Homer

Ceccacci Michele, Magnani Simone, Monticelli Alessandro

April 9, 2023

Abstract

Contents

Chapter 1

Analysis

1.1 Requirements

HOMER (HOMe EmulatoR) is an emulated domotic controller connected to lights, windows, and other domestic facilities. The smart environment is managed through a dashboard which allows the user to control devices, monitor sensors and electrical consumptions. Each device should support updates in variable time units. Some devices should act in a somewhat random way (but still predictable), while others should be fully deterministic.

Functional Requirements

- The software should allow creating and controlling lots of different devices, such as:
  - Lights
  - Electrical outlets
  - Doors
  - Locks
  - Windows
  - Blinds
  - Temperature changer devices
- Devices can be added/removed at runtime.
- A logger is needed in order to allow users to keep track of state changes. It also makes debugging easier for developers.

- The software should have graph views for temperature and air quality state.
- The users should be able to see electrical consumption.
- The users should be able to set the desired temperature in the course of the day.
- The users should be able to view the trend of temperature, consumptions and air quality.

Non-Functional Requirements

- Should be performant enough to run on a desktop/laptop.
- User interface should be intuitive and user-friendly
- The project should be portable, and work on Windows, MacOS and Linux devices.

1.2 Analysis and domain model

1.2.1 Domain analysis

The application will emulate a domotic environment, some smart devices and facilities monitoring. Smart devices are common domestic devices as lights, electrical outlets, windows, doors and various sensors for monitoring indoor temperature and air quality along with electrical absorption. The user will interact with the devices via the controller through a dashboard.

The main challenge will be modelling the communication between the controller and the different devices, which can have very different states.

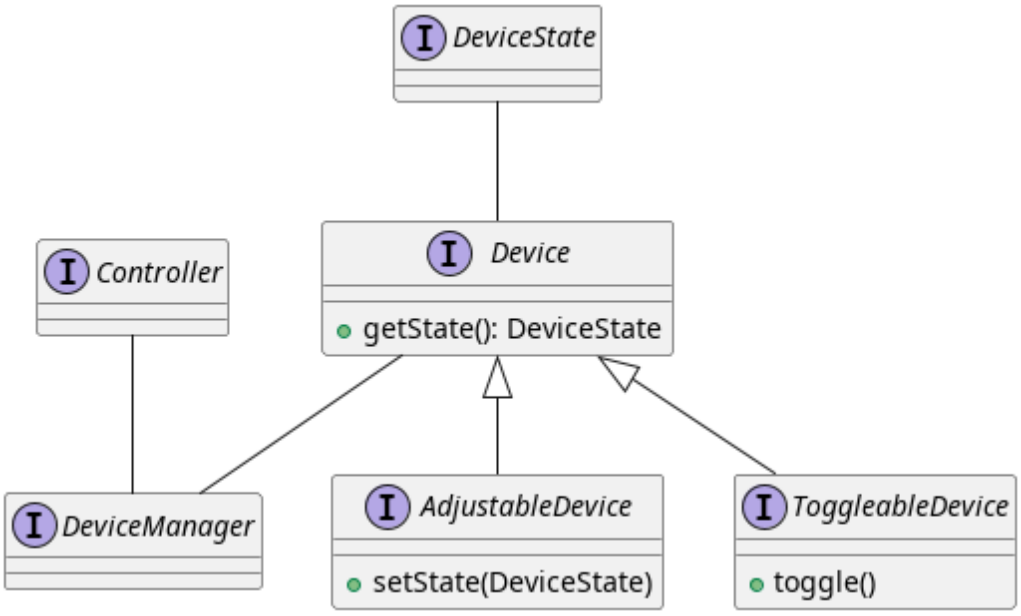


Figure 1.1: UML diagram of the domain

Chapter 2

Design

2.1 Architecture

HOMER is built on the MVC (model-view-controller) architecture. The controller is responsible for getting inputs from the views, and updating the model consequently. The controller also sends back state updates to the various views. The model part is composed by the devices and electrical outlets. Each device has a state of its own, that the controller gets and sends to the views. Each view is independent from each other. When a view commands a state update, the controller processes the change and sends an update signal to all the views.

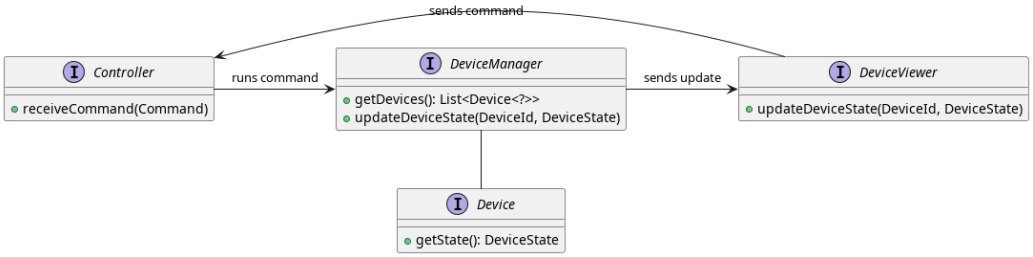
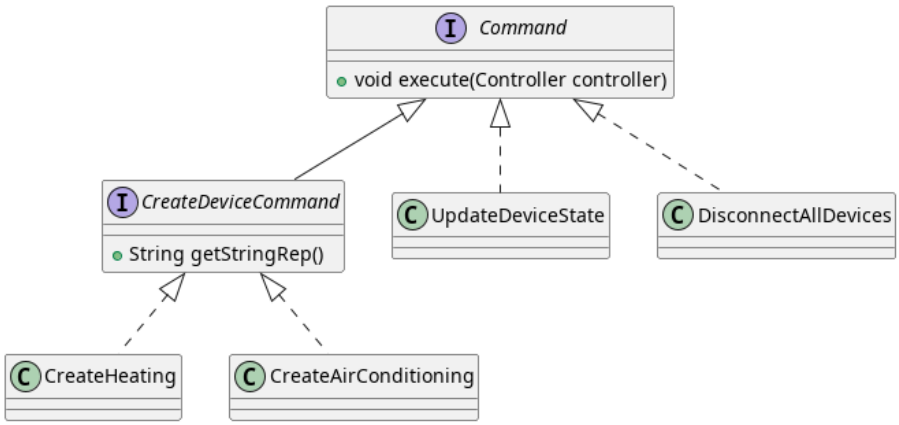


Figure 2.1: UML diagram of the architecture

2.2 Detailed Design

Problem: Sending updates from the view to the controller. We need a common way for all the UI components to send the same kind of updates to the controller without repeating code. We also want the controller to have more control over what is actually execute, and when it is executed. So

the controller acts as a receiver. All the commands implement an execute method, that is called inside the controller. Solution: Used the command pattern, which allows different views to send the same updates to the controller, avoiding code duplication Commands that create devices need to be displayed to the user, so we created a new interface called CreateDeviceCommand and we decided to opt for string representations instead of enums to allow for further future expansibility.



2.2.1 Michele Ceccacci

Logger

Problem: The system needs a composable logger, which has to track both state updates in devices and connections/disconnections of devices. The need for composition stems from the fact that different use cases of the software might need just a subset of the logging capabilities. The logger must also be dynamically composable for maximum flexibility. Solution: used the Decorator design pattern, which allows to wrap dynamically objects at runtime and compose them. Not using abstract classes also takes away the problems of inheritance.

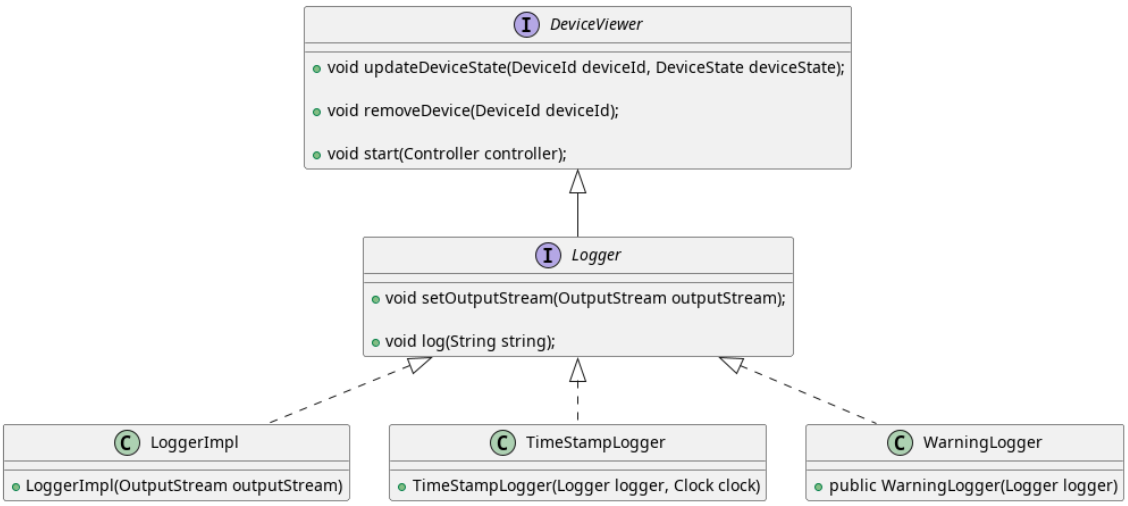


Figure 2.2: UML diagram of the logger

Air conditioning and Heating

Problem: Air conditioning and Heating devices have really similar behaviours. The difference between the two, is that an air conditioning device decreases the environment's temperature, whereas the heating device raises it. Solution: Since the only function that actually changes between the two implementations is the one responsible for updating the environment's temperature, the template method pattern was used. The method updateTick, responsible to update the environment and electrical consumption when called by the controller, is a template method, and calls the abstract method updateTemperature.

