Programação Orientada a Objetos

Trabalho prático

2023/2024

Meta 2

Licenciatura de Engenharia Informática

Kevin Rodrigues - 2013010749@isec.pt

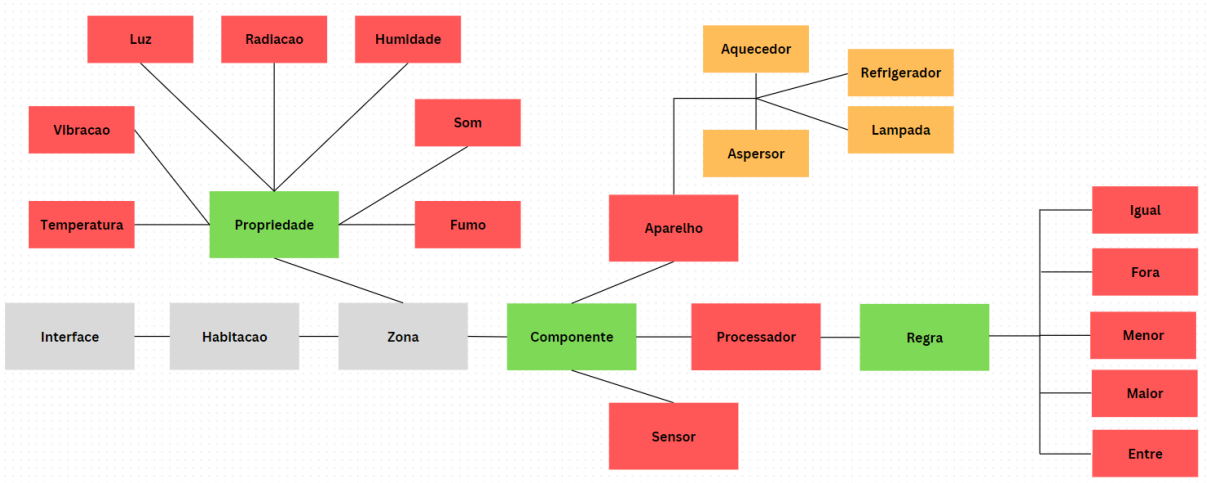
Renato Ferreira – a2019134345@isec.pt

1 - Requisitos

	implementado	parcialmente implementado	não implementado
prox	x		
avanca	x		
hnova	x		
hrem	x		
znova	x		
zrem	x		
zlista	x		
zcomp	x		
zprops	x		
pmod	x		
cnovo	x		
crem	x		

rnova	x		
pmuda	x		
rlista	x		
rrem	x		
asoc	x		
ades	x		
acom	x		
psalva	x		
prepoe	x		
prem	x		
plista	x		
exec	x		
sair	x		

2 - Estrutura



3 - Classes

Durante o desenvolvimento do trabalho prático, foram aplicadas conceitos e metodologias de programação orientada a objetos com ênfase na delegação de responsabilidades correspondentes a cada objeto.

Não foi implementado nada que comprometesse o encapsulamento das classes, sendo que não serão encontradas funções friend nem variáveis /métodos “protected” no projeto.

A herança foi aplicada quando necessário, sendo ela de 2 ou 3 camadas.

A maioria das classes base são abstratas. Isto é devido ao polimorfismo aplicado quando uma ou mais métodos executam operações diferenciadas.

Composição/Agregação foram implementados conforme a necessidade.

Smart pointers nem templates foram implementados no trabalho prático.

3.1 Interface

```
bool map = false;
habitacao *habit;
std::map<string, processador*> ps;
```

A classe “interface” é responsável pela inicialização do simulador, controle gráfico e interpretação dos comandos inseridos pelo utilizador.

Dentro da classe em questão, por composição, apresenta um ponteiro de “habitação”, sendo esta criada no comando “hnova”.

Até a criação desta habitação, validado pelos limites definidos no enunciado, qualquer ação, à exceção dos comandos “exec”/”sair”/”hnova”, são bloqueados pela flag interna (bool map) do objeto interface.

