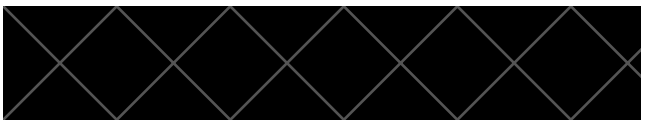


Python Smart Home Coursework

Scenario

A fictional UK-Finnish company, Alikoti, develops eco-friendly smart home technologies. Since 2015, they have focused on fully integrated smart home solutions, establishing a strong foundation in the industry. In 2024, they expanded their offerings to include standalone smart home devices, such as smart lights and smart fridges, designed to be supported by third-party software. Last year, the company decided to build its own smart home app to unify its ecosystem and enhance user experience.

Alikoti has asked you to develop a prototype GUI application with two levels of functionality. The first level, designed for individual home users, must provide a working application that manages all smart devices within a single property, allowing users to monitor and control devices efficiently. The second (challenge) level, intended for property managers, must provide a working application to remotely manage smart devices across multiple homes. Each home must have its own dedicated interface, enabling property managers to monitor and control devices independently for each home.

Tasks

Complete the tasks in the given order. The challenge task can be considered optional and should only be attempted after successfully completing Tasks 1–5 if you feel confident extending the project. Each task includes core requirements; more difficult requirements are available for earning higher marks.

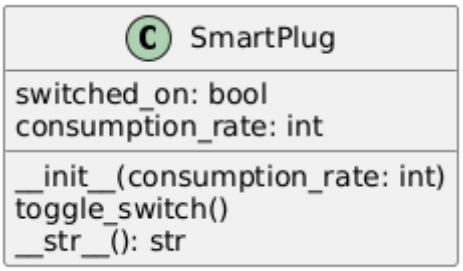
Testing for each task will be used in part of the demo session, where you must demonstrate your implementation's functionality. Your test functions should be well-structured and verify key features before submission.

When submitting, compress the folder containing all necessary Python (**.py**) files into a single **.zip** file. Use multiple files only if required—avoid unnecessary file splitting. Do not include extra files. **Do not submit any other file format.**

Submit your **.zip** file via the Moodle link under **Assessments** titled **"SUBMISSION: Item 6 — Coursework."**

Task 1 — SmartPlug Class [10 marks]

Write a `SmartPlug` class that satisfies the requirements below. Use the class diagram below as your starting point:



Each `SmartPlug` object should have two attributes: `switched_on` and `consumption_rate`. The `switched_on` attribute tracks whether the plug is on or off and should be either `True` or `False`. The `consumption_rate` attribute represents the power consumption in watts and must be an integer between 0 and 150 (inclusive).

The constructor must accept only one parameter, `consumption_rate`, and should set the `switched_on` attribute to `False` by default. A `toggle_switch()` method should be implemented to toggle the value of `switched_on` between `True` and `False`. This method should not take any parameters. The `__str__()` method must be implemented to return the string representation of a `SmartPlug` object. When a `SmartPlug` object is printed, if it is switched on, it should display:

```
SmartPlug is on with a consumption rate of 120
```

with "120" replaced by the actual consumption rate. If it is switched off, it should display:

```
SmartPlug is off with a consumption rate of 120
```

Ensure the class follows these requirements exactly.

Harder: Update your `SmartPlug` class so that the value of `consumption_rate` cannot be updated to anything outside the specified range. You are free to implement this in

various ways, for example by making some attributes private, adding extra class or instance variables and/or raising exceptions (see lecture 14). Remember to add getters and setters if required. Note that all exceptions, if any, must be handled by the test functions as well as the front end of your app (see later tasks).

Testing

To begin testing the core functionality of the `SmartPlug` class, design and implement a `test_smart_plug()` function that systematically verifies initialisation, toggling, and updating. This function should first instantiate a `SmartPlug` object with a `consumption_rate` of 45 watts. Immediately after creation, print its state to verify that it has been initialised correctly, ensuring that the `switched_on` attribute is `False`. The expected output at this stage should be:

```
SmartPlug is off with a consumption rate of 45
```

Next, test the plug's ability to toggle its state. Call the `toggle_switch()` method to turn the `SmartPlug` on, then print its state again to confirm the update. The expected output should now be:

```
SmartPlug is on with a consumption rate of 45
```

