

**Instituto Politécnico de Viseu  
Escola Superior de Tecnologia e Gestão de Viseu  
Departamento de Informática**

Unidade Curricular: Programação Orientada aos Objetos

Projeto

**Realizado por:**

Gonçalo Abreu- 22996

Leandro Dias- 23028

Viseu, 2022

Índice

[**Introdução** 3](#_Toc122338286)

[**Bibliotecas utilizadas** 4](#_Toc122338287)

[ Fabrica.h 4](#_Toc122338288)

[ Admin.h 5](#_Toc122338289)

[ Motor.h 6](#_Toc122338290)

[ MCombustao.h, MEletrico.h e MInducao.h 7](#_Toc122338291)

[ Normal.h 7](#_Toc122338292)

[ Objetos.h 7](#_Toc122338293)

[ RelogioFabrica.h 8](#_Toc122338294)

[ Sensor.h 8](#_Toc122338295)

[ Sfogo.h, Sluz.h, Smissel.h e Stemperatura.h 9](#_Toc122338296)

[ User.h 9](#_Toc122338297)

[ Uteis.h 9](#_Toc122338298)

[ Visitante.h 9](#_Toc122338299)

[ XML.h 9](#_Toc122338300)

[ LerXML.h 10](#_Toc122338301)

[**Função main** 11](#_Toc122338302)

[**Conclusão** 12](#_Toc122338303)

# **Introdução**

Com o presente trabalho pretende-se colocar em prática todos os conhecimentos adquiridos ao longo do semestre na Unidade Curricular de Programação Orientada aos Objetos (POO).

Para isso, foi-nos proposto um projeto no qual teremos de realizar uma simulação da gestão de uma dada fábrica. A fábrica possui diferentes utilizadores, vários tipos de motores e vários tipos de sensores que permitam manter o bom estado das instalações.

Quanto aos utilizadores, estes possuem diferentes níveis de acesso e responsabilidades.

Os motores possuem diferentes estados que permitam alertar o utilizador sobre o seu funcionamento atual, por exemplo, se se encontram a uma temperatura elevada ou então se há alguma probabilidade de avaria.

Os sensores terão a função de detetar e alertar o utilizador acerca das condições de ambiente e segurança atual da fábrica. Teremos sensores de luz, humidade, fumo e mísseis (que permitirá detetar a aproximação de mísseis da fábrica).

Foi-nos fornecido pelo docente da Unidade Curricular um ficheiro em XML com os dados da fábrica como hora de abertura e fecho, intervalos de temperatura e consumo por hora dos motores, posições (X, Y) dos diferentes componentes da fábrica, ID e marca de cada um dos componentes, etc.

A partir do ficheiro em XML, teremos de criar um programa de simulação da fábrica.

Para isso, seguiremos a metodologia orientada a objetos.

# **Bibliotecas utilizadas**

As bibliotecas são uma maneira simplificada de gerir diferentes funções para diminuir o tamanho do código principal (main).

Criaram-se diferentes bibliotecas para o efeito:

### Fabrica.h

Esta biblioteca tem incluídas todas as outras bibliotecas existentes (como mostra a Fig. 1).

Uma imagem com texto

Descrição gerada automaticamente  
Fig. 1- Bibliotecas incluídas em Fabrica.h

Começou por se criar a classe “Fabrica”.

Definiu-se como privado as funções de listas ordenadas de utilizadores, motores, sensores, objetos, motores quentes, motores avariados e sensores avariados.

Como público, definiu-se as funções:

* Fabrica(User \*ut)- Esta constrói a fábrica, dado o utilizador atual.
* **bool** Load(**const** string &ficheiro)- Como as configurações da fábrica são dadas em ficheiro XML, esta função permite carregar esse ficheiro enquanto o simulador estiver a correr.
* **bool** Add(User \*ut)- Permite adicionar utilizadores à fábrica;
* **bool** Add(Motor \*m)- Permite adicionar motores à fábrica;
* **bool** Add(Sensor \*s)- Permite adicionar sensores à fábrica;
* **void** Listar(ostream &f = std::cout)- Lista o estado atual da fábrica;
* **void** Desligar(**int** id\_motor)- Desliga um motor específico;
* ESTADO\_MOTOR **Get\_ESTADO**(**int** id\_motor)- Permite saber o estado de um motor específico;
* list<Motor \*> Listar\_Tipo(string Tipo, ostream &f)- Lista e devolve todos os motores de um tipo específico;
* **bool** **Manutencao**()- Permite fazer a manutenção dos componentes da fábrica como motores e sensores;
* list<string> Ranking\_Dos\_Fracos()- Lista por ordem crescente de fiabilidade, as marcas de motores que mais avariam. Ou seja, da marca que mais avaria para a que menos varia;
* list<Motor \*> Ranking\_Dos\_Mais\_Trabalhadores()- Lista os motores que mais trabalharam por ordem crescente;
* **void** **Relatorio**(string fich\_xml)- Permite a realização de um relatório em XML do estado da fábrica;
* **int** **Aviso\_Humidade**(list<Motor \*> &lm,**int** x,**int** y,**int** nVizinhanca)- Devolve o número de motores que foram desligados quando o sensor de humidade envia um aviso;
* **int** **Aviso\_Fumo**(list<Motor \*> &lm, string fich\_video)- Desliga todos os motores da fábrica e reproduz um vídeo com todas as saídas desta quando deteta a existência de fumo;
* **int** **Aviso\_Luz**(string fich\_video)- Reproduz um vídeo que deseja um bom regresso a casa quando deteta que a luz está abaixo de um dado valor;
* **void** **Aviso\_Missel**(string fvideo, string festado = "Estado.txt")- Quando deteta um míssil, reproduz um vídeo a mostrar os abrigos e desliga a fábrica por completo, guardando o estado atual desta.
* **Gets**

