. A composição diz que sendo B parte de A, A não existe sem pelo menos uma instância de B, o que é, também, o caso de "Carro tem Motor", dado que um Carro não existe sem Motor. Assim sendo, na Figura 1.31, especificamos ainda mais a relação entre Carro e Motor.

composição

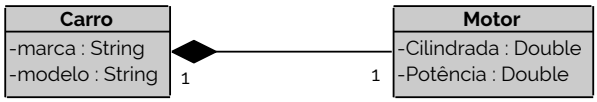


figura 1.31

O terceiro tipo de associação que referimos que existe, mais atrás, toma partido de uma das beneces do paradigma de orientação a objetos: a **herança**. Assim sendo, estamos a falar de uma relação de **generalização**, a qual se designa através de uma seta completa, como podemos ver na Figura 1.32.

herança

generalização

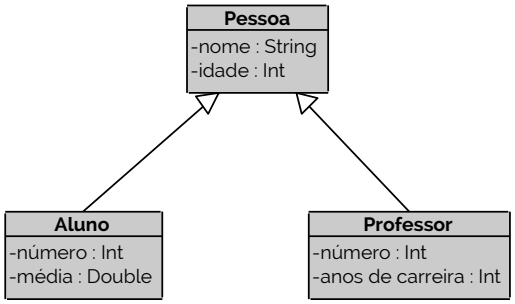


figura 1.32

As associações também não têm de ser tão lineares como as que temos vindo a representar ao longo das figuras anteriores. Tal como entre duas classes podemos ter mais do que uma associação, mesmo que de tipos diferentes, é importante saber que também é possível ter relações com a própria classe, quase que de forma recursiva, isto é, **reflexiva**. Por exemplo, um Empregado pode gerir por outro Empregado (sendo este gerido pelo primeiro), como podemos ver na Figura 1.33.

reflexiva

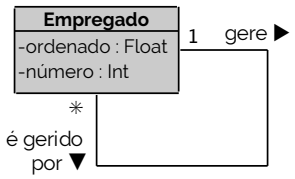


figura 1.33

Com as associações também podemos ter **classes de associação**, isto é, classes que provêm da relação entre duas classes. Consideremos um exemplo do pagamento do serviço Via Verde, ou outro semelhante, na passagem de identificadores em pórticos da autoestrada. Para representar uma Passagem consideramos que esta é obtida quando há uma relação entre o Identificador que se encontra no carro e o Pórtico que contém um leitor para o identificador, de forma a poder saber quem é que passou. Vejamos assim a Figura 1.34, onde representamos a classe de associação com uma linha a traço interrompido orientado à relação entre as classes Leitor e Pórtico.

classes de associação

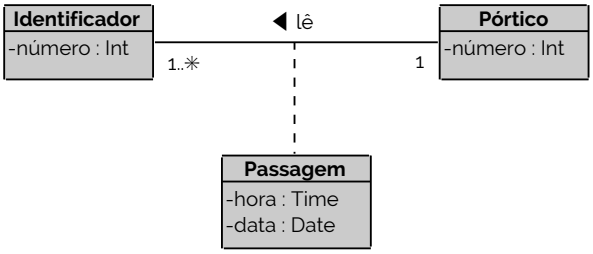
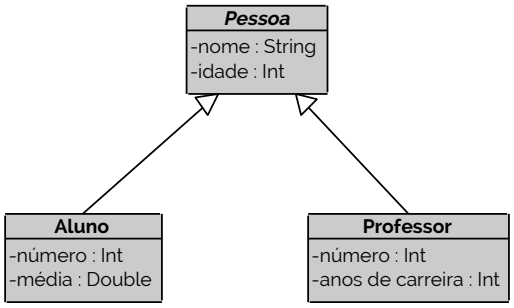


figura 1.34

Em Programação III (a2s1) também referimos, no contexto da programação orientada aos objetos, em **classes abstratas**. Uma classe abstrata é tal que não é possível (porque não é interessante para o domínio do problema) criar objetos dessa classe. Em termos de UML é possível designar uma classe abstrata através da formatação itálica do nome da classe, como podemos ver na Figura 1.35.

classes abstratas

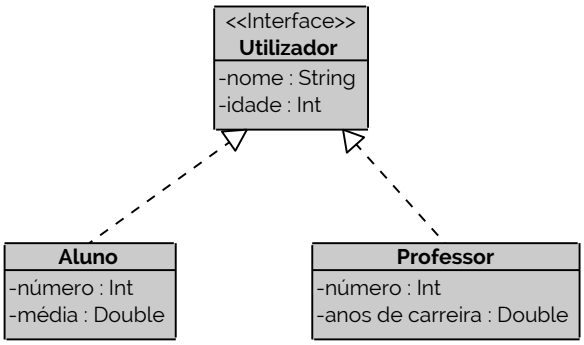
figura 1.35



Tal como referimos os casos de classes abstratas, também referimos a importância das **interfaces**, como se usa em linguagens de programação como Java. Estas, em UML, são representadas usando o estereótipo `<<interface>>` o qual, para serem implementadas, são representadas pelas mesmas setas que na figura anterior, mas a traço interrompido, como podemos ver na Figura 1.36.