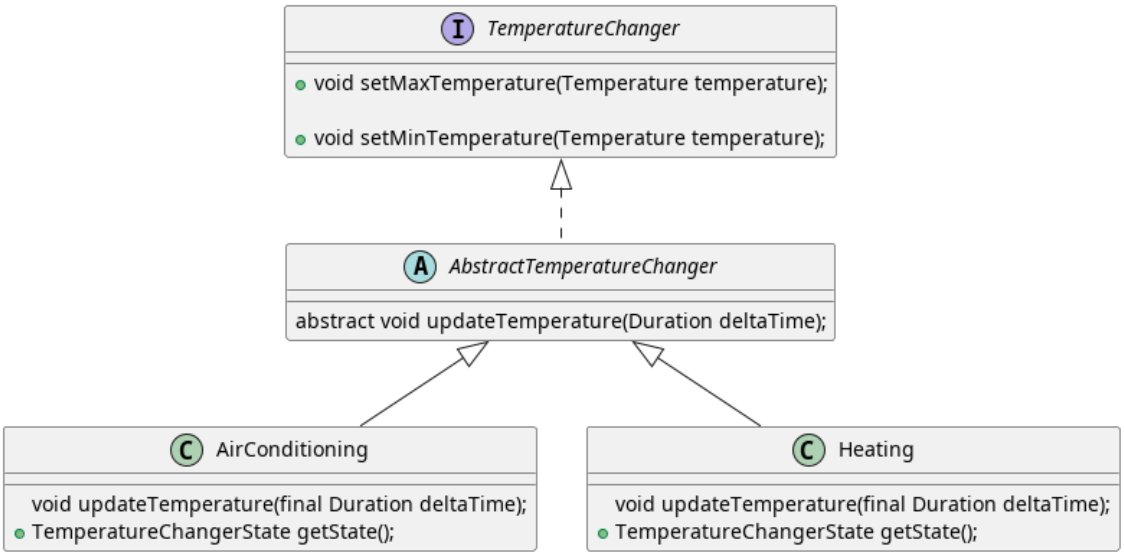


Figure 2.3: TemperatureChangers package uml diagram

Passing state updates to the view

Problem: devices have really different states, and they all need to passed down to the view. I took inspiration from REST applications, and thought about implementing a serialization/deserialization protocol. Since the view and the model don't need to communicate over a network, in the end i decided to just implement a DeviceState interface, which all device states implement. This also respects the single responsibilty principle, since every device's state can be updated independently. Then i defined a single command for state updates, so that the view could create a component for the specific device in case the device wasn't already present. To handle device removals, i just created a single removeDevice Method in the DeviceViewer interface, based on the deviceId, since it identifies the device univocally.
Another problem related to this, is the fact that when a view sends an update to the controller, all the other views need to be notified too. This was solved by adding a ViewManager to the controller, in charge of passing down updates to all the views. ViewManager extends the View interface, so we can be sure that all the valid view updates are also valid DeviceManager updates. The other only method ViewManager defines is addView, allowing to add other views dynamically.

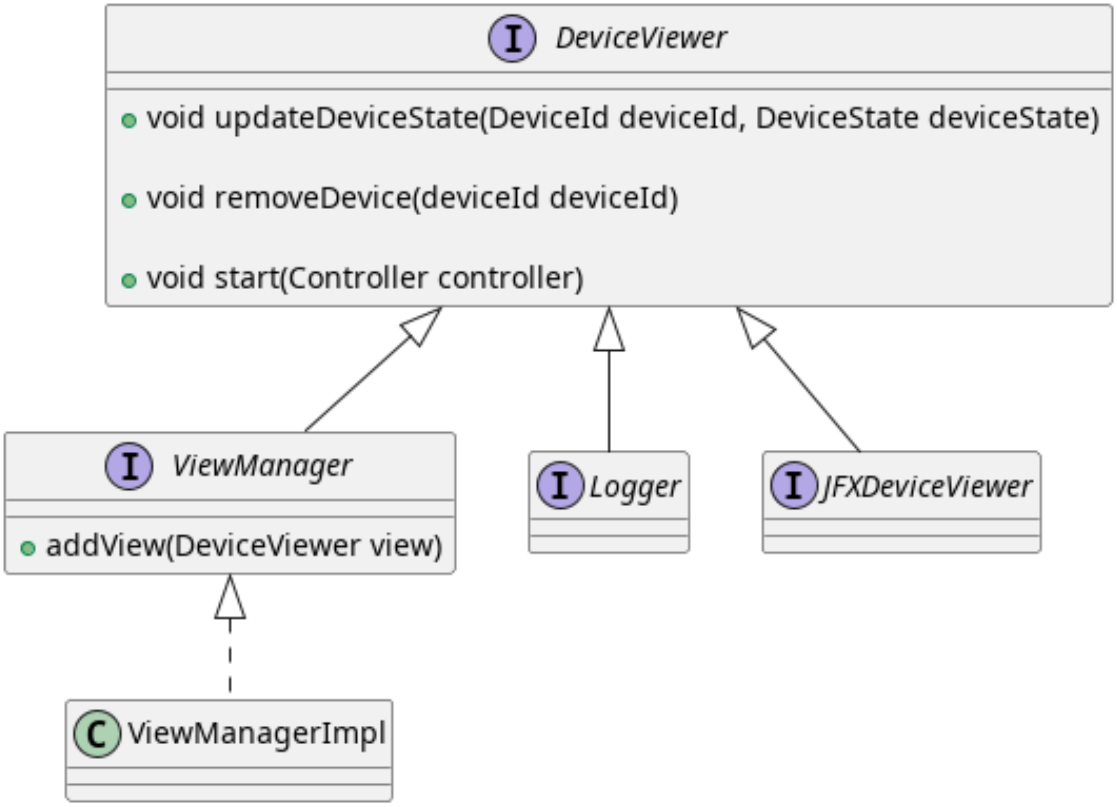


Figure 2.4: ViewManager uml diagram

2.2.2 Simone Magnani

Simulation

The simulation is a `discrete-time simulation`, where each tick represents an amount of time that has passed. This allows to control the `delta time`, therefore allowing to speed up/down the simulation.

Any component that requires to update its state in steps and know how much time has passed will implement the interface `DiscreteObject`.

Figure 2.5: UML diagram of the DiscreteObject interface

Problem Since the simulation can be sped up, we have to track the time in the simulation environment

Solution Creation of the concept of `Clock` that stores the time and is updated at each tick.

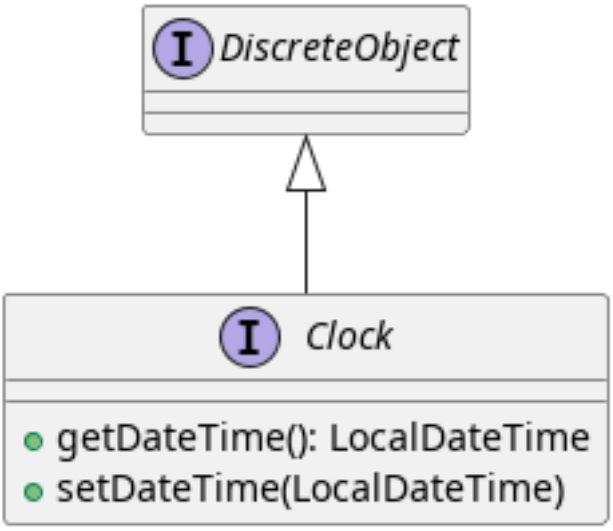


Figure 2.6: UML diagram of the clock

Control of the simulation

The game loop is handled in the simulation manager.

The simulation manager logic is split between the `SimManager` and `SimManagerViewObserver`.

Any `DiscreteObject` can ask to be updated by the game loop with the `Observer` pattern,via the `SimManager` method `addObserver`.

The simulation manager interfaces with its view counterpart to allow pause/resume and time rate control via the `SimManagerViewObserver` using the `Observer` pattern.