Em relação ao armazenamento de processadores em memória, o resultado dessa operação é guardado nesta classe num map<string,processador*>.

3.2 Habitação

```
vector <zona*> zonas;
```

Na classe “habitacao”, é gerido a informação externa relativa às zonas. Por composição, todas as zonas criadas ao longo do simulador são guardadas num vector de ponteiros nesse objeto “habitacao”.

3.3 Zona

```
vector<sensor*> sensores;
vector<aparelho*> aparelhos;
map<string,propriedade*> propriedades;
vector<processador*> processadores;
```

Cada zona tem as suas propriedades e componentes de domótica, sendo estes guardados em vectores de ponteiros(aparelhos, sensores e

processadores) e num map (propriedades). Esta gestão é feita pela classe “zona” sendo esta responsável pelo conteúdo dele (composição).

3.4 Propriedades

```
virtual void set_valor(int v) = 0; // valida os limites do valor da propriedade
```

As propriedades são geradas quando uma zona é criada, mesmo sem sensores ou aparelho, isto porque são independentes, ou seja, a sua existência não é bloqueada pela existência de componentes de domótica, apenas os seus valores não poderão ser lidas nem alteradas durante a passagem do tempo.

A classe “propriedade” é uma classe base abstrata, tendo esta várias classes derivadas, tais como, “temperatura”/”vibracao”/”luz”/”radiacao”/”humidade”/”som”/”fumo”. As classes derivadas têm a responsabilidade de validar a alteração de valor proposto pelo aparelho através dos limites definidos na criação das propriedades.

3.5 Componente

```
class componente {

public:
    static int generetedId() ;

private:
    static int genereted_id;
};
```

A classe “componente” foi pensada de forma que os identificadores dos componentes de domótica (aparelhos, processadores e sensores) sejam sequenciais e únicos.

3.5.1 Aparelho

```
class aparelho : componente{
public:
    aparelho(string type);
    ~aparelho();
    string get_id() const; // retorna id do aparelho
    int get_id_aparelho() const; //retorna valor do id static para
    void set_id_aparelho() const; // define o id do proximo aparel
    string get_type() const; // retona o tipo de aparelho
    virtual aparelho* clone() = 0; // retorna objeto com novo pon
    int get_instance() const; // retorna a instancia atual interna
    void set_instance(); // define a instancia atual interna
    bool get_isOn() const; // devolve o estado do aparelho atual
    void set_isOn(); // define o estado atual do aparelho
    virtual void set_val_change(string user_cmd,string cmd) = 0;

private:
    string id;
    static int id_aparelho;
    string type;
    int instance;
    bool isOn;
```

A classe “aparelho” é uma classe derivada da classe “componente”. É uma classe abstrata que apresenta classes derivadas desta classe, que são o “aquecedor”/”aspersor”/”refrigerador”/”lampada”.

Os aparelhos, dependendo do comando fornecido pelo processador e validação interna relacionado a passagem do tempo, enviam o valor a incrementar ou decrementar a propriedade para validar.

3.5.2 Sensor

```
class sensor : componente{
public:
    sensor(string cmd, propriedade *pp); /
    ~sensor(); // destrutor por defeito
    string get_id() const; // retorna id d
    static int get_id_sensor() ; // retorn
    static void set_id_sensor(); // define
    string get_prop() const; // retorna a
    sensor* clone(); // retorna o objeto c
    int get_prop_val() const; // retorna v

private:
    string id;
    string prop;
    propriedade *p;
    static int id_sensor;
```