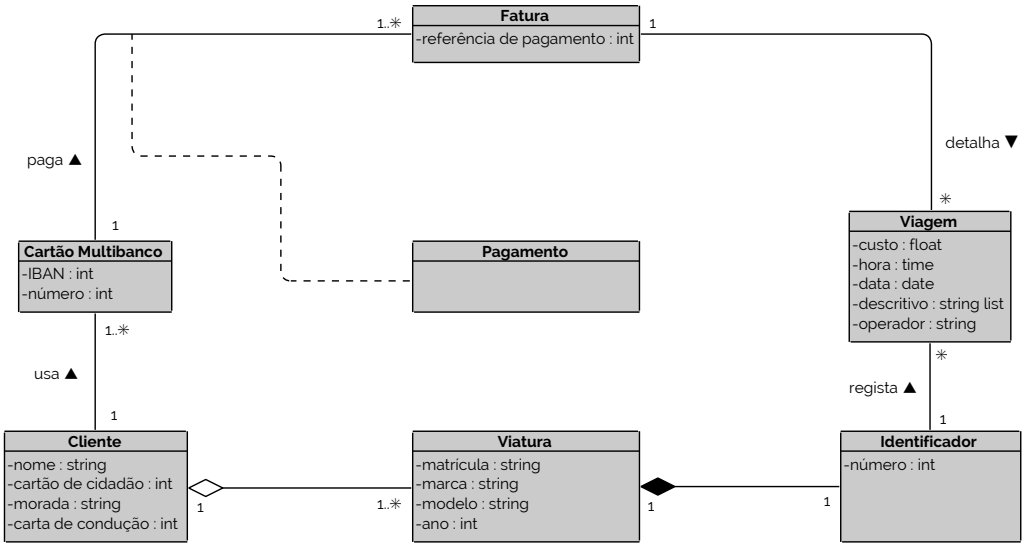
interfaces

figura 1.36



Consideremos agora, num exemplo um pouco mais complexo, o sistema Via Verde, onde é feita uma Fatura, por mês, por todas as passagens que são cumpridas por uma viatura, nos pórticos da autoestrada. Vejamos a Figura 1.37 [11].

figura 1.37



Comecemos por analisar a Figura 1.37 da Fatura para a direita. Uma Fatura é uma classe cujos objetos são definidos por uma referência de pagamento e que detalha viagens - mesmo que não existam viagens haverá sempre uma fatura que indica que não houve viagens, daí a multiplicidade ser 0 ou mais viagens. Uma Viagem é uma classe de objetos definidos por um custo, uma hora, uma data, um operador da viagem e um descritivo. Várias viagens, ou nenhuma, são registadas pelo único Identificador que está presente em

cada Viatura. A Viatura tem necessariamente de ter um Identificador, para desfrutar do serviço Via Verde. Esta viatura, entre outras possíveis, têm de estar registadas com o nome de um Cliente, mesmo que este Cliente não seja o condutor efetivo das primeiras. Dado o registo do Cliente é este o titular de um, ou mais, Cartões Multibanco que servirão para pagar o detalhe da Fatura com que iniciámos a análise. Neste último processo é gerado um Pagamento, detalhado aqui como uma classe de associação entre Cartão Multibanco e Fatura.

2. Arquitetura de Software

Não obstante a tudo o que fora contemplado no capítulo anterior, o presente capítulo pretende referir como é que os projetos de software são desenhados - o que são e como são executados os seus processos e interações.

Todos nós conhecemos a marca Lego® e sabemos as construções que podem ser feitas com as peças desta marca. Consideremos assim uma peça Lego como um componente de um software que está em desenvolvimento. À medida que o projeto vai avançando as peças começam a encaixar-se, mas no fim o conjunto tem de ser íntegro, sem pontas soltas e equilibrado.

Criação de componentes ou módulos

As várias peças de um bloco Lego têm um significado diferente para cada montagem, isto é, para cada projeto de software onde são montadas. Por exemplo, se montarmos um carro, uma das peças pode servir para fazer o parachoques dianteiro, enquanto que noutro projeto qualquer pode servir como uma base para uma grua - mas a peça é a mesma, o que é que mudou? Tomando partido dos conhecimentos de acoplamento e de coesão, que estudámos anteriormente, podemos verificar que os módulos têm responsabilidades diferentes dependendo da sua aplicação, embora sejam os mesmos módulos.

Para criarmos um módulo podemos pegar num diagrama de casos de utilização e cortar a especificidade dos atores em causa, por outras palavras, a responsabilidade dos módulos muda principalmente com os atores que interagem com o sistema. Se um dos atores tiver necessidade de níveis de segurança bastante apertados, então é importante que um determinado módulo tenha um cuidado especial com a segurança. Pelo contrário, se um determinado ator for muito genérico, essa preocupação não deverá ser tão tida em conta. Assim sendo, vejamos a Figura 2.1, onde pegámos na Figura 1.16 e retirámos os atores.

