Programação Orientada a Objectos 2021/2022

Exercícios

Ficha Nº 6

Exercícios com múltiplas classes
Classes com dependência e interação bidirecional

1. Escreva um programa que simule a existência de peixes num aquário. Deve modelizar cada um destes conceitos (Peixe e Aquário) em classes distintas e aplicar devidamente os conceitos do encapsulamento. Significa isto que não vai ter o aquário a manipular ou alterar o estado interno dos peixes aquário como se estes fossem marionetas (por exemplo, fazer o aquário incrementar o peso do peixe quando este é alimentado – errado: o peixe come; se transforma essa comida em peso, ou se tem um problema de metabolismo e deita tudo fora, o peixe é que sabe, não o aquário).

Acerca dos peixes sabe-se o seguinte:

- Um peixe tem: Nome da espécie (texto), cor (texto), peso (gramas), número de série (inteiro sempre crescente)

- Faz / permite
  - Ser alimentado com uma dada quantidade em gramas de alimento que é somada ao seu peso. Se a soma da quantidade de alimento oferecida ao peixe com o seu peso ultrapassar 50g:
    (i) Em 50% dos casos o peixe reduz o seu peso a 40g (limite de peso – peso inicial de um novo peixe) e gera um novo peixe;
    (ii) Em 50% dos casos o peixe emagrece 50%, não come nada nas 4 vezes seguintes em que o aquário distribuir alimentos e morre na vez seguinte.

    > **Nota importante**: a reação que o peixe tem quando é alimentado (aumentar de peso, gerar um novo peixe, emagrecer ou morrer em certas circunstâncias faz parte de uma funcionalidade do peixe. Não é o aquário que toma essas decisões. Considere que podia haver várias espécies de peixe e de alguma forma a decisão variava em função da espécie. A função do aquário é simplesmente "dizer" ao peixe "toma esta comida, come e faz o que entenderes com ela, eu vou passar ao próximo peixe". A alternativa "toma isto, come. Estás com mais de 50 gramas? Então gera um novo peixe ou emagrece" vai contra os princípios do encapsulamento e é considerada um erro (e dá mais trabalho a fazer). O peixe, em certas circunstâncias morre. O peixe não se tira a ele próprio do aquário – o peixe não faz ideia como é que o aquário guarda os peixes internamente.

  - Obter a descrição textual em string.

- Obedece às seguintes regras
    - A construção dos objetos exige sempre o nome da espécie. A cor é opcional, sendo "cinzento" se nada for especificado. O peso inicial é de 10 g. O número de série é um valor automaticamente atribuído aumentando de um em um a cada novo peixe criado explicitamente (a construção por cópia não aumenta este valor). O primeiro peixe tem o valor 500.
    - Quando são criados, os peixes não estão necessariamente em aquário nenhum. Podem ser postos num aquário à *posteriori*. Quando um peixe é posto no aquário, este assume a sua posse e controlo total, passando a comandar o destino do peixe (poder de vida e morte sobre o ~~objec...~~ peixe).

Acerca dos aquários sabe-se que:

- Um aquário tem
    - Uma quantidade de peixes. Eventualmente muitos.
- Faz / permite
    - Inserir um peixe novo.
    - Verificar se um peixe se encontra no aquário dado o número de série dele.
    - Alimentar os peixes todos com uma determinada quantidade de gramas (a mesma quantidade para todos, um por um).
    - Eliminar os peixes mortos.
- Um aquário obedece às seguintes regras:
    - Os aquários são criados inicialmente sem peixe nenhum.

Observações

- Este exercício introduz o aspeto importante de interação complexa e bidirecional entre duas classes e recomenda-se a quem não vá às aulas que o resolva todas as alíneas na mesma e peça depois uma opinião qualitativa ao docente sobre o código elaborado. A interação bidirecional é a seguinte:
    1. O aquário indica aos peixes que têm mais uma dose de alimento (aquário → peixe).
    2. O peixe indica ao aquário que deseja acrescentar um novo peixe (peixe → aquário).

