Software Design Patterns

Continuous Assessment 2 – CA2

Nicolas D’Alessandro Calderon

Design and Responsibilities Assignment Patterns

Question 1

a) Before choosing a specific pattern, I will analyze **the type** of problem that we are facing here. The learning material distinguishes between **two important kinds** of patterns for this unit:

- **Responsibility Assignment Patterns** (WHO should do something): Helps in decide which class should take on which responsibility. Useful when assigning logic on who creates an object, who handles a request or who has enough knowledge to perform an operation.
- **Design Patterns** (HOW to do something technically): These are for offering reusable solutions to common software design problems specially at the technical level. Useful when we need to define how to control access to an object, how to decouple component or how to handle the creation of objects.

In our exercise, the problem is described as the need to define **how to manage a unique, globally accessible and critical configuration in a way that prevents inconsistencies across the system**. As we can see, this is a HOW to do technical problem, focused on the structure and control of a single shared instance, which clearly falls into the category of *design patterns*.

After classifying this problem, I will evaluate which of the available **design patterns** in the learning material is the most suitable for our use case:

1. The statement of this first question tells us that:
   - The configuration is critical to the functioning of the system.
   - It must be **managed in a unique wa**y (only one instance must exist).
   - It must be **accessible globally** (from anywhere in the system).
   - It must ensure **is consumed and modified in a secure way** and avoid inconsistencies.

2. Based on this description, we can highlight three **key technical needs**:
   - **Unique management**
   - **Global accessibility**
   - **Consistency and safety**

3. We can now discard some of the patterns that may be not full aligned with these key technical needs:
   - Patterns like **Strategy**, **Command** or **State** are more for encapsulate behaviour or manage dynamic changes but *they don't provide global access or unique management.*
   - **Iterator**, **Adapte**r, **Decorator** or **Proxy** are more for structural and traversal decision but *not related with the global system coordination and they don't ensure a single instance.*
   - **Factory Method**, **Template Method** and **Abstract Server** are more focused on how to create objects and server abstraction. *They are more for deciding how to create multiple objects rather than managing the global configuration.*
   - **Observer** is more for automatic updates and **Null Object** helps avoid null checks, but *they don´t guarantee a unique instance or a secure modification.*
   - **Facade** simplifies the interface usage but *doesn't provide by itself unique management or access control.*

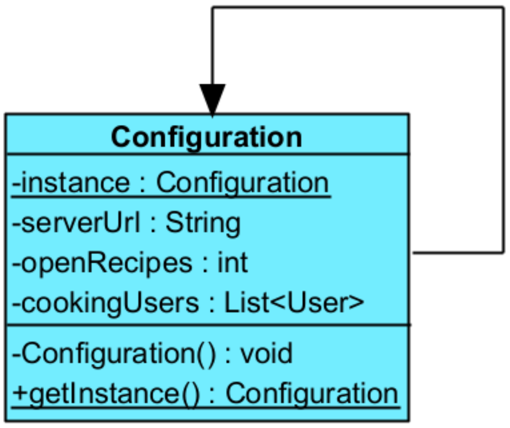
4. So, the pattern that may address all the three key technical needs is the **Singleton** pattern. In the material it is mentioned that this pattern "*ensures for a class that can only be one instance"*, which is exactly what is expected in this Guided Cooking system requirement. Even though the statement is mentioning the "coordination of the execution of all the devices" which can in my opinion easily lead to misunderstand as the need of coordinate actions between objects applying patterns like **Command**, **Mediator** or **Observer**, my decision is focused on the need of **one unique instance of configuration, accessible from any part of the system and consistent**.

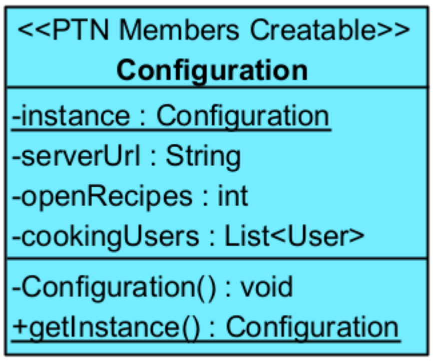
5. Advantages of applying **Singleton** pattern compared with other alternatives:
   - Ensure there's only one configuration object in the whole system.
   - Allows any part of the system, such as the device controllers or the recipe managers, to access the same configuration.
   - It will help to avoid problems like duplication or conflict settings.
   - It is a simple and practical solution, easy to apply and without the need of any extra change in other classes.

6. Disadvantages to be considered:
   - It is difficult to replace and hard for unit tests.
   - If the system is using multiple threads, and several devices connect at the same time, this pattern will need and extra configuration to not break (thread safe).