Following this, update the `consumption_rate` to 75 watts and print the `SmartPlug` once more. This step ensures that valid updates to the consumption rate are properly applied. The expected output at this point should be:

```
SmartPlug is on with a consumption rate of 75
```

Finally, toggle the `SmartPlug` off and print its state again. This ensures that the `toggle_switch()` method correctly turns the device off regardless of previous changes. The expected output should now be:

```
SmartPlug is off with a consumption rate of 75
```

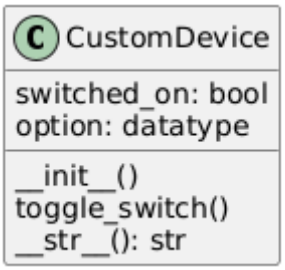
To verify the harder requirements, the `test_smart_plug()` function should extend into a second phase focusing on constraint enforcement and error handling. First, attempt to set `consumption_rate` to an invalid value, such as -10 watts and 200 watts, and catch any exceptions that occur. If implemented correctly, the program should reject these values, and the attribute should remain unchanged. Print a message confirming that invalid updates were prevented, and print the `SmartPlug` object to show its current (unchanged) state.

Finally, verify that the `SmartPlug` class cannot be instantiated with an invalid `consumption_rate`. Attempt to create two `SmartPlug` objects, one with -5 watts and another with 160 watts, and confirm that exceptions are raised, preventing their creation. If an exception occurs, print a message stating that invalid initialisation was blocked.

Task 2 — Custom device [15 marks]

Create two classes representing the custom devices listed in Table 1. To determine which devices to implement, identify the **last two unique digits** of your student number. For example, if your student number is **2635111**, the last two unique digits are **1 and 5**, so you would implement the devices corresponding to those numbers in Table 1.

Use the class diagram below as your guide and remember to change “CustomDevice”, “option” and “datatype” to the values specified in Table 1.



Each custom device has two attributes: `switched_on`, which functions the same way as in the `SmartPlug` class from Task 1, and an option attribute listed in the “Option attribute” column of Table 1. The valid range for this attribute is specified in the “Permitted Values” column, and any values outside this range must be rejected. The default value for the option attribute is provided in the “Default Value” column and must be set by the constructor. The `switched_on` attribute must always default to `False`, just like in the `SmartPlug` class.

Each class must include a `toggle_switch()` method that allows the user to switch the device on or off. When called, this method should change the value of `switched_on` between `True` and `False`, ensuring that the device state updates correctly.

To manage the option attribute, each class must provide a mechanism for setting and getting its value. This mechanism must allow the user to update the option attribute, and there must be a way to retrieve its current value. The implementation of these mechanisms is flexible and may use methods, properties, or another suitable approach.

Additionally, the class must implement a `__str__()` method that returns a formatted string that includes the device name, its on/off status, and the current value of the option attribute. For example, a `SmartLight` should display:

```
SmartLight is off with a brightness of 50
```

Table 1. Each device has a unique option attribute, which must be set within a defined range of permitted values. The default value for each attribute is specified in Table 1 and must be assigned upon initialisation.

Digit	Custom device name	Option attribute	Permitted values	Default Value
0	`SmartLight`	`brightness`	Percentage (1–100)	`50`
1	`SmartFridge`	`temperature`	1, 3 or 5 degrees Celsius	`3`
2	`SmartHeater`	`setting`	Whole numbers between 0–5	`2`
3	`SmartTV`	`channel`	Whole numbers between 1–734	`1`
4	`SmartSpeaker`	`streaming`	Amazon, Apple, or Spotify	`Amazon`
5	`SmartDoorBell`	`sleep_mode`	Boolean values	`False`
6	`SmartOven`	`temperature`	0–260 degrees Celsius	`150`
7	`SmartWashingMachine`	`wash_mode`	Daily wash, Quick wash, or Eco	`Daily wash`
8	`SmartDoor`	`locked`	Boolean values	`True`
9	`SmartAirFryer`	`cook_mode`	Healthy, Defrost, or Crispy	`Healthy`

Harder: Modify your custom device class to ensure that the option attribute cannot be set to an invalid value outside its permitted range, as defined in Table 1. Implement a mechanism to enforce this constraint (see the challenge section of Task 1 for additional guidance). To improve code reusability and maintainability, you may consider using inheritance. Analyse the functionality shared between smart devices and plugs to determine what should be included in a base class within an inheritance hierarchy. For example, a base class could include common elements such as the `switched_on` attribute and the `toggle_switch()` method. Individual custom device classes could then extend this base class, adding their own `option` attribute while ensuring that allowed values align with Table 1.