    Esta situação pressupõe o conhecimento mútuo entre ambos os objetos.
- Deve utilizar uma estratégia gradual, com um passo de cada vez (use um diagrama/esquema incluindo ambas as classes *Peixe* e *Aquário*):
    1. Comece por representar apenas os dados e funções necessários a cada entidade independentemente da outra.
    2. Depois deduza os dados e funções que devem existir para além daqueles diretamente especificados atrás em consequência da existência de interação entre peixes e aquário.

        A título de exemplo, considere a situação de um peixe, quando decide fugir do aquário necessita de saber em que aquário se encontra para que peça que seja removido dele.

Um aspeto novo e importante neste exercício consiste na interação e conhecimento mútuos existentes entre os objetos de aquário e peixes e uma parte significativa do exercício é consiste em descobrir se vão ser necessário dados e funções nas classes que não são mencionadas explicitamente no enunciado: quais, para que efeito, e como são usados.

a) Planeie as classes necessárias a este exercício tendo em atenção o que foi descrito e as observações indicadas.

b) Implemente as classes que planeou na alínea anterior. Coloque cada classe num par de ficheiros .h/.cpp independente ficando o projeto com 5 ficheiros ao todo: um *.h* e um *.cpp* por cada classe e um *.cpp* para a função *main*. Esta organização irá fazer surgir uma situação de inclusão circular *A precisa de B mas B precisa de A* (o aquário tem peixes mas o peixe tem que saber em que aquário se encontra). Esta situação é inevitável dada a interação existente entre peixes e aquário e o seu aparecimento neste exercício é propositado para ver como se resolve. Trata-se de uma situação muito comum que é necessário saber resolver.

> **Nota: a inclusão circular *A.h* inclui *B.h* e *B.h* inclui *A.h* não funciona**. Das duas, uma:
> - Ou ambos os ficheiros vão ser incluídos repetidamente até ao infinito: *A.h* inclui *B.h* que inclui *A.h* (2ª vez) que inclui *B.h* (2ª vez) que inclui *A.h* (3ª vez) etc. → Não funciona, não é aceite.
> - Ou os ficheiros estão protegidos contra inclusão repetida (diretiva *pragma* ou a construção #*ifndef* - #*define* - #*endif*): neste caso, *A.h* inclui *B.h* que já não consegue incluir *A.h* e. portanto. *B.h* não compila porque lhe falta *A.h*. → Não funciona.
>
> A solução passa por um dos ficheiros *A* ou *B* desistir de incluir o outro e mencione apenas a existência da classe que precisa. No caso de apenas precisar de um ponteiro, basta mencionar que a classe (objetos) apontada existe (se precisar de um objeto mesmo, então terá que incluir mesmo o *.h* todo). Se não foi ao laboratório em que este exercício é resolvido, deve averiguar junto do docente como esta situação se resolve e ver as notas introdutórias ou ficheiro adicional no *moodle* sobre esta situação.

c) Escreva uma função *main* que permita a seguinte funcionalidade:

- Adicionar peixes a um aquário mediante dados especificados pelo utilizador
- Listar os peixes que estão no aquário (ver a descrição textual de cada um)
- Alimentar todos os peixes dada uma quantidade de comida especificada pelo utilizador.
- Verificar se um determinado peixe se encontra no aquário dado o seu número de série.
- Remover um peixe dado o seu número de série.

d) A solução para a classe *Aquário* envolve o uso de um vector segundo duas alternativas:

- De objetos de *Peixe*. Esta solução costuma ser a mais intuitiva para quem está a começar, pois evita ponteiros e fica-se logo com os objetos guardados. No entanto, tem várias desvantagens:
  - Apenas é adequado a situações de composição.
  - Não permite o polimorfismo.
  - Vai um pouco contra o enunciado, que diz que o aquário toma posse do peixe, o qual já existia. Usando vetores de objetos, o aquário guarda, quando muito, uma cópia do peixe original, passando a existir dois objetos distintos, o que pode original incoerência de dados mais tarde no programa (depende da situação).
  - O vector muda os objetos de sítio quanto se insere ou apaga um objeto. Isto faz que qualquer ponteiro que estivesse a apontar para um objeto armazenado no vector deixe de ser válido. Faz também com que seja complicado apagar um objeto de um vector a partir de um ciclo que está a percorrer esse mesmo vector (o efeito menos nocivo é o de "saltar" um objeto. Os outros efeitos são ainda piores, tais como corromper ou modificar o objeto errado, ou fazer uma iteração a mais e trabalhar sobre um objeto que já não existe).
  - As operações inserir e remover são menos eficientes em vetores de objetos do que em vetores de ponteiros.