Based on this comparative analysis of the pattern alternatives offered in the learning material, I found the **Singleton** pattern as the best choice to apply in this use case.

b) NOTE: Unfortunately, I didn´t find any UML Singleton representation example in the learning material nor the solution examples, so I searched on internet different ways to represent it. I am adding here two options:


Configuration
-instance : Configuration
-serverUrl : String
-openRecipes : int
-cookingUsers : List<User>
-Configuration() : void
+getInstance() : Configuration


<<PTN Members Creatable>>
Configuration
-instance : Configuration
-serverUrl : String
-openRecipes : int
-cookingUsers : List<User>
-Configuration() : void
+getInstance() : Configuration

***Option 1**: Self reference representing the class contains an instance of itself.*

Source: https://refactoring.guru/design-patterns/singleton#structure

***Option 2**: Using Visual Paradigm <<PTN Members Createtable>> Stereotype indicating that the class may contain aditional methods beyond the strictly neccesary for the Singleton pattern implementation.*

Source: https://www.visual-paradigm.com/tutorials/singletonpattern.jsp c)

```
public class Configuration
{
    // Static attribute having the single instance of the class
    private static Configuration instance;

    // Configuration data attributes
    private String serverUrl;
    private Integer openRecipes;
    private List<User> cookingUsers;

    // Private constructor to prevent instantiation from outside
    private Configuration()
    {
        this.serverUrl = "";
        this.openRecipes = 0;
        this.cookingUsers = new List<User>();
    }

    // Static method to access the single instance
    public static Configuration getInstance()
    {
        if (instance == null) // Ensure it is created only once
            instance = new Configuration();

        return instance;
    }
}
```

Question 2

a) In this case, the problem consists of designing a system that **allows users to explore their collections of saved recipes in multiple, flexible ways** such as filtering by vegan recipes, sorting by cooking or adding new visualizations and export options in the future.

This is also a clear “HOW to do” technical problem, because we must **define a technical way to allow the flexible exploration of the collections** in a maintainable and extensible manner.

After classifying this problem, I will evaluate which of the available design patterns is the most suitable for our use case:

1. The question statement highlights the following requirements:
    - We want to define **different strategies for exploring recipes** *(filtering, sorting, etc).*
    - The system must **allow for future addition of new strategies** without modifying the existing code.
    - There may be **multiple filters or criteria** with its own logic.

2. From this definition, we can extract these **key technical needs**:
    - **Flexibility** *(to define and apply different exploration behaviours).*
    - **Separation of concerns** *(the logic for exploring should be independent from the main collection).*
    - **Open/Closed principle** *(the system must be open to new strategies but closed for modification of existing logic).*

3. Based on all these, we can discard several patterns alternatives from our learning material catalogue
    - **Adapter** and **Facade** are structural patterns used more for interface compatibility or simplification, *but they are not direct related with flexibility behaviour as we need*.
    - **Command** and **Observer** may offer extensibility (like trigger actions or reacting to events) but *they don’t address the need of selecting different exploration behaviours*.
    - **Decorator** is suitable for adding responsibilities dynamically *but not for changing the collection traversal logic as it is required*.

