Tay Jing Xuan – Project Portfolio for Farmio

About the project

Farmio is a game that aims to teach children computational thinking in a fun and interesting way. During gameplay, children learn computational concepts and write pseudocode to fulfil the game objectives. The extensive code simulations aid them in better understanding how their code works.

The image below shows the command line interface for Farmio during gameplay:

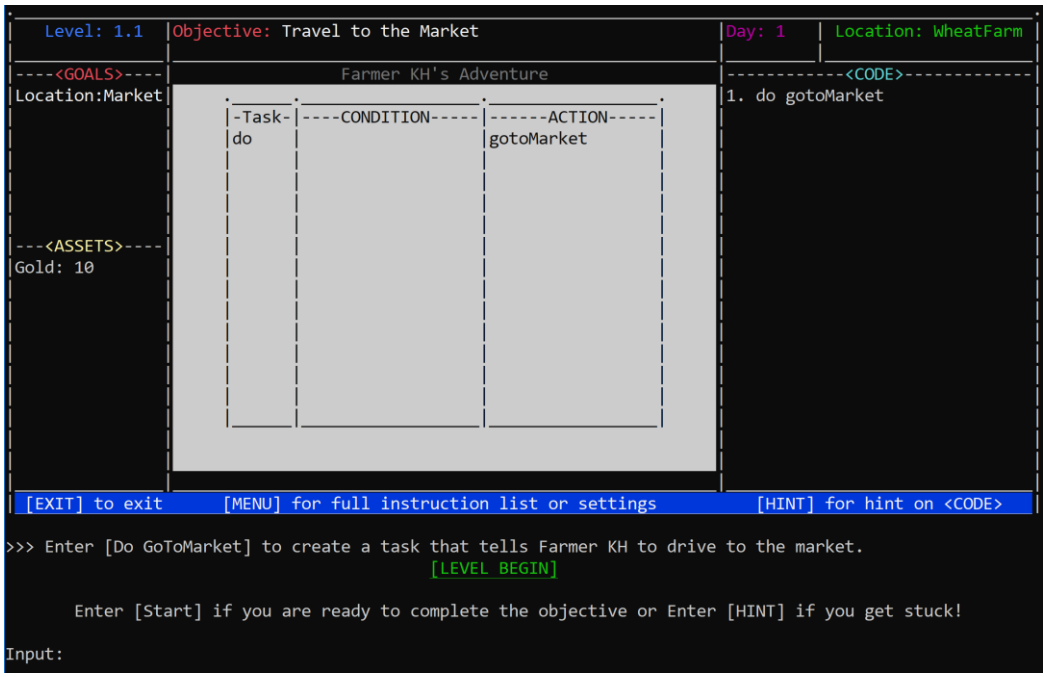

Figure 1. The Command Line Interface for Farmio

My role was to design and implement the framework for the game tasks, conditions and actions, as well as the features for the users to create, insert, edit and delete tasks. The following sections explain these features in greater detail, as well as provide an excerpt of the relevant documentation I added to the user and developer guides in relation to these features.

Note the following symbols and formatting used in this document:

This symbol indicates important information

`delete all` A grey highlight indicates that this is a command that can be input into the command line interface and executed by the game

`Task` A grey highlight with blue text indicates a component, class or object in the architecture of the application

Summary of Contributions

This section shows a summary of my contributions of my coding, documentation, and other relevant contributions to the team project.

Key Functionalities Added:

1. Design and Implementation of Framework for Tasks, Conditions and Actions

- What it is: The framework is implemented as part of the logic component in the game. It adds the functionality to create, store, and execute user created tasks.
- Highlights: This framework ensures high modularity of the tasks, conditions and actions, allowing users to create any task with any combination of conditions and actions. Analysis of various design alternatives was required to create a framework that is highly maintainable, as well as easily implemented and understood. Various implementation alternatives of the `Condition` and `Action` classes were analysed as well, in order to make these classes easy to expand upon as the development of the game progressed.

2. Design and Implementation of Commands for Creating, Editing, Inserting and Deleting Tasks

- What it does: Users are able to manipulate tasks in four different ways using these commands:
    - `do ACTION` / `if CONDITION do ACTION` – creates a `DoTask` / `IfTask`.
    - `insert POSITION TASK` – inserts a new task at the specified position
    - `edit POSITION TASK` – edits the task at the specified position
    - `delete POSITION` / `delete all` – deletes one task at the position/deletes all tasks
- Justification: These features create a smooth user experience. In addition to creating tasks, users can easily modify tasks if they made a mistake using the edit and insert commands. If they wish to start over with an empty task list, they can easily use the `delete all` command to remove all tasks or use the `delete POSITION` command to remove a single task.
- Highlights: The code base for these features are implemented in a way that they can be incorporated into future commands. Deep analysis of various design alternatives was done to ensure that the methods used for parsing, validation and creation of `Tasks`, `Condition`, and `Action` objects are highly maintainable, and easy to expand upon for future development. Effort was put into creating a smooth user experience, by ensuring that all commands are case and whitespace insensitive, reducing instances where the user's command fails due to a small typing error.

Additional Functionalities Added:

- Added feature for application to automatically detect operating system and remove incompatible colour and character codes (Pull requests #215, #295)
- Added feature to speed up and slow down animations for ease of testing (Pull requests #337, #339)

Code Contributed: To view a sample of my code, please follow this link.

Contributions to the User Guide

I was responsible for the parts of the Farmio User Guide that pertained to task manipulation, the feature which I had implemented. The following excerpt from the Farmio User Guide shows some additions that I made for the task creation feature. To view my full contributions, please read sections 3.4.1. to 3.4.4. of the Farmio User Guide.

Interactive Segment

The interactive segment comes after the story segment. This stage of the game allows you to write code to create tasks to fulfil the level objectives. This section describes the commands that can be used during the interactive segment.

Creating a New Task

Creates a new task and adds it to the bottom of your task list. There are two types of tasks:

1. **DoTask**

   Farmer always executes the task

2. **IfTask**

   Farmer executes the task only if the condition is fulfilled

Creating a DoTask:

Format: `do ACTION`

Examples of valid DoTask creation commands:

- `do sellGrain`
- `do buySeeds`

Creating an IfTask:

Format: `if CONDITION do ACTION`

Examples of valid task creation commands:

- `if hasWheat do harvestWheat`
- `if gold greater than or equals 10 do buySeeds`

You can only have a maximum of 18 tasks in your task list

This command is case insensitive

For example, to create a task for the farmer to go to market, you would enter

`do goToMarket`

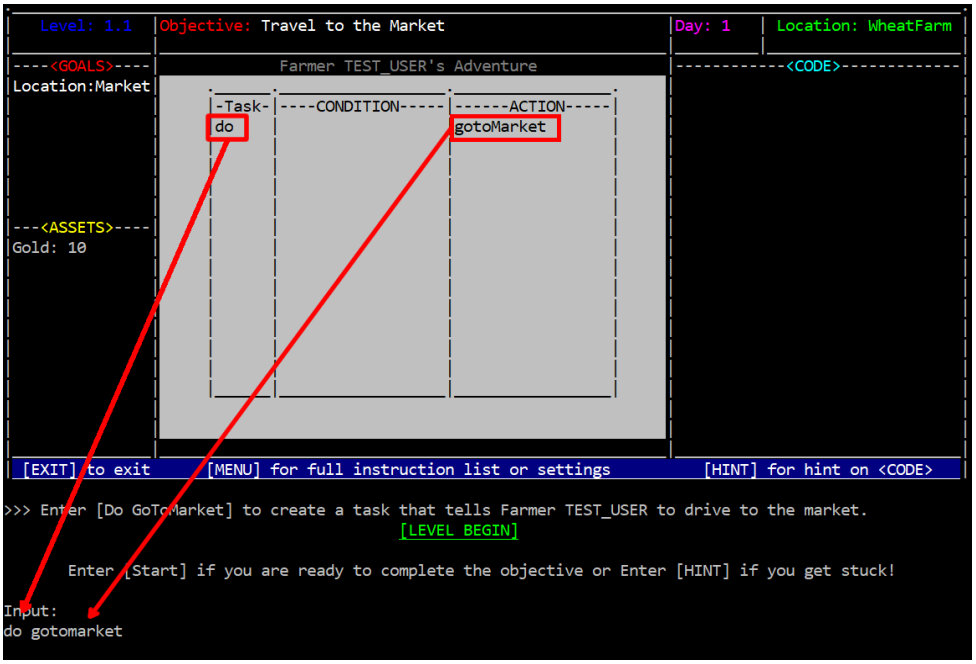

Figure 2. Creating a New Task

After the task has been successfully created, it will be added to the `CODE` section.

Figure 3. Successful Creation of Task

Contributions to the Developer Guide

My contributions to the Farmio Developer Guide consist of the documentation for all `Task` manipulation features that I have added, as well as documentation of the `logic` component. The following excerpt from the Farmio Developer Guide shows the additions I made for the `Task` object and the task creation feature. To view my full contributions, please read sections 4.3.1. to 4.3.5 of the Farmio Developer Guide.

Implementation and Manipulation of `Task` objects

This section describes the implementation of the Task object as well as the four ways users can interact with the `Task` object: creating a new `Task`, inserting a new `Task`, editing a `Task`, and deleting a `Task`.

`Task` Object

The `Task` object is central to gameplay. Any tasks that users create will be represented with instances of `Task`. During gameplay, these user created tasks will be executed to modify the game assets. The final state of these game assets will then determine if the user has reached the game objectives.

Implementation

The following class diagram illustrates the implementation of the `Task`, `Condition` and `Action` classes.

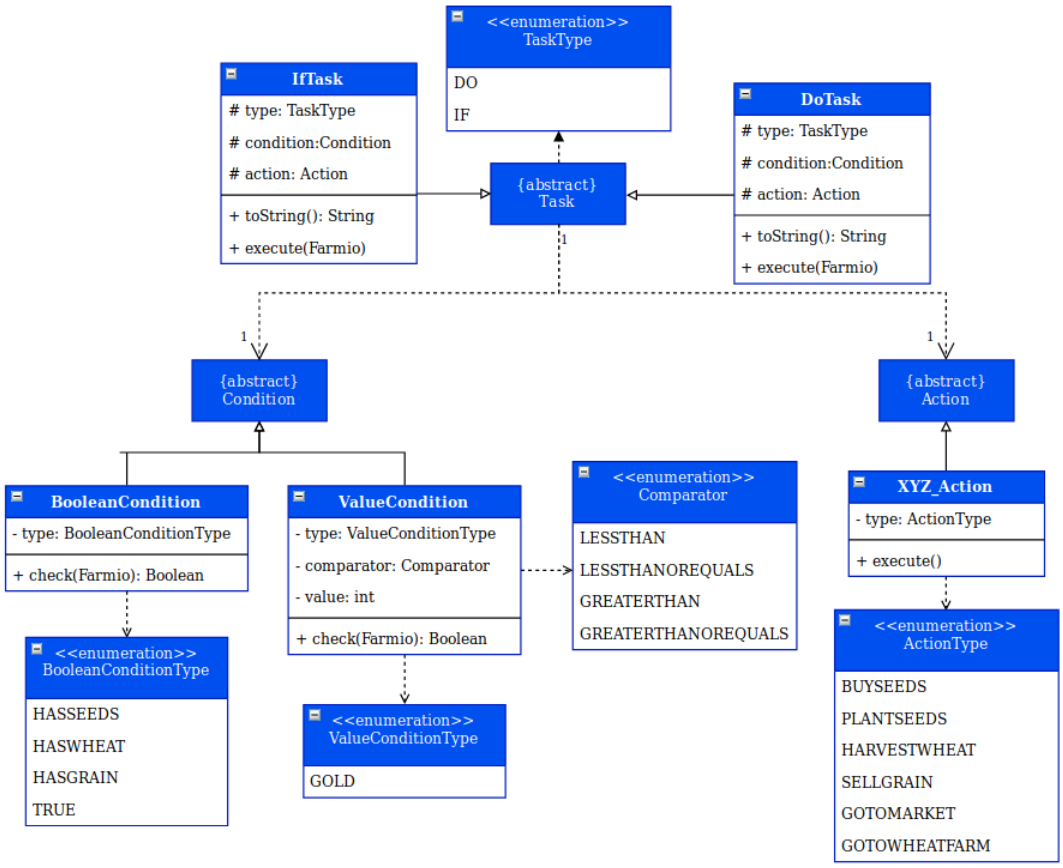

Figure 4. Class Diagram of `Task`, `Condition` and `Action` Classes

Creating a new `Task`

This feature facilitates creation of new tasks.

Implementation

The task creation mechanism is done through the `Logic` class, which utilizes `Parser` and `Command` to implement this feature. It allows users to create a new `Task` to be added into the `TaskList`. Additionally, it utilizes the following methods:

- `Parser#parseTask()` - Returns a `Task` created from user input.
- `Action#isValidAction()` - Validates the user input action.
- `Action#toAction()` - Creates an `Action` object from the user input.
- `Condition#isValidCondition()` - Validates the user input condition.
- `Condition#toCondition()` - Creates a `Condition` object from the user input.

The steps below describes the high level behaviour of the task creation mechanism for the example where the user inputs the command `do buySeeds`.

Step 1. The user inputs the command `do buySeeds`. `Logic` will then invoke `Parser#parse()`, which then calls `Parser#parseTask(do buySeeds)`.

Step 2. The method then extracts two substrings, the condition and action. `Condition#isValidCondition()` is used to validate the condition, and `Action#isValidAction()` used to validate the action.

If either of the validation functions fail, a `FarmioException` is thrown, notifying the user which part of their command is invalid

Step 3. If both the condition and action are valid, the corresponding `Condition` and `Action` objects are created using `Condition#toCondition()` and `Action#toAction()`. The `Task` object is then created and returned to `Parser#parse()`.

Step 4. The `CommandTaskCreate` object is created by `Parser#parse()`, and returned to `Logic`, which executes it, adding the new `Task` to the `TaskList`.
The sequence diagram below illustrates the high-level process of creating the `CommandTaskCreate` object, and executing it.

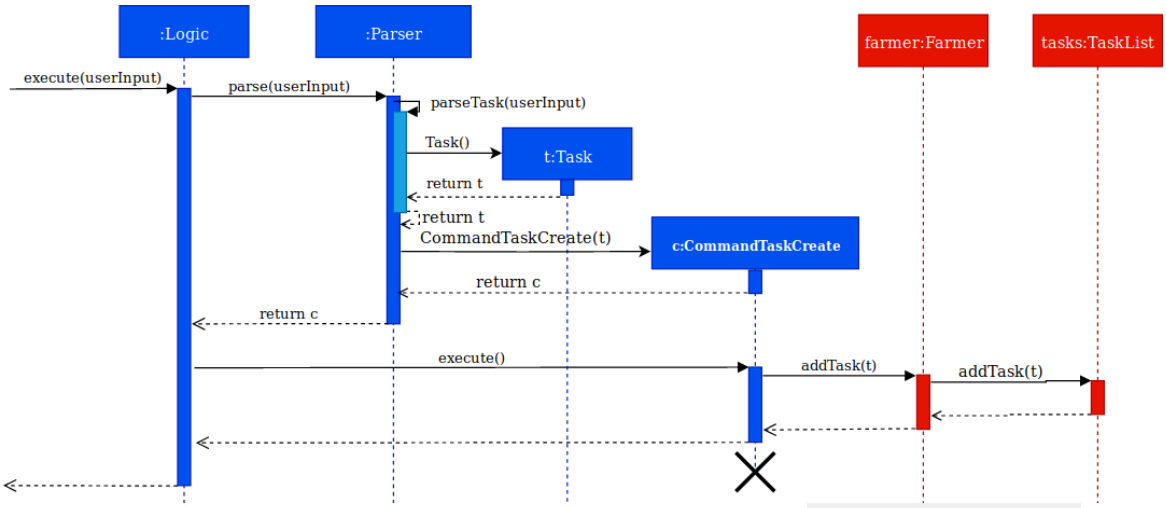

Figure 5. Sequence Diagram for Creating and Executing `CommandTaskCreate`

The object diagram below shows the different `Task` objects that can be created using this feature. `t1` is a `DoTask` that is created from the user command `do plantSeeds` while `t2` is an `IfTask` that is created from user command `if gold greater than 9 do buySeeds`.

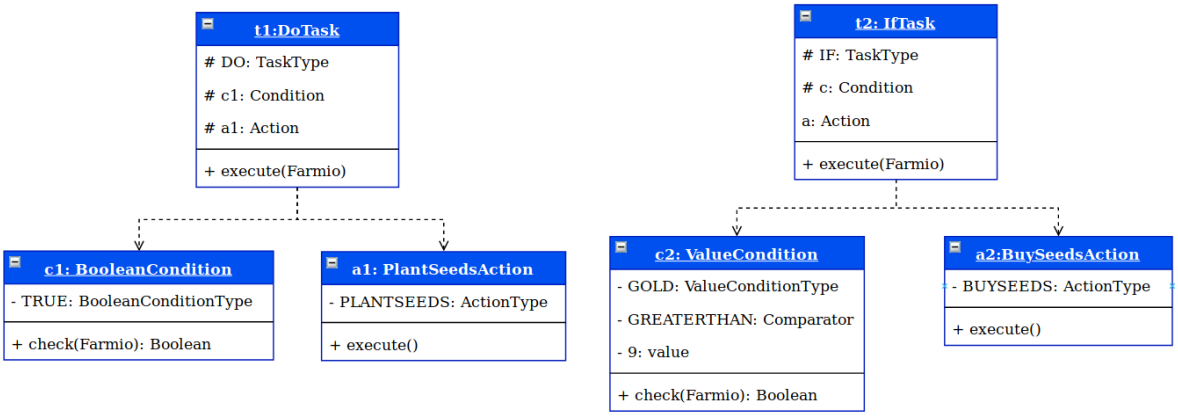

Figure 6. Class Diagram of Two Different `Task` Objects, `t1` and `t2`

A maximum of 18 tasks are allowed. Once that limit is exceeded, the user cannot add any more tasks.

Design Considerations

Aspect: Validation of conditions and actions

- Alternative 1 (current choice): Implement the validation methods as static methods in `Condition` and `Action` classes
    - Pros: Improves cohesion, as the methods for validating conditions and actions belong in their respective classes, and `Parser` only needs to call those methods. This also improves maintainability of code.
    - Cons: Increases coupling of code, as Parser class is increasingly dependent on the `Condition` and `Action` classes
- Alternative 2: Implement it within the `Parser#parseTask()` method
    - Pros: Easy to implement, as there is no abstraction of code
    - Cons: Poor cohesion as `Parser` would then be responsible for validation of the conditions and actions instead of the `Condition` and `Action` classes, and the `Parser#parseTask()` method would become difficult to maintain as more conditions and actions are added to the game.