- De ponteiros para objetos de *Peixe*. Esta solução acaba por ser menos considerada por envolver ponteiros, mas é uma solução melhor pelas seguintes razões:
  - Suporta diretamente os casos de agregação uma vez que os objetos apontados residem no exterior do objeto que tem os ponteiros, e também suporta os casos de composição, como é o caso deste exercício (no caso de composição exigirá que se duplique/copie os objetos apontados em operador de atribuição e construtor por cópia, e que se destruam no destrutor).
  - É mais eficiente no tratamento de operações de inserção e de eliminação.
  - Possibilita o polimorfismo (ou qual exige ponteiros ou referências).

Nesta alínea pretende-se confirmar que se entende bem a diferença entre armazenar diretamente os objetos na estrutura de dados (vector ou matriz) e armazenar ponteiros para os objetos na estrutura de dados (vector ou matriz). Confirme que entende a diferença usando diagramas que forem necessários.
Na classe *Aquario*, se tiver usado um *vector de objetos Peixe*, explore a dificuldade acrescida que esta opção envolve usando a funcionalidade sugerida abaixo. Se tiver usado um *vector de ponteiros para objetos Peixe*, passe adiante.

A função *main* deve permitir:

- Apagar um peixe dado o seu número.
- Apagar todos os peixes maiores que um determinado peso (operação feita internamente pelo aquário, para respeitar o encapsulamento).
- Mudar a cor de um peixe dado o seu número.

e) Se não tiver usado um vector de ponteiros na classe *Aquario*, adapte o código que já tinha de forma a ter os objetos *Peixe* fora do objeto *Aquario*, passando a ter um *vector de ponteiros para objetos Peixe* em vez de um *vector de objetos Peixe*. Sabendo que a situação apresentada neste exercício é um caso de composição, garanta que a sua classe *Aquario* tem um comportamento coerente e correto nas seguintes situações:

- Cópia de objetos (parâmetro, retorno de funções, inicialização por cópia)
- Atribuição
- Construção e destruição

f) Na sequência da alínea anterior, sabendo quais são os mecanismos usados nas situações de cópia de objetos e de atribuição, se lhe dissessem que "garanta que não é possível copiar ou atribuir objetos desta classe" qual seria a forma mais rápida de garantir isso? Proponha essa alteração, implemente-a e teste-a tentando atribuir ou copiar objetos de Aquário.

A alteração é simples e rápida de fazer. Se o compilador rejeitar operações de cópia e atribuição, mas tudo o resto se mantiver a funcionar bem (terá que colocar essas instruções de cópia e atribuição em comentários para ver se o resto continua a funcionar bem), então, em princípio, terá acertado. Confirme com o professor.

Na resolução desta alínea coloque as alterações que fez na alínea anterior em comentários para não perder esse código, e depois reponha-o.

Objetivos do exercício

- Introdução aos objetos dinâmicos.
- Treinar a definição de classes em cenários com alguma complexidade e que envolvem várias classes que se utilizam mutuamente.
- Conhecer a situação de visibilidade e comunicação bidirecional entre objetos e lidar com tais situações.
- Conhecer e resolver a situação de inclusão circular e entender as causas que levam a essa situação.
- Consolidar o conhecimento acerca da diferença entre armazenar objetos e ponteiros para objetos (por exemplo, num vector).
- Relembrar e treinar o conceito de classes robustas para operações atribuição, cópia e destruição.

2. Pretende-se ter um programa em C++ que consiga gerir passeios de bicicleta. Este objetivo envolve a manipulação de informação acerca de pessoas e de passeios. O exercício vai focar-se em classes, ficando a função *main* com um papel secundário para efeitos de testes, como habitualmente (interação complexa com utilizador fica para o trabalho prático).

a) Defina a classe *PasseioDeBicicleta*. Esta classe representa um passeio organizado em que participa pelo menos uma pessoa, que será o líder, e um conjunto de participantes adicionais sem limite. Faz também parte do passeio de bicicleta a seguinte informação: o nome da vila onde vai decorrer o passeio e a data (texto). A classe *PasseioDeBicicleta* deve permitir:

- Associar/desassociar um novo participante. A pessoa já existia e até pode participar em várias coisas. A mesma pessoa só pode participar uma vez no passeio.
- Mudar o destino (implica remover automaticamente todos os participantes já existentes exceto o líder).
- Listar os participantes.

