
Instituto Superior de Engenharia de Coimbra

Engenharia Informática

# Programação Orientada a Objetos

# Trabalho Prático 2021/22

Nuno Santos - 2019110035

Rafael Gil - 2020136741

# Índice

# Introdução

Neste relatório vamos descrever os pormenores em relação ao trabalho prático de Programação Orientada a Objetos.

Este trabalho consiste na criação de um jogo single-player, usando apenas a linguagem c++.

O jogo consiste numa ilha, subdividida por zonas, onde cada zona pode ter um edifício e vários trabalhadores. A ilha tem um sistema de recursos, obtidos através de mecanismos do jogo. O objetivo do jogador é fazer a melhor gestão dos seus recursos e dos seus trabalhadores, de maneira a tentar sobreviver o máximo tempo possível, uma vez que o jogo acaba assim que ficar sem recursos ou sem trabalhadores.

# Classes

O programa é composto por diferentes classes base e as suas respectivas classes derivadas, que representam os vários conceitos que compõem o jogo.

## Classe UI

Esta classe tem a função de servir como interface entre o utilizador e o programa, sendo que todas as instruções que o utilizador indicar ao programa, vão ser "colhidas" por esta classe que por sua vez vai enviar ao programa. O mesmo aplica-se ao retorno do programa para o utilizar, sendo que o programa retorna algo e esta classe é que se encarrega de mostrar esse algo ao utilizador. Esta classe é responsável pelo Jogo.

## Classe Jogo

Esta é a classe responsável pelo funcionamento do jogo, sendo que é ela que vai conter a ilha do jogo e vai gerir os ciclos do jogo. Tem métodos para a inicialização da ilha, que por sua vez vai invocar os métodos respetivos da classe Ilha, passando-lhes a informação necessária ao seu funcionamento.

## Classe Ilha

Esta é a classe responsável por armazenar o tabuleiro de jogo e os recursos. Faz isso ao ter um array bidimensional de ponteiros para conseguir usar as classes derivadas da Zona, o qual é alocado dinamicamente com as dimensões desejadas pelo utilizador, e tem um vetor de ponteiros para Recursos, conseguindo assim ir armazenando e gerindo os recursos ao decorrer do jogo. O tabuleiro e os recursos são da responsabilidade da Ilha. Guarda também a informação sobre o seu tamanho, linhas e colunas, e sobre os dias que se passaram ao decorrer do jogo, e tem ainda uma flag para que indica se ao decorrer do jogo já ocorreu alguma contratação de trabalhadores, o que vai ser útil para verificar se o jogo já terminou, uma vez que o jogo começa com 0 trabalhadores contratados e uma das condições para terminar o jogo é não haver nenhum trabalhador contratado.

Em relação aos métodos desta classe existem vários que retornam os valores dos atributos e depois outros que servem propósitos específicos, como o *criaIlha()* que vai alocar memória para o tabuleiro de jogo e criar um tipo de zona para espaço do array bidimensional

aleatoriamente fazendo um shuffle de um vetor de strings que armazena todos os tipos existentes de zonas e um sorteio de um inteiro para escolher uma posição do vetor e o tipo de zona selecionado vai ser criado, garantindo assim uma grande aleatoriedade na criação das zonas do tabuleiro de jogo. O método *iniciaRecursos()* que vai ser chamado no início do programa para criar os recursos e colocar os seus ponteiros no vetor indicado. O método *mostrallha()* que vai retornar uma representação visual do tabuleiro agradável e fácil de interpretar, mostrando os detalhes de cada zona, como qual é o seu tipo, qual o tipo do edifício nela presente, quais os tipos de trabalhadores nela presentes, até ao quinto primeiro trabalhador a ir para aquela zona, e a quantidade de trabalhadores que se encontra naquela zona. O método *mostraZona()*, que mostra a informação detalhada de uma zona em específico, e o seu método primo *mostraTodasZonas()* que vai chamar o método *mostraZona()* para todas as zonas do tabuleiro. O método executa(), o qual vai validar os comandos introduzidos pelo utilizador e que caso sejam válidos vão desencadear as ações destinadas a cada comando diferente. o método executaFich(), que tem precisamente o mesmo funcionamento do método descrito anteriormente, diferenciando-se apenas na forma como obtém os comandos uma vez que os vai buscar a um ficheiro. Tem um método para fazer o tratamento da zona, invocado para cada zona do tabuleiro e sempre que começar um novo dia, que por sua vez vai chamar os métodos de tratamento dos dados da zona, tais como os trabalhadores e os edifícios. Os métodos para fazer a gestão dos recursos, como o *gastaRecursos()* que vai receber a quantidade e o tipo de recurso que quer gastar, verificando se tem a quantidade de recurso suficiente, e só após confirmar que sim é que vai gastar; e o método *aumentaRecursos()* que vai receber a quantidade e o tipo de recursos que quer aumentar, procedendo a aumentar a quantidade do recurso indicado. O método *game()* vai verificar se as condições para o término do jogo foram cumpridas, sendo estas se o utilizador gastou todos os recursos ou se perdeu todos os seus trabalhadores.