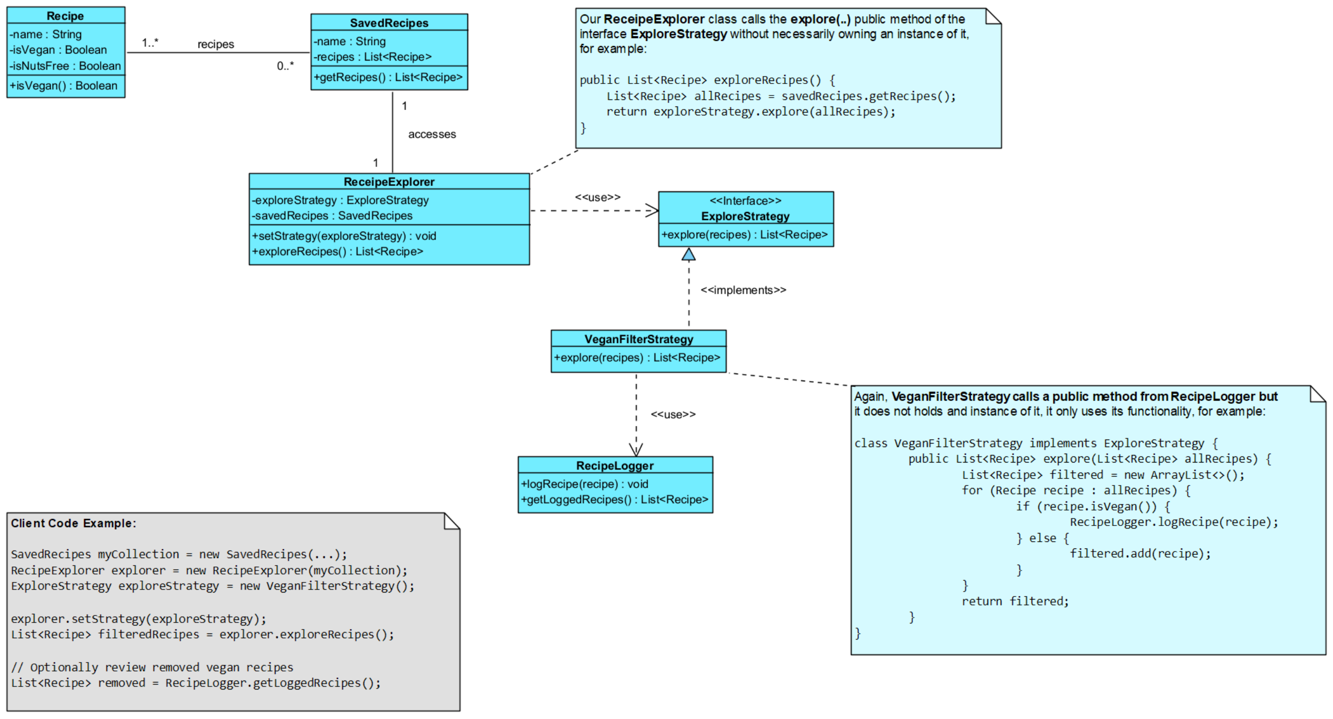
4. **Template Method** and **Strategy** pattern seem to be the most suitable for this use case. The learning material mentions that *“The Strategy design pattern allows solving very similar problems to those that the Template method pattern solves …”*. So, analysing these two options, we may say that the **Template Method** would partially work if we provided a fixed algorithm, but as it is described in the material explanation, it requires inheritance and a predefined algorithm structure, which will limit the flexibility and will requires subclassing every time a new strategy is needed. On the other hand, the **Strategy** pattern is specifically designed for this kind of scenario because it allows the algorithms (in our case the exploration logic) to be defined in a family of classes. So, these strategies can be “interchanged at runtime” and based on the user’s preference. We can have a core class that doesn´t need to know how each strategy works, it just used the selected one.

5. Advantages of applying the **Strategy** pattern compared with other alternatives:
   - Supports **open/closed principle** since new exploration strategies can be added without modifying the existing classes.
   - We will **prioritize composition over inheritance** since strategies are passed as object rather than implemented by subclasses.
   - Allows users or system components to dynamically select how they want to explore the recipe collections.
   - The **exploring logic implemented**, for example veganFilterStrategy is **decoupled** from SavedRecipes class (better for maintainability).

6. Disadvantages to be considered:
   - If there are too many strategies of exploration classes, we will have code that is difficult to navigate and understand.
   - Clients must know the differences between strategies to be able to select a proper one so naming is key.
   - If we have only few algorithms and they do not change we may overcomplicate the program with the new classes and interfaces.

Based on this comparative analysis of the pattern alternatives offered in the learning material, I found the **Strategy** pattern as the best choice to apply in this use case.

A possible class diagram, that can be then represented in the sequence diagram requested, can be:


Recipe
-name : String
-isVegan : Boolean
-isNutsFree : Boolean
+isVegan() : Boolean
1..*
recipes
0..*
SavedRecipes
-name : String
-recipes : List<Recipe>
+getRecipes() : List<Recipe>
1
accesses
1
ReceipeExplorer
-exploreStrategy : ExploreStrategy
-savedRecipes : SavedRecipes
+setStrategy(exploreStrategy) : void
+exploreRecipes() : List<Recipe>
<<use>>
<<Interface>>
ExploreStrategy
+explore(recipes) : List<Recipe>
<<implements>>
VeganFilterStrategy
+explore(recipes) : List<Recipe>
<<use>>
RecipeLogger
+logRecipe(recipe) : void
+getLoggedRecipes() : List<Recipe>

Our **ReceipeExplorer** class calls the **explore(..)** public method of the interface **ExploreStrategy** without necessarily owning an instance of it, for example:

```
public List<Recipe> exploreRecipes() {
    List<Recipe> allRecipes = savedRecipes.getRecipes();
    return exploreStrategy.explore(allRecipes);
}
```

