



**Instituto Superior de Engenharia de Coimbra**

**2023/2024**

**Licenciatura em Engenharia Informática**  
**Programação Orientada a Objetos**

**Trabalho Prático**  
**Home Sim**



**Bruno Pinto**, nº 2021129642

**Diogo Ferreira**, nº 2021129669

## Índice

<b>Introdução .....</b>	<b>2</b>
<b>Estrutura de Classes .....</b>	<b>3</b>
Habitação .....	4
Zona .....	5
Componente.....	6
Aparelho.....	7
<b>Arquitetura do Sistema .....</b>	<b>8</b>
<b>Conclusão .....</b>	<b>15</b>
<b>Bibliografia .....</b>	<b>16</b>

## Introdução

O Trabalho Prático de **Programação Orientada a Objetos** consiste na criação de um simulador de uma habitação de domótica interligados entre si, utilizando a **linguagem C++** juntamente com a biblioteca **PDcurses**<sup>1</sup> para criar a interface do simulador.

A habitação inclui **Zonas**, cada uma com **Propriedades** como temperatura, luz, etc. Essas Propriedades são monitorizadas por **Sensores**, que coletam dados para **Processadores** que tomarão ações com base em **Regras** determinadas pelo usuário e nas leituras dos sensores.

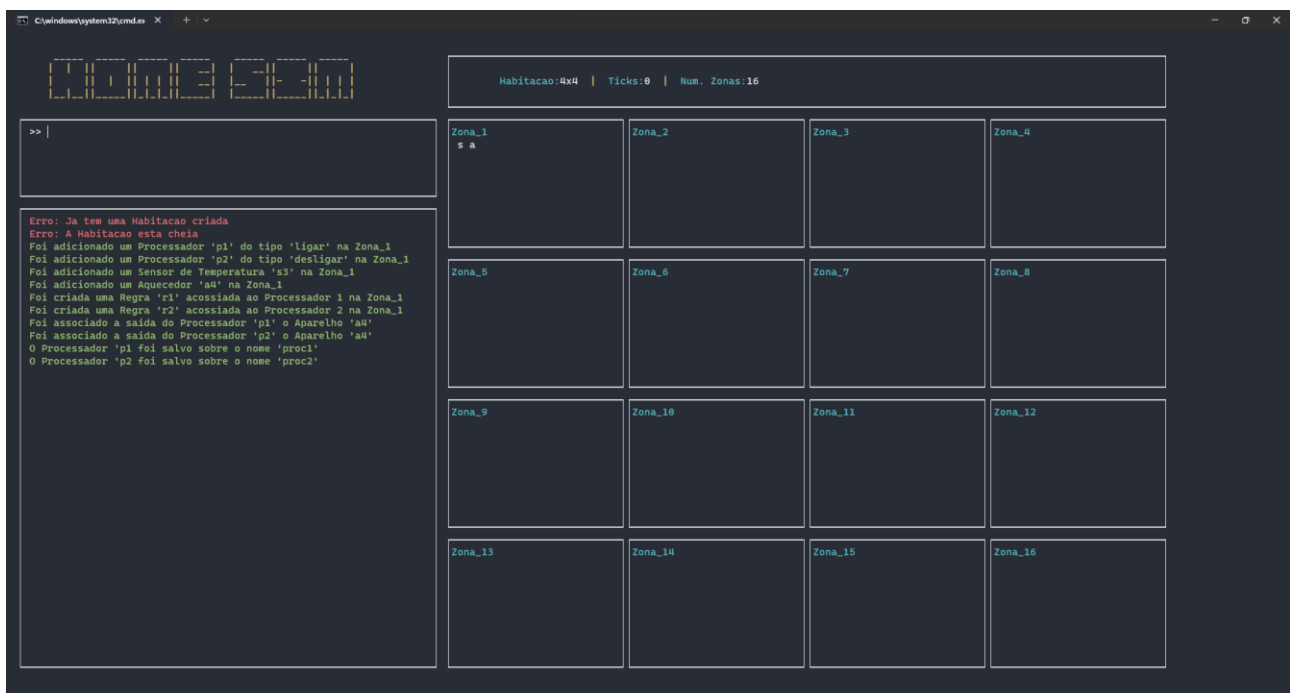


Figura 1 - Layout do Simulador

Para ajudar no desenvolvimento e na *User Experience*<sup>2</sup> foi criado um comando **help**. O comando lista todos os comandos existentes do Simulador.