Testing

To verify the correct implementation of the custom device classes, write a function called `test_custom_device()` that tests two distinct custom devices, one of each type. Begin by creating instances of the custom devices and print their initial states. This should show that the `switched_on` attributes default to `False` and that the option attributes are correctly set to the default values, as specified in Table 1.

Next, toggle the switch for both devices using the appropriate method and print their states again. This should confirm that toggling correctly switches the `switched_on` attribute between `True` and `False`. Following this, update the option attribute of each device by setting it to a new valid value from Table 1. Print both objects again to verify that the changes have been applied correctly.

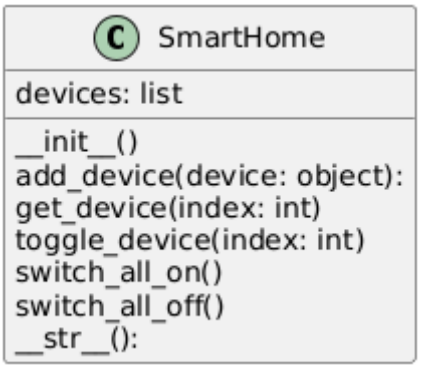
For the harder requirements, test whether the custom device classes properly enforce constraints on the option attribute. Attempt to set the option attribute of each device to an invalid value—one that falls outside the permitted range in Table 1. The test should confirm that the class prevents the change or raises an exception, ensuring that only valid values are accepted.

Task 3 — `SmartHome` Class [15 marks]

In this task, you will create a `SmartHome` class that acts as a centralised controller for managing multiple smart devices. The `SmartHome` class will handle a collection of devices, including the `SmartPlug` objects created in Task 1 and the two custom devices from Task 2, which are chosen based on the last two unique digits of your student number.

The `SmartHome` class must allow users to add devices to the collection, remove devices, retrieve specific devices, and toggle their on/off states. This class is responsible for controlling and interacting with all of the devices in the smart home system.

In the core task, focus on managing these devices within the `SmartHome` class and implementing the required functionality for adding, removing, retrieving, and toggling devices. Use the class diagram below as your starting point.



The `SmartHome` class must store a collection of `SmartPlug` objects and custom smart devices from Table 1. It should support multiple instances of each device type, allowing users to add various smart devices to the system.

The class must include an `__init__()` constructor to initialise the collection of devices. Users must be able to add new smart devices to the system using an `add_device()` method. To retrieve a specific device, the `get_device(index)` method should return the corresponding object from the collection. To control devices, they must be able to toggle a specific device's `switched_on` attribute by providing its position in the collection. Additionally, the `SmartHome` class should provide functionality to turn all devices on or off at once using the `switch_all_on()` and `switch_all_off()` methods.

When printed, the `SmartHome` should display a summary of the number of devices it contains, followed by details of each device. The `__str__()` method should generate an output in the following format:

```
SmartHome with 3 device(s):
1- SmartPlug is on with a consumption rate of 120
2- SmartOven is on with a temperature of 200
3- SmartPlug is off with a consumption rate of 90
```

Harder: Enhance the `SmartHome` class by introducing a `max_items` limit to restrict the number of devices that can be added to the collection. Once the limit is reached, any further attempts to add devices should be rejected with appropriate error handling.

Implement a `remove_device(index)` method that allows for the deletion of a device from the collection by specifying its position. If a valid index is provided, the device should be removed, and the list should update accordingly. If an invalid index is given, the method should raise an appropriate error.

Implement an `update_option(index, value)` method that allows modification of the option attribute of a device by specifying its position in the collection and a new option value. The attribute being updated will depend on the device type; for example, a `SmartPlug` will update its `consumption_rate`, while a `SmartLight` will change its `brightness`. Introduce error handling within the `SmartHome` class to ensure that all modifications remain within valid limits.

Testing

To evaluate the functionality of the `SmartHome` class, implement a test function called `test_smart_home()`. This function will systematically verify that devices can be added, toggled, updated, and that all constraints are properly enforced.

To begin testing the core functionality, create a `SmartPlug` and one of each of the two custom devices from Task 2. Initialise a `SmartHome` object and add all three devices to it. Print the state of the `SmartHome` to confirm that the devices have been correctly added.