Para este exercício defina e use uma classe *Cidadao* simples, com os dados fundamentais tais como nome e BI, com um construtor que recebe esses dados, e funções que permitam atualizar o nome mas não o BI, e consultar o nome e BI (se tiver uma classe "Pessoa" de exercícios anteriores, use essa). Se não tiver e achar que a elaboração de uma classe *Cidadao* com estas características é muito complicada, use simplesmente o código que está abaixo (e agarre-se ao livro quanto antes).

```
// fazer a classe Cidadao e ver este código só se achar que é muito complicado
class Cidadao {
   string nome;
   int bi;
public:
   Cidadao(string n, bi b) : nome(n), bi(b) {}
   void setNome(string n) { nome = n; }
   string getNome() const { return nome; }
   int getBi() const { return bi; }
};
```

b) Teste a sua classe com uma função *main* simples na qual cria alguns objetos *Cidadao* para fornecer a um objecto de *PasseioDeBicicleta* ou dois.

c) As pessoas (objeos de *Cidadao*) existem fora do passeio com a sua vida e existência própria (deve ter reparado que este é um caso de agregação). Não é o passeio de bicicleta que comanda a criação e destruição dos objetos de *Cidadao*. Isto significa que esses objetos têm que ser geridos por alguma outra parte do programa. Até agora tem-se criado alguns objetos na função *main* para efeitos de teste, mas esta solução não é muito organizada. Construa uma classe *ArquivoDeIdentificação* que faz a gestão da vida dos objectos *Cidadao*. O código da classe *ArquivoDeIdentificacao* é o único local onde são criados ou destruídos objetos de *Cidadao*. Deverão existir funções membro para:

- Criar um novo Cidadao. Os dados necessários são passados por parâmetro.
- Obter um cidadão (ponteiro ou referência – não tem sentido ser por cópia – explique porquê) dado o seu BI.

> **Observações**: O texto seguinte retira alguma dificuldade ao exercício, diminuindo assim o seu valor. Se quiser salte o texto seguinte.
>
> Na classe *ArquivoDeIdentificacao* pode usar um vector para armazenar os cidadãos. Se usar um vector de objetos, sempre que insere um novo cidadão vai fazer com que todos os restantes mudem de sítio de memória, fazendo com que os ponteiros existentes na classe *PasseioDeBiciceta* se tornem inválidos (recordar que os vectores mudam os objetos de sítio quando se insere ou apaga – é assim que a sua memória dinâmica interna funciona). Por outras palavras, vai ter que usar um vector de ponteiros para Cidadão se quiser que o seu programa funcione como deve ser (relembrar as observações dadas na alínea d do exercício 1).

d) Acrescente ao seu programa a capacidade de ler e gravar as pessoas (objetos de *Cidadao*) em disco num ficheiro de texto. Está-se a falar das pessoas que existem, não as que que participam num determinado passeio. A capacidade de ler e gravar pessoas deve ficar encapsulada na entidade *ArquivoDeIdentificacao*.
Se não resolveu nenhum dos exercícios desta sequência, construa apenas uma classe *ArquivoDeIdentificacao* com a capacidade de criar/manter/destruir pessoas, armazenadas através de um vector de ponteiros para *Cidadao* (utilize o código fornecido para esta classe) e acrescente ao *ArquivoDeIdentificacao* a capacidade de ler e gravar as pessoas em disco.

e) Considere agora que as pessoas pretendem passear mas apenas num determinado conjunto de vilas e aldeias (representados pelos seus nomes). Cada pessoa tem um conjunto diferente de localidades onde gosta de passear.

> **Representação das preferências de locais de passeio:**
> Poder-se-ia pensar em adicionar estes dados à classe *Cidadao*, mas essa opção, apesar de razoável, iria acrescentar à classe *Cidadao* dados que apenas interessam ao *PasseioDeBicicleta*. Por outro lado, esses dados também não são do passeio de bicicleta pois variam de pessoa para pessoa. Conclui-se que estes dados devem existir na entidade que representa a interação entre uma pessoa (cidadão) e um passeio de bicicleta.