<sup>1</sup> **PDcurses**: biblioteca responsável pela criação da parte gráfica ( <https://pdcurses.org> )

<sup>2</sup> **User Experience**: Experiência do utilizador

## Estrutura de Classes

Para o funcionamento do projeto foram criadas algumas classes:

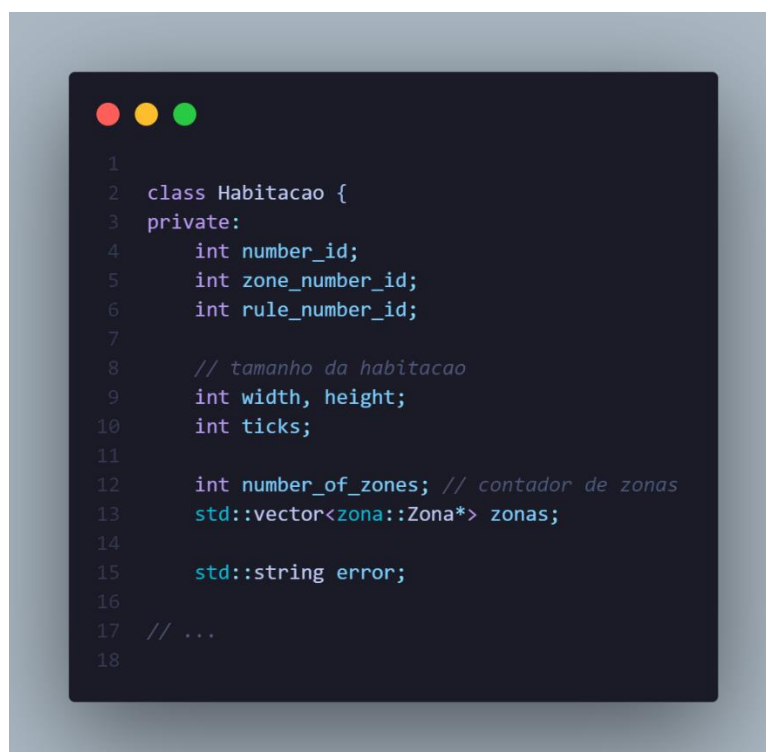
- **Simulador:** classe responsável por todo o funcionamento do simulador.
- **Comando:** classe que controla o funcionamento dos comandos do simulador usando ***Command Design Pattern***<sup>3</sup>.
- **Habitação:** classe com toda a informação da habitação.
- **Zona:** responsável por gerir todos os **Componentes** de cada zona.
- **Propriedades:** classe que guarda todas as propriedades da **Zona**.
- **Componente:** classe geral usada para a informação em comum dos diferentes **Aparelhos** e **Sensores**.
- **Aparelho:** reservada para os diferentes aparelhos e engloba um **Processador**.
- **Processador:** reservada para cada processador englobando uma ou várias **Regras**.
- **Regra:** reservada a cada regra.
- **Sensores:** reservada para cada Sensor

---

<sup>3</sup> **Command Design Pattern:** Conceito utilizado para executar os comandos ( <https://refactoring.guru/design-patterns/command/cpp/example> )

## Habitação

Cada **Habitação** guarda o número do ID para cada **Componente** novo, **Zona** e **Regra**. Além disso sabe o seu tamanho, quanto tempo (**ticks**) passou, quantas **Zonas** criadas existem e uma *string*<sup>4</sup> que guarda a última mensagem de erro ( não é necessariamente um erro, mas na maioria das vezes que é usada é para uma mensagem de erro).



```
1
2 class Habitacao {
3 private:
4     int number_id;
5     int zone_number_id;
6     int rule_number_id;
7
8     // tamanho da habitacao
9     int width, height;
10    int ticks;
11
12    int number_of_zones; // contador de zonas
13    std::vector<zona::Zona*> zonas;
14
15    std::string error;
16
17    // ...
18
```