Again, **VeganFilterStrategy calls a public method from RecipeLogger but** it does not holds and instance of it, it only uses its functionality, for example:

```
class VeganFilterStrategy implements ExploreStrategy {
        public List<Recipe> explore(List<Recipe> allRecipes) {
                List<Recipe> filtered = new ArrayList<>();
                for (Recipe recipe : allRecipes) {
                        if (recipe.isVegan()) {
                                RecipeLogger.logRecipe(recipe);
                        } else {
                                filtered.add(recipe);
                        }
                }
                return filtered;
        }
}
```

Client Code Example:

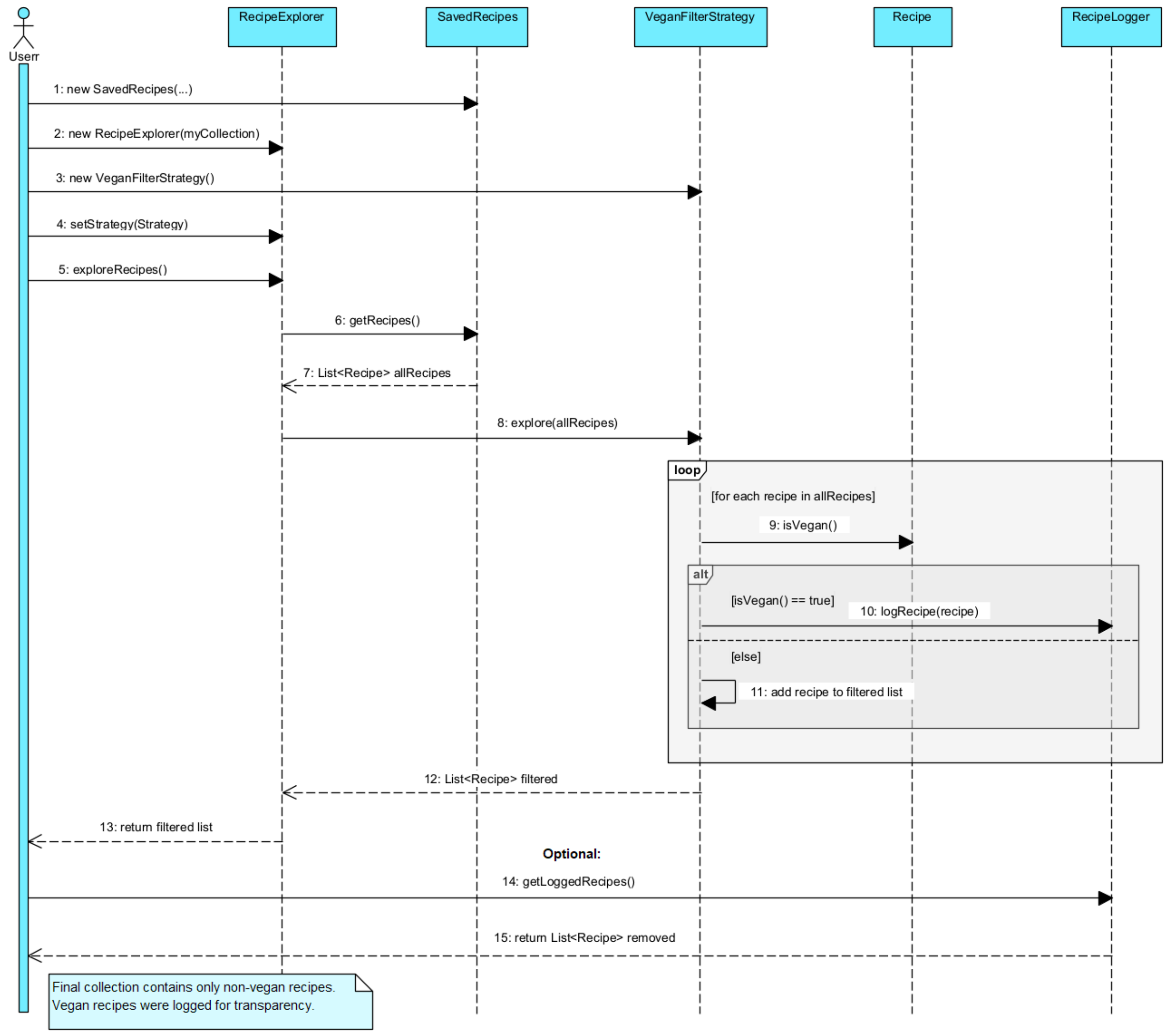
```
SavedRecipes myCollection = new SavedRecipes(...);
RecipeExplorer explorer = new RecipeExplorer(myCollection);
ExploreStrategy exploreStrategy = new VeganFilterStrategy();

explorer.setStrategy(exploreStrategy);
List<Recipe> filteredRecipes = explorer.exploreRecipes();

// Optionally review removed vegan recipes
List<Recipe> removed = RecipeLogger.getLoggedRecipes();
```