Retrieve each device using the `get_device(index)` method and verify that the returned object matches the expected device at that position. Print the retrieved device details to confirm correctness.

Toggle each device individually using the `toggle_device(index)` method and print the `SmartHome` again to ensure that their `switched_on` attributes have been updated as expected.

Next, use the `switch_all_on()` method to turn on all devices at once, followed by `switch_all_off()` to turn them all off again. Print the final state of the `SmartHome` to verify that all devices have been switched off successfully and that their option attributes remain unchanged.

For the harder requirements, verify that the `max_items` limit is enforced by attempting to add more devices than allowed. The test should confirm that excess devices are rejected and that appropriate error handling is in place.

Modify the option attribute of both the `SmartPlug` and both the custom devices using `update_option(index, value)`. First, assign a valid value from Table 1 and print the SmartHome to confirm that the change was successfully applied. Then, attempt to set the attribute to an invalid value and check that the update is rejected.

Test the `remove_device(index: int)` method to ensure that devices can be deleted correctly from `SmartHome`. First, delete a device from the collection and print the updated state of `SmartHome` to verify that it has been removed. Then, attempt to remove a device using an invalid index (e.g., a negative index or an index beyond the number of devices in the collection). The method should raise an appropriate error, preventing unintended deletions.

Test the `update_option(index, value)` to ensure that error handling is working as expected by trying to assign or update values outside the permitted range for each device's option attribute. If properly implemented, the `SmartHome` class should prevent these invalid updates, either by raising an exception or maintaining the current value.

Finally, print the `SmartHome` state one last time to verify that all changes, including toggling, option updates, and deletions, have been correctly applied and that invalid operations were successfully blocked.

Task 4 — `SmartHomeApp` [10 marks]

In this task, you will develop a Graphical User Interface (GUI) for the `SmartHomeApp` using Tkinter. The `SmartHomeApp` class will serve as a graphical interface for interacting with an instance of `SmartHome`, allowing users to view their smart devices in an organised display. The focus of this task is on designing the layout of the GUI, ensuring that devices and their statuses are displayed correctly. In this task, buttons will function as placeholders without performing any actions. Interactive functionality, such as toggling devices and modifying settings, will be implemented in Task 5.

The `SmartHomeApp` class should initialise and store an instance of `SmartHome`, which contain one of each of the two custom devices from Task 2 and a SmartPlug from Task 1 These custom devices, including the smart plug, should be created with default values, eliminating unnecessary console interaction, as these setups have already been tested in

previous tasks. These devices must be added to `SmartHome` using the appropriate methods from previous tasks.

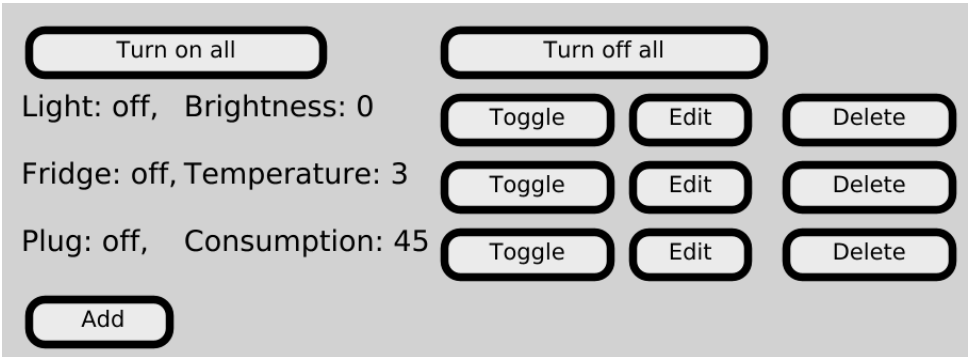
The class should then create a Tkinter-based GUI layout that visually represents the smart home system, ensuring that each device is displayed with its name (i.e. type) and current status (on/off state or attribute value). The figure below shows the recommended GUI layout and required widgets. Since this task focuses solely on layout design, interactive functionality is not required at this stage. The goal is to correctly display all devices and their statuses within the interface. The ability to toggle devices and modify settings will be implemented in Task 5.