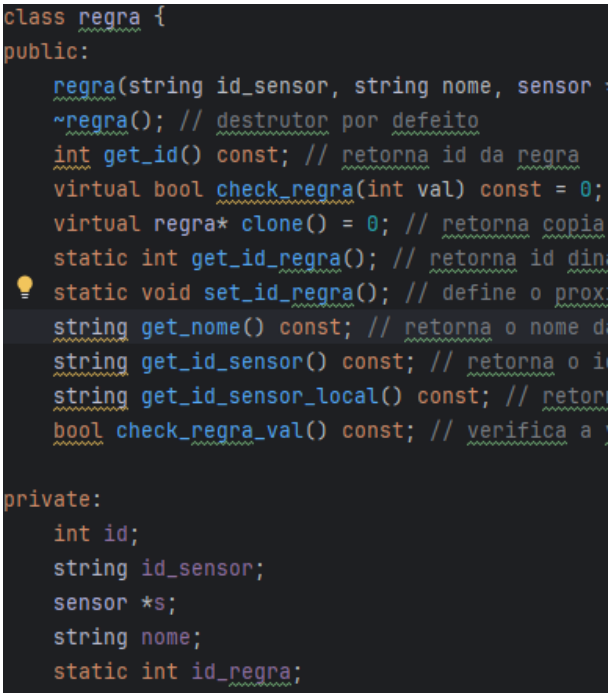
Na simulação só existe um tipo de sensor, que na criação define-se que propriedade irá ler da zona inserida. Por agregação, é associado o ponteiro de propriedade da zona inserida e a leitura é feita por essa via.

3.5.3 Processador de regras

```
vector<regra*> regras;
vector<aparelho*> AllAparelhos;
```

Os processadores é um dispositivo que permite guardar várias regras e é responsável por elas (composição). Também permite fazer a ligação a vários aparelhos, por agregação. Caso todas as regras se tornem verdadeiras, o processador envia o seu comando associado aos aparelhos ligados ao processador.

3.6 Regras



```
class regra {
public:
    regra(string id_sensor, string nome, sensor
    ~regra(); // destrutor por defeito
    int get_id() const; // retorna id da regra
    virtual bool check_regra(int val) const = 0;
    virtual regra* clone() = 0; // retorna copia
    static int get_id_regra(); // retorna id din
    static void set_id_regra(); // define o prox
    string get_nome() const; // retorna o nome d
    string get_id_sensor() const; // retorna o i
    string get_id_sensor_local() const; // retor
    bool check_regra_val() const; // verifica a

private:
    int id;
    string id_sensor;
    sensor *s;
    string nome;
    static int id_regra;
```

A classe “regra” é uma classe base abstrata que apresenta várias classes derivadas, sendo elas “igual”/”fora”/”entre”/”maior”/”menor”. Na definição de cada regra, é associado um sensor (agregação) e o valor da propriedade que esse sensor lê, é utilizado para a validação da regra. Se for verdadeiro, retorna “true”, caso contrário será “false”.

No retorno de todas as regras verdadeiras, é sinalizado ao processador a enviar o comando aos aparelhos.

4 - Decisões

4.1 Identificadores

Todos os componentes apresentam ID 's em string. Dependendo do tipo de componente, o seu tipo é concatenado com o inteiro fornecido pela classe “componente”. Por exemplo, um sensor tem o ID “s1”, um processador tem o ID “p2” e um aparelho tem o ID “a3”.

As zonas e as regras apresentam ID 's numéricos e as propriedades não tem ID direto mas a sua pesquisa é através do map<string,propriedade*> declarado nas zonas onde o campo string é o seu tipo.

4.2 Cópia

Em contexto do enunciado, ao longo do desenvolvimento apenas fez sentido desenvolver um construtor por cópia na classe processador com suporte ao operator=.
Em relação aos restantes métodos denominados por “clone”, é utilizado o construtor por cópia default.

4.3 Eliminação

Quando é invocado o comando de eliminação de sensores ou aparelhos, é verificada a existência de dependências. Quando é os sensores, é verificado se existe alguma regra, dessa zona, com esse sensor associado. Caso exista, o simulador informa o utilizador que a ação não foi concluída. Quando é os aparelhos, é verificado os processadores com aparelhos associados e caso exista, o simulador informa que a ação não foi concluída.