Trabalho Prático

Projeto: *Simulador de habitação*

LEI – 2023/24
Unidade Curricular: POO

Docente: Prof. João Durães
Autor: Pedro Pereira – a2021130905
Fábio Pereira – a2021129697

ISEC ENGENHARIA

CLion

Índice

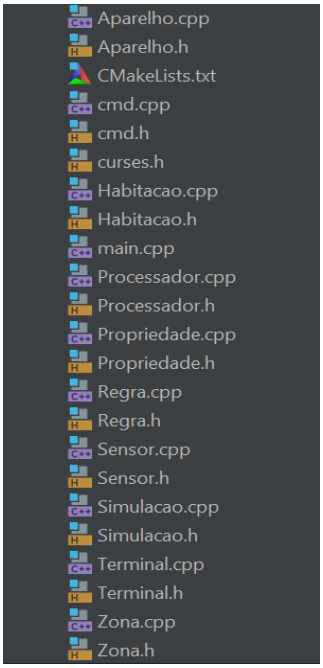
1. Introdução 3
2. Desenvolvimento 4
2.1 Estrutura do projeto 4
2.2 Classes 4
2.2.1 Simulacao 4
2.2.2 Habitacao 5
2.2.3 Zona 6
2.2.4 Regra 7
2.2.5 Sensor 8
2.2.6 Processador 8
2.2.7 Propriedade 10
2.2.8 Aparelho 10
2.3 Requisitos 11
2.4 Ficheiros 12
3. Conclusão 13
4. Bibliografia 13

1. Introdução

Este trabalho tem como objetivo a criação de um simulador de habitação controlada por componentes de domótica interligados entre si, desenvolvido na linguagem de programação C++ e usando o paradigma de programação orientada a objetos (POO).

2. Desenvolvimento

2.1 Estrutura do projeto



O programa está organizado em vários ficheiros de **código fonte (.cpp)** e os seus respetivos **"*headers*"**, incluindo os necessários para usar a **biblioteca ncurses** para que seja possível ter um aspeto gráfico melhorado. Os ficheiros principais para o funcionamento do programa são **main.cpp** (onde o programa lida com o texto introduzido pelo utilizador e faz a leitura de um ficheiro de texto com comandos ou envia o que "leu" para ser processado no **cmd.cpp**), **cmd.cpp** (onde o comando lida com todos os comandos exceto **'exec'**), **Terminal.cpp** e **curses.h**.

Para além desses, existem ainda vários ficheiros onde estão as classes necessárias para cada componente que faz parte do ambiente de simulação da habitação.

2.2 Classes

2.2.1 Simulacao

As classes são fundamentais para o funcionamento deste programa usando o paradigma de programação orientada a objetos. Cada componente da habitação tem a sua classe, sendo que se interligam entre si.

A classe que coordena o funcionamento do programa é a **classe Simulacao** que é responsável pela gestão da Habitação, do ID para as zonas criadas na habitação e do vetor onde ficam os processadores guardados tal como os métodos para gerir esses mesmos processadores.

```
class Simulacao {
private:
    Habitacao* habitacao;
    int idZona;

    std::vector<Processador> processadoresGuardados;

public:
    Simulacao();
    ~Simulacao();
    int getIdZona() const;
    void incrementaIdZona();
    Habitacao* getHabitacao() const;
    void setHabitacao(Habitacao* habitacao);
    int getNumProcessadoresGuardados() const;
    Processador getProcessadorGuardado(int indice) const;
    int guardaProcessador(Processador processador , const std::string nome);
    int removeProcGuardado(const std::string nome);
};
```

Figura 1 - Classe Simulacao

2.2.2 Habitacao

A **classe Habitacao** gere as zonas e tudo a que lhes pertence, sendo por isso nesta classe onde está a matriz de ponteiros necessária para controlar aceder às zonas criadas e tendo por isso certa ligação com a **classe Zona**. É através dos métodos desta classe que se adiciona **(adicionaZona)** e remove **(removeZona)** zonas na habitação que pertence à classe Simulador.

```
class Habitacao {
private:
    int linhas;
    int colunas;
    Zona*** matrizZonas;

public:
    Zona* getZona(int linha, int coluna) const;
    int getLinhas() const;
    int getColunas() const;
    Habitacao(int linhas, int colunas);
    void adicionaZona(const Zona& zona, int linha, int coluna);
    void removeZona(int linha, int coluna);
    ~Habitacao();
};
```