Figura 2 - Classe da Habitação

---

<sup>4</sup> **string**: cadeia de caracteres

## Zona

Cada **Zona** guarda o seu *id*, a sua posição (*x,y*), as suas **Propriedades** e os **Componentes** que fazem parte da mesma. Além dessa informação também guarda o último erro ocorrido na Zona (o mesmo conceito da **Habitação**)

```
1  namespace zona {
2
3      class Zona {
4      private:
5          int id;
6
7          // coordenadas
8          int pos_x, pos_y;
9
10         // propriedades
11         std::vector<propriedades::Propriedade*> props;
12
13         // componentes
14         std::vector<componente::Componente*> comps;
15         int count_Sensors;
16         int count_Processors;
17         int count_Gadgets;
18
19         std::string error;
20
21     public:
22         Zona();
23         Zona(int number_id, int x, int y);
24     //...
```

Figura 3 - Classe Zona

## Componente

Cada **Componente** sabe o seu próprio *id*, nome e o seu tipo (se é um **Aparelho**, **Sensor** ou **Processador**)

Esta classe é a principal de onde todos os componentes do **Simulador** derivam desta.

```
1 namespace componente {
2
3     enum class ComponenteType : char {
4         APARELHO = 'a',
5         SENSOR = 's',
6         PROCESSADOR = 'p'
7     };
8
9     class Componente {
10     private:
11         std::string id;
12         std::string name;
13         char type;
14
15     public:
16         Componente(int id, char type, std::string name);
17     // ...
18     // -- FUNCS VIRTUAIS
19     virtual ~Componente(){}
20
21     virtual char getType() const;
22     // devolve o caracter que o descreve
23
24     virtual std::string getInfo() const;
25     // devolve a informacao da componente
26
27     virtual std::string run(std::vector<propriedades::Propriedade *> &props);
28     // corre o componente
29
30     // -- FUNCS GLOBAIS
31     std::string getID() const;
32     // devolve o ID do componente
33
34     std::string getName() const;
35     // devolve o nome do componente
36
37     };
38
39 } // componente
40
```

Figura 4 - Classe Componente

O método **virtual run()** retorna uma mensagem do que aconteceu no **Componente**.

## Aparelho

A Classe **Aparelho** é a classe “mãe” de cada tipo de aparelho (Aquecedor, Aspersor, etc).

Cada Aparelho sabe o seu estado (*ligado/desligado*), o seu tipo e quanto tempo está num determinado estado.

```
1  enum class AparelhoType : char {
2      AQUECEDOR = 'a',
3      ASPERSOR = 's',
4      REFRIGERADOR = 'r',
5      LAMPADA = 'l'
6  };
7
8  class Aparelho : public componente::Componente{
9  private:
10
11      bool isOn;
12      // estado de desligado ou ligado
13
14      int ticks_passed;
15      // ticks passados desde a ultima mudança no ambiente
16
17      AparelhoType type;
18      // tipo do aparelho
19
20  public:
21      Aparelho(int id, AparelhoType type, std::string name);
22
23      bool getIsOn() const;
24      // devolve o estado do aparelho
25
26      int getTicks() const;
27      // devolve ticks passados desde a ultima mudança no ambiente
28
29      void resetTicks();
30      // redefine os ticks passados desde a ultima mudança no ambiente
31
32      virtual char getType() const override;
33      // devolve o caracter que o descreve
34
35      virtual std::string getInfo() const override;
36      // devolve a informacao do componente
37
38      void turnOn();
39      void turnOff();
40      // Liga/desliga o aparelho
41
42      virtual void runCommand(std::string command);
43      // corre um comando recebido
44
45      std::string run(std::vector<propriedades::Propriedade *> &props) override;
46      // correr o Aparelho
47
48  };
```

Figura 5 - Classe Aparelho



## Arquitetura do Sistema

Todo o programa baseia-se na classe **Simulador**, onde é processador todos os comandos do **Utilizador** e onde é gerida toda a **Habitação**.