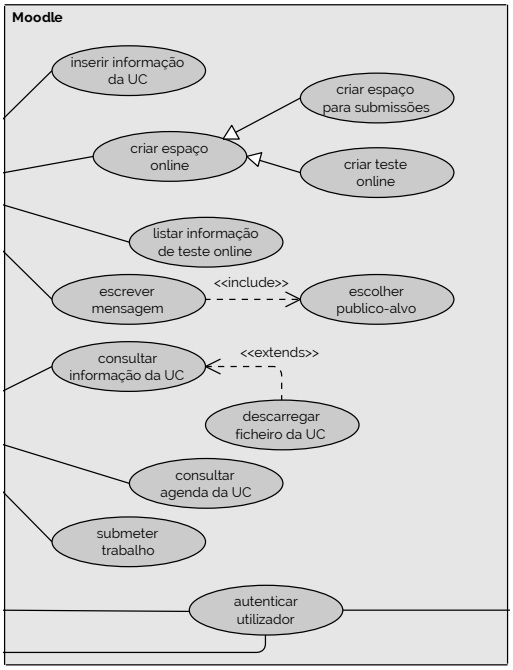


figura 2.1

Olhando para a Figura 2.1 podemos reparar que agora temos um sistema Moodle cujas interações ainda não estão definidas, isto é, foram deixadas em aberto. Assim sendo, podemos referir que criámos uma **API** ou apenas uma **interface de utilizador**. Neste caso, dependendo dos atores que lhe forem atribuídos, cada módulo terá uma responsabilidade diferente para os sistemas externos, mas a mesma responsabilidade para o sistema interno, dado o nível de coesão que já fora insituído em fases anteriores.

API, interface de utilizador

Tendo agora um **componente**, podemos juntá-lo a toda uma rede de outros componentes através do estudo da **arquitetura** de software [12]. Olhando então para a arquitetura podemos imaginá-la como uma rede de nós - como um grafo - onde cada nó é um componente e cada aresta é um canal de comunicação. Cada componente tem uma ligação com o canal de comunicação - tem uma **interface**. Mais, com um conjunto de vários componentes podemos gerar um **subsistema**. Isto permite que a arquitetura seja modular, de forma a facilitar a correspondência de responsabilidades entre os vários membros do sistema. Podemos agora definir a arquitetura de software como a descrição de subsistemas e de componentes de um sistema de software e a relações entre eles.

componente
arquitetura
interface
subsistema

Diagrama de Componentes

Agora que aumentámos a nossa escala de abstração mais um nível, olhando para um sistema como um conjunto de vários subsistemas que, por si, são conjuntos de interações entre vários componentes, é possível, sob a representação da linguagem de modelação UML, representar um **diagrama de componentes**, onde se detalham as várias interações entre vários componentes. Nestes diagramas passamos agora a descrever a organização de software estático, como códigos de programação, ficheiros, entre outros, pacotes e organizações em camada.

diagrama de componentes

Neste tipo de diagrama o elemento mais atómico, sendo um diagrama de componentes, é um componente, que em UML representamos por um retângulo com duas partículas de interação com o exterior, como podemos ver na Figura 2.2.

diagrama de componentes

figura 2.2

Podemos designar como componente uma parte física e substituível de um sistema que fornece a realização de um conjunto de operações num conjunto de interfaces. Dentro deste tipo de diagramas os blocos não variam muito, sendo que os vários blocos distinguem-se, essencialmente, pelo estereótipo que contêm. No caso de um componente eles têm em si escrito `<<component>>`, mas noutros casos têm outras designações, detalhadas na Figura 2.3 [13].

figura 2.3

estereótipo	tradução	descrição
subsystem	subsistema	componente como unidade de decomposição para sistemas maiores
process	processo	componente baseado em transações
service	serviço	componente funcional, stateless
specification	especificação	classificação que especifica um domínio de objetos sem definir a implementação física de tais objetos
realization	realização	classificação que especifica um domínio de objetos e que também define a implementação física de tais objetos
implement	implementa	definição de componente da qual não se pretende qualquer especificação por si. Equivalente a um estereótipo <<specification>> com dependências.

Os componentes, sejam eles quais forem, se interagem com o exterior, então necessitam de um ponto de conexão. Nestes diagramas chamamos-lhe de **porta** e representamo-las através de quadrados anexos aos contornos da figura de componente. A cada um

porta

destes portos também podemos ter uma ou mais interfaces. Estas interfaces podem ser de dois tipos. Consideremos o sistema de uma ficha de eletricidade. A tomada na parede é uma interface para uma ligação de energia com um dispositivo externo. Para nos ligarmos a tal tomada temos uma ficha específica, com a mesma forma que o encaixe que nos é fornecido. A tomada aqui tomará o sentido de uma interface que fornece um conteúdo, que em termos de UML se representa através de um círculo vazio. Por outro lado, a ficha será o instrumento que iremos usar para receber o conteúdo que um componente está a partilhar, o qual se representa por um semi-círculo aberto. Assim sendo, e como ambas as interfaces funcionam num sistema análogo ao de chave-fechadura, quando ambas se encontram ligadas, significa que pelo menos um serviço está a ser partilhado entre dois componentes. Veja-se a Figura 2.4.

figura 2.4

Para conectar várias interfaces de um componente podemos usar linhas, análogas a cabos, reencaminhando sinais de um componente para outro. Por exemplo, na Figura 2.5 temos o exemplo de uma interação entre um componente de uma loja online, com autenticação de um cliente, que suporta uma segunda loja que contém um serviço de pesquisa.