- ***ReceipeExplorer*** *represent the context of the strategy pattern. It receives the collection and a strategy and execute* ***exploreRecipes()*** *that calls internally to strategy.explore(…).*
- ***ExploreStrategy*** *represent the Interface that defines the contract for the strategies.*
- ***VeganFilterStrategy****: Concrete strategy implementation of* ***ExploreStrategy****. In this use case it filters vegan recipes and uses* ***RecipeLogger*** *to log the ones it removes.*
- ***RecipeLogger*****:** *Class responsible for logging discarded recipes (vegan ones in this case).*

**Source: https://refactoring.guru/design-patterns/strategy#structure* b)


Userr
RecipeExplorer
SavedRecipes
VeganFilterStrategy
Recipe
RecipeLogger
1: new SavedRecipes(...)
2: new RecipeExplorer(myCollection)
3: new VeganFilterStrategy()
4: setStrategy(Strategy)
5: exploreRecipes()
6: getRecipes()
7: List<Recipe> allRecipes
8: explore(allRecipes)
loop
[for each recipe in allRecipes]
9: isVegan()
alt
[isVegan() == true]
10: logRecipe(recipe)
[else]
11: add recipe to filtered list
12: List<Recipe> filtered
13: return filtered list
Optional:
14: getLoggedRecipes()
15: return List<Recipe> removed
Final collection contains only non-vegan recipes.
Vegan recipes were logged for transparency.

Question 3

a) Before choosing a specific pattern, I will analyse the type of problem that we are facing in this question. We are required here to design a system that can supports flexible execution of commands (like start, pre-heat, stop, etc.) on smart cooking devices, with the additional need of supporting undo and redo operations.

This is clearly a **technical "HOW to do" problem**, since we are focused on how to technically organize and control the actions, rather than assigning responsibilities.

After classifying this problem, I will evaluate which of the available design patterns is the most suitable for our use case:

1. The question statement highlights the following requirements:
   - We want to define **a variety of commands that need to be executed on devices.**
   - The system must **allow for future addition of new commands** without modifying the existing code.
   - There must be a **history of executed commands** to support undo and redo the operations.

2. From these requirements, we can extract these **key technical needs**:
   - **Flexibility** (*to encapsulate each command as a reusable objects*).
   - **Extensibility** *(to support adding new command types easily).*
   - **History management** *(to implement undo/redo operations).*

3. Based on all these, we can discard several patterns alternatives from our learning material catalogue:
   - Patterns like **Observer** or **Strategy** are more useful for event notification or variation of the algorithms *but don't focus on supporting undo/redo logic.*
   - **Template Method, Adapter,** or **Façade** help with reusability and simplification, but do not encapsulate executable actions as it is required.
   - **Singleton, Factory Method** are more related to object control, but don't offer action tracking history.

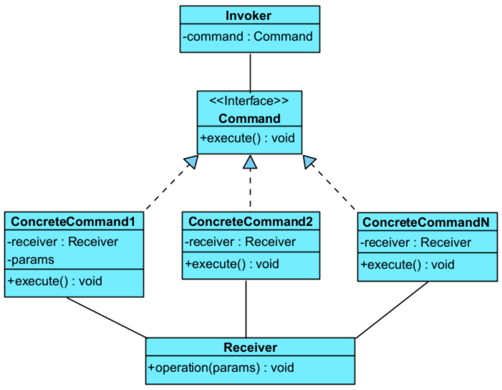
4. The pattern that may best satisfies all the needs described above is the **Command** pattern. As it is mentioned in the Learning Material, this pattern is useful when “*we want to handle calls to operations as objects...”.* It also allows for a centralized execution mechanism (like the remote-control class) and supports flexible architecture where adding a new command will not require to change the existing code (open/closed principle).

5. Main reasons for applying the **Command** pattern in our use case:
   - Allows each device instruction to be **encapsulated as a separate command** class.
   - Make it easy to **implement the undo and redo** operations required.
   - **It provides a central mechanism for executing the commands,** which is what is expected with the idea of a remote control for giving instructions to the devices.
   - **It will ensure that the system is open for extension** (creating new command classes) and **closed for modification** because we don´t need to change the existing command execution logic to add new instructions.
   - It is perfectly **aligned with the requirement of flexible control** and historical tracking of device actions, allowing us to implement the most advanced features described such as switching back from fast pressure to slow, etc.