Example Setup

If a your number is **2635011** (the last two unique digits are **0 and 1)** you should create the following three devices:

- A `SmartLight`
- A `SmartFridge`
- One `SmartPlug` (with a consumption rate of 45W)

These devices should be stored in the `SmartHome` and then displayed in the GUI, which should look something like this:



Harder: Enhance the `SmartHomeApp` to ensure that the GUI layout remains clear and readable regardless of the number of devices displayed or variations in text length. The interface should be able to adapt dynamically when displaying a large number of devices, preventing clutter or misalignment while maintaining readability.

Additionally, consider how devices with wider or narrower text labels affect the layout. Ensure that the display does not break or become difficult to interpret when different device names or option values vary in length. The solution should improve the scalability

and readability of the interface without requiring major structural changes when new devices are added.

The approach to solving these challenges is left open-ended, allowing flexibility in how students design their GUI to handle these variations effectively.

Testing

To ensure that the `SmartHomeApp` is correctly implemented, write a test function called `test_smart_home_system()`. This function should guide incremental testing, focusing on visual inspection to confirm that the interface is correctly structured and that devices are properly displayed.

Before creating the GUI, your program should verify that `SmartHome` contains the expected devices by printing it to the console. The output should match the list of devices added to `SmartHome`, confirming correct initialisation. If any elements fail to display in the GUI, debugging should start by checking this output to ensure that the `SmartHome` instance has been properly set up.

The Tkinter window should open without errors, displaying all expected elements. The GUI layout should be structured so that labels and buttons are aligned properly.

For the harder requirement, visually inspect whether the GUI properly adapts when larger numbers of devices are added or when text varies in width. Devices should remain clearly displayed, and no text or buttons should overlap or become unreadable.

Task 5 — GUI interactivity [15 marks]

In this task, you will add interactive functionality into the `SmartHomeApp` GUI, ensuring that user actions modify both the `SmartHome` instance and the frontend interface. The `SmartHomeApp` class should serve as a graphical interface for `SmartHome`, meaning every GUI action should trigger an equivalent method in `SmartHome`, keeping the interface and system in sync. Interactivity is introduced here in progressive levels of complexity to allow for step-by-step implementation.

The first level introduces basic UI controls that update both the `SmartHome` instance and the interface. Any changes made in the GUI should be reflected in the devices managed by `SmartHome`. For example, clicking "Turn on all" or "Turn off all" should iterate through the collection of smart devices in `SmartHome`, updating their `switched_on` state accordingly, while immediately displaying these changes in the GUI. Similarly, clicking a

device's "Toggle" button should update its `switched_on` state within `SmartHome` and reflect the updated status in the interface. These interactions ensure that the backend logic and user interface remain fully synchronised.

The next level introduces intermediate functionality, where user actions modify the collection of devices managed by `SmartHome` and trigger a UI update. Clicking "Delete" should remove the selected device from `SmartHome`, ensuring that it is no longer part of the system. The UI should then be dynamically redrawn to reflect the updated device list. This ensures that deleted devices are completely removed from both the SmartHome instance and the GUI, without requiring a restart.

In the final level, interactions require additional UI elements and system updates that modify the state of individual devices or the collection managed by `SmartHome`. Clicking "Edit" should open a new window where users can modify a specific device's settings, such as brightness or consumption rate. Once confirmed, these changes should be applied directly to the device within `SmartHome`, ensuring that its updated state is accurately reflected in both the system and the GUI.

Clicking "Add" should open a new window that allows users to configure and add a new `SmartPlug` or custom device. Once created, the new device must be added to the collection in `SmartHome`, modifying the managed set of devices. The GUI must then dynamically update to display the newly added device, ensuring that the system remains synchronised without requiring a refresh.

By structuring interactivity in this progressive difficulty sequence, the simpler UI updates can be implemented and tested first before introducing more complex modifications that involve both SmartHome synchronisation and dynamic UI rendering.

Testing

The `test_smart_home_system()` function should systematically verify each interactive component, ensuring that changes are correctly reflected in both the `SmartHome` instance and the GUI. Begin by testing the basic UI controls. Click "Turn on all" and "Turn off all", confirming that the UI updates correctly to reflect the change in device states. Next, test individual "Toggle" buttons, ensuring that each device's state visually changes when clicked.