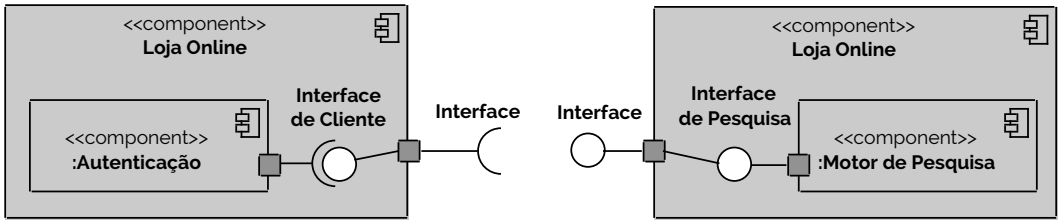


figura 2.5

Este diagrama, tal como os outros que estudámos, complementa-se com outros tipos de diagrama UML, especialmente com um cujo nível de abstração é ainda maior - o **diagrama de instalação**. Um diagrama de instalação (em inglês *deployment diagram*) é tal que interage diretamente com os diagramas de componentes através de **manifestos**, onde se definem todas as caraterísticas dessa comunicação (políticas, regras, ...). A nível interno é aqui que se representam as ligações entre vários nós que executam software, sob a forma de **artefactos**, isto é, sob a forma de elementos que podem assumir vários valores, dependendo do caso de aplicação. Neste tipo de diagrama o elemento mais básico é o **nó**, representado por um cubo, sendo um elemento físico que existe em tempo de execução e que representa um sistema computacional, entre outros... Na Figura 2.6 podemos ver um nó.

diagrama de instalação
manifestos
artefactos
nó
diagrama de instalação

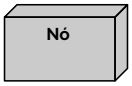


figura 2.6

Os componentes integrados nos nós, que são geridos por manifestos, também podem ter dependências entre si, representadas por setas a traço interrompido. Na Figura 2.7 temos o caso do desenvolvimento do HTML num servidor Web [14].

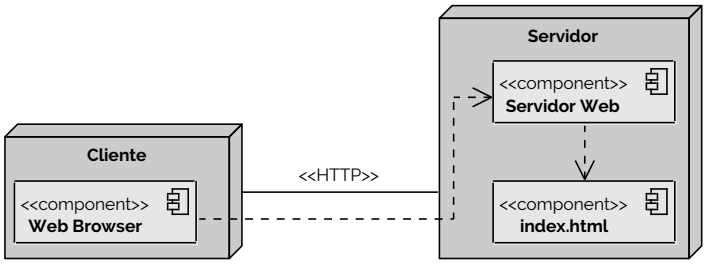


figura 2.7

Se pretendermos ser ainda mais globais em termos de abstração, então podemos organizar todos os elementos com atributos comuns numa só representação - um **pacote**. Através de um **diagrama de pacotes** é possível simplificar diagramas mais extensos e ver o primeiro nível de comunicações entre os vários pacotes. Vejamos a Figura 2.8, onde se representam pacotes genéricos e a respetiva ligação entre eles.

pacote
diagrama de pacotes
diagrama de pacotes

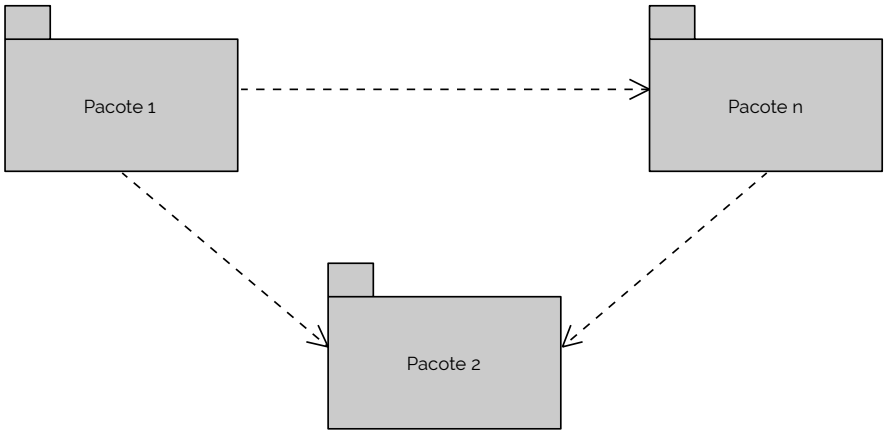


figura 2.8

Fases de desenvolvimento, programação e riscos

Tendo estado a discutir as várias fases de desenvolvimento já devemos saber se existem diferenças com as fases de programação. Como pudemos ver logo no início, os projetos têm várias fases de desenvolvimento, sobre as quais iteramos várias vezes até obtermos o resultado pretendido. A **programação** em si é uma parte dessas fases, mais em particular, da fase de elaboração e construção, sendo ela uma ferramenta para a construção de um projeto.