## Classe Zona - Base

Esta classe representa uma quadrícula individual do tabuleiro de jogo. Esta tem o conhecimento de que tipo é, qual o edifício que se encontra nela e quais os trabalhadores que se encontram nela. Cada zona individual pode ter apenas um edifício de cada vez, sendo que não tem qualquer restrição na quantidade de trabalhadores que nela se encontram. Tem ainda uma espécie de *placeholder,* com um limite de 5, que armazena os tipos dos trabalhadores que nela se encontram, puramente por motivos estéticos. É uma classe abstrata.

Esta classe dispõe de um vasto conjunto de métodos, sendo que muitos deles servem apenas para devolver os valores das variáveis, sendo os métodos mais relevantes o *trataTrabalhadores(),* que vai ser chamado sempre que se passar para o novo dia e que vai chamar os métodos do Trabalhador para tratar de cada trabalhador que se encontrar na zona; o método *trataEdificios()*, que vai ser chamado sempre que se passar para o novo dia e que vai chamar os métodos do Edifício para tratar o edifício que se encontra nessa zona; o método *verificaDespedimento(),* que vai verificar se algum dos trabalhadores da zona se despediu e caso se tenha despedido vai eliminar esse trabalhador do vetor que armazena os ponteiros para os trabalhadores. Tem ainda os métodos que têm como função acrescentar edifícios ou trabalhadores à zona. O método *defineTrab()*, recebe como parâmetro uma string, que indica qual é o tipo de trabalhador que vai adicionar, o dia do jogo e um ponteiro para a ilha, para conseguir verificar se é possível gastar a quantidade de recursos necessária à contratação do

trabalhador (que caso seja possível vai gastar os recursos), e tem como função adicionar um trabalhador novo à sua zona. O método *defineEdificio()* serve para "construir" um novo edifício na sua zona, sendo que vai receber uma string que indica o tipo de edifício a construir, um ponteiro para a ilha para conseguir verificar os recursos disponíveis para saber se pode proceder ao consumo destes e que por sua vez vai levar à construção do edifício, e ainda uma flag que indica se foi o comando do "admin" para proceder à construção pretendida sem que seja necessário preocupar-se com os recursos. Tem algumas funções que vão servir para eliminar trabalhadores, sejam todos os trabalhadores daquela zona ou um em específico, tem outro método semelhante para o edifício, que vai apagar o edifício que se encontra na zona e também um método que vai vender o edifício, que vai aumentar uma quantidade respectiva do recurso respectivo à venda de qual seja o edifício presente naquela zona.

Esta classe e todas as classes derivadas vão ter um método destinado à duplicação polimórfica, chamado duplica, e o construtor por cópia, o que nos vai permitir copiar os dados de cada zona, sem que estes sejam partilhados indevidamente. Têm ainda o método *atribiu()* que vai servir de substituto ao operador de atribuição, uma vez que nos deparamos com algumas dificuldades na redefinição do operador de atribuição.