Para inicializar a simulação é necessário criar um objeto **Simulador**, incluindo um objeto **Terminal**<sup>5</sup>, e chamar o método **run()**.



```
1  int main() {
2      // criar o terminal
3      term::Terminal &t = term::Terminal::instance();
4
5      // criar simulador
6      simulador::Simulador sim(t);
7
8      _sleep(500);
9
10     // começar a simulacao
11     sim.run();
12
13     t.clear();
14     return 0;
15 }
```

Figura 6 - Inicialização do Simulador

No método **run()** é onde são inicializadas as principais janelas e onde entramos num ciclo onde são lidos todos os comandos do utilizador até o mesmo desejar sair, usando o comando “sair”.

---

<sup>5</sup> **Terminal:** Tipo de objeto da biblioteca responsável por manipular a consola

Para que cada comando seja corretamente executado é usado o método do Simulador *executeCommand()*, onde é verificado qual é o comando que foi pedido.

```
1 void Simulador::executeCommand(std::string& prompt, term::Window& output, bool isFromExec) {
2
3     // dividir comando
4     std::string cmd, args;
5     splitCMD(prompt, &cmd, &args);
6
7     // guardar os argumentos
8     std::istringstream iss(args);
9
10    Comando* exe = nullptr;
11
12    // parte visual
13    if(!isFromExec)
14        output.clear();
15    output << term::set_color(COLOR_DEFAULT);
16
17
18    if(cmd.empty()){
19        output << term::set_color(COLOR_ERROR) << "Comando vazio";
20    } else if(h == nullptr && cmd != "hnova" && cmd != "exec" && cmd != "sair" && cmd != "help"){
21        output << term::set_color(COLOR_ERROR) << "Tem primeiro de criar uma habitacao: hnova <numLinhas> <numColunas>\n";
22    } else if(cmd == "prox"){
23
24        exe = new Prox;
25
26        // ...
27
28    }
```

Figura 7 - estrutura do executeCommand()

Após a ter sido reconhecido o comando é executado.

**Nota: toda a verificação da sintaxe do comando é feita diretamente na execução.**

```
1 // ...
2
3 if(exe == nullptr)
4     return;
5
6 if(exe->Execute(*h, args))
7     output << term::set_color(COLOR_SUCCESS) << " " << exe->getError() << "\n";
8 else
9     output << term::set_color(COLOR_ERROR) << " Erro: " << exe->getError() << "\n";
10
11 }
```

Figura 8 - Execução do Comando

Para que os comandos fossem verificados e executados foi usado o método de **Command Design Pattern**, onde cada comando é um objeto de uma classe.

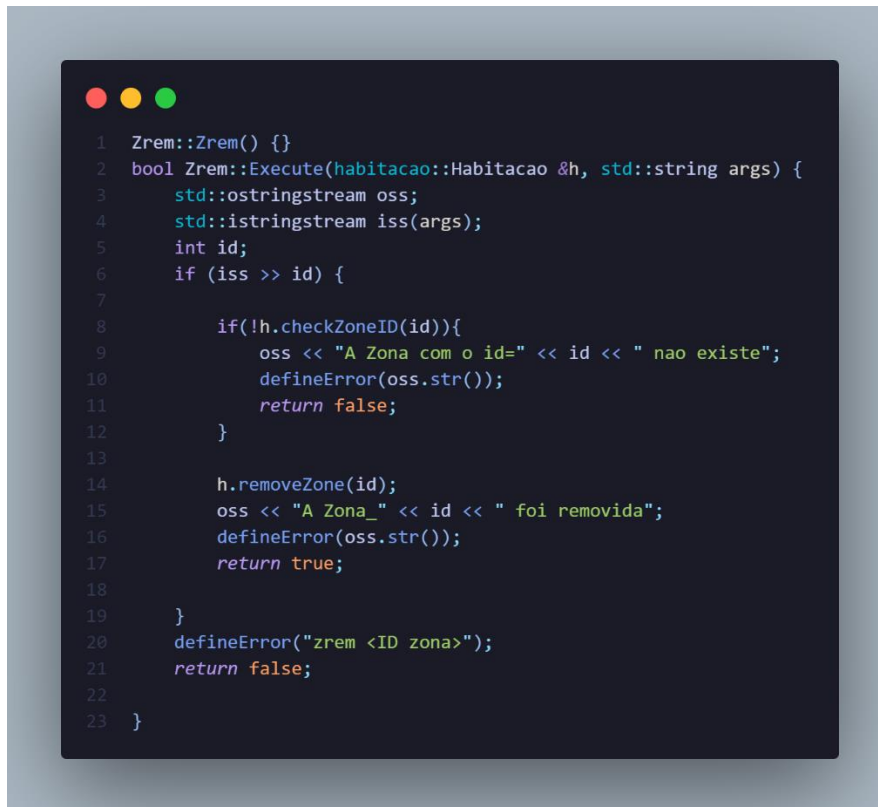
```
1  class Comando {
2  private:
3      std::string error;
4      std:: string args;
5
6  public:
7      // -- FUNCS VIRTUAIS
8      virtual ~Comando(){}
9
10     virtual bool Execute(habitacao::Habitacao &h, std::string args) = 0;
11     // funcao para executar o comando
12
13     // -- FUNCS GLOBAIS
14     void defineError(std::string error_message);
15     // funcao para definir o erro
16
17     std::string getError() const;
18     // funcao para recuperar o erro
19
20 };
21
22 class Prox : public Comando {
23 public:
24     Prox();
25
26     bool Execute(habitacao::Habitacao &h, std::string args) override;
27 };
28
29
30 class Znova : public Comando {
31 public:
32     Znova();
33
34     bool Execute(habitacao::Habitacao &h, std::string args) override;
35 };
36
37 // ...
38
```