programação

Ao longo de um processo de desenvolvimento dos sistemas podemos ser confrontados com um largo conjunto de **riscos** que provém das ações que nele realizamos. Estes riscos nem sempre são comtemplados na análise de requisitos ou previstos na fase de conceção, sendo episódios não desejados que ocorrem ao longo do projeto. No entanto, por ser indesejado, não significa que tenha de produzir um resultado negativo em 100% das vezes. Ainda assim, se o risco não for negativo, damos-lhe vulgarmente o nome de **oportunidade**. Quando isto acontece há que conseguir explorar, reorganizando o projeto de forma a garantir que a oportunidade reapareça, aumentar o mesmo, definindo ações para aumentar a probabilidade do impacto do risco, e partilhar a oportunidade para quem mais possa capturar a oportunidade pelo bem do projeto.

riscos
oportunidade

Em termos de **riscos negativos** há que saber evitá-los, reorganizando o projeto para garantir que nunca volte a acontecer, atenuá-los, reduzindo a probabilidade ou o impacto do risco e até transferi-los, reorganizando o projeto para que outros lidem com eles, transferindo-lhes a responsabilidade.

riscos negativos

Só na fase de conceção é que se podem iniciar as possibilidades de haver risco ao longo do projeto e, em simultâneo, por ser a primeira fase de desenvolvimento, esta é a fase em que o valor é diminuto, mas o risco é o maior. Assim sendo, a grande probabilidade de ocorrerem riscos provêm de não tomar atenção aos requisitos mais cruciais, não fazer a representação mais adequada dos clientes, modelar requisitos apenas funcionais e tentar aperfeiçoar requisitos antes de os experimentar na construção [15].

3. Realização de testes

Quando criamos um produto, seja ele feito de código ou de outro tipo de ferramentas, é bom que, antes da publicação deste, seja feitos conjuntos de **testes** para averiguar a boa funcionalidade dos módulos que o constituem.

testes

Todos os produtos que existem já tiveram boas e más fases ao longo do seu desenvolvimento. Estas más fases, de forma a serem corrigidas, necessitam de ser identificadas primeiro, mas num produto final é praticamente impossível conseguir designar uma falha a um componente. Vejamos um exemplo de uma falha grave que ocorreu em janeiro de 1990, nas centrais telefónicas da AT&T [16]: a 15 de janeiro de 1990, cerca de 60 000 clientes de longa distância tentaram fazer chamadas também de longa distância, como habitual, só que não obtiveram qualquer resposta. O que acontecera é que os *swtiches* encarregues das comunicações de longa distância, todos esses 114 equipamentos, entraram num estado de reinício sem cessar. Embora a AT&T tenha assumido que estaria, muito provavelmente, a ser atacada por nove horas sucessivas, a companhia tentou desvendar o problema. No final, identificaram o problema como sendo do novo *software* que fora instalado. Os *switches*, na altura, se ficassem congestionados, enviavam (cada congestionado), uma mensagem "*do not disturb*" para que outros não o perturbassem. Assim, um segundo *switch* que receba tal mensagem, passado um tempo, re-verifica o primeiro e, se detetar atividade, reinicia para mostrar que o *switch* 1 está novamente ativo. Acontece, no entanto, que nesta atualização de *software* os *switches* enviavam não uma, mas duas mensagens "*do not disturb*", a segunda, sendo recebido pelos *switches* enquanto estes estavam a reiniciar, voltando a reiniciar e passando a mesma mensagem para os seguintes.

Tendo problemas pequenos como o anterior em vista, que causaram um enorme impacto nos negócios - neste caso fez a AT&T perder cerca de 60 milhões de dólares americanos - é imporante, enquanto uma equipa vai produzindo um produto, ir verificando o mesmo através de vários testes à aplicação - tanto específicos a um módulo, como genéricos para aferir a capacidade do sistema num todo.

Verificação e validação

Dois dos parâmetros, entre os quais um já referimos anteriormente, importantes para aferir a qualidade de uma solução são a validade e a verificação. Comecemos pela verificação.

verificação

A **verificação** traduz-se por aferir se o produto está a ser criado nas suas devidas condições. Por outras palavras, na elaboração de um projeto é importante saber se as várias soluções que estamos a expor estão coerentes com a melhor forma de apresentar um resultado correto, isto é, por muitas soluções que tenhamos, o resultado das mesmas pode ser obtido de boas ou más maneiras - quem julga isso é a verificação.

validação

Por outro lado, podemos até ter um produto que nos dá algo com qualidade, mas não propriamente o que pretendemos - é aqui que a **validação** atua. A validação pretende assim aferir se o produto certo está a ser criado.

qualidade

Note-se assim que a validação possui uma importância maior que a verificação, não obstante, não se poderá ter um projeto que não delibere as suas ações tomadas sem ambos os critérios de teste [17]. Isto acontece principalmente porque o sucesso de um projeto depende da sua **qualidade**, aferida pelos critérios em estudo.

Testar a qualidade de um produto

Aquando do teste de uma aplicação ou produto há que saber identificar que parte é que deve ser testada e porque razão. Podemos assim efetuar várias questões às nossas criações: questões como "qual é o propósito dos testes" ou "o que são testes" podem ser respondidas com a noção do projeto (o conceito e idealização em si). Por outro lado, questões mais técnicas como "como testar", "quão bem conheço eu o projeto", "quando é que iniciamos/paramos os testes", "com que outros produtos comparamos o nosso" ou "com o que é que testamos o nosso produto" esperam respostas mais próximas do âmbito do projeto, conhecimento, processo, referência e *frameworks* a serem usadas.

