Contents

[**Smart Home Manager** 2](#_Toc187507063)

[**1. Program Structure and Development** 2](#_Toc187507064)

[**2. Encapsulation in the Program** 2](#_Toc187507065)

[**3. Reflection on the Solution** 3](#_Toc187507066)

[**4. Abstraction and Inheritance** 3](#_Toc187507067)

[**5. Polymorphism** 4](#_Toc187507068)

[**6. Memory Management and Avoiding Memory Leaks** 4](#_Toc187507069)

[**7. Design Decisions** 4](#_Toc187507070)

[**8. Conclusion** 5](#_Toc187507071)

[**References** 5](#_Toc187507072)

[**Appedix** 6](#_Toc187507073)

**Smart Home Manager**

**Overview**

You can control your home's smart gadgets like an expert with the help of the Smart Home Manager, a C++ software. Household appliances, lighting, temperature controls, security cameras, washing machines, and more are organised and controlled by room. Users may manage devices, do searches, activate and deactivate devices, and handle various rooms with customised device configurations using the software. Encapsulation, inheritance, polymorphism, and abstraction are some of the Object-Oriented Programming (OOP) concepts used by the Smart Home Manager to make the software easy to use, scalable, and maintain.

**1. Program Structure and Development**

A class-based architecture underpins the Smart Home Manager, with SmartHomeManager serving as the system orchestrator for the whole smart home network. There are a lot of different kinds of devices and rooms included in the system, such as SecurityCameras, SmartLights, Thermostats, etc. The application makes use of several OOP concepts, including:

* **Inheritance:** Devices are derived from the base class SmartDevice.
* **Encapsulation:** Internal attributes are protected and accessed via public getter and setter methods.
* **Polymorphism:** Virtual functions allow for flexible device interactions and cloning.

To manage the installation, removal, and activation of SmartDevice objects across rooms, the SmartHomeManager class interacts with Room objects.

**2. Encapsulation in the Program**

An essential component of object-oriented design is encapsulation. Entity coupling is the practice of encapsulating data (attributes) and methods (functions) that manipulate the data into a single unit, usually a class, and limiting direct access to certain components of the object. This restricts access to the object's internal state to those methods with well defined interfaces, making it secure.

**Example from the SmartDevice class:**

class SmartDevice {

protected:

int id;

string name;

string manufacturer;

bool status; // true = active, false = inactive

public:

virtual shared\_ptr<SmartDevice> Clone() const = 0; // Pure virtual method for cloning

SmartDevice(int id, const string& name, const string& manufacturer);

virtual ~SmartDevice();

virtual void InteractionEvent() const = 0;

void Activate();

void Deactivate();

bool GetStatus() const;

virtual void ViewInfo() const;

virtual bool CheckConnection() const;

int GetID() const;

void SetID(int id);

void SetName(const string& name);

const string& GetName() const;

};

The protected id, name, manufacturer, and status characteristics in this class may only be accessed by methods or derived classes within the same class. The device's state may be changed with the Activate and Deactivate methods, which are more secure than directly accessing the status property.

Other derived classes also accomplish encapsulation, guaranteeing that each device may communicate with its attributes in a safe and predictable manner.

**3. Reflection on the Solution**

This smart home management system's solution satisfies the fundamental needs of controlling smart devices in different rooms and enabling actions like adding, uninstalling, and interacting with gadgets. The architecture is clear and extensible because to the usage of object-oriented programming concepts; adding new device types requires little to no modifications to the current code.

Making sure that device interactions were handled appropriately was a difficulty throughout development, particularly when dealing with various kinds of devices (e.g., SmartLight vs. SmartDoorLock). Making sure the proper gadgets were turned on or off at the right times was another challenge when controlling their statuses across rooms.

**4. Abstraction and Inheritance**

The idea of abstraction is to display just the most important aspects of an item while concealing its implementation details. Here, we accomplish abstraction by building the abstract base class SmartDevice. This class allows us to communicate with devices without knowing their underlying details. It exposes just the methods that are essential, such as Activate(), Deactivate(), and ViewInfo().

Code reuse is encouraged by inheritance since new classes may be built based on old ones. The abstract methods specified in SmartDevice, including InteractionEvent() and ViewInfo(), must be implemented by every device type since they all derive from SmartDevice.

**Example from IrrigationSystem:**

class IrrigationSystem : public SmartDevice {

int wateringDuration; // in minutes

string schedule; // e.g., "8:00 AM, Daily"

int waterUsage; // in liters

shared\_ptr<SmartDevice> Clone() const override;

public:

IrrigationSystem(int id, const string& name, const string& manufacturer, int duration = 30, const string& sched = "6:00 AM, Daily", int usage = 0);

void SetWateringDuration(int duration);

void SetSchedule(const string& sched);

void CheckWaterUsage() const;

void InteractionEvent() const override;

void ViewInfo() const override;

};

With its own implementation of the InteractionEvent() and ViewInfo() methods, the IrrigationSystem class shows how the device functions uniquely while still complying to the base class interface. It derives from SmartDevice.

**5. Polymorphism**

Different classes may give their own distinct implementations of the same method or function; this is called polymorphism. This is especially helpful when working with items (such as devices) that belong to the same family but have diverse behaviours despite sharing an interface.

When the Smart Home Manager calls the InteractionEvent() or ViewInfo() methods on multiple kinds of devices, polymorphism is used. By communicating with them via pointers to the SmartDevice base class, the manager may provide flexibility even if the device kinds may change.

**Example of polymorphism:**

shared\_ptr<SmartDevice> device = make\_shared<SmartLight>(1, "Light", "Philips", 100, "Warm");

device->InteractionEvent(); // Calls SmartLight's InteractionEvent

In this example, the InteractionEvent() function will be invoked for the real object type, SmartLight, even if device is a reference to SmartDevice.

**6. Memory Management and Avoiding Memory Leaks**

The application reduces the danger of memory leaks by automatically handling memory management because it utilises shared\_ptr for device management. When an object is no longer required, its reference count will decrease to zero, and the shared\_ptr will destroy them.

Nevertheless, developers should use caution when dealing with circular references. Careful design decisions are required to prevent memory from being inadvertently kept in situations where devices contain references to other objects.

**7. Design Decisions**

We went with a class structure that would allow us to simply add new device kinds without having to change any of the current code. When managing memory, using shared\_ptr makes ownership semantics easier and guarantees correct cleaning. A unified controller that interacts with rooms and devices in a structured manner is the SmartHomeManager class.

**8. Conclusion**

A well-thought-out program, Smart Home Manager, handles the task of controlling smart home gadgets. The solution's modularity, extensibility, and ease of maintenance are all fruits of the use of abstraction, polymorphism, inheritance, and encapsulation. In the future, we may see enhancements to the user interface, support for new device kinds, and speed optimisation for bigger houses with several devices. With careful attention to the requirements for Object-Oriented Programming (OOP) concepts, testing procedures, design choices, reflection, and reference use, this project/report intends to attain a grade band of 74% to 100% as all the requirements and the optional work has been done.

**References**

1. *Object-Oriented Programming in C++* by Robert Lafore (Pearson)
2. *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley)
3. *C++ Programming: From Problem Analysis to Program Design* by D. S. Malik (Cengage Learning)

**Appedix**