Estes são funções que procuram acessar a dados que possam ser introduzidos pelo utilizador ou que possam ser acessados no ficheiro XML.

Para esta classe “Fabrica” foram definidos os seguintes gets:

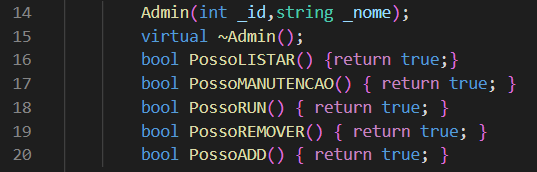
Uma imagem com texto, ecrã

Descrição gerada automaticamente

/\* por terminar\*/

### Admin.h

Esta subclasse “Admin” da classe “User” corresponde ao gestor/ administrador da fábrica. Este tem acesso a tudo e pode adicionar, remover, gerir os motores, realizar a manutenção e listar.



Estas permissões são dadas pelo comando “return true” de cada uma das funções apresentadas, que correspondem às ações referidas.

### Motor.h

Iniciou-se por definir os diferentes estados que os motores podem comportar e a cor das temperaturas associadas a cada um dos motores em que a cor verde indica que o motor se encontra a funcionar bem. O amarelo indica que o motor está a aquecer e deve ser enviado um aviso ao gestor. O vermelho indica que o motor está muito quente, então o gestor deve ser avisado e o motor deve entrar no estado STOP.



De seguida, criou-se a subclasse “Motor” da classe “Objetos”.

Definiu-se como privado o consumo por hora, o estado do motor e a cor do motor.

Encontram-se protegidas a temperatura e as horas de trabalho do motor.

Como públicas há a identificação do motor com o seguinte atributo:



Definiu-se então os estados do motor. Estes apresentarão no ecrã o output que se apresenta entre aspas e os diferentes estados encontram-se desativados (“return false”).

Uma imagem com texto

Descrição gerada automaticamente

Os gets usados foram os seguintes. Estes foram definidos para retornar à função principal as variáveis que se encontram na figura a seguir. Por exemplo, no caso do **int** **getCONSUMO\_HORA**(){**return** CONSUMO\_HORA;}, este lerá o consumo por hora do motor e retornará esse valor à função principal pelo comando “return”.

Uma imagem com texto

Descrição gerada automaticamente

Os sets usados foram os seguintes. Foram definidos os tipos de cada uma das variáveis.

Uma imagem com texto

Descrição gerada automaticamente

### MCombustao.h, MEletrico.h e MInducao.h

Para cada uma destas bibliotecas, o código usado é idêntico, difere apenas no nome.

Iniciou-se por se criar uma subclasse da subclasse “Motor” criada na biblioteca “Motor.h”. Nada foi colocado em privado ou protegido, apenas em público.

Cada motor possui 4 estados: START para iniciar o funcionamento; RUN, que indica que está em funcionamento; STOP para parar o funcionamento; AVARIADO, que indica que está avariado.

Cada um destes estados ou está ativo (ON- True- 1) ou está inativo (OFF- False- 0), logo, as funções correspondentes a cada um destes estados devolverá uma variável do tipo boolean.

Uma imagem com texto

Descrição gerada automaticamente

De seguida, criou-se um método que permite escrever num ficheiro o tipo de motor.

### Normal.h

A subclasse “Normal” da classe “User” refere-se a um utilizador normal.  
Este tem permissões para tudo, exceto remover, pois está a cargo do gestor (ou seja, na função **bool** PossoREMOVER()em vez de retornar *true*, retorna *false*).

Uma imagem com texto

Descrição gerada automaticamente

### Objetos.h

Esta classe engloba todos os objetos da fábrica, os motores e sensores.

Define os atributos dos objetos no geral (cria funções get para retornar esses atributos) como o ID, Marca, posição X e Y, número de avarias que houve em todos os objetos, etc…

Uma imagem com texto

Descrição gerada automaticamente

### RelogioFabrica.h

Esta função será a simulação do relógio da fábrica. Usou-se a biblioteca “time.h” para se concretizar o código.



Criou-se a classe “RelogioFabrica” e declararam-se as variáveis “START” do tipo “time\_t” e “VELOCIDADE” do tipo “int”.