Based on the analysis of the requirements and different pattern characteristics, I believe that the **Command** pattern is the most appropriate pattern choice for address the problem in our use case. It will provide a clean, extensible, and maintainable way to control the smart cooking devices, including support for undo and redo functionalities.

b) To apply this pattern in our use case, I applied the following reasoning:

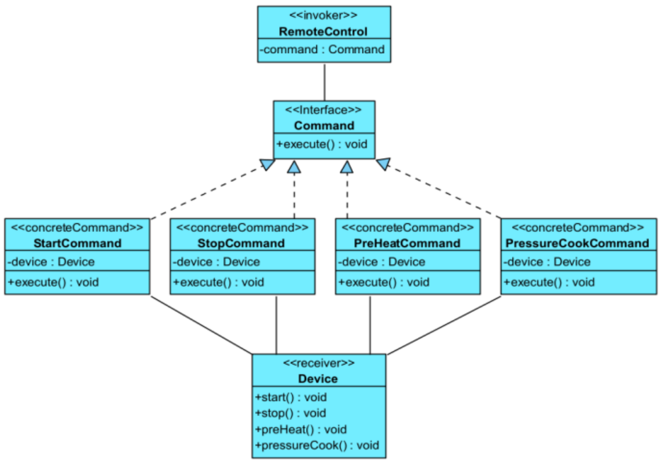
***Step 1** - Base Structure of the Command Pattern*



First, I defined a mental model with the basic structure of the **Command** pattern and its components:

- The **Invoker** who sends the command.
- The **Command** interface that defines what will be executed.
- the **ConcreteCommand** classes that implement various kind of requests and pass the call to one of the business logic objects.
- The **Receiver** who actually does the real work.

***Step 3** - Adapt the model to the smart devices use case*



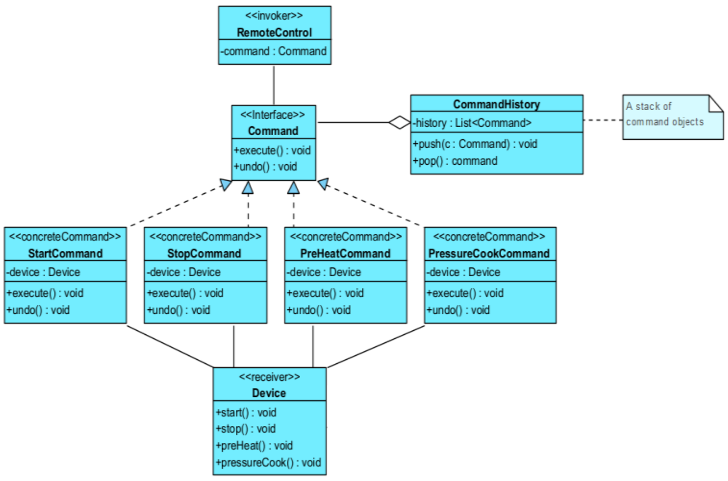
Then, I adapted this model to the classes names that fits the smart devices system proposed:

- The **Invoker** became **RemoteControl**.
- The **Receiver** became **Device**.
- the **ConcreteCommand** classes are the ones described in the statement.

This helped to test **how the pattern works in practice** and imagine how the system would behave when a user sends a command to a device.

This confirm **that the system will be flexible**, meaning that if we want to add a new action, we can just write a new command class.

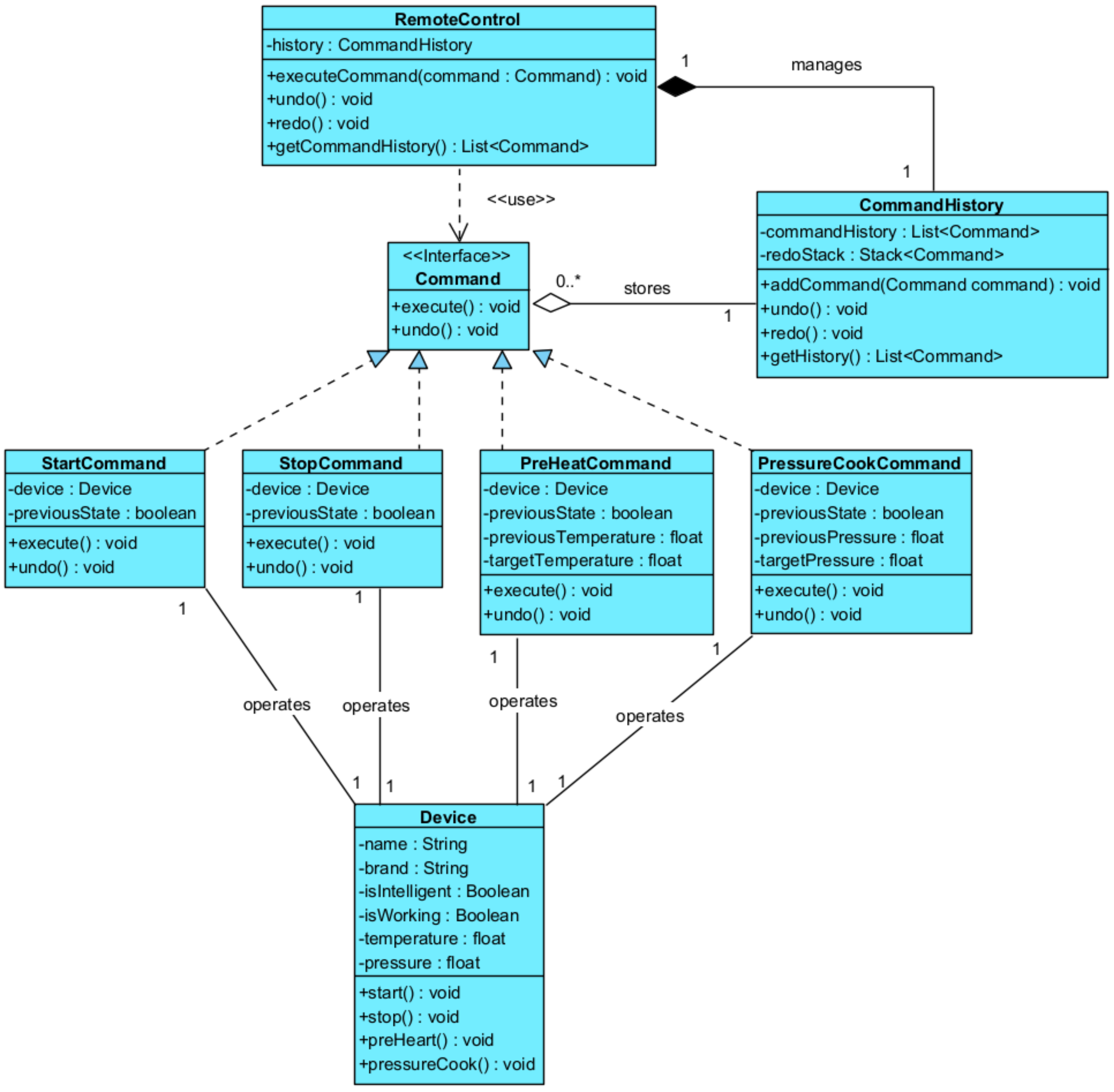
***Step 2** - Add new class for history features*



Finally, I added the history and undo/redo support that is required by the assignment. To do so, I added a **CommandHistory** class that keeps a list of executed commands.

The idea of separate this functionality in a new class will be to avoid mixing the logic inside the other classes and keep things clean.

Based on this model, I propose the following UML class diagram for our use case:



c)

```
// Invoker Class
public class RemoteControl {
    private CommandHistory history;

    public RemoteControl() {
        this.history = new CommandHistory();
    }

    // Execute the command and is added to the history
    public void executeCommand(Command command) {
        command.execute();
        history.addCommand(command);
    }

    // Delegate the operations to Command Hisotry
    public void undo() {
        history.undo();
    }

    public void redo() {
        history.redo();
    }

    public List<Command> getCommandHistory() {
        return history.getHistory();
    }
}
```

```java
// Class for managing the history of command execution
public class CommandHistory {
    private List<Command> commandHistory = new ArrayList<>();
    private Stack<Command> redoStack = new Stack<>();
    private int currentIndex = -1;

    // Add a command to the history
    public void addCommand(Command command) {
        // Clear any commands in the redo stack when a new command is executed
        if (!redoStack.isEmpty()) {
            redoStack.clear();
        }

        commandHistory.add(command);
        currentIndex = commandHistory.size() - 1;
    }

    // Undo the last executed command
    public void undo() {
        if (currentIndex >= 0) {
            Command command = commandHistory.get(currentIndex);
            command.undo();
            redoStack.push(command);
            currentIndex--;
        } else {
            System.out.println("Nothing to undo");
        }
    }

    // Redo and undo command
    public void redo() {
        if (!redoStack.isEmpty()) {
            Command command = redoStack.pop();
            command.execute();
            currentIndex++;
        } else {
            System.out.println("Nothing to redo");
        }
    }

    // Return the commands history
    public List<Command> getHistory() {
        return new ArrayList<>(commandHistory);
    }
}
```

```
// Command Interface
public interface Command {
    void execute();
    void undo();
}

// Receiver
public class Device {
    private String name;
    private String brand;
    private boolean isIntelligent;
    private boolean isWorking;
    private float temperature;
    private float pressure;

    public Device(String name, String brand, boolean isIntelligent) {
        this.name = name;
        this.brand = brand;
        this.isIntelligent = isIntelligent;
        this.isWorking = false;
        this.temperature = 0;
        this.pressure = 0;
    }

    public void start() {
        isWorking = true;
        System.out.println(name + ": started.");
    }

    public void stop() {
        isWorking = false;
        System.out.println(name + ": stopped.");
    }

    public void preHeat(float temp) {
        temperature = temp;
        System.out.println(name + ": preheated to " + temp);
    }

    public void pressureCook(float pressure) {
        this.pressure = pressure;
        System.out.println(name + ": pressure cook at " + pressure);
    }
    public float getTemperature() {
        return temperature;
    }

    public float getPressure() {
        return pressure;
    }

    public boolean isWorking() {
        return isWorking;
    }
}

// StartCommand
public class StartCommand implements Command {
    private Device device;
    private boolean previousState;

    public StartCommand(Device device) {
        this.device = device;
    }

    public void execute() {
        previousState = device.isWorking();
        if (!previousState) {
            device.start();
        } else {
            System.out.println(device.getName() + " was already started.");
        }
    }

    public void undo() {
        if (!previousState) {
            device.stop();
        }
    }
}
```

```
// StopCommand
public class StopCommand implements Command {
    private Device device;
    private boolean previousState;

    public StopCommand(Device device) {
        this.device = device;
    }

    public void execute() {
        previousState = device.isWorking();
        if (previousState) {
            device.stop();
        } else {
            System.out.println(device.getName() + " was already stopped.");
        }
    }

    public void undo() {
        if (previousState) {
            device.start();
        }
    }
}

// PreHeatCommand
public class PreHeatCommand implements Command {
    private Device device;
    private boolean previousState;
    private float previousTemperature;
    private float targetTemperature;

    public PreHeatCommand(Device device, float targetTemperature) {
        this.device = device;
        this.targetTemperature = targetTemperature;
    }

    public void execute() {
        previousState = device.isWorking();
        previousTemperature = device.getTemperature();
        device.preHeat(targetTemperature);
    }

    public void undo() {
        device.preHeat(previousTemperature);
    } }
```

```
// PressureCookCommand
public class PressureCookCommand implements Command {
    private Device device;
    private boolean previousState;
    private float previousPressure;
    private float targetPressure;

    public PressureCookCommand(Device device, float targetPressure) {
        this.device = device;
        this.targetPressure = targetPressure;
    }

    public void execute() {
        previousState = device.isWorking();
        previousPressure = device.getPressure();
        device.pressureCook(targetPressure);
    }

    public void undo() {
        device.pressureCook(previousPressure);
    }
}
```

Client Usage Example:

```
public class Main {

    public static void main(String[] args) {

        // Create the Smart Device

        Device pressureCooker = new Device("Pressure Cooker", "Phillips", true);


        // Create the remote control

        RemoteControl remote = new RemoteControl();


        // Send the command to the device

        Command start = new StartCommand(pressureCooker);

        remote.executeCommand(start); // Pressure Cooker: started.


        // Preheat to 180 grades

        Command preheat = new PreHeatCommand(pressureCooker, 180.0f);

        remote.executeCommand(preheat); // Pressure Cooker: preheated to 180.0


        // Define slow pressure

        Command slowPressure = new PressureCookCommand(pressureCooker, 1.0f);

        remote.executeCommand(slowPressure); // Pressure Cooker: pressure cook at 1.0


        // Change to fast pressure but we decide to undo

        Command fastPressure = new PressureCookCommand(pressureCooker, 3.0f);

        remote.executeCommand(fastPressure); // Pressure Cooker: pressure cook at 3.0


        // User realizes it was too strong and undo

        remote.undo(); // Pressure Cooker: pressure cook at 1.0

        // Then decide to redo the change

        remote.redo(); // Pressure Cooker: pressure cook at 3.0


        // Finally stopped the device

        Command stop = new StopCommand(pressureCooker);

        remote.executeCommand(stop); // Pressure Cooker: stopped.

}
```

Question 4

TODO:

a)
b)