Proceed to test the delete functionality by clicking "Delete" on a device. Confirm that the deleted device is also removed from the GUI, with the display adjusting dynamically. For editing functionality, click "Edit", modify the device's attributes. Ensure that the GUI correctly displays the modified values to reflect the updated device settings. Finally, test

adding new devices by clicking "Add", configuring a new `SmartPlug` or custom device, and verifying that it is successfully added to the GUI without requiring a restart.

Challenge Task [25 marks]

This challenge extends the `SmartHomeApp` by introducing a new `SmartHomesApp` class, which acts as a centralised hub for managing multiple smart homes within a single system. Instead of controlling just one smart home, users should be able to view, add, remove, and modify multiple smart homes from a structured interface. Each smart home operates independently, but the `SmartHomesApp` provides access to all their devices, allowing users to switch between homes and manage their settings efficiently.

A collection of smart homes must be managed within the `SmartHomesApp`, enabling users to select and interact with individual smart homes while maintaining an overview of all available homes. The system must include a graphical user interface (GUI) that serves as a main menu, allowing users to navigate between smart homes, modify their settings, and interact with their devices. This menu should provide clear summary information for each smart home, such as the number of devices in the home and how many of them are currently switched on. Presenting this information at a glance will allow users to quickly assess the status of their smart homes before selecting one to manage. While this challenge primarily focuses on implementing this functionality in the frontend, a more complete design could involve structuring the backend to handle a collection of SmartHome instances, ensuring robust data management.

To support persistent data management, smart home configurations should be stored in a file. Users should then be able to add new smart homes, which will be incorporated into the system, and remove existing ones when they are no longer needed. The system should allow users to retrieve and manage a specific smart home, providing access to its settings, device configurations, and smart plug management.

To ensure that system state is maintained across sessions, all changes must be saved to a file when the user exits the application so that smart homes and their devices are restored upon reopening it. CSV is the recommended storage format, though JSON or other structured formats can also be used to ensure user preferences and configurations remain intact.

Testing should verify that the system correctly handles adding, removing, and retrieving smart homes, as well as ensuring that data is reliably saved and reloaded. The design of the user interface is open-ended, allowing flexibility in how multiple smart homes are

displayed and managed. However, the interface must provide a clear, logical way to switch between smart homes, ensuring that all key functions remain easily accessible.

Demonstration & Mark Allocation

You need to demonstrate your program to a member of staff in your Programming practical session, timetabled on 20th or 21st March.

You will first complete an exercise that tests your comprehension of your submitted code, and then a member of staff will mark your program and give you feedback. All the marks for the assignment will be awarded at the demonstration. You must attend and complete both parts; failure to do so will result in your work being recorded as zero (and possibly as a non-submission). Demonstrating your program late may result in your mark for the assignment being capped under University rules. Late demonstrations will take place in practicals during the first week of Teaching Block 2, and you must demonstrate your work by 4th April 2025, or it will be recorded as a non-submission.

Formal written feedback and your functionality and code quality marks will be sent to you via email immediately after your demonstration has been completed. If you do not receive this email, then your mark may not have been recorded, and it is your responsibility to inform Nadim if this happens.

Code comprehension [10 marks]

You will need to demonstrate your understanding of the code you have submitted by means of a computer-based cloze exercise, similar to those given in the group activities in the practical classes. The system will display your code with comments removed and with 10 symbols (keywords, variables, function names, operators, etc.) replaced by blanks. You will have 10 minutes to fill in these blanks with the correct symbols from your program. In the rare case that your submission is too short for the system to generate 10 blanks, then a smaller number will be presented and your mark for this exercise will be limited. If your score for this cloze exercise leads the module team to suspect that you did not write the complete program yourself, your work will be subject to an academic misconduct process. This may result in your assignment mark being reduced to zero.

Functionality and code quality [90 marks]

Each task is assessed on both functionality and code quality. Functionality will be assessed through the testing specified in each task. Assessment of code quality assessment will involve evaluating its structure, readability, and adherence to object-oriented principles.

Task	Functionality	Code	Total Marks
Task 1 – SmartPlug Class	7	3	10
Task 2 – Custom Device classes	11	4	15
Task 3 – SmartHome Class	11	4	15
Task 4 – SmartHomeApp GUI	7	3	10
Task 5 – GUI Interactivity	11	4	15
Challenge Task	21	4	25

Functionality will be assessed through the test cases outlined in each task. Code quality will be evaluated based on maintainability, clarity, and adherence to good programming practices.

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]