Proponha uma forma de representar estas preferências e reconstrua o esquema de classes (neste momento não precisa de implementar nada). Confirme junto do professor a sua solução.

f) Na sequência do anterior, deve ter deduzido que faria sentido a criação de uma nova entidade (classe) que representa a inscrição de um cidadão num passeio de bicicleta. Trata-se de um pedaço de papel com a lista de destinos preferidos (conjunto sem limite de *strings* – nomes de vilas) e a indicação da pessoa em questão. Construa a classe *FichaDeInscricao* não esquecendo que esta não controla a existência da pessoa que lhe diz respeito. Afinal, trata-se de um mero papel e a pessoa existe fora dele. Esta alínea diz respeito apenas a esta classe isoladamente das restantes.

g) Refaça o exercício do passeio de bicicleta. Agora, o que fica armazenado no passeio são fichas de inscrição e não os ponteiros para pessoas. Quando se associa uma nova pessoa ao passeio, é exigido também a lista de destinos preferidos, que ficam armazenados. O passeio de bicicleta cria um objecto *FichaDeInscrição* com esses dados e armazena-o.

> **Quanto ao armazenamento das fichas de inscrição.**
> *Analise o problema e decida como armazenar as fichas:* ***objectos ou ponteiros para objectos****. Tente decidir por si e* ***depois (e só depois)*** *leia este texto.*
>
> Repare que o passeio pode guardar diretamente os objetos *FichaDeInscrição* em vez de ponteiros (também pode guardar ponteiros mas dá mais trabalho). Se fossem pessoas (objetos de Cidadao) só devia guardar os ponteiros e não os objetos, como visto no caso anterior. Essa diferença deve-se a:
>
> - As pessoas existem fora e independentemente do passei de bicicleta. O passeio de bicicleta não pode ter "pessoas dentro de si" e a vida (duração) desses objetos não pode ser condicionada à vida (duração) do objeto *PasseioDeBicicleta*.
> - Os objetos ficha de inscrição apenas fazem sentido no contexto de um passeio de bicicleta: trata-se de objetos auxiliares às operações internas deste, e não faz sequer sentido que existam no seu exterior. Faz sentido é que a sua existência seja "oculta" no interior do objeto *PasseioDeBicicleta*.

h) Na atualização da classe *PasseioDeBicicleta* que está a fazer deve cumprir o seguinte:

- Quando está a registar um novo participante no passeio, é indicada a lista de destinos preferidos (tal como dito atrás), e a inscrição só é aceite de o destino do passeio coincidir com um dos destinos desejados da pessoa.
- Quando o destino do passeio muda, as pessoas inscritas não são automaticamente removidas. Em vez disso mantêm-se e o passeio transmite essa alteração a cada pessoa já inscrita (mais especificamente, à entidade *FichaDeInscricao*); as pessoas que não desejem o novo destino removem-se desse passeio. A verificação de destino é/não-é desejado deve ser devidamente encapsulada pela entidade que trata dessa questão (obs.: o líder não tem destino desejado). Esta característica exemplifica um caso de interação bidirecional entre objetos de classes diferentes: o objeto *PasseioDeBicicleta* comunica aos objectos *FichaDeInscricao* o novo destino, e estes podem (ou não) comunicar ao *PasseioDeBicicleta* que desejam sair do passeio, tal como no caso Aquário – Peixes do exercício anterior.

> Objetivos do exercício
>
> - Treinar a definição de classes em cenários com alguma complexidade e que envolvem várias classes que se utilizam mutuamente.
> - Consolidar a resolução de situações de inclusão circular e entender as causas que levam a essa situação.
> - Consolidar o conceito de visibilidade e comunicação bidirecional entre objetos.
> - Consolidar o conhecimento acerca da diferença entre armazenar objetos e ponteiros para objetos (por exemplo, num vector).
> - Conhecer e utilizar da classe ifstream para acesso a ficheiros de texto.
> - Relembrar e treinar o conceito de classes robustas para operações atribuição, cópia e destruição.