## Classe Montanha – derivada da Zona

Esta classe vai representar o tipo de zona montanha, que vai ter os atributos próprios que correspondem ao aumento de produção que causa às minas, ao aumento da probabilidade de demissão que causa aos trabalhadores e a quantidade de ferro que pode armazenar.

```
class Montanha : public Zona{
public:
    Montanha(string t, int x, int y);
    Montanha(const Montanha& outro);

    void trata(ilha& i) override;
    Montanha& atribui(const Zona& outro);
    Zona * duplica() const override;
    double getAumentoProbDem() const override;
private:
    float probDemissao, aumentoProd, quantFerro;
};
```

Tem dois métodos distintos das outras classes derivadas, sendo eles o método *trata()* e o método *getAumentoProbDem()* . O método *trata()* é chamado sempre que um novo dia começa e é responsável por fazer a produção automática de recursos característica da montanha. Já o método *getAumentoProbDem()* apenas devolve o valor do aumento da probabilidade de demissão que vai causar aos trabalhadores.

Esta zona vai gerar 0.1 unidades de ferro sempre que se passar ao turno seguinte.

## Classe Deserto – derivada da Zona

Esta classe vai representar o tipo de zona deserto e apenas tem um atributo próprio que vai guardar a redução de produtividade que vai causar às minas que estiverem nesta zona.

```
class Deserto : public Zona{
public:
    Deserto(string t, int x, int y) : Zona( t, x , y), prod(0.05){}
    Deserto(const Deserto& outro);

    double obtemRedProd() override;
    Deserto& atribui(const Zona& outro);

    Zona * duplica() const override;
private:
    double prod;
};
```

Esta classe tem um método adicional, chamado *obtemRedProd(),* que vai devolver

o valor da redução de produção que vai causar aos edifícios do tipo mina;

## Classe Floresta – derivada da Zona

Esta classe vai representar a zona do tipo floresta. Os atributos pertencentes a esta classe vão guardar o número de árvores que se encontram nessa zona, o número máximo de árvores que pode haver nessa zona e um contador para os dias, pois a cada dois dias, produz uma árvore extra.

Tem os métodos *defineArvores(),* que vai gerar um numero aleatório entre 20 e 40 para inicializar o numero de árvores da floresta, o método *getNArvores(),* que vai devolver o numero de árvores que existem na floresta e o método *trata()* que vai ser invocado sempre que se passa ao dia seguinte e que vai verificar se existe algum edifício na zona e caso se verifique que é verdade vai começar a decrementar o numero de arvores, caso seja falso vai verificar os dias, e se já tiverem passado dois dias sem ter produzido uma árvore, então vai produzir uma nova árvore. Verifica também a quantidade de lenhadores que existem, produzindo uma quantidade equivalente ao número de trabalhadores que existem nessa zona;  é também neste método que se faz a gestão do contador dos dias, sendo que se se verificar que já passaram dois dias, à

variável que representa os dias vai ser atribuído o valor 0, caso contrário, vai se incrementar essa variável.

```
class Floresta : public Zona{
public:
    Floresta(string t, int x, int y) : Zona( t t, x , y), nArvores(defineArvores()),  nArvores_max(100), dias(0){}
    Floresta(const Floresta& outro);

    int defineArvores();
    int getNArvores() const override;
    void trata(ilha& i) override;
    Floresta& atribui(const Zona& outro);
    Zona * duplica() const override;
private:
    int nArvores, nArvores_max , dias;
};
```

## Classe Pantano – derivada da Zona

Esta classe vai representar a zona do tipo pântano, tendo apenas um atributo próprio que representa um contador para os dias. Após 10 dias, o pântano vai demolir o edifício, caso haja, e despedir os trabalhadores, caso existam.

```
class Pantano : public Zona{
public:
    Pantano(string t, int x, int y);
    Pantano(const Pantano& outro);

    void trata(ilha& i) override;
    Pantano& atribui(const Zona& outro);
    Zona * duplica() const override;
private:
    int dias;
};
```

Dispõe de um método *trata(),* que vai ser invocado sempre que for um novo dia, sendo que verifica quantos dias se passaram desde a última ação; se já se tiverem passado 10 dias, vai verificar se existe algum edifício, e caso exista apago-o,

representando assim a sua demolição, procedendo a apagar todos os trabalhadores que lá se encontrem e a atribuir o valor 0 à variável que representa os dias; caso não se verifique que já passaram 10 dias, simplesmente incrementa a variável dos dias.

### Classe Pastagem – derivada da Zona

Esta classe apenas vai herdar os atributos da classe base, não tendo os seus atributos próprios. A pastagem tem como função receber os trabalhadores recém-contratados e provém aos trabalhadores um sitio para descansar sendo que, desde que o trabalhador se encontre nesta zona, previne qualquer trabalhador de se demitir.

```
class Pastagem : public Zona{
public:
    Pastagem(string t, int x, int y) : Zona( t, x , y){}

    Zona * duplica() const override;
    Pastagem& atribui(const Zona& outro);
};
```

### Classe Zonax – derivada da Zona

Esta classe apenas vai herdar os atributos da classe base, não tendo os seus atributos próprios. A função desta classe é aumentar a produção de qualquer edifico em 10%.

```
class ZonaX : public Zona{
public:
    ZonaX(string t, int x, int l);

    Zona * duplica() const override;
    ZonaX& atribui(const Zona& outro);

};
```

### Classe Recursos - Base

Esta classe representa um recurso do jogo e tem conhecimento da quantidade de recursos que tem e de que tipo é.

```
class Recursos {
public:
    Recursos(string t, int q = 5);
    virtual ~Recursos() = default;

    virtual double vende(double quant);
    virtual int producao();
    virtual double producaoBAF();
    virtual double producaoBAC();
    void aumenta(double quant);
    bool gasta(int quant);
    string obtemTipo();
    double obtemQuantidade() const;
    virtual Recursos* duplica() = 0;

private:
    string tipo;
    double quantidade;
};
```

Esta classe dispõe de um vasto conjunto de métodos, sendo que muitos deles servem apenas para devolver valores das variáveis, sendo os métodos mais relevantes o método vende(), que permite a venda dos recursos, o método aumenta(), permite aumentar a quantidade de um determinado recurso, o método gasta(), que vai retirar uma quantidade opcional à quantidade de um recurso e o método duplica() que é destinado à duplicação polimórfica.

### Classe Ferro - derivada da Recurso

Esta classe representa o recurso ferro, sendo o seu único atributo o preço de venda deste recurso.

Tem o método vende(), que vai retornar o valor a que o ferro é vendido.

```
class Ferro: public Recursos{
public:
    Ferro();

    double vende(double quant) override;
    Recursos * duplica() override;

private:
    int custoVenda;
};
```

### Classe BarraAco - derivada da Recursos

Esta classe representa o recurso barra de aço, em que os seus atributos são o preço de venda deste recurso, custo do ferro e o custo do carvão pois pode-se obter este recurso através da transformação de ferro e carvão.

```
class BarraAco: public Recursos{
public:
    BarraAco();

    double vende(double quant) override;
    double producaoBAC() override;
    double producaoBAF() override;
    Recursos * duplica() override;
private:
    double custoFerro, custoCarvao;
    int custoVenda;
};
```

Como é possível obter barras de aço através da transformação de ferro e carvão, vamos precisar dos seguintes métodos, o método vende(), que vai retornar o valor a que a barra de aço é vendida, o método producaoBAC(), que vai retornar a quantidade necessária de carvão para se transformar numa barra de aço e por fim tem o método producaoBAF(), que vai retornar a quantidade necessária de ferro para se transformar numa barra de aço.

### Classe Carvao - derivada da Recursos

Esta classe representa o recurso carvão, os seus atributos são o preço de venda deste recurso e a quantidade necessária de madeira para a transformação em carvão.

```
class Carvao: public Recursos{
public:
    Carvao();

    double vende(double quant) override;
    int producao() override;
    Recursos * duplica() override;
private:
    int custoVenda, custoProducao;
};
```

Como é possível obter carvão através de madeira, vamos precisar dos seguintes métodos, o método vende(), que vai retornar o valor a que a barra de aço é vendida e o método producao(), que vai retornar a quantidade necessária de madeira para se transformar em carvão.

### Classe Madeira - derivada da Recursos

Esta classe representa o recurso madeira, o seu único atributo é o preço de venda deste recurso.

```
class Madeira: public Recursos{
public:
    Madeira();

    double vende(double quant) override;
    Recursos * duplica() override;

private:
    int custoVenda;
};
```

Tem o método vende(), que vai retornar o valor a que a madeira é vendida.

### Classe VigasMadeira - derivada de Recursos

Esta classe representa o recurso Vigas de madeira, os seus atributos são o preço de venda deste recurso e a quantidade necessária de serração para a transformação em madeira.

```
class VigasMadeira: public Recursos{
public:
    VigasMadeira();

    double vende(double quant) override;
    int producao() override;

    Recursos * duplica() override;
private:
    int custoProducao, custoVenda;
};
```

Como é possível obter as vigas de madeira através de serração, vamos precisar dos seguintes métodos: o método vende(), que vai retornar o valor a que vigas de madeira são vendidas e o método producao(), que vai retornar a quantidade necessária de

serração para se transformar madeira em vigas de madeira.

### Classe Eletricidade - derivada da Recursos

Esta classe representa o recurso Eletricidade, os seus atributos são o preço de venda deste recurso e a quantidade necessária de carvão para a transformação em Eletricidade.

```
class Eletricidade: public Recursos{
public:
    Eletricidade();

    double vende(double quant) override;
    int producao() override;

    Recursos * duplica() override;
private:
    int custoProducao;
    double custoVenda;
};
```

Como é possível obter Eletricidade através da queima de carvão, vamos precisar dos seguintes métodos, o método vende(), que vai retornar o valor a que a Eletricidade é vendida e o método producao(), que vai retornar a quantidade necessária a queimar de carvão para se transformar em Eletricidade.

### Classe Edificio - Base

Esta classe representa um edifício, tem conhecimento da ilha onde está introduzida, os seus atributos são o preço de venda deste edifício, o tipo de edifício, o seu nível, a posição da zona onde está e o seu estado (ligado ou desligado).

```
class Edificio {
public:
    Edificio(ilha* i, string t, int c, int x, int y);
    virtual ~Edificio() = default;

    void ligaDesliga();
    bool procuraTrabalhador(string str) const;
    string obtemTipo();
    int getonoff() const;
    void vende();
    virtual void produz();
    bool gastaRecursos(string t, double quant);
    void aumentaRecursos(string t, double quant);
    bool verificaLaterais(string t);
    int obtemCustoDinheiro() const;
    virtual int obtemCustoSubs();
    int getNivel() const;
    void incrementaNivel();
    virtual void melhora();
    virtual Edificio* duplica() const = 0;
    bool MNT();
    virtual int desaba();
    int obtemQuantTrab();
    bool ZNX();
    bool DSR();

    Edificio& operator=(const Edificio& outro);
private:
    string tipo;
    int onoff, custoDinheiro, nivel;
    ilha* i;
    int x, y;
};
```

Esta classe dispõe de um vasto conjunto de métodos, sendo que muitos deles servem apenas para devolver os valores das variáveis, sendo os métodos mais relevantes o método procuraTrabalhador(), que vai pedir à ilha para procurar um trabalhador numa zona indicada retornando um booleano, o método vende(), que vai retornar o valor de custo do tipo de edifício, o método gastaRecursos(), que vai perguntar á ilha se pode gastar aqueles recursos invocando a função gastaRecursos() da ilha que irá retornar um booleano, o método aumentaRecursos(), vai dar á ilha determinados recursos e a ilha simplesmente aumenta os recursos consoante a quantidade que for dada, o método verificaLaterais(), que vai pedir à ilha para verificar se numa dada posição existe ou não aquele edifício, invocando a função verificaLaterais() da ilha que irá retornar um booleano, o método melhora(), é um método virtual que vai aumentar o nível de cada edifício e o método duplica(), que vai fazer um duplicado dela mesma de maneira a que não corrompa os seus próprios dados.

## Classe minaFerro - derivada de Edifício

Esta classe representa o edifício do tipo Mina de Ferro, o qual é responsável pela produção do recurso do tipo ferro. Tem como atributos o seu custo de substituição, uma vez que este edifício pode ser construído recorrendo a diferentes tipos de recursos, a quantidade que produz, o custo em dinheiro do upgrade, o custo dos recursos do upgrade, a quantidade de ferro que consegue armazenar e a probabilidade que este edifício tem em desabar.

Quanto aos métodos que tem, destaca-se o método *produz()* , que é chamado sempre que ser der início a um novo dia, e este método verifica se existe algum trabalhador do tipo mineiro na zona em que se encontra, e se houver vai proceder a fazer as verificações necessárias a que consiga produzir ferro da maneira indicada, ou seja, vai verificar qual é o tipo de zona em que se encontra, uma vez que se se encontrar numa montanha a sua produção vai ser aumentada em 100% ou se estiver num deserto a sua produção vai ser diminuída em 50% e temos ainda o caso da Zona-X, que vai aumentar a produção em 10%; caso não se encontre em alguma das zonas que influencia a produção então vai proceder a produzir de maneira natural. Tem também o método *desaba()* que vai gerar um número aleatório, que caso seja mais baixo que a probabilidade de o edifício desabar vai dar a indicação à zona. Tem ainda o método *melhora()* que vai proceder a verificar se existem recursos suficientes para que o edificio seja melhorado, e caso haja vai aumentar os atributos devidos.

```
class MinaFerro: public Edificio{
public:
    MinaFerro(ilha* i, int x, int y);

    void melhora() override;
    int desaba() override;
    int obtemCustoSubs() override;
    void produz() override;
    MinaFerro& operator=(const Edificio& outro);
    Edificio * duplica() const override;
private:
    int custoSubs, quantProd, upgradeDinheiro, upgradeRecurso, quantArmazenamento, probDesabar;

};
```

## Classe minaCarvao - derivada de Edifício

Esta classe representa o edifício do tipo mina de carvão, o qual é responsável pela produção do recurso carvão. Tem como atributos o seu custo de substituição, uma vez que este edifício pode ser construído recorrendo a diferentes tipos de recursos, a quantidade que produz, o custo em dinheiro do upgrade, o custo dos recursos do upgrade, a quantidade de ferro que consegue armazenar e a probabilidade que este edifício tem em desabar.

Em relação aos métodos, esta classe tem exatamente os mesmo métodos que a classe minaFerro tem, uma vez que ambas têm o mesmo comportamento com a única diferença que esta produz carvão em vez de ferro.

```
class MinaCarvao : public Edificio{
public:
    MinaCarvao(ilha* i, int x, int y);
    MinaCarvao(const MinaCarvao& outro);

    void melhora() override;
    int desaba() override;
    int obtemCustoSubs() override;
    void produz() override;
    Edificio * duplica() const override;

    MinaCarvao& operator=(const Edificio& outro);
private:
    int custoSubs, quantProd, upgradeDinheiro, upgradeRecurso, probDesabar, quantArmazenamento;
};
```

## Classe bateria - derivada de Edifício

Esta classe representa um edifício do tipo bateria, que vai ser responsável por armazenar a eletricidade que é produzida pela central elétrica. Esta classe tem como atributos a quantidade de eletricidade que consegue armazenar e o custo em dinheiro do upgrade.

Tem como métodos, para além do *duplica()* que serve para proceder à duplicação polimórfica, o *melhora(),* que à semelhança dos que foram previamente mencionados, vai verificar se há recursos suficientes para proceder à melhoria do edifício e caso haja vai melhorar, aumentar as suas características.

```
class Bateria : public Edificio{
public:
    Bateria(ilha* i);
    Bateria(const Bateria& outro);

    void melhora() override;
    Bateria& operator=(const Edificio& outro);
    Edificio * duplica() const override;

private:
    int quantEletricidade, upgradeDinheiro;
};
```

## Classe CentralEletrica - derivada de Edifício

Esta classe vai representar um edifício do tipo central elétrica, que vai ser o responsável por produzir eletricidade. Tem como atributos a quantidade de carvão que é necessário gastar para produzir eletricidade.

Tem como métodos o *produz(),* que vai ser chamado sempre que se iniciar um novo dia e que vai ser responsável por verificar se as condições para proceder à produção estão cumpridas, sendo estas condições: ter uma bateria e uma floresta numa das suas arestas e ter carvão suficiente para gastar, sendo estas verificações feitas através de métodos da classe ilha, da qual a classe Edifício guarda um ponteiro; feitas estas verificações e caso se verifiquem verdadeiras então vai produzir eletricidade.

```
class CentralEletrica : public Edificio{
public:
    CentralEletrica(ilha* i, int x, int y);

    void produz() override;
    CentralEletrica& operator=(const CentralEletrica& outro);
    Edificio * duplica() const override;
private:
    int quantCarvao;
};
```

## Classe Edificio-X - derivada de Edifício

Esta classe vai representar um edifício do tipo edifico-x, que vai ter a funcionalidade de produzir uma quantidade de todos os recursos se tiver os três tipos de trabalhadores presentes na mesma zona. Não tem nenhum atributo próprio.

Tem o método *produz()* que vai proceder à verificação dos trabalhadores que se encontram na mesma zona que o edifício, sendo que precisa de pelo menos um trabalhador de cada tipo para ser capaz de produzir; se esta condição se verificar, então vai produzir 5 unidades de todos os recursos.

```
class Edificiox : public Edificio{
public:

    Edificiox(ilha *i, int i1, int i2);
    void produz()override;

   Edificio * duplica() const override;

};
```

## Classe fundicao - derivada de Edifício

Esta classe representa um edifício do tipo fundição, que é responsável pela produção de barras de aço.

Tem o método *produz()* que vai verificar as condições para a produção de barras, que são a existência de uma dada quantidade de ferro e de carvão; caso se verifique que estas condições podem ser cumpridas, então vai proceder à produção de uma barra de aço.

```
class Fundicao : public Edificio{
public:
    Fundicao(ilha* i, int x, int y);

    void produz() override;
    Edificio * duplica() const override;
};
```

## Classe serração - derivada de Edifício

Esta classe representa um edifício do tipo serração, que é responsável pela produção de vigas de madeira.

Tem o método *produz()* que vai verificar as condições para a produção de vigas, que são a existência de uma dada quantidade de madeira; caso se verifique que estas condições podem ser cumpridas, então vai proceder à produção de uma viga.

```
class Serracao : public Edificio{
public:
    Serracao(ilha* i);

    void produz() override;
    Edificio * duplica() const override;
};
```

## Classe Trabalhador - Base

Esta classe representa os trabalhadores, apresentando um conjunto de atributos para a distinção deste: qual o tipo de trabalhador que é, o custo que contratação deste trabalhador, a probabilidade de este trabalhador se despedir, qual o dia de jogo em que se encontra e qual o dia de jogo em que foi contratado, o seu ID e uma flag que indica se ele já se movimentou ou não. Tem ainda um ponteiro para a zona, pois este tem o conhecimento da zona em que se encontra.

Tem um grupo de métodos, sendo que muitos deles servem para obter informação sobre o trabalhador, mas tem outros métodos que vão desempenhar funções importantes como o *pedeDemissao(),* que vai verificar se o trabalhador pediu a demissão ou não, o *redefineZona(),* que tem a função de redefinir o ponteiro da zona em que encontra, sendo este método preciso devido ao trabalhador poder movimentar entre as zonas, e um conjunto de métodos que vão verificar o tipo de zona em que se encontram pois há zonas que influenciam o comportamento dos trabalhadores. Todas as classes derivadas vão ter o método *duplica()* que tem como função proceder à duplicação polimórfica.

```
class Trabalhador{
public:
    Trabalhador(string t, int c, double p, int dias, int d, Zona* z);
    Trabalhador(string t, int c, double p, int dias, int d, Zona* z, int i, int m);
    Trabalhador(const Trabalhador& outro);
    virtual ~Trabalhador() = default;
    int ID() const;//obtem id_trab
    string obtemTipo();
    string obtemID() const;
    int obtemDiasSim() const;//obtem dias
    virtual int pedeDemissao();//ver se pede demissao
    void aumentaDias();//incrementa dias
    virtual int vidaBoa();//ver se o lenhador está no dia de descanso
    int obtemCusto() const;
    double obtemProb() const;
    virtual int obtemDescanso();//obtem descanso do lenhador
    int obtemDID();//obtem d
    Trabalhador& operator=(const Trabalhador& outro);
    bool previneDesp();//verificar se impede o despedimento devido a estar numa pastagem
    void movimenta();
    int obtemMovim() const;
    void redefineZona(Zona* z_a);
    virtual Trabalhador* duplica() const = 0;
    double aumentoProbDem();
    bool isMNT();
private:
    string tipo;
    int custo; //Custo do trabalhador
    double probEmbora; //Probabilidade do trabalhador ir embora
    int dias, d; //Dias da simulacao / dia do jogo em que o trabalhador foi contratado
    int id_trab; //id do trabalhador
    static int id;
```

## Classe Lenhador - derivada de Trabalhador

Esta classe representa o trabalhador do tipo lenhador. Tem como atributo uma flag que serve para verificar se é o dia de ele obter o seu descanso ou não, uma vez que este tipo de trabalhador previne o seu despedimento ao ter dias de descanso em que não produz.

Tem como métodos o *vidaBoa(),* que vai verificar se é o dia de descanso dele ou não, que caso for vai retornar essa indicação e vai repor a flag e caso não seja vai incrementar a flag e retornar a indicação que não é o dia de descanso. Tem também o método *obtemDemissao(),* que vai verificar se este trabalhador se encontra numa montanha, pois essa é a única maneira de ele pedir demissão, e caso esteja vai prosseguir a fazer os cálculos para verificar se pede demissão ou não.

```
class Lenhador: public Trabalhador{
public:
    Lenhador(int dia, Zona* z);
    Lenhador(int a, int c, double d, int e, int f, int g, Zona *z, int m);
    Lenhador(const Lenhador& outro);

    int vidaBoa() override;
    int obtemDescanso() override;
    int pedeDemissao() override;
    Trabalhador * duplica() const override;


    Lenhador& operator=(const Trabalhador& outro);
private:
    int descanso;

};
```

## Classe Operario - derivada de Trabalhador

Esta classe vai representar um trabalhador do tipo operário.

Tem como método o *pedeDemissao(),* que vai verificar se já se passaram 10 dias desde a sua contratação pois é essa a condição que tem para poder pedir demissão; caso já tenham passados 10 dias desde a sua demissão, vai proceder a fazer os cálculos necessários para verificar se vai pedir demissão ou não.

```
class Operario: public Trabalhador{
public:
    Operario(int dia, Zona* z);
    Operario(int a, int c, double d, int e, int f, Zona *z, int m);

    Operario& operator=(const Operario& outro);
    int pedeDemissao() override;
    Trabalhador * duplica() const override;
};
```

## Classe Mineiro - derivada de Trabalhador

Esta classe vai representar o trabalhador do tipo mineiro.

Tem como método o *pedeDemissao(),* que vai verificar se já se passaram 2 dias desde a sua contratação pois é essa a condição que tem para poder pedir demissão; caso já tenham passados 2 dias desde a sua demissão, vai proceder a fazer os cálculos necessários para verificar se vai pedir demissão ou não.

```
class Mineiro: public Trabalhador{
public:
    Mineiro(int dia, Zona* z);
    Mineiro(int a, int c, double d, int e, int f, Zona *z, int m);

    int pedeDemissao() override;
    Trabalhador * duplica() const override;
};
```