Para termos um bom julgamento de cada uma destas categorias é importante assim saber designar uma diferença entre várias fases de um problema. Podemos assim identificar três fases especiais: a primeira é a **falta**, a qual se identifica por ser uma discrepân-

falta

cia num determinado estado do sistema, porque um componente não está presente, por exemplo (uma linha de código ausente, um módulo de um motor que está inexistente, logo não permite que este ligue, ...); uma segunda fase é a **falha**, que sendo comportamental, poderá ser detetada e resolvida enquanto está em execução; a última fase é a de **erro**, a mais grave de todas e que se define como a causa geral das falhas.

falha

erro

Caixa negra e branca (black e white boxes)

A execução de testes é então feita, dependendo do âmbito, genericamente em duas abordagens diferentes: numa perspetiva exterior ou interior de um produto. Quando nos referimos ao exterior de um produto queremos apontar a sua interface. Consideremos para o efeito uma função, de uma linguagem de programação arbitrária, chamada `addFour(x)`. Com esta função esperamos assim que se dermos o elemento '6' como entrada à mesma, nos seja dado como valor de retorno essa quantidade somada de 4 unidades, isto é, o valor '10'. Na verdade, para o teste desta função, se a considerarmos como algo a ser avaliado por fora (como uma **caixa negra** - em inglês *black box*) então não nos interessa saber como é que ela funciona por dentro, mas sim apenas se faz o que lhe compete ou não. Assim sendo, podemos efetuar testes sobre os resultados delas obtidas, com um determinado conjunto de parâmetros de entrada. Por exemplo, se dermos a quantidade '8' como entrada esperamos um '12' à saída, havendo um erro caso tal não aconteça.

caixa negra

Tal como podemos avaliar um determinado produto pelo seu exterior, também podemos avaliá-lo num paradigma de **caixa branca** (em inglês *white box*), sendo que temos acesso ao interior do mesmo. Voltando ao exemplo da função `addFour(x)` podemos agora exemplificar o caso da *white box* avaliando os vários estados da função em execução ao longo do tempo.

caixa branca

Consideremos assim as funções do Código 3.1.

código 3.1

```
# função a)
def addFour(x):
  return x + 4

# função b)
def addFour(x):
  sum = x
  for i in range(1, 5):
    sum += 1
  return sum
```

Quando olhamos para ambas as funções do Código 3.1 quais são as diferenças entre ambas? Ora, numa perspetiva de caixa branca, em que temos acesso aos seus interiores, elas são diferentes porque enquanto que uma adiciona uma constante ao que lhe entregamos como argumento, a outra executa ciclos de incremento de forma ingénua. No entanto, numa perspetiva mais exterior (de *black box*) temos que as duas funções são exatamente iguais, isto é, qualquer que seja o argumento de entrada a saída é a mesma e a assinatura das funções são perfeitamente iguais.

Mas quais são as vantagens e desvantagens de ambas as perspetivas? Ora, como vantagens de uma avaliação de caixa preta temos que: podemos aplicar testes mesmo não sabendo a forma como os módulos estão implementados; os produtos acabam por ser mais robustos face a implementações, dado que apenas estamos a avaliar a interação das interfaces para com o utilizador, e não algum sinal aberto dentro de uma implementação que possa existir. Uma última vantagem visível para os sistemas de caixa preta é que os testes poderão ser desenhados mesmo antes de existir qualquer implementação.

Em termos de vantagens para os sistemas de caixa branca temos que podemos aferir melhor os resultados dos nossos testes, tendo noção dos vários componentes que se encontram dentro dos nossos equipamentos (em inglês diria inclusive, *under-the-hood*). No entanto, contrariamente às *black boxes*, estas têm desvantagens grandes, como o elevado número de locais a serem cobertos por testes, como o número de linhas de código, verificação de cada estado dos registos numa execução, ... (exemplos para uma aplicação computacional).

O que acontece, em suma, é que enquanto que com uma perspetiva de caixa negra podemos testar um módulo na sua funcionalidade mais orientada para o cliente, com uma perspetiva de caixa branca temos uma noção de teste mais orientado para pormenores técnicos do produto, de forma ajustada a quem os cria.

Fases e tipos de teste

Os testes, por si, são elementos funcionais que devolvem valores lógicos: falhou ou foi aceite. Assim sendo, olhando ao uso que um determinado cliente poderá fazer de um produto é conveniente que este já se encontre num estado **fechado** para o seu uso como um todo, isto é, não expondo o cliente a qualquer detalhe mais técnico de desenvolvimento do mesmo.

fechado

Até que um sistema possa ser fechado geralmente passa por um conjunto de fases de testes que vão ampliando o espetro de clientes experimentais e o número de funções fechadas. Inicialmente, quando um produto está para ser levado para os clientes é criada uma fase de nome **alfa** que permite que os clientes operem os sistemas, mas em ambientes controlados, não de produção. Esta fase permite assim que os desenvolvedores (*developers*) possam ter consciência de algumas consequências não esperadas de funcionamento dos seus produtos.