Figura 9 - Estrutura de Comandos usando CDP<sup>6</sup>

---

<sup>6</sup> CDP: Command Desgin Pattern

A estrutura geral de todos os comandos passa por desmontar os argumentos do comando ( mesmo que esses sejam inexistentes ). Após serem separados, são executados os métodos necessários na **Habituação** para obter o resultado pretendido.

A screenshot of a code editor window with a dark background and light-colored text. The code is written in C++ and shows the implementation of a 'Zrem' command. It includes a constructor, a main execution function, and logic to check if a zone ID exists and then remove it. The code is numbered from 1 to 23 on the left side of the editor.

```
1  Zrem::Zrem() {}
2  bool Zrem::Execute(habitacao::Habitacao &h, std::string args) {
3      std::ostringstream oss;
4      std::istringstream iss(args);
5      int id;
6      if (iss >> id) {
7
8          if(!h.checkZoneID(id)){
9              oss << "A Zona com o id=" << id << " nao existe";
10             defineError(oss.str());
11             return false;
12         }
13
14         h.removeZone(id);
15         oss << "A Zona_" << id << " foi removida";
16         defineError(oss.str());
17         return true;
18     }
19     defineError("zrem <ID zona>");
20     return false;
21 }
22
23 }
```

Figura 10 - Estrutura geral de um Comando

Dentro da **Habitação** podem existir várias **Zonas** que são responsáveis por gerir todos os **Componentes** nela inseridos.

```

1
2 class Zona {
3 private:
4     int id;
5
6     // coordenadas
7     int pos_x, pos_y;
8
9     // propriedades
10    std::vector<propriedades::Propriedade*> props;
11
12    // componentes
13    std::vector<componente::Componente*> comps;
14    int count_Sensors;
15    int count_Processors;
16    int count_Gadgets;
17
18    std::string error;
19
20    // ...
21

```

Figura 11 - Estrutura de uma Zona

Durante a simulação, na passagem de tempo ( quer seja por meio do comando **prox** ou **avanca** ), é percorrida cada **Zona**, que por sua vez percorre cada elemento em si inserido, todos os componentes e executado o método **run()**.

```

1 bool Habitacao::tick() {
2     std::ostringstream oss;
3     ticks++;
4
5     for(auto& zona : zonas) {
6         zona->tick();
7         oss << zona->getError();
8     }
9
10    error = oss.str();
11    return true;
12 }

```

```

1 bool Zona::tick() {
2     std::ostringstream oss;
3     std::string output, cmd;
4
5     for (auto& comp : comps) {
6         // correr cada Componente
7         std::istringstream iss(comp->run());
8
9         // percorrer e correr o output
10        while (std::getline(iss, cmd))
11            if (!executeMod(cmd))
12                oss << error << "\n ";
13    }
14
15    error = oss.str();
16    return true;
17 }

```

Figura 12 - Passagem de tempo da Simulação

Cada **Componente** recebe as **Propriedades** da **Zona**, porém a sua utilização vai depender se necessita dessa informação ou não, por exemplo os **Processadores** não necessitam dessa informação pois o seu funcionamento não depende diretamente das **Propriedades**.

Um exemplo contrário é os **Aparelhos**, que com o passar do tempo alteram algumas **Propriedades** conforme a sua situação atual (se está *ligado* ou *desligado*).

```
1  std::string Lampada::run(std::vector<propriedades::Propriedade*> &props) {
2      Aparelho::run(props);
3      std::ostringstream oss;
4
5      // primeiro instante ligado
6      if(getIsOn() && getTicks() == 1) {
7          auto it = std::find_if(props.begin(), props.end(), [](const propriedades::Propriedade* p) {
8              return p->getType() == propriedades::PropriedadeType::LUZ;
9          });
10
11         if (it != props.end()){
12             if((*it)->getValue() == propriedades::UNSET)
13                 (*it)->setValue(0);
14             else
15                 (*it)->setValue((*it)->getValue() + 900);
16         }
17
18         oss << "A Luz foi alterada para " << (*it)->getValueStr() << "\n";
19         return oss.str();
20     }
21
22     // primeiro instante desligado
23     if(!getIsOn() && getTicks() == 1) {
24         auto it = std::find_if(props.begin(), props.end(), [](const propriedades::Propriedade* p) {
25             return p->getType() == propriedades::PropriedadeType::LUZ;
26         });
27
28         if (it != props.end()){
29             if((*it)->getValue() == propriedades::UNSET)
30                 (*it)->setValue(0);
31             else
32                 (*it)->setValue((*it)->getValue() - 900);
33         }
34
35         oss << "A Luz foi alterada para " << (*it)->getValueStr() << "\n";
36         return oss.str();
37     }
38
39     return "";
40 }
41 }
```

Figura 13 - método run() da classe Lâmpada

Os **Processadores** no seu **run()** verificam se todas as suas **Regras** são verdade, caso sejam notificam os **Aparelhos** a qual estão ligados.

```
1  bool Processador::areRulesTrue() {
2      if(rules.empty())
3          return false;
4
5      for(const auto &r : rules) {
6          if (!r->check()) {
7              sent = false;
8              return false;
9          }
10     }
11
12     return true;
13 }
14
15 std::string Processador::notifyGadgets() {
16     if(sent)
17         return "";
18
19     for(const auto& ap : gadgets)
20         ap->runCommand(getAction());
21
22     sent = true;
23     return "O Processador " + getID() + " enviou o comando \'' + command + "'\n";
24 }
25
26 std::string Processador::getAction() const {
27     return command;
28 }
29
30 std::string Processador::run(std::vector<propriedades::Propriedade *> &props) {
31     std::ostringstream oss;
32
33     if(areRulesTrue()){
34         oss << notifyGadgets();
35         return oss.str();
36     }
37
38     return "";
39 }
```

Figura 14 - run() dos Processadores

## Conclusão

Concluindo, no desenvolvimento do **Home Sim** exploramos os conceitos fundamentais da Unidade Curricular de **POO**. Exploramos os diferentes conceitos que abordámos nas aulas como: Classes e Objetos, Herança, Polimorfismo, Encapsulamento, *Overloading* de funções e Gestão de memória.

Este trabalho permitiu nos pôr em prática a matéria lecionada e consequentemente aprofundar o nosso conhecimento em **C++** e de **Programação Orientada a Objetos**.



## Bibliografia

cppreference.com. (s.d.). *cppreference.com*. Obtido de cppreference.com: <https://en.cppreference.com>

Durães, J. A. (2023). *NONIO IPC*. Obtido de inforestudante.ipc.

Refactoring.Guru. (2014). *Factory Method in C++ / Design Patterns*. Obtido de refactoring.guru: <https://refactoring.guru/design-patterns/factory-method/cpp/example>

Refactoring.Guru. (2014). *Command in C++ / Design Patterns*. Obtido de refactoring.guru: <https://refactoring.guru/design-patterns/command/cpp/example>

Tutorialspoint. (2023). *C++ Tutorial*. Obtido de tutorialspoint: <https://www.tutorialspoint.com/cplusplus/index.htm>