Uma imagem com texto, relógio

Descrição gerada automaticamente

Fez-se uma chamada de função “time\_t GetTime” para acessar ao tempo simulado.

Uma imagem com texto, ecrã, captura de ecrã

Descrição gerada automaticamente

### Sensor.h

Criou-se uma subclasse “Sensor” da classe “Objetos” para se definirem os sensores.

No início definiram-se os 3 estados possíveis dos sensores que podem estar a funcionar (RUN), avariados (AVARIADO) ou sem estado (SEMESTADO).



Os sensores são identificados pelo ID, marca, posição Y, posição X, probabilidade de avaria e valor do aviso.



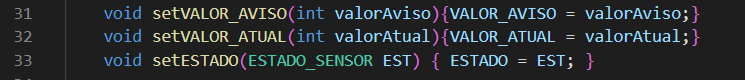
Cada sensor terá uma função “ESTOU\_AVARIADO” que devolve um boolean, para retornar *true* ou *false* em caso de avaria.

Usaram-se gets para pegar no estado, probabilidade de avaria, valor do aviso e o valor atual do sensor.

Uma imagem com texto

Descrição gerada automaticamente

Usaram-se também os sets para definir o valor do aviso, o valor atual e o estado.



### Sfogo.h, Sluz.h, Smissel.h e Stemperatura.h

Para cada uma destas bibliotecas, o código usado é idêntico, difere apenas no nome.

Iniciou-se por se criar uma subclasse da subclasse “Sensor” criada na biblioteca “Sensor.h”. Nada foi colocado em privado ou protegido, apenas em público.

De início há uma identificação do sensor:



De seguida, criou-se um método que permite escrever num ficheiro o tipo de motor.

### User.h

A classe “User” define os utilizadores da fábrica. É esta classe que vai ler o ID e o Nome dos utilizadores e estabelecer as diferentes permissões que estes possam vir a ter. Por exemplo, o administrador terá acesso a todas as funções. O visitante só terá acesso a listar e um utilizador terá acesso a todas as funções, exceto remover.

Uma imagem com texto, ecrã, interior, captura de ecrã

Descrição gerada automaticamente

Em privado foram anunciadas as variáveis ID (do tipo int) e NOME (do tipo string).

### Uteis.h

Nesta biblioteca estarão disponíveis algumas funções úteis para o código como uma função que define a hora atual, uma função que separa strings, ou seja, através de uma string completa, a função separa a string em vetores diferentes, tantos quantos definirmos.

### Visitante.h

Criou-se uma subclasse “Visitante” da classe “User” e colocou-se como público os métodos que permitem a identificação do visitante e o método **virtual** **bool** **PossoLISTAR**() {**return** true;} que dá permissão a que o visitante possa listar (através do “return true”).

Uma imagem com texto

Descrição gerada automaticamente

### XML.h

Nesta biblioteca encontram-se funções uteis para o que tenha a ver com o XML.

### LerXML.h

O LerXML.h é uma biblioteca na qual se fez uma estrutura que conseguisse ler um ficheiro XML e adicionasse os dados aí indicados no programa.

Recorreu-se à biblioteca rapidXML que fornece funções úteis para o efeito.

xml\_node<> \* DEFINICOES\_node = root\_node->first\_node("DEFINICOES"); é uma função que permite criar um nó filho. Neste exemplo foi criado um nó chamado “DEFINICOES\_node”, filho do nó raíz (definido anteriormente no código) e este aparecerá no ficheiro com o nome “DEFINICOES”.



O código apresentado criou um nó para o nome da fábrica, filho do nó “DEFINICOES\_node”. De seguida, utilizou-se o código set\_nomeEmpresa(NOMEFABRICA\_node->value()); . Este código lerá o que se encontra escrito e definirá esse valor.

O restante código é idêntico ao já explicado. Foi-se criando a estrutura do XML, criando os nós necessários. No nó dos motores e sensores, após realizar o código para ler os valores pretendidos, usou-se o comando Sensor \*s= **new** Smissel(**int** \_id,string \_marca,**int** \_posicaoY,**int** \_posicaoX,**int** \_probAvaria,**int** \_valorAviso, **this**); Add(s); (por exemplo). Este exemplo indica a criação de um novo sensor de misseis, após se ler os dados no XML.

# **Função main**

Esta é a função principal. A partir desta será executado o menu principal e, a partir deste, executados os restantes.

# **Conclusão**

Ao longo deste trabalho, tivemos oportunidade de entender melhor os conceitos de Programação Orientada aos Objetos, para além de termos aprendido e melhorado programação em linguagem C++. Permitiu-nos aumentar a nossa capacidade de pesquisa para os casos em que não sabíamos como concretizar um dado código.

Este projeto fez-nos pensar em eventuais dúvidas que poderíamos ter tido e que não nos surgiram de imediato ao aprender a matéria nas aulas teóricas.

Fez também com que começássemos a perceber em que a Programação Orientada aos Objetos se aplica no mundo real e permitiu-nos estudar e trabalhar nesse caso, pelo que nos será de uma mais- valia no futuro.