alfa

De seguida, após uma fase alfa, os módulos são sujeitos a uma fase **beta**, isto é, a uma fase onde já se transita para uma execução em ambiente de produção, contudo, com um controlo do mesmo. É análogo a levar um equipamento que fazemos dentro das nossas casas, que funcionou connosco, mas que agora pretendemos que alguém, fora das nossas casas, também teste, no entanto, antes de toda a gente a poder usar.

beta

Nos dias que correm surgem ainda mais fases de teste, ainda mais em simultâneo com o uso esperado dos equipamentos, isto é, em **testes paralelos**. Consideremos a aplicação da Apple, o TestFlight, que permite que as aplicações estejam sobre testes aplicacionais enquanto os utilizadores as usam nas suas fases finais de atualização. Este tipo de testes permite um olhar mais apurado sobre o real uso das suas criações e possíveis transtornos de quem as usa, tal como criar um melhor canal de comunicação entre quem usa um produto e quem o cria [18].

testes paralelos

Desenvolvimento orientado a testes (TDD)

Um possível paradigma a aplicar para a criação e uso de testes no desenvolvimento de um projeto é o **TDD** (sigla inglesa para *Test-Driven Development*), como já tivemos oportunidade de verificar na disciplina de Laboratórios de Informática (a1). Neste paradigma criam-se três estados possíveis ao longo de toda a criação de um projeto: um estado para as falhas, um estado para aceitação e outro estado para o *refactoring*. Vejamos cada um destes estados com mais detalhe.

TDD

Inicialmente quando se cria um projeto geram-se testes para aferir a qualidade deste. Assim sendo, e porque nenhum teste foi ainda realizado ou falharia, caso fosse, estaríamos num estado de falha (ao qual, por uma analogia, denominaremos de estado vermelho). Quando os testes estiverem prontos e o sistema os passar, então este transita para um estado, por analogia, verde, onde se denotam os testes como passados. Estando neste último estado é agora necessário recomeçar a reorganização e acrescento de funções ao nosso sistema, criando novas tarefas para tal, entrando num novo estado, o de *refactoring*. Este processo encontra-se simbolizado na Figura 3.1.

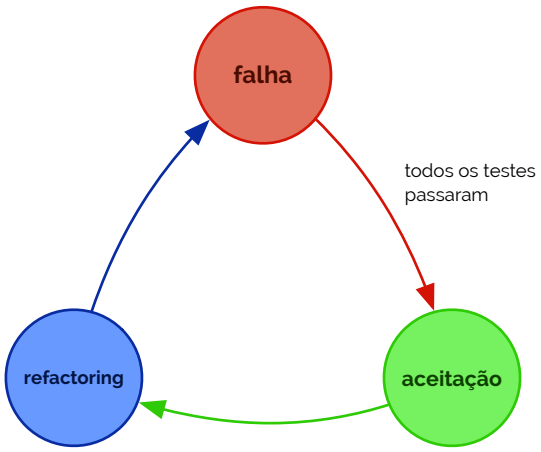
Note-se que este paradigma, embora possa ser bastante seguido (inclusive sem se saber), não tem de ser o melhor em algumas vezes. O próprio *refactoring* não significa que a reforma que possa ser implementada no que já está feito possa ser melhor ou beneficiador para o nosso projeto. Pelo contrário, até pode acontecer uma generalização dos processos, tornando o nosso sistema mais ineficaz.

Este tipo de paradigma, no entanto, consegue tornar as várias tarefas designadas num grupo de trabalho bastante automáticas, isto é, permite que o nível de automação

seja de tal forma elevado que mal uma parte do sistema esteja concluído numa determinada fase de desenvolvimento, sejam logo efetuados testes à medida, inclusive testes criados de forma automaticamente por cada detalhe da unidade sobre teste (**UUT**).

UUT

figura 3.1



Desenvolvimento orientado ao comportamento (BDD)

Outro modo de testarmos os vários componentes é partirmos da perspetiva de quem os usa. Inicialmente quando criamos um projeto necessitamos de ter uma ideia para a sua conceção. Nela devemos especificar um conjunto de estórias para a sua utilização. Com estas estórias podemos agora verificar a funcionalidade do nosso sistema para resolver os vários problemas nelas designados. Quando estudamos um sistema nesta perspetiva estamos a fazer um desenvolvimento orientado ao comportamento, em inglês, **BDD** (sigla para *Behavioural-Driven Development*).

BDD

Em termos muito básicos, aqui define-se o que é que uma função, por exemplo, deverá e não deverá fazer, numa ponte entre os utilizadores, os desenhadores das interfaces de contacto com e os clientes.

Para efetuar este tipo de testes existem ferramentas que interpretam condições de aceitação ao estilo de silogismos simples, usando palavras como "dado", "quando" e "então". Tais testes são vulgarmente denominados de **Given When That** e assumem a forma do Código 3.2.

Given When That

código 3.2

```
Given some initial context,
When an event occurs,
Then ensure some outcomes.
```

Mantendo as normas deste documento, iremos exemplificar o uso destes testes em português, seguindo o estilo do Código 3.3.

código 3.3

```
Dado alguns contextos iniciais,
Quando um evento acontece,
Então há que assegurar que algo é produzido.
```

Este tipo de linguagem é desenvolvido no **Gherkin**, uma linguagem próxima do inglês usada para uma *framework* denominada de Cucumber. Um exemplo de aplicação é o do Código 3.4, onde a estória onde se enquadra é "Utilizador levanta dinheiro" [19].