Figura 2 - Classe Habitacao

2.2.3 Zona

A **classe Zona** é essencial para o bom funcionamento do programa pois é nela que é armazenada toda a informação relacionada a cada uma das zonas da habitação. É aqui que estão os vetores das propriedades, sensores, processadores e aparelhos, sendo que os três últimos são vetores de ponteiros para objetos das classes **Sensor**, **Processador** e **Aparelho** e o primeiro é um vetor de objetos da classe **Propriedade**. É também através dos métodos desta classe que, por exemplo, são feitas as "medições" das propriedades pelos sensores ou que é feita a reposição de um processador que tinha sido guardado em memória (**substituiProc**).

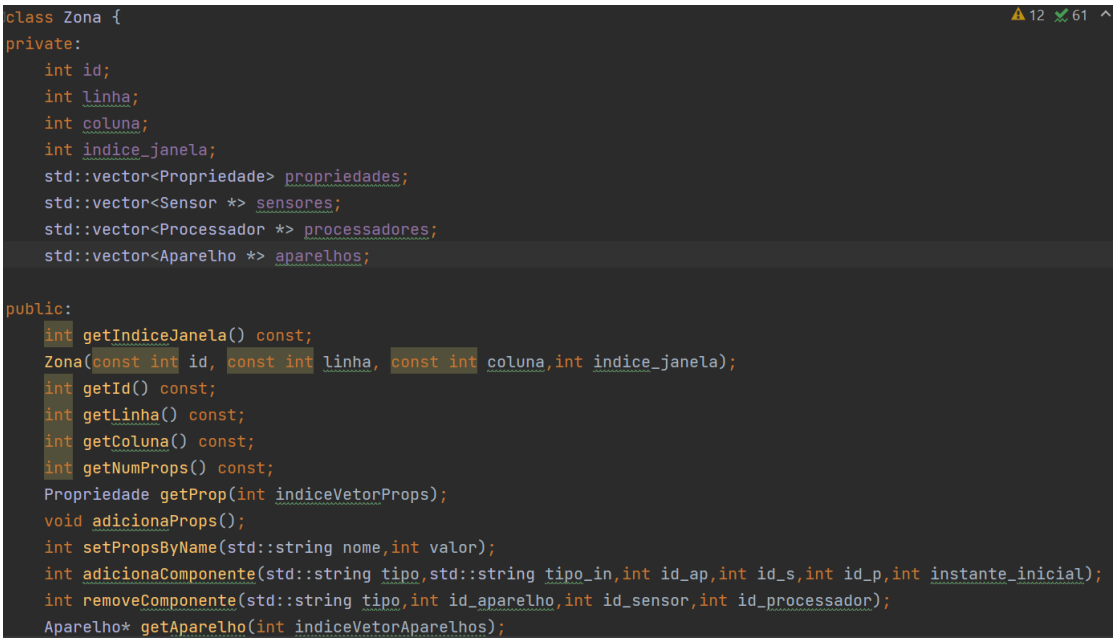


Figura 3 – Classe Zona

Figura 4 – Classe Zona (2)

2.2.4 Regra

A **classe Regra** serve para que os processadores de regra possam avaliar as propriedades presentes em cada zona consoante as regras para depois realizar os comandos necessários. Esta classe retorna um valor verdadeiro caso todas as regras e condições (uma dessas condições é o comando definido aquando da criação do processador) sejam cumpridas, caso contrário devolve um valor falso e o processador não realiza qualquer ação. A regra apenas tem um sensor associado ao qual vai verificar se o valor medido por este está de acordo com a sua restrição (da regra) para depois ela própria ser avaliada pelo processador.

```
class Sensor;

class Regra {
private:
    Sensor * sensorAssociado;
    int id_regra;
    int x;
    int y;
    int valor_avaliar;
    std::string tipo;
    bool var_regra=false;
public:
    Sensor *getSensorAssociado() const;
    Regra(std::string tipo,int id ,int x, int y,int valor_avaliar,Sensor* sensor);
    void setVar_Regra(bool valor);
    bool getVar_Regra() const;
    int getValor_avaliar();
    int getID();
    std::string getTipo();
    void setValor_avaliar(int valor);
    void avaliar();
    int getX() const;
    int getY() const;
```

Figura 5 – Classe Regra

2.2.5 Sensor

A **classe Sensor** é onde está a classe base e as suas derivadas (uma para cada propriedade presente nas zonas). A função destas classes é obter os valores das propriedades presentes nas zonas para que as regras possas avaliá-las e desse modo "reportar" ao processador de regras correspondente. Cada sensor tem um ID único e apenas mede uma propriedade específica de uma zona apenas.

```
class Regra;

class Sensor {
    std::string prop_observada;
    std::string letra_visualizacao;
    int id;
    int valor_medido;

public:
    Sensor(std::string prop_observada,std::string letra_visualizacao,int id);
    const std::string &getPropObservada() const;
    const std::string &getLetraVisualizacao() const;
    int getValorMedido() const;
    void setValorMedido(int valor);
    int getId()const;
    virtual void get_propriedade(std::vector<Propriedade> propriedades);
};
```

Figura 6 – Classe Sensor

```
class sTemperatura : public Sensor{
public:
    sTemperatura(int id);
    void get_propriedade (std::vector<Propriedade> propriedades) override;
};
```

Figura 7 – Exemplo de classe (sTemperatura) derivada da classe Sensor

2.2.6 Processador

A **classe Processador** representa um componente capaz de avaliar um conjunto de regras lógicas sobre um determinado valor. Esta classe percorre um vetor de todas as regras para verificar se todas elas devolvem um valor verdadeiro como foi referido anteriormente. Cada Processador tem a sua própria lista de regras a avaliar (vetor de objetos do tipo Regra), lista de

aparelhos (vetor de objetos do tipo Aparelho), um identificador único, um comando, um identificador da zona associada e um nome (que apenas é utilizado para gerir os processadores a guardar, para os repor e também evitar guardar repetidos).

```
class Processador {                                                                                    ⚠ 6  ✓ 66
private:
    std::vector<Regra> regras;
    std::vector<Aparelho> aparelhos;
    int id_processador;
    std::string comando;
    int idZonaAssociada;
    std::string nome;

public:
    Processador(std::string comando,int id, int idZona);
    Processador(int id,std::string comando,int zona_associada,std::vector<Regra> regras,std::vector<Aparelho> aparelh
    bool mandar_comando(int instante,int mandar_comando);
    int getId_Proc();
    std::string getComando();
    int getIdZonaAssociada();
    std::vector<Regra> getVetorRegras() const;
    std::vector<Aparelho> getVetorAparelhos() const;
    void setVetorRegras (const std::vector<Regra>& novoVetorRegras);
    void setVetorAparelhos (const std::vector<Aparelho>& novoVetorAparelhos);
    int adicionaRegra(std::string tipo,int id,int x, int y,int valor_avaliar,Sensor* sensor);
    int removeRegra(int id_regra);
    int getNumRegras()const;
```

Figura 8 – Classe Processador

```
    void setComando(const std::string &comando);
    void setNome(const std::string &nome);
    const std::string getNome();
    void atualizaRegra();
    std::string lista_regras();
    int adicionaAparelho(std::string letra,bool ligado,int id,int instante_inicial);
    int removeAparelho(int id);
    int getNumAparelhos()const;
    Aparelho getAparelho(int indiceVetorAparelhos);
    Regra getRegra(int indiceVetorRegras);
    ~Processador();
};
```

Figura 9 – Classe Processador (2)

2.2.7 Propriedade

A **classe Propriedade** é a responsável por representar todas as variáveis reais de um ambiente, como por exemplo a temperatura e o ruído de cada zona sendo que de maneira genérica todas têm um valor atual, máximo e mínimo. O atributo chave de cada Propriedade funciona como um id único que serve para identificar o tipo de propriedade. No contexto do programa todas as propriedades são inicializadas a zero aquando da criação das zonas apenas numa questão de simplificar e não estar a atribuir valores aleatórios a cada propriedade e que poderiam nem fazer sentido com o observado no contexto real. De qualquer maneira é possível ao utilizador de definir os valores que pretender para cada propriedade após criar a zona.

```
class Propriedade {
    std::string chave;
    int valor;
    int valor_min;
    int valor_max;
public:
    Propriedade(std::string chave,int valor,int valor_min,int valor_max);
    const std::string &getChave() const;
    int getValor() const;
    void setValor(int valor);
    int getValor_min() const;
    int getValor_max() const;
};
```

Figura 10 – Classe Propriedade

2.2.8 Aparelho

A **classe Aparelho** é onde está a classe base (para um aparelho) e as suas derivadas (uma para cada tipo de aparelho diferente). A função destas classes é obter os valores das propriedades presentes nas zonas para que as regras possas avaliá-las e desse modo "reportar" ao processador de regras correspondente. Cada aparelho tem um ID único composto por um número e uma letra (que depende do tipo de aparelho) e um estado (ligado/desligado). Os restantes atributos são referentes aos instantes do

programa de maneira a registar quando o aparelho mudou de estado e os valores a incrementar.

```
class Aparelho {
    std::string letra;
    bool ligado;
    int id;
    int instante_inicial;
    int instante_ligou;
    int instante_desligou;
    int ultimo_incremento=0;
public:
    Aparelho(std::string letra, bool ligado,int id,int instante_inicial);
    const std::string &getLetra() const;
    bool isLigado() const;
    void setLigado(bool ligado);
    int getId()const;
    void setLetra(const std::string &letra);
    int getInstante()const;
    int getInstante_ligou() const;
    void setInstante_ligou(int valor);
    int getInstante_desligou() const;
    void setInstante_desligou(int valor);
    int getUltimo_Incremento() const;
    void setUltimo_Incremento(int valor);
};
```

Figura 11 – Classe Aparelho

```
class Aquecedor : public Aparelho {
public:
    Aquecedor(bool ligado, int id, int instante_inicial);
};
```

Figura 12 – Exemplo de classe (Aquecedor) derivada da classe Aparelho

2.3 Requisitos

Todos os requisitos foram totalmente implementados exceto a mudança de letra no identificador de cada aparelho consoante o seu estado (ligado/desligado). Neste caso, quando o estado muda através do processador (dependendo do que foi avaliado pelas regras) letra apenas se transforma após a inserção do comando seguinte. Quando a alteração é feita pelo **comando acom** a letra muda imediatamente. De qualquer maneira isto é apenas uma incongruência visual pois o aparelho é ligado/desligado imediatamente após a mudança nas propriedades que origina que o processador receba um valor verdadeiro/falso das regras.

2.4 Ficheiros

Em relação aos ficheiros excluindo os de código-fonte, é possível testar as funcionalidades do programa usando um ficheiro com os comandos pretendidos.

```
hnova 2 2
znova 1 1
znova 2 2
cnovo 1 a a
cnovo 1 s t
cnovo 1 p liga
zcomp 1
rnova 1 1 entre 1 20 50
rlista 1 1
asoc 1 1 1
```

Figura 13 - Exemplo de ficheiro teste.txt

3. Conclusão

Neste momento apenas estão presentes os ficheiros necessários para que seja apresentado o ambiente de simulação no terminal, sendo que em relação às classes estão criadas de forma geral as necessárias para o funcionamento do simulador da habitação apesar de algumas não estarem ainda implementadas uma vez que não são precisas no momento.

Através do programa apresentado neste relatório é possível fazer uma simulação de uma habitação controlada por componentes de domótica interligados entre si. É possível perceber sempre qual dos componentes foi utilizado para realizar uma determinada ação uma vez que têm todos identificadores únicos seja através de um identificador numérico juntamente com a letra do seu tipo ou então pelo nome no caso dos processadores guardados em memória.

4. Bibliografia

1. **Stack Overflow**
2. **TutorialSpoint**
3. **Cplusplus**
4. **ChatGPT**
5. **Material de apoio da disciplina disponibilizado no Inforestudante**