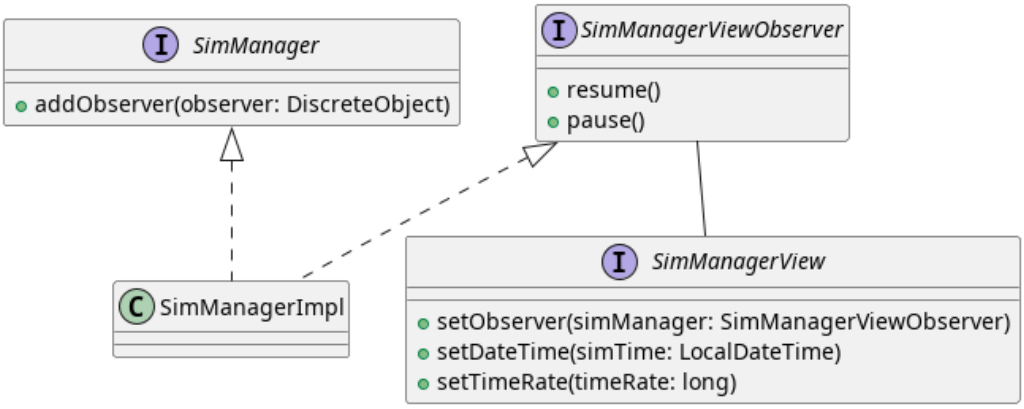


Figure 2.7: UML diagram of the SimulationManager

Devices

Devices will be detailed separarely between read-only, toggleable and adjustable ones.

Thermometer

The thermometer is modelled as an extension of `Device` since it's not meant to be controllable but only to return a `DeviceState` to the Controller, and `DiscreteObject`, to define how should the temperature be sensed (in a simple case, it could be updated directly, or in another case imperfections such as lag can be modelled).

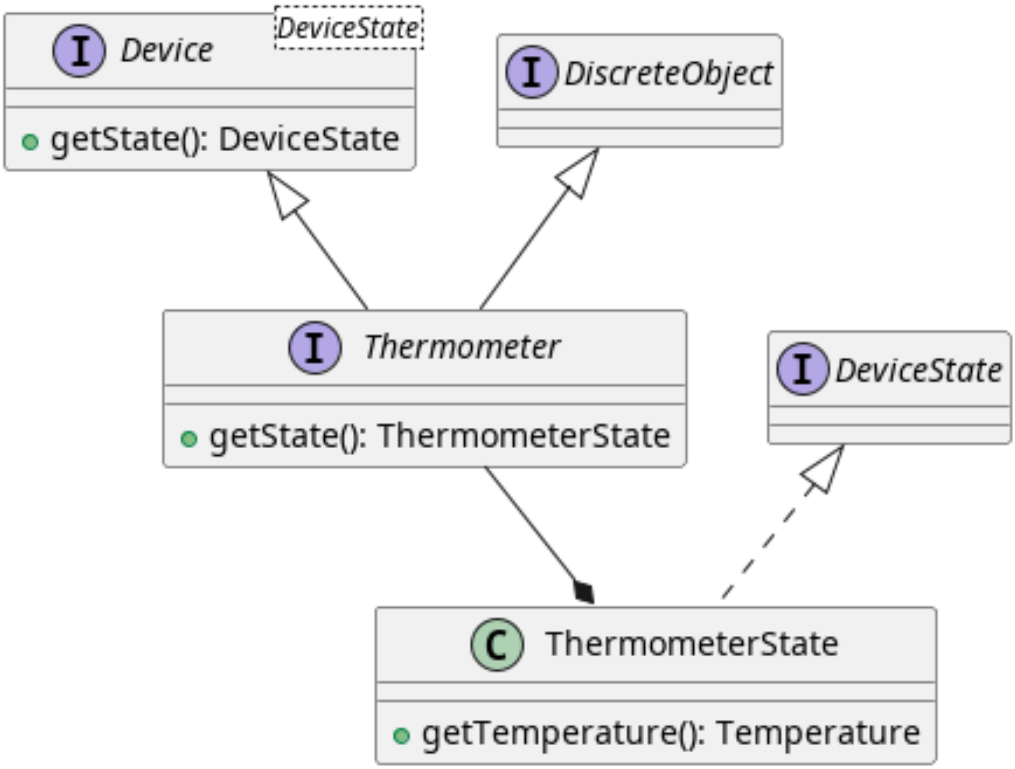


Figure 2.8: UML diagram of the thermometer

Problem There are multiple devices that need to interact with a common temperature

Solution Creation of the concept of `Environment` which represents the physical simulated environment with its temperature parameter. In the future, this would also allow to simulate different environments, eg. different rooms or completely separate buildings.

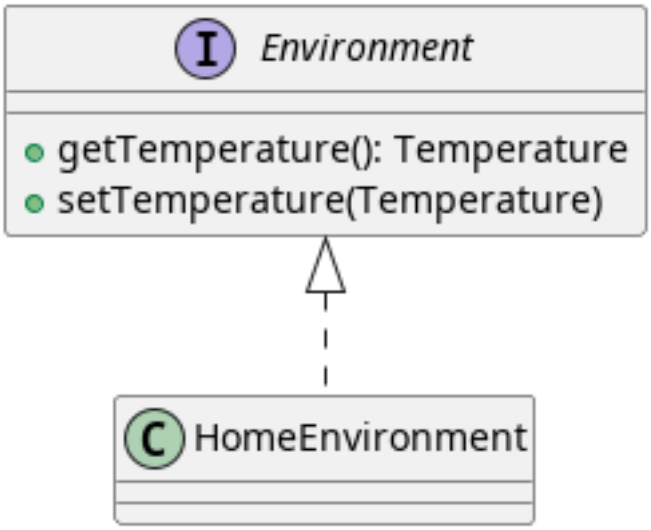


Figure 2.9: UML diagram of the environment

Lock

The Lock has been modelled as a `ToggleableDevice`. Its logic is really simple, the state can vary between locked and unlocked. The lock is represented in figure Figure 2.10

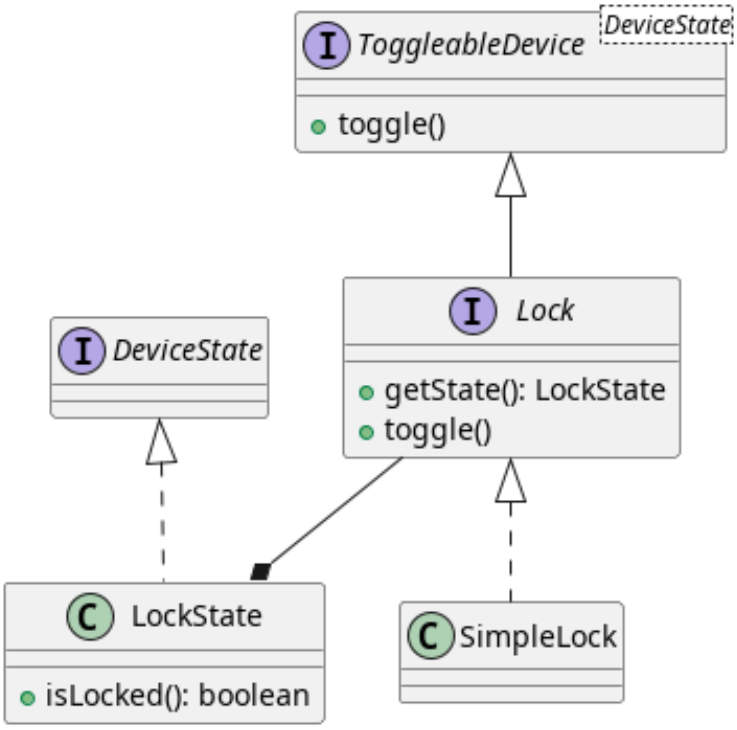


Figure 2.10: UML diagram of the lock

Window, Door, Blinds

The devices of type `Window, Door and Blinds` are modelled as `AdjustableDevices`.

Problem Each of these devices can be considered to be controllable between a range of values. Also, if those devices were meant to be remotely controllable, there should be something that moves them.

Solution Create the concept of `Actuator`. This also allows someone to decide however they want to model the movement mechanism (as simplistically or realistically as preferred) => `Strategy` pattern. It also allows to compartmentalize away the logic of the movement, adhering to the single responsibility principle.

Problem Min, max values for ranges (eg. actuator movement range) require to be managed in several parts. It is also necessary to make sure that the order of the values is correct. This would lead to a lot of duplicated code.

Solution Create a `Bounds` class encapsulating the concept of boundaries, which also allows to check the correct order of the bounds internally.

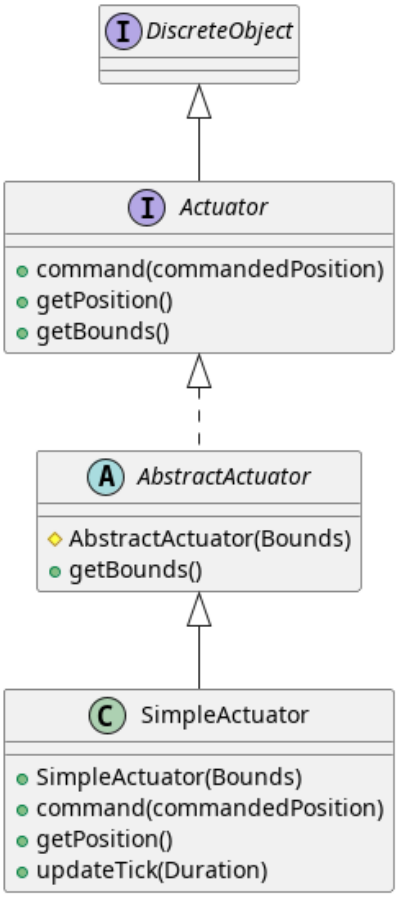


Figure 2.11: UML diagram of the actuator

Problem Reuse of code for the different devices.

Solution Create the concept of `ActuatedDevice`. Then, create an `AbstractActuatedDevice` which wraps an `Actuator`. It is then possible to create several different implementations and choose which actuator implementation to use, without having to create other device implementations that would be practically identical.

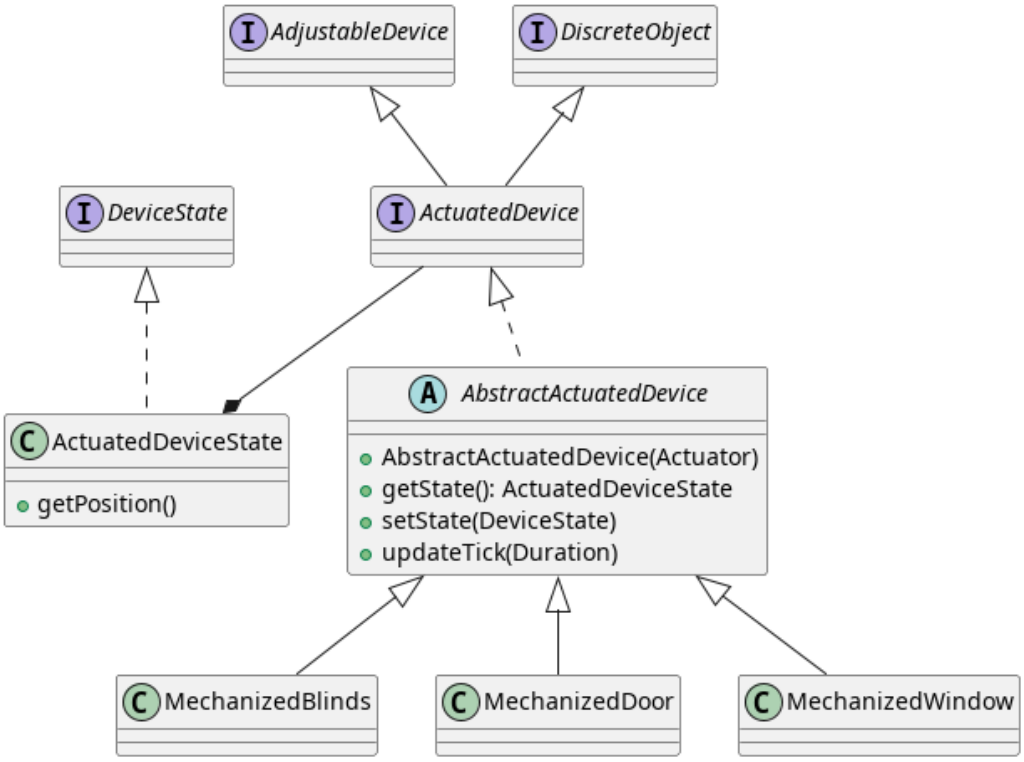


Figure 2.12: UML diagram of the ActuatedDevice and how the devices derive from it

Temperature Scheduler

The temperature scheduler is a component with the task of attaining the user's desired temperature.
It is external of the controller, but interfaces with it in the implementation. The scheduler has been separated between model, controller and view. Generics have been used to allow reusability.

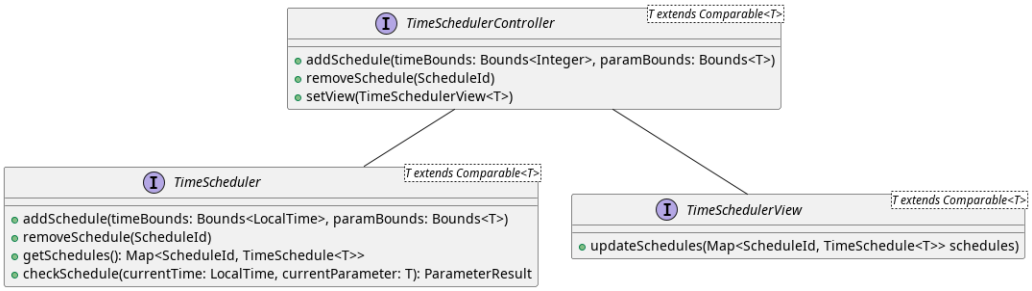


Figure 2.13: UML diagram of the scheduler overall structure

Scheduler Model

The scheduler model manages the storage of the schedules and the checking of the current parameter against the schedule.

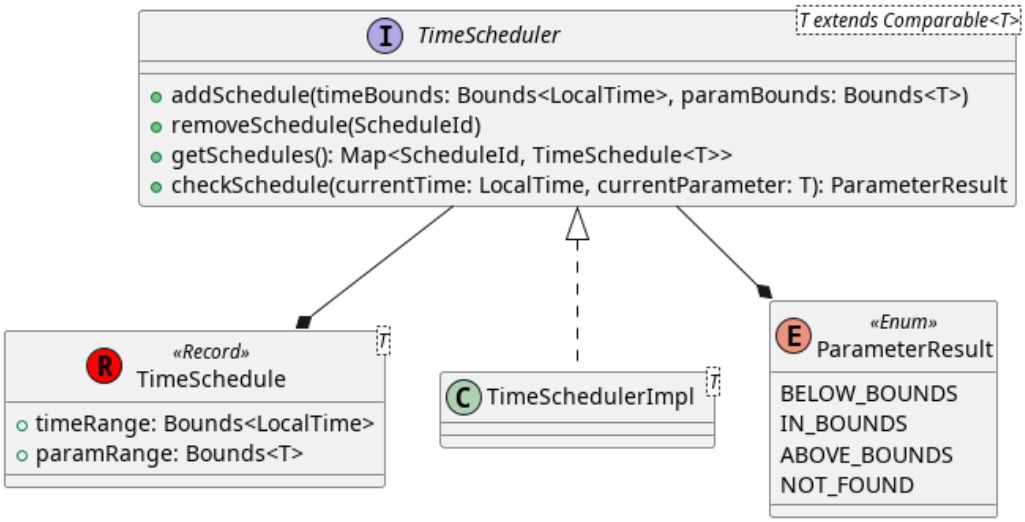


Figure 2.14: UML diagram of the scheduler model

Scheduler Controller

The scheduler controller handles the communication between the model and the view, and also implements the `DiscreteObject` interface to periodically ask the model if the target parameter is met. When the target parameter is not met, the scheduler will send a command to the domotic controller.

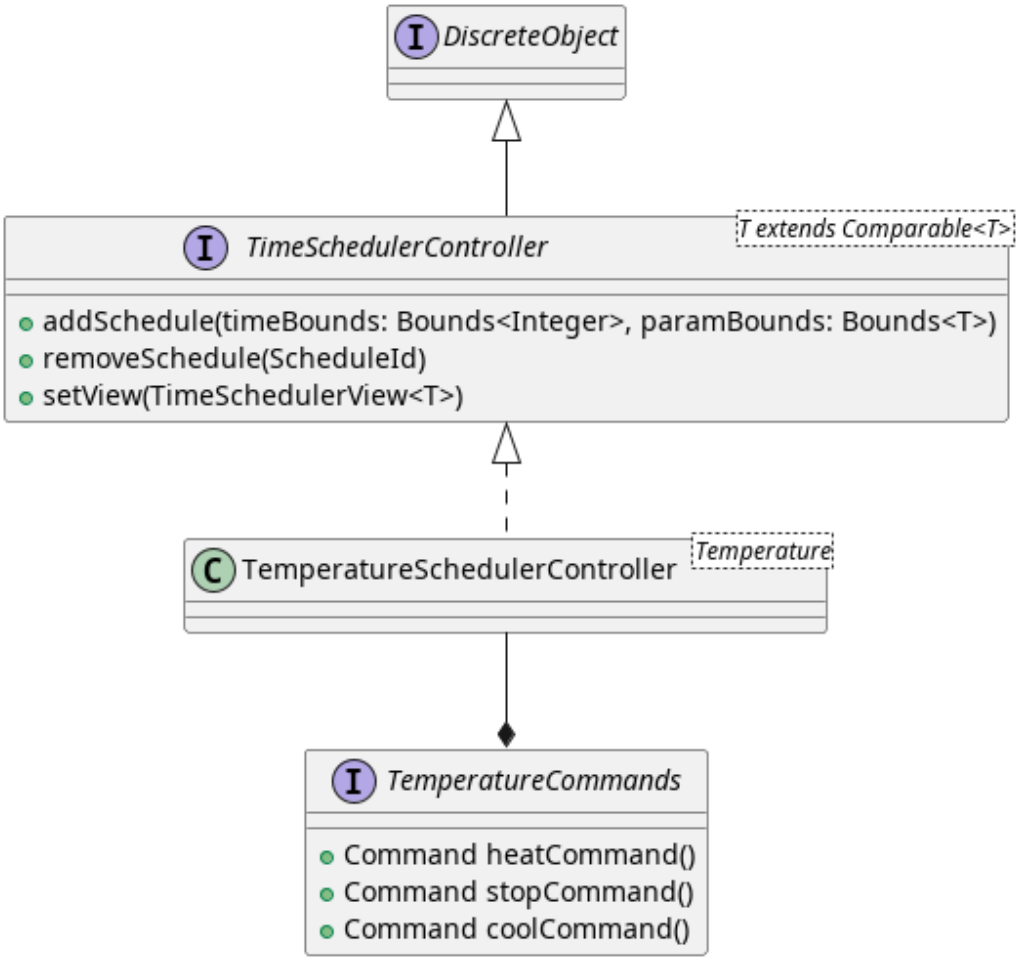


Figure 2.15: UML diagram of the scheduler controller

A `TemperatureCommands` interface has been created, following the `Strategy` pattern, to specifically deal with the control of the temperature.
Three separate commands are expected: heating, cooling, and stopping when the parameters are met. In the proposed implementation, the heaters and air conditioners are instructed to set their state to the minimum or maximum possible intensity, but for example, another implementation could take into account the difference between the current temperature and the desired temperature, and regulate the intensity proportionally.

A `Template` pattern has been used in `TemperatureCommand` to create this implementation, with an abstract method `Function` to reuse the same code for both heating and cooling. The template method is `setState`, as shown in figure Figure 2.16. This method is then used for both the heating and cooling functions.

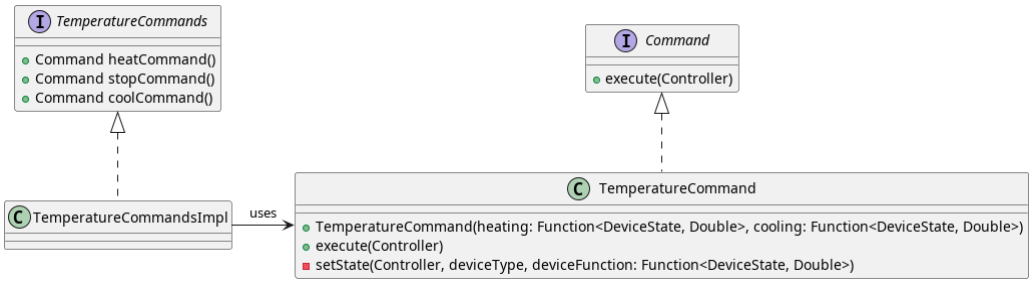


Figure 2.16: UML diagram of the implementation of TemperatureCommands with the use of a template method in TemperatureCommand

Scheduler View

The scheduler view has two tasks: allow the user to add/remove schedules, by calling the controller methods, and displaying the current schedules, when updated from the controller.

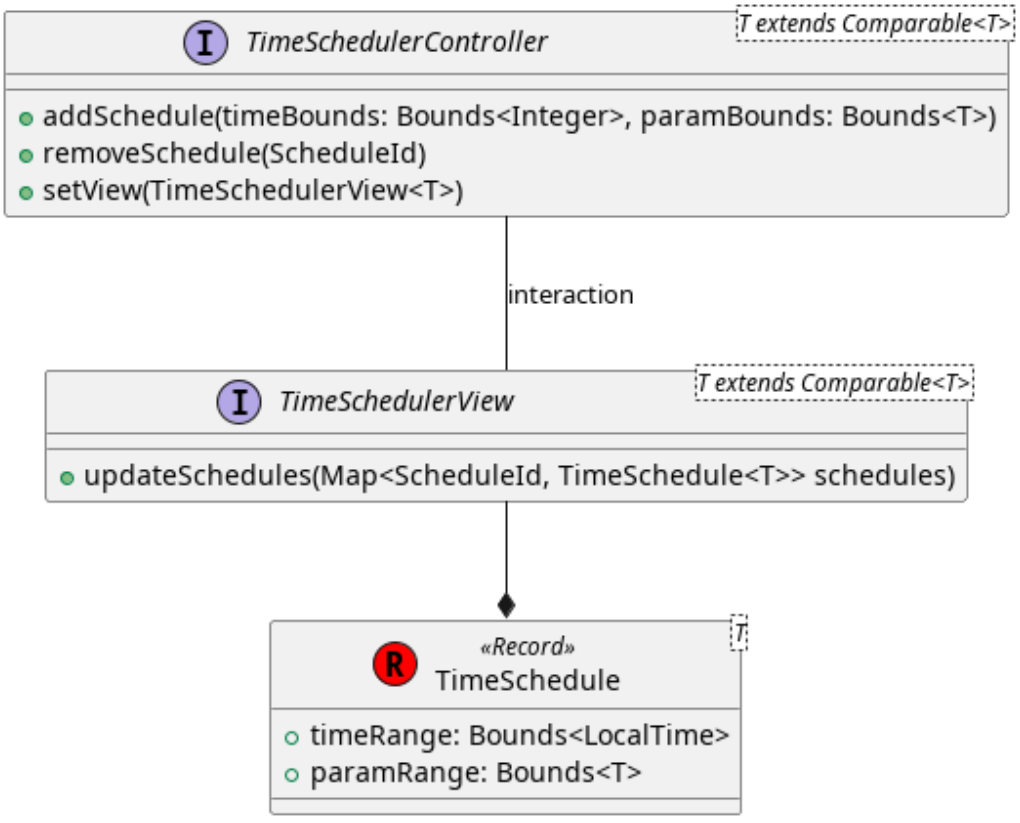


Figure 2.17: UML diagram of the scheduler view

Graphs

As with the scheduler, the implementation of the graphs has been divided between model, controller, and view components.

Graphs Model

The model has the simple task of storing the logged data, and providing the dataset upon request.

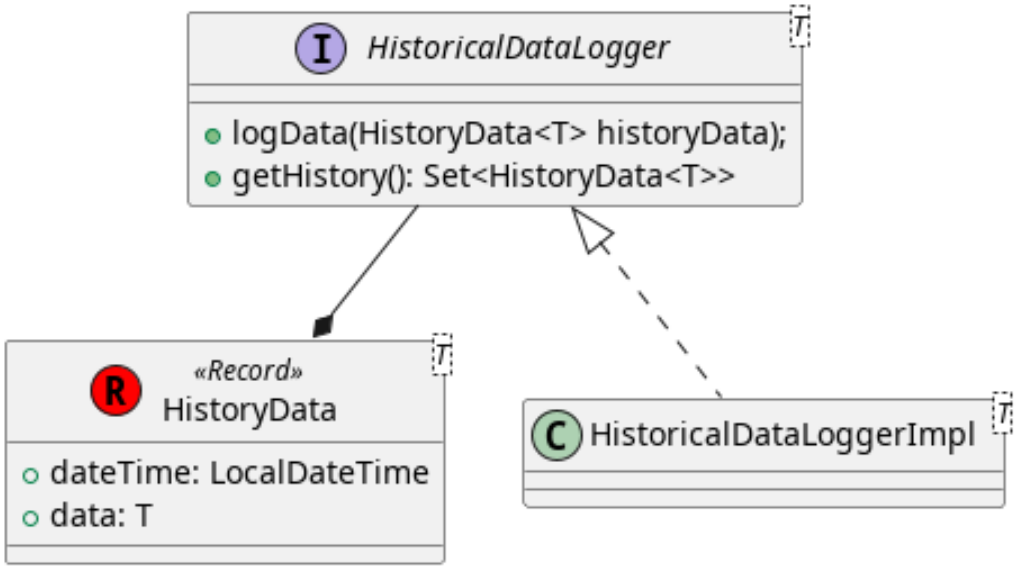


Figure 2.18: UML diagram of the graph model

Graphs Controller

The controller has the task of supplying the data to log to the model, and to refresh the view. It has been generalized with the use of a `Supplier` in an abstract implementation `AbstractLogger`.
The supplier decides how to retrieve the data to log, effectively becoming an abstract method, and the updateTick (where the data is logged) becomes a template method.

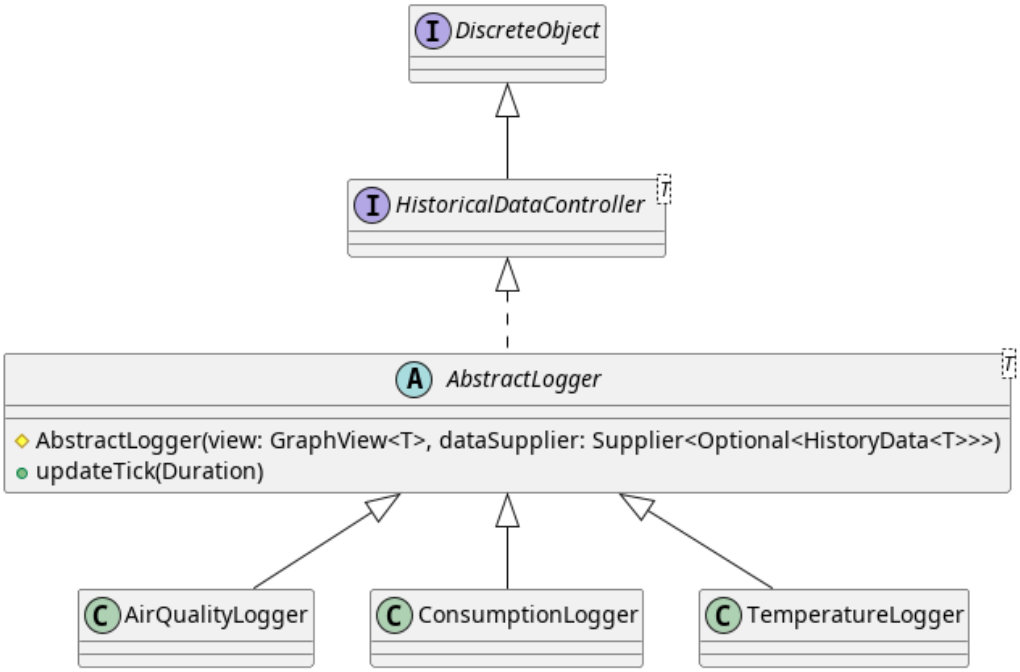


Figure 2.19: UML diagram of the graph controller

Problem We have to choose which thermometer to log.

Solution Use the first/any thermometer found. This solution would not work with multiple thermometers in different environments. In that case it would be better to choose a particular thermometer to log, and/or to have multiple logs for all the different thermometers.

Problem We have to choose how to log the data.

Solution I chose to log the data at each (simulation) hour for design simplicity. Could also have logged each tick, but it would have led to some pretty confusing graphs if the time rate were to be modified due to the time axis not being linear. In the future, the time between each log could be configured. It could also be possible to log more frequently but choose to display at a larger interval of time.

Graphs View

The graph view has the task of displaying the historical data, sent from the controller, to the user.

Figure 2.20: UML diagram of the graph view

Problem How to handle the display of the new data.

Solution I chose to refresh all the data each update for design simplicity. Another way would have been to send only the new data from the graph controller to the view, this would have required the view to decide whether to discard the oldest data record, and due to time constraints, I just decided to go with the simplest way. It also allowed me to better separate the logic and view, in fact it's the controller deciding what to display.

2.2.3 Alessandro Monticelli

Electrical Meter

When we first ideated HOMER, we agreed to build an Electrical Meter allows the user check the electrical consumption of various devices during the simulation.

Its main role is to retreive the state of the Outlets and calculate the instantaneous power absorption and the overall electrical consumption. It also implements an automatic safety check on the consumption as it will cut the power to the most consuming Outlet when the global power absorption reaches the limit of 4kW. The Electrical Meter acts as part of the Controller in the MVC pattern.

I implemented the ElectricalMeter interface in the ElectricalMeterImpl class, which updates its state in the updateTick method, where it retreives the state of the Outlets, performs the safety check, and the notifies the View through the view manager, that keeps the references to the elements of the View for the FXML loader, and implements the methods for retreiving User input and updating the View.

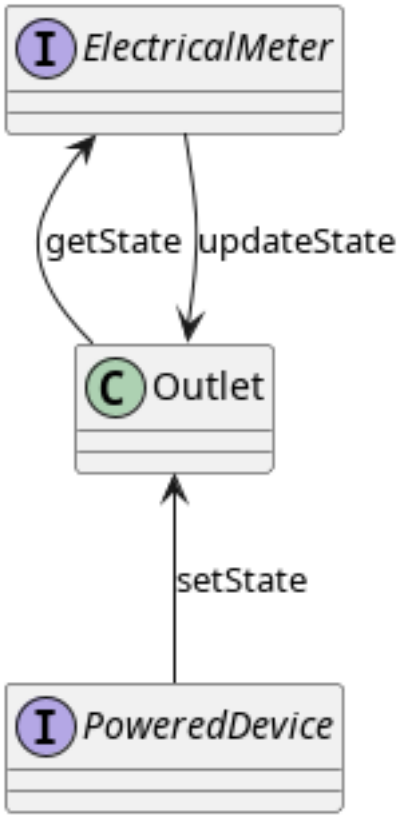


Figure 2.21: ElectricalMeter simple diagram

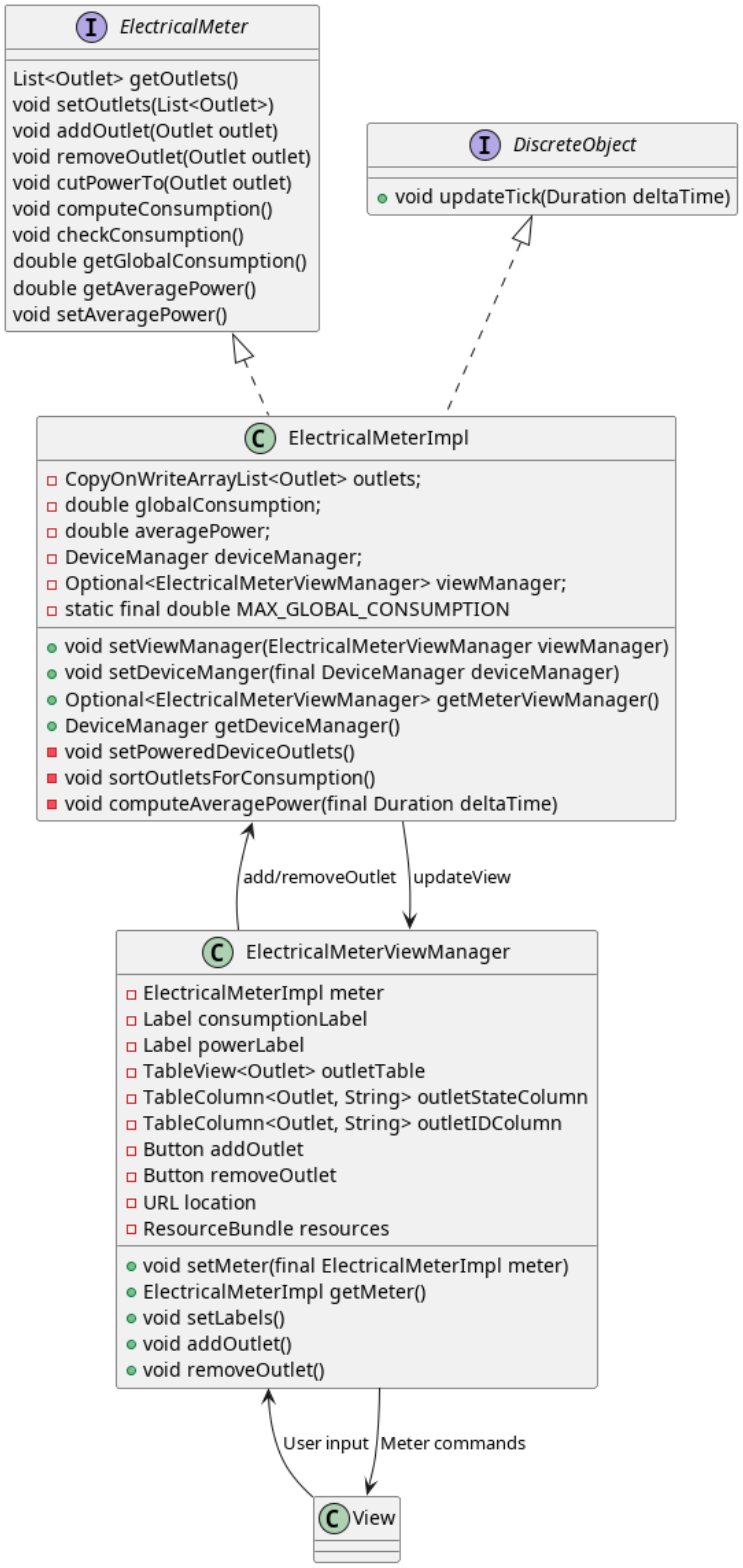


Figure 2.22: ElectricalMeterImpl detailed diagram

Outlets

As mentioned earlier, one of the key features of HOMER is the ability to control electrical equipment.
To accomplish this, I created the Outlet class, which is modeled as an AdjustableDevice.
The Outlet's state is represented by the OutletState class, which tracks the consumed power over time in Watt-hours (Wh) from any device that is plugged into the Outlet. Additionally, OutletState stores the minimum and maximum power consumption values for the device.

To maintain a realistic simulation, the Outlet class does not have knowledge of whether a device is plugged in or not. Instead, I designed the Outlet to let devices set their own power consumption and send that information to the Outlet when needed. To facilitate standardization of Outlets, I implemented the Factory pattern, which provides two standard options for creating Outlets: C-type Outlets (also known as Schuko), with a maximum power absorption of 3.5 kW, and L-type Outlets (the standard Italian outlet), with a maximum power absorption of 2 kW.

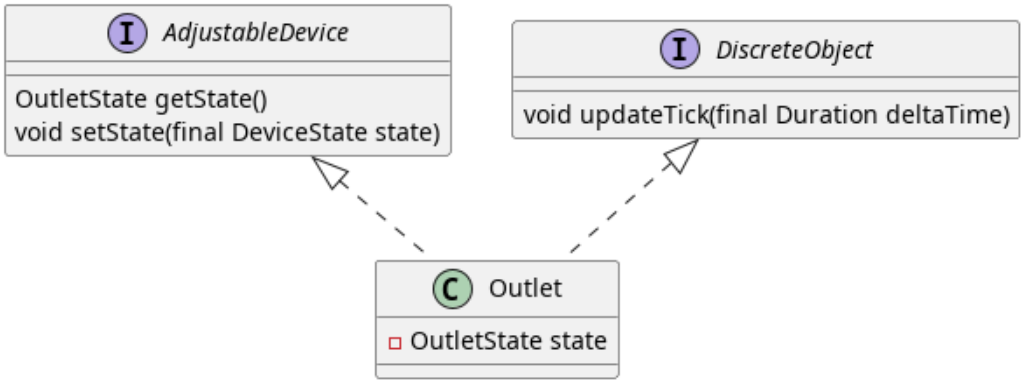


Figure 2.23: Outlet simple diagram

Powered Devices

In order to model power-consuming devices, I created the PoweredDevice and PoweredDeviceInfo interfaces.
The PoweredDevice interface includes methods for controlling the device's power consumption, retrieving the corresponding PoweredDeviceInfo object, and setting a new outlet for the device. Meanwhile, the PoweredDeviceInfo interface provides methods for retrieving the outlet that the device is plugged

into, as well as the minimum and maximum power consumption values for the device.

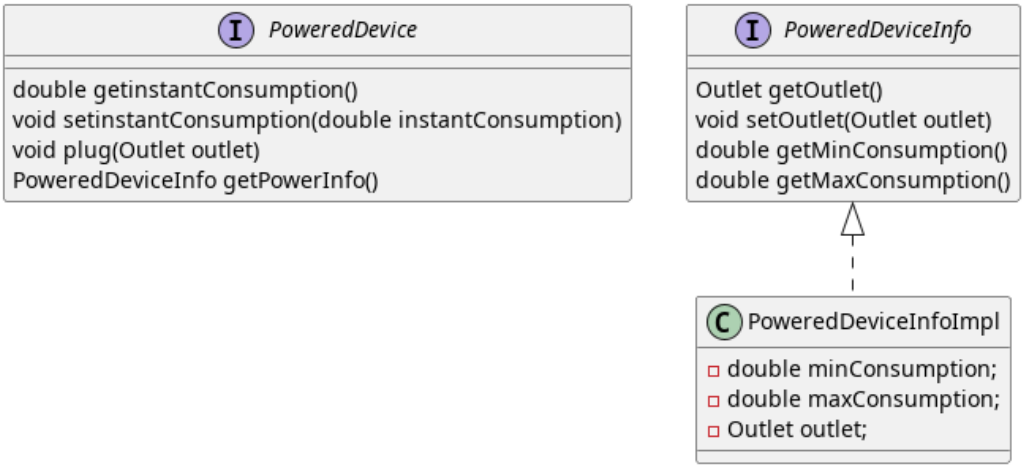


Figure 2.24: PoweredDevice and PoweredInfo diagrams

Lights

The Light class represent a very basic implementation of a ToggleableDevice and a PoweredDevice, its state is held in the LightState class, which has methods to retreive the state of the Light (ON/OFF).

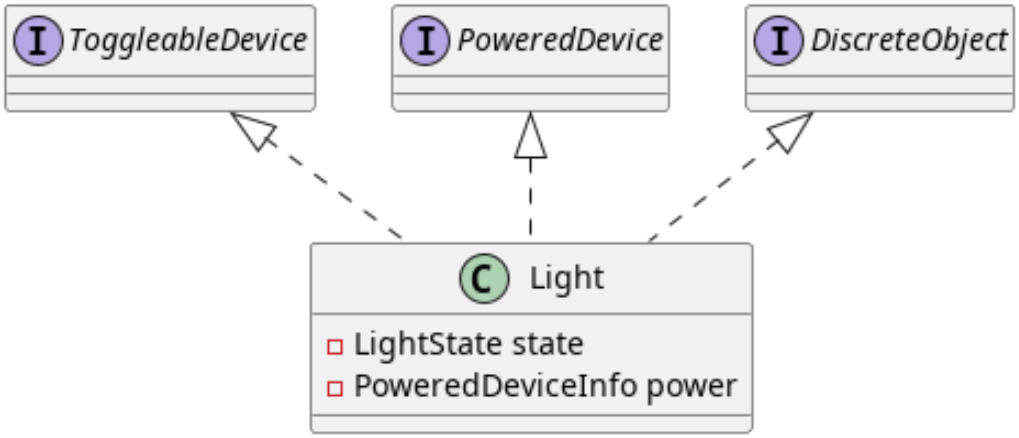


Figure 2.25: Light diagram

Chapter 3

Development

3.1 Automated testing

We used unit testing especially on the model to prevent regressions, and we tested observable behaviour only. Our unit test suite uses JUnit. We decided not to test UI components, since our UI is not the main focus, and other areas of the model would benefit more from more granular testing.

Tested classes:

- Temperature
- DurationConverter
- Outlets
- ElectricalMeter
- Light
- temperaturechangers
- Logger

3.1.1 Michele Ceccacci

- TemperatureTest
- AirqualityStateTest
- AbstractTemperatureChangerTest
- AirConditioningTest

- HeatingTest
- LoggerImplTest

3.1.2 Simone Magnani

- BoundsTest
- LimitTest
- ClockImplTest
- SimpleActuatorTest
- AbstractActuatedDeviceTest
- SimpleLockTest
- TemperatureSchedulerTest
- MechanizedWindowTest

3.1.3 Alessandro Monticelli

- ElectricalMeterTest
- OutletTest
- LightTest

3.2 Workflow

The first step in our process was domain analysis, and then the insights gained were applied by modeling the project's main interfaces by using UML. We used Git as our DCVS. We used gitflow and feature branching, with dev as our development branch, and main as our stable branch for releases. The workflow is based on pull requests and forks, allowing for asynchronous feedback when needed. Group meetings were only needed to discuss refactors.

3.2.1 Michele Ceccacci

- Utility classes
  - `homer.common.temperature.*`
  - `homer.common.time.DurationConverter`
- Model
  - `homer.model.temperaturechangers.*`
  - `homer.model.airquality.*`
- Logger
  - `homer.view.logger.*`
- JavaFx view
  - `homer.view.javafx.deviceview.*`
  - `homer.view.javafx.JFXDeviceViewer` Responsible for most of the implementation
  - `homer.view.javafx.*` Did most of the work on JFXDeviceViewer and everything in the other classes.
- Controller
  - `homer.controller.DeviceManagerImpl` Minor contributions to the implementation
  - `homer.controller.ViewManagerImpl` Contributed to most of the implementation

I contributed to the architecture of DeviceViewer and the Controller class.

I also have minor contributions to the controllerImpl class, namely sending updates on temperature and air quality changes. Whenever i noticed repeated code in the codebase i took charge of implementing a common solution, such as DurationConverter.

I also helped designing the architecture of Controller's ViewManager and DeviceManager.

3.2.2 Simone Magnani

- Utility classes
  - `homer.common.bounds.*` for the abstraction of boundary values
  - `homer.common.history.*` for the concept of data associated to a time
  - `homer.common.limit.*` for a common and reusable way to limit values
  - `homer.common.time.Clock*` for the concept of tracking custom time
- For the simulation core
  - `homer.core.*`
  - `homer.view.sim.*`
- For the implementation of the temperature scheduler
  - `homer.controller.scheduler.*`
  - `homer.model.scheduler.*`
  - `homer.view.scheduler.*`
- For the implementation of the graphs
  - `homer.controller.history.*`
  - `homer.model.history.*`
  - `homer.view.graph.*`
- Devices
  - `homer.model.lock.*`
  - `homer.model.thermometer.*`
  - `homer.model.environment.*` for the concept of physical environment
  - `homer.model.actuator.*` for the concept of actuator and the abstraction of actuated devices
  - `homer.model.blinds.*`
  - `homer.model.door.*`

  - `homer.model.window.*`

I have contributed to the establishing of the architecture eg. `Device` and `Controller` interfaces.

When I had some ideas that could be used by someone else eg. `Environment, Bounds, Limit`, I would open a pull request, which allowed everyone to be aware of it, and who could chime in and provide their feedback.

I have also contributed in minor parts to the entrypoint and `Controller` implementation, for the integration of my code, and `JFXDeviceViewer`.

3.2.3 Alessandro Monticelli

- APIs and Utility
  - `homer.api.PoweredDevice`
  - `homer.api.PoweredDeviceInfo`
  - `homer.api.PoweredDeviceInfoImpl`
- Controller
  - `homer.controller.electricalmeter.*`
- Model
  - `homer.model.lights.*`
  - `homer.model.outlets.*`
- View
  - `homer.view.javafx.sensorsview.ElectricalMeterViewManager`

I contributed to the architecture of the Electrical Meter and the simulation of electrical consumption (through the PoweredDevice interface, along with PoweredDeviceInfo).
I also contributed to the mathematical and theoretical modeling of the physics (e.g., functions for changes in temperature and electrical consumption over time) and helped establish the architecture of the application during the analysis phase.

3.3 Development Notes

3.3.1 Michele Ceccacci

- **Optional**s were used when the value could either be present or not
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/8779c6742ac88985e781856c26cf6815686ea9c7/src/main/java/homer/model/temperaturechangers/AbstractTemperatureChanger.java#L34
- **Streams** were used to access and modify data, especially data that used optional types
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/8779c6742ac88985e781856c26cf6815686ea9c7/src/main/java/homer/controller/ControllerImpl.java#L47
- **Records** to make data immutability more explicit, and to avoid reimplementing constructors/HashCode methods
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/8779c6742ac88985e781856c26cf6815686ea9c7/src/main/java/homer/controller/command/DisconnectDevice.java#L9
- **Lambda** functions and functional interfaces were used to make code more readable and often complemented streams
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/8779c6742ac88985e781856c26cf6815686ea9c7/src/main/java/homer/view/logger/LoggerImpl.java#L112
- **JavaFX** was used to develop the main view.
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/8779c6742ac88985e781856c26cf6815686ea9c7/src/main/java/homer/view/javafx/DisconnectDeviceButton.java#L11

3.3.2 Simone Magnani

- **Generics and bounded generics** used in several parts of my code
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/java/homer/common/limit/Limit.java#L23

- **Lambda expressions**
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/java/homer/controller/scheduler/TemperatureCommandsImpl.java#L12
- **Functional interfaces** such as `Function`, `Supplier`, `BiConsumer`
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/java/homer/controller/scheduler/TemperatureCommand.java#L45
- **Stream**
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/java/homer/model/scheduler/TemperatureScheduler.java#L56
- **Optional**
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/java/homer/controller/scheduler/TemperatureSchedulerContr
    java#L81
- **Record** for the implementation of `HistoryData`
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/java/homer/common/history/HistoryData.java#L12
- **JavaFX** for the implementation of the sim, scheduler and graph views
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/java/homer/view/sim/SimManagerViewFxImpl.java#L20
- **ControlsFX** for the range slider
  - https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/java/homer/view/scheduler/AddTemperatureScheduleViewFx.java#L28

To allow the air quality graphs FlowPane (placed inside of a ScrollPane) to automatically resize these two lines were used https://stackoverflow.com/a/36264110.

3.3.3 Alessandro Monticelli

- **Generics** Use of generics was fundamental for keeping a clean clode and reach high extensibility. https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/8779c6742ac88985e781856c26cf6815686ea9c7/src/main/java/homer/model/lights/Light.java#L21
- **Optional** https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/315b0f02d5e7a9dc0812e59ee195f5052ce3ddb1/src/main/java/homer/controller/electricalmeter/ElectricalMeterImpl.java#L37
- **Stream** The use of streams allowed me to keep my code clean and readable, I made large use of this Java feature, which I think is among the most interesting. https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/8779c6742ac88985e781856c26cf6815686ea9c7/src/main/java/homer/controller/electricalmeter/ElectricalMeterImpl.java#L109
- **Lambda** https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/315b0f02d5e7a9dc0812e59ee195f5052ce3ddb1/src/main/java/homer/view/javafx/sensorsview/ElectricalMeterViewManager.java#L100
- **JavaFX**
  - **Scene Builder** It proved to be a fundamental tool for building the view. I quickly learned to use it, and it allowed me to create a consistent UI without too much hassle.
  - **FXML** I really like the idea of creating the UI through a structured markup language. I think it's functional and clean. I also appreciate the way the view is managed through a controller class. https://github.com/progetto-oop-22-23/OOP22-HOMER/blob/main/src/main/resources/homer/view/javafx/sensorsview/ElectricalMeterVi
    fxml

Chapter 4

Final comments

4.1 Self evaluation and final comments

4.1.1 Michele Ceccacci

This project definitely improved my teamworking and software architecture skills. This was my first time participating in a greenfield project this big, and my first time taking architectural decisions that would carry over the whole software's lifecycle. In my previous work experience and open source contributions, i was more focused on adding new features to an existing product or bugfixing, rather than actual softare architecture. I was already proficient at git, but i still made some occasional mistakes. I think that working in a team without a more senior figure acting as lead definitely made me more self reliant, and allowed me to dig deeper into problems.

4.1.2 Simone Magnani

Although this was not my first project of such dimensions to be designed from scratch, it was the first one to be made in a team, and the first one where I had the awareness that certain design choices had a name and are commonly used. Altough the minimum requirements were met, I expected to reach further. I now feel to have grasped how git works and am comfortable in using it in collaborative projects. I will definitely carry over the knowledge I have acquired in architecture design and design patterns onto my future projects.

4.1.3 Alessandro Monticelli

I believe this course has been the most informative since the start of my academic path.
I thoroughly enjoyed attending the classes, and even though I had to miss some for personal reasons, I learned a lot from all three professors.
I appreciated the emphasis placed on the art of clean code and programming patterns.
The project itself was immensely helpful because I had never worked in a team or created a project of this size from scratch.
I learned many new things during these months of hard work and became very curious about the Gradle tool, which I think is really powerful. Although I was already proficient with Git, I had never used a strict workflow like the one we decided to follow at the start of the project, and I found it to be incredibly helpful. Overall, I think the general workflow allowed us to work in a highly efficient manner.

Appendix A

User guide

The main view is divided in two sections horizontally.
Above there are 3 tabs: DEVICES, SCHEDULER and GRAPHS.
At the bottom there is the SIMULATION information and control section, which reports the simulation time and the time rate, and allows the user to pause and resume the simulation, or change the time rate.
In the DEVICES tab, there is an add window button. After selecting a device, it will be created (and will be inserted at the bottom). Each device has a remove device button used to disconnect it, and may have some kind of utility to set its state.
In the SCHEDULER tab, the user can select the desired temperature range in time of the day intervals (schedule).
No interval can be overlapping with another. The application will prevent the user from trying to do so, and notify them.
When a schedule is present, the domotic controller will try to attain the desired parameters, provided that:

- A thermometer is connected
- For heating: a heater is connected
- For cooling: an air conditioner is connected

In the GRAPHS tab, the user can visualize temperature, global consumption, and air quality parameters, logged at intervals of 1 hour.
In order to visualize temperature and air quality parameters, either a thermometer or an air quality sensor, respectively, should be connected.

The electrical meter view is relatively basic, consisting of a table with two columns.
The first column indicates the name of the Outlet, while the second column

displays its State (i.e., the power absorption in watt-hours).
At the bottom of the table, there are two buttons: one to add a new outlet and another to remove the most recently added outlet (the adding and removal process is organized in a LIFO structure).
Additionally, the state column can be sorted in either ascending or descending order by clicking on the column heading.

Appendix B

Lab Assignments

B.1 michele.ceccacci@studio.unibo.it

B.2 simone.magnani7@studio.unibo.it

- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=117044#p173103
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=118995#p175305
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=119938#p176561
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=121130#p177406

B.3 alessandr.monticell4@studio.unibo.it

- Laboratorio 03: https://virtuale.unibo.it/mod/forum/discuss.php?d=112846#p168116
- Laboratorio 04: https://virtuale.unibo.it/mod/forum/discuss.php?d=113869#p169459
- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=114647#p169801
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=115548#p171347

Bibliography