Gherkin

código 3.4

```
Como Utilizador
Quero levantar dinheiro da ATM
Para que possa obter dinheiro quando o banco está fechado

Cenário 1: Conta tem fundos suficientes
Dado que o saldo bancário é de 100€
  E o cartão é válido
  E a ATM contém dinheiro suficiente,
Quando o Utilizador pede 20€,
Então a ATM deve dispensar 20€
```

```
  E o saldo da conta deve ser 80€
  E o cartão deve ser devolvido.

Cenário 2: Conta não tem fundos suficientes
Dado que o saldo bancário é de 10€
  E o cartão é válido
  E a ATM contém dinheiro suficiente,
Quando o Utilizador pede 20€,
Então a ATM não deve dispensar nada
  E a ATM deve alertar que o Utilizador não tem dinheiro suficiente
  E o saldo da conta deve ser 10€
  E o cartão deve ser devolvido.

...
```

E assim terminam os apontamentos da disciplina de Análise e Modelação de Sistemas (a3s1), prosseguindo o estudo para as disciplinas de Projeto em Engenharia Informática (a3s2) e Engenharia de Software (a4s2).

31 ANÁLISE E MODELAÇÃO DE SISTEMAS

1. Introdução a um Sistema de Informação

Uma relação entre o cliente e o produto final 2
Processo de negócio e diagramas de atividade 3
Requisitos de um sistema/processo 8
Critérios para o desenho de projetos e diagramas de sequência 13
Responsabilidades de módulos e diagrama de classes 18

2. Arquitetura de Software

Criação de componentes ou módulos 22
Diagrama de Componentes 23
Fases de desenvolvimento, programação e riscos 25

3. Realização de testes

Verificação e validação 26
Testar a qualidade de um produto 26
Caixa negra e branca (black e white boxes) 27
Fases e tipos de teste 28
Desenvolvimento orientado a testes (TDD) 28
Desenvolvimento orientado ao comportamento (BDD) 29

As referências abaixo correspondem às várias citações (quer diretas, indiretas ou de citação) presentes ao longo destes apontamentos. Tais referências encontram-se dispostas segundo a norma IEEE (as páginas Web estão dispostas de forma análoga à de referências para livros segundo a mesma norma).

[1] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 2 ed. New Jersey: Prentice-Hall.

[2] J. M. Fernandes, "Analysing and Modelling IS workflows," I. Oliveira, Universidade de Aveiro, 2016.

[3] D. Rosenberg and M. Stephens, Use Case Driven Object Modeling with UML: Theory and Practics, 1 ed. New York: Apress, 2007.

[4] W. A. Fischer. Jr. and J. W. Jost, "Comparing Structured and Unstructured Methodologies in Firmware Development," *Hewlett-Packard Journal,* vol. 40, pp. 83-85, April 1989.

[5] P. Eeles. (2005, 21 October 2016). *Capturing Architectural Requirements*. Available: http://www.ibm.com/developerworks/rational/library/4706.html#N100A7

[6] K. Wiegers and J. Beatty, *Software Requirements*, 3 ed.: Microsoft Press, 2013.

[7] IEEE, "IEEE Recommended Practice for Software Requirements Specifications," Std. 830-1998, ed. New York, NY: IEEE, 1998.

[8] I. Oliveira, "Descrição dos Casos de Utilização - Plataforma eLearning da Universidade de Aveiro," Universidade de Aveiro, Aveiro, 2014.

[9] K. Fakhroutdinov. (2009, 23 October 2016). *UML Sequence Diagrams*. Available: http://www.uml-diagrams.org/sequence-diagrams.html

[10] D. Bell. (2004, 22 October 2016). *The class diagram - An introduction to structure diagrams in UML 2*. Available: http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/

[11] A. Cruz, N. Costa, R. Ramos, and R. Lopes, "Guião Prático 3 - Modelo de Domínio," Aveiro, 6 October 2016.

[12] D. Garlan and M. Shaw, "An Introduction to Software Architecture," School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1994.

[13] J. M. Fernandes, "Software Architecture," I. Oliveira, Universidade de Aveiro, 2016.

[14] T. A. Pender, *UML Weekend Crash Course*, 1 ed. Indianapolis, IN: Wiley Publishing, Inc., 2002.

[15] Unknown, "Sistema de Informação," in *Análise e Modelação de Sistemas*, Universidade de Aveiro, 1 ed.

[16] D. Burke, "All Circuits are Busy Now: The 1990 AT&T Long Distance Network Collapse," California Polytechnic State University, San Luis Obispo, November, 1995.

[17] S. A. Pardo, *Validation and Verification*: Springer International Publishing, 2016.

[18] J. M. Fernandes, "Testing," I. Oliveira, ed. Universidade de Aveiro, 2016.

[19] J. M. Fernandes, "Behavioural-Driven Development," I. Oliveira, ed. Universidade de Aveiro, 2016.

Apontamentos de Análise e Modelação de Sistemas
1ª edição - dezembro de 2016

Autor: Rui Lopes
Outros recursos: Aulas de Análise e Modelação de Sistemas
Agradecimentos: José Maria Fernandes e Ilídio Oliveira

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas.