CS2113T SOFTWARE ENGINEERING AND OBJECT-ORIENTED PROGRAMMING

Semester 1 2019/2020

Group F14-2
Farmio Developer Guide

Authors:

Name	Matric Number
Tan Yeh Han John	A0182665W
Tan Zheng Wen	A0183615A
Tay Jing Xuan	A0185096U
Yew Jing Hui	A0184187U
Umar Moiz	A0180913H

1. Introduction

1.1. Purpose

This document specifies the architecture and software implementation for the game, Farmio. The document is meant for developers who wish to understand the design and implementation of any features, so as to expand on the game.

1.2. Overview of Farmio

Farmio is a game that targets children aged between 11 and 14. We aim to teach children basic computing methodology in a fun and interactive way. Through this gameplay, students are exposed to problems which can be solved using programming concepts such as “if-else” statements, “for” and “while” loops.

1.3. Document Organization

Section	Purpose
Section 2: System Architecture	To describe the overall system architecture as well as its components
Section 3: Implementation	To describe the implementation of various features in Farmio

Note the following symbols and formatting used in this document:

	This symbol indicates important information
`New`	A grey font indicates a button or clickable option.
`start`	A grey highlight indicates that this is a command that can be typed into the command line and executed by the game
`Task`	A grey highlight with blue text indicates a component, class or object in the application

2. Setting Up

2.1. Pre-requisites

1. Java SE Development Kit (JDK) 11 and above.
2. Recommended: Intellij IDE.
3. Fork and clone project repository to local storage.

2.2. Setting Up the project

1. Fork this repo, and clone the fork to your computer
2. Open Intellij
3. Set up the correct JDK version
    a. Click `Configure` > `Structure for new Projects`.
    b. Click `New...` and select the directory for JDK.
    c. Click `OK`.
4. Click `Import Project`.
5. Locate the project directory and click `OK`.
6. Select `Import project from external model`.
7. Select `Gradle` and click `Next.`
8. Click `Finish`.

3. Design

These sections describe the high level architecture of the overall program, as well as its components.

3.1 Overall Architecture

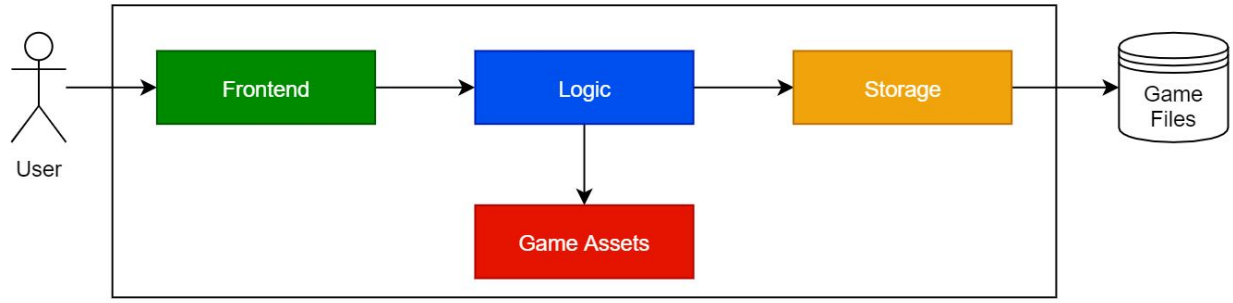


Figure 3.1.1. Overall Architecture Diagram

The architecture diagram above explains the high-level design of the game.
The game is made up of four main components:

- `Frontend` : The user interface of the game
- `Logic`: Parses user commands and executes them
- `Game Assets`: Holds data of game assets and modifies them
- `Storage`: Reads data from, and writes to external user files

The following sequence diagram shows how the components interact with each other for the scenario where the user issues the command `do buySeeds`.

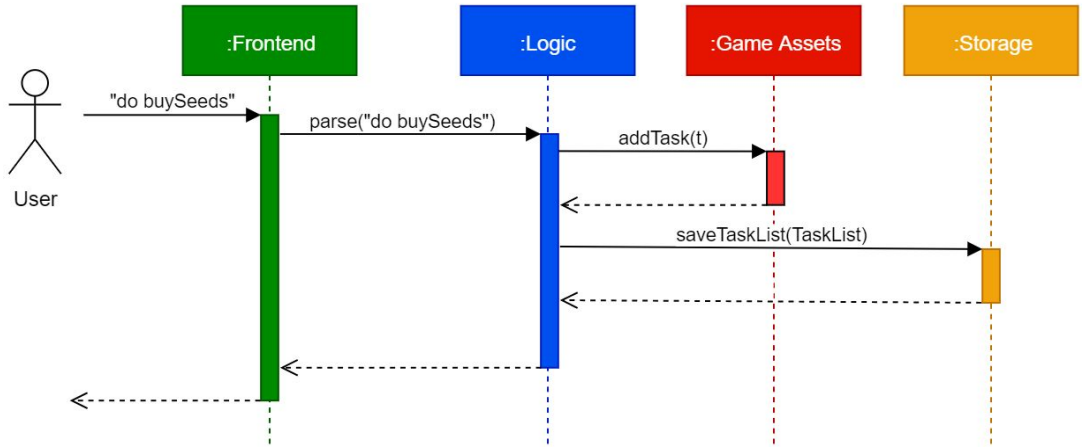


Figure 3.1.2. Component Interaction for `do buySeeds` command

3.2. Frontend Component

The Frontend component is responsible for all game user interface and the creation of the virtual game console. The diagram below is a high level representation of the component.

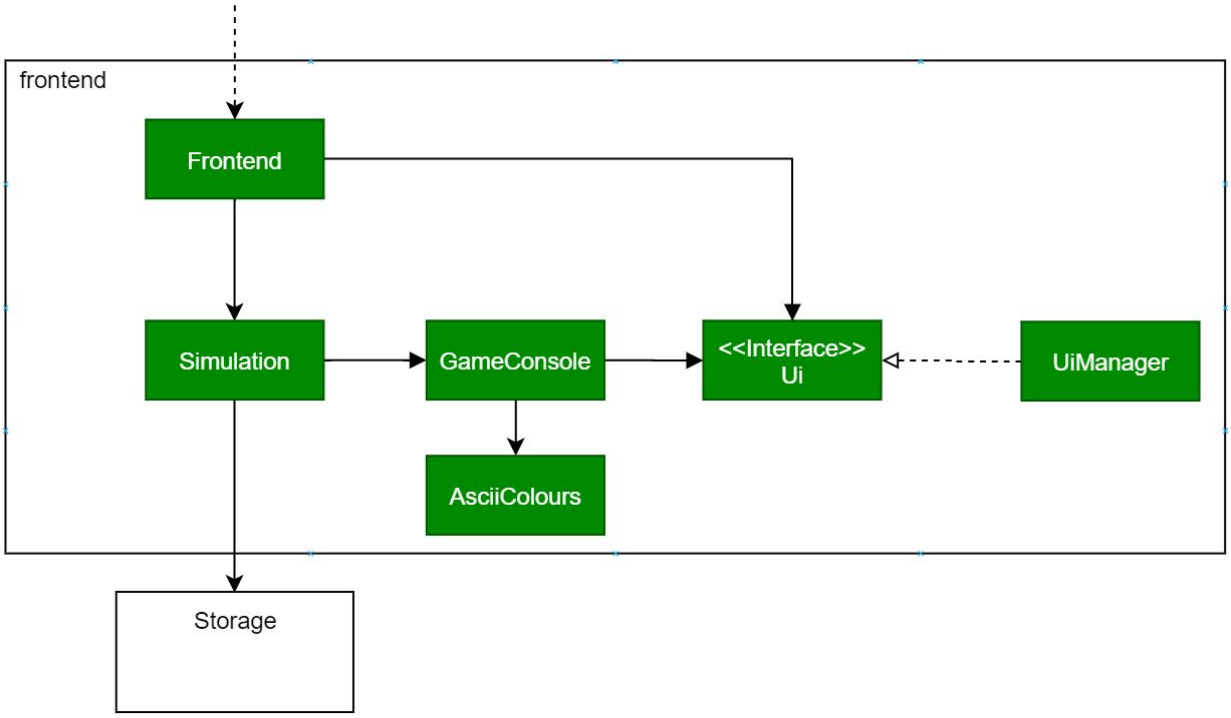


Figure 3.2.1. Structure for the Frontend Component

API: `Frontend.java`

1. Commands can call `Frontend` to display the game console or text.
2. `Frontend` may call either `Simulation` or `Ui` depending on the command.
3. `Simulation` will animate the text formatted by `GameConsole` and `AsciiColours`. It may retrieve an ascii image using `Storage`.
4. `Ui` will then be called to generate these formatted outputs on a suitable platform.

3.3. Logic Component

The Logic component is responsible for all game logic, parsing and carrying out user instructions. The diagram below is a high level representation of the component.

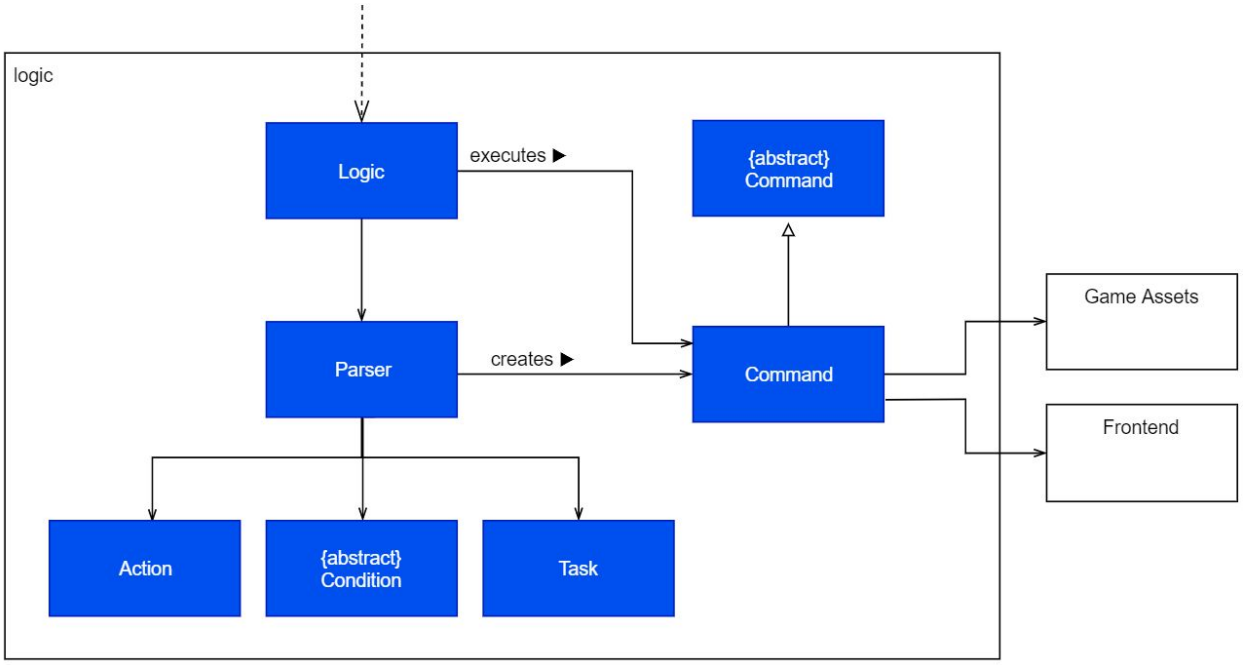


Figure 3.3.1. Structure for the Logic Component

API: `Logic.java`

1. `Logic` uses the `Parser` class to parse the user command
2. This creates a `Command` object which is executed by `Logic`
3. The command execution can create `Action`, `Condition` and `Task` objects.
4. In addition, the command execution can affect `Game Assets`, and it will use `Frontend` to notify the user of the changes

3.4. Game Assets Component

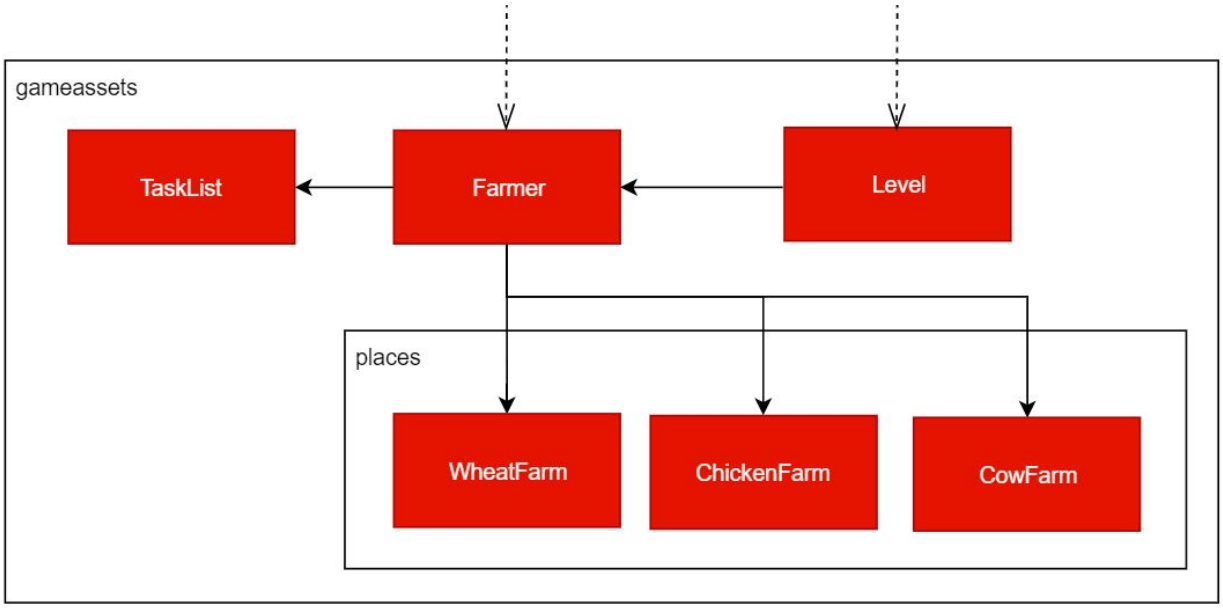


Figure 3.4.1. Structure for the Game Assets Component

API: `Farmer.java` , `Level.java`

`Farmer` class

- Stores `WheatFarm`, `ChickenFarm`, and `CowFarm` objects
- Modifies `WheatFarm`, `ChickenFarm`, and `CowFarm` objects
- Stores a `TasksList` that contains `Task` objects

`Level` class

- Checks if `Farmer` has fulfilled objectives
- Generates feedback depending on the state of `Farmer`

3.5. Storage Component

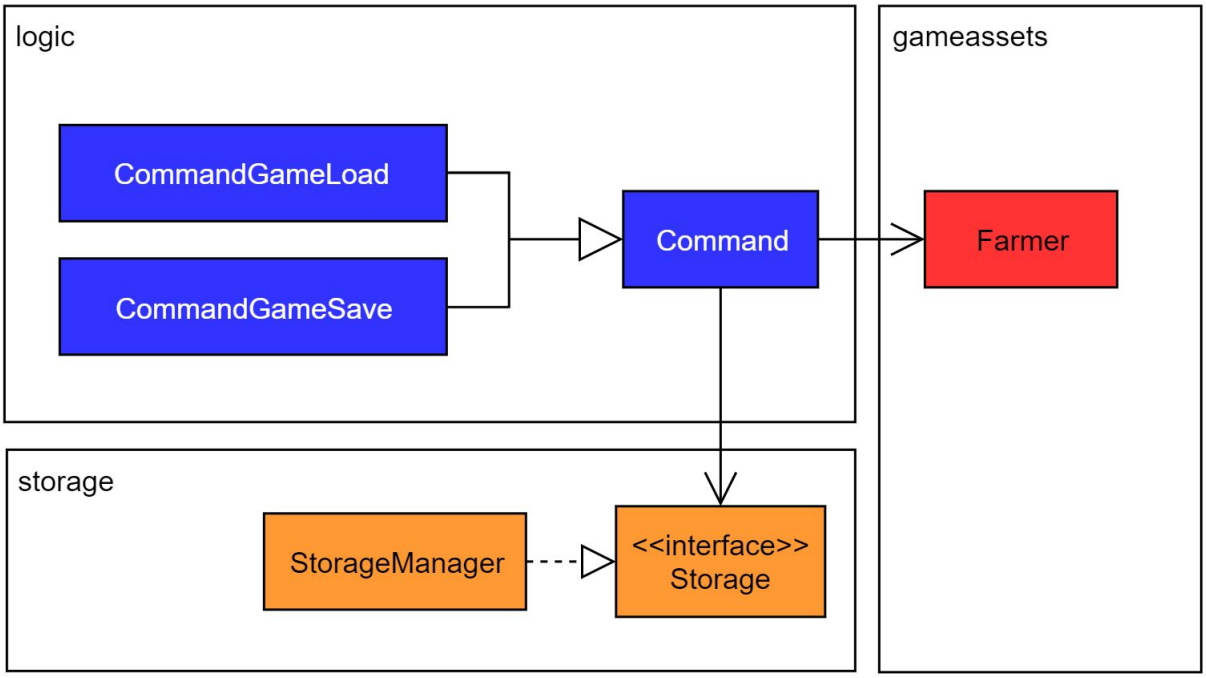


Figure 3.5.1. Structure for Storage Component

API: `Storage.java`

The `Storage` class has the following functions:

- Write `Farmer` object into save.json
- Read JavaScript Object Notation (JSON) object from save.json
- Read plain text and JSON files from resources folder

4. Implementation

This section describes some noteworthy details on how certain features are implemented.

4.1. Save/Load Game Feature

Save and load game features use save.json file located in the project directory. Data in save.json file is formatted in JSON formatting standards and is formatted using the orF.json.simple package.

4.1.1. Save Game Feature

4.1.1.1 Implementation

The save game feature allow users to save existing game data to be restored in the future. The feature is mainly handled by `StorageManager` and initiated by user save game command. The following steps describes how the feature operates.

Step 1. `Storage#storeFarmer()` is called by `CommandSaveGame` when user enters `save game` command. `Farmer` will be passed into the method as a parameter.

Step 2. In `Storage#storeFarmer()`, `Farmer#toJson()` will be called to retrieve JSON representation of `Farmer` object.

Step 3. When `Farmer#toJson()` is called, a series of method calls will be executed for every no primitive data type variable in `Farmer`. Non-primitive data type objects will have either a `toJson` or `toString` method that returns objects to be added into the `JSONObject` object.

The following sequence diagram shows the process of generating JSON data of `Farmer`.

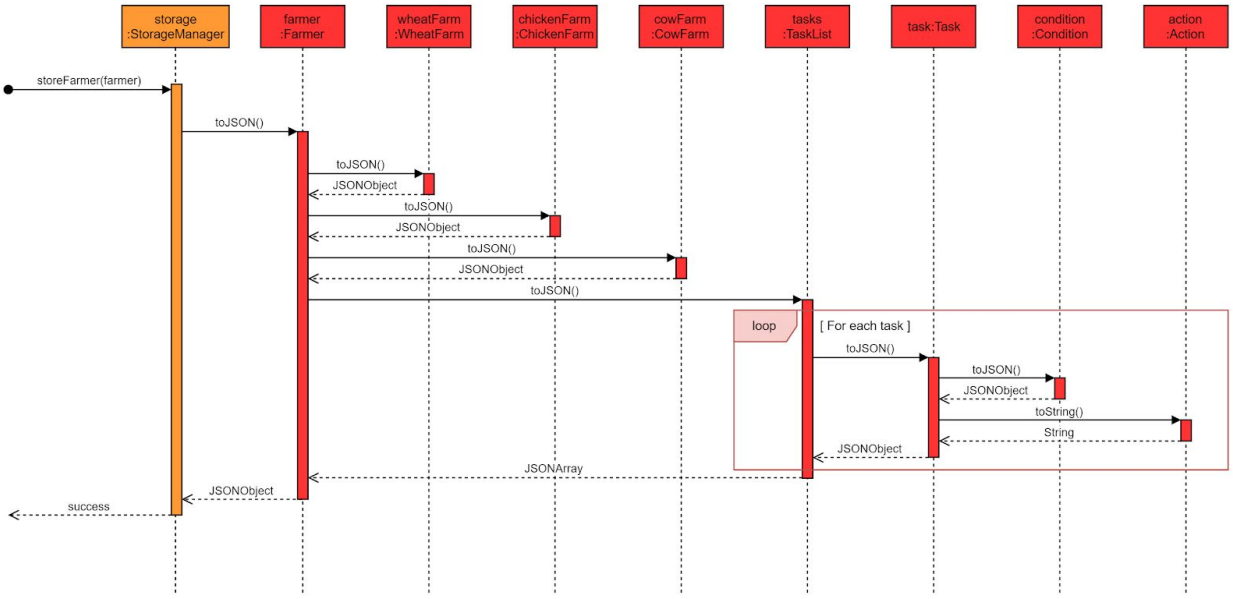


Figure 4.1.1.1.1. Generate JSON Behaviour

Step 4. After retrieving `Farmer` json data, the method then proceeds to write the JSON data into save.json file. If save.json file exists, the content of the file will be overwritten. Otherwise save.json will be created and written to.

Step 5. Upon successful `Storage#storeFarmer()` execution save.json is ready to read using `Storage#loadFarmer()`.

4.1.1.2 Design Considerations

Aspect: Game save date format

Alternative 1 (current choice): Use JSON format

- Pros: JSON standard is well known to the public, anyone can view and modify the save file when necessary.
- Cons: Limited data type supported to write directly in to file. Only primitive data types are supported to write in JSON formatted files.

Alternative 2: Use object serialization

- Pros: Any serializable object can be stored into save files. Custom created object can be made serializable as long as the class implements java.io.Serializable.
- Cons: Users cannot cannot easily view or edit the save file when necessary.

Aspect: JSON conversion process
Alternative 1 (current choice): Create toJSON method for every custom class.

- Pros: Simplify code in the method and split conversion responsibility to respective class. Reduce debugging difficulty as each class will handle part of the conversion.
- Cons: Every new class that needs to be stored in a JSON file will require an extra toJSON method.

Alternative 2: All conversion is done in loadFarmer.

- Pros: Localize all data conversion, reduce the need to reference from other classes.
- Cons: Length of code may increase and debugging difficulty for every new custom class and new variable.

4.1.2. Load Game Feature

4.1.2.1. Implementation

Users can use load game feature to restore previously saved game session from save.json file in the current working directory. Implementation of this feature is handled in `StorageManager` and uses the following methods:

- `Storage#loadFarmer()` - Retrieve `Farmer` data from save file.
- `Storage#getLevel()` - Retrieve `Level` data from resource folder.
- `Farmer#setJSON()` - Sets `Farmer` variables with data from JSONObject.
- `Farmio#setFarmer()` - Replace existing `Farmer`.
- `Farmio#setLevel()` - Replace existing `Level`.

The procedure for the game to be loaded will be in steps below, from when the user initiates the feature.

Step 1. The user enters the command load game. The command will be parsed and executed by `Farmio` and calls `Storage#loadFarmer()`.

Step 2. Storage will attempt to locate save.json, parse the content of save.json and check the validity of the level. The level is valid if the level number exist in the resource folder. The following activity diagram will illustrate the logical process of reading the content in save.json.

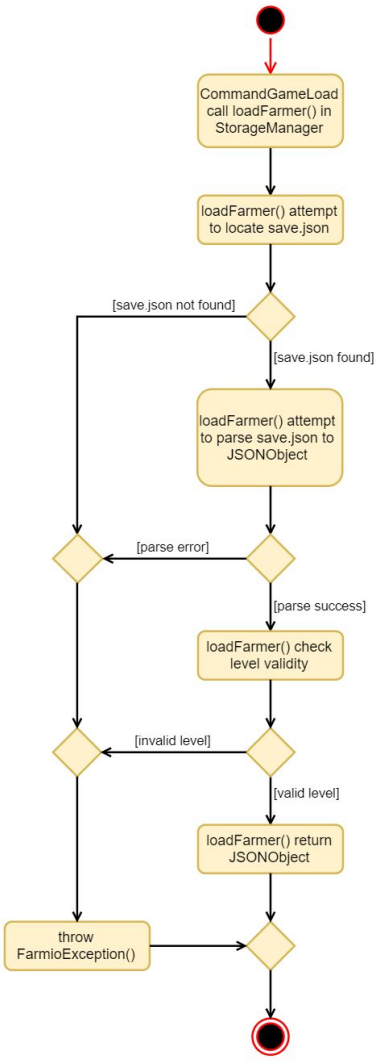


Figure 4.1.2.1.1. Fetch save.json Process

Step 3. If retrieval of JSON data from `Storage` is successful, initialise a new `Farmer` object and call `Farmer#setJSON` with the JSON data as the argument. `Farmer#setJSON` will extract values from the given parameter and set variable in Farmer accordingly. For non-primitive data type variables, initiate new object instance and pass the corresponding JSON data from the parameter into the constructor. Behaviour of the initialization is similar to the JSON generation process from Figure 4.1.1.1.

Step 4. Call `Farmio#setFarmer()` and pass previously created `Farmer` as argument to replace existing `Farmer` instance in `Farmio` which effectively change current game session to the previous saved state.

Step 5. Call `Storage#getLevel()` to retrieve JSON data from level file. Level files are located in the resource/levels folder. All level files are named by the corresponding level number contained in the file.

Step 6. If level JSON data retrieval is successful, initialise a new `Level` object with the obtained JSON data as an argument in the constructor. Level constructor will populate the level object with the data from the JSON parameter accordingly.

Step 7. Set level in `Farmio` using `Farmio#setLevel` with the previously created `Level` object. This will replace existing level in `Farmio` and load the level that was saved.

After level in farmio is restored to previous save point, the game have successfully loaded and the control will be returned to the user to continue with the narrative of the loaded level.

4.1.2.2. Design Considerations

Aspect: Extraction of variable data from JSON data.

- Alternative 1 (current choice): Distribute the data extraction to the respective objects.
    - Pros: Improves code cohesion, as new custom objects can be added in/loaded from the save file without modifying other classes.
    - Cons: Difficult to implement since all objects will need to include both conversion to and from JSONObject independently.
- Alternative 2: Localize extraction in CommandGameLoad#execute()
    - Pros: Easier implementation since all variables can be validated together.
    - Cons: Poor cohesion as new variable added in custom object, changes have to be made to CommandGameLoad#execute to extract data accordingly.

4.1.3. Auto Save Feature

4.1.3.1 Implementation

This feature is an extension of the save game feature which prevents users from losing valuable game data if the user fail to save the game before exiting the program.
The game attempts to save at specific points of the game to maximise the chances of full recovery upon system failure. These strategic points are:

- The start of every level.
- When user enter start command.

4.1.3.2. Design Considerations

Aspect: Auto save frequency.

- Alternative 1 (current choice): Only save at the start of level and start of game simulation.
    - Pros: Less impact on game performance.
    - Cons: Fully recovery upon failure reduce slightly.
- Alternative 2: Save on every task manipulation command.
    - Pros: High possibility for full recovery of game data.
    - Cons: High impact on game performance.
    - Cons: Makes save game redundant.

4.2. Add Name Feature

This section describes the implementation of how the user is able to customize their name for their in-game character.

4.2.1 Implementation

Users will use the add name feature whenever they start a new game. The add name mechanism is implemented by using `Parser`, `Command` and `Logic` class. It allows user to create an in-game name for their character which will be stored within the `Farmer` class. Additionally, it implements the following methods:

- `Command#CommandAddName(String userInput)` - Changes the farmer name to what the user has entered.
- `Farmer#inputName(String name)` - Stores the name that the user has inputted.
- `Farmer#isValidName(String name)` - Checks whether the name is valid from the save file.

Step 1: This implementation for this feature is done by using the `Command#CommandAddName()` Method.

Step 2: Within `Command#CommandAddName(String userInput)` method, the method will check whether the name that the user inputs satisfies all the requirements. We have limit the name to have a maximum of 15 characters and have no special characters with the exception of underscores.

Step 3: By storing the userInput inside a string inside `Command#CommandAddName()`, we will be passing the string into the method `Farmer#inputName(name)`.

Step 4: The method `Farmer#inputName(name)` will change the String variable name in Farmer to whatever the user has input. The name will be printed during the narrative, hints, objectives and feedback segments so as to make the game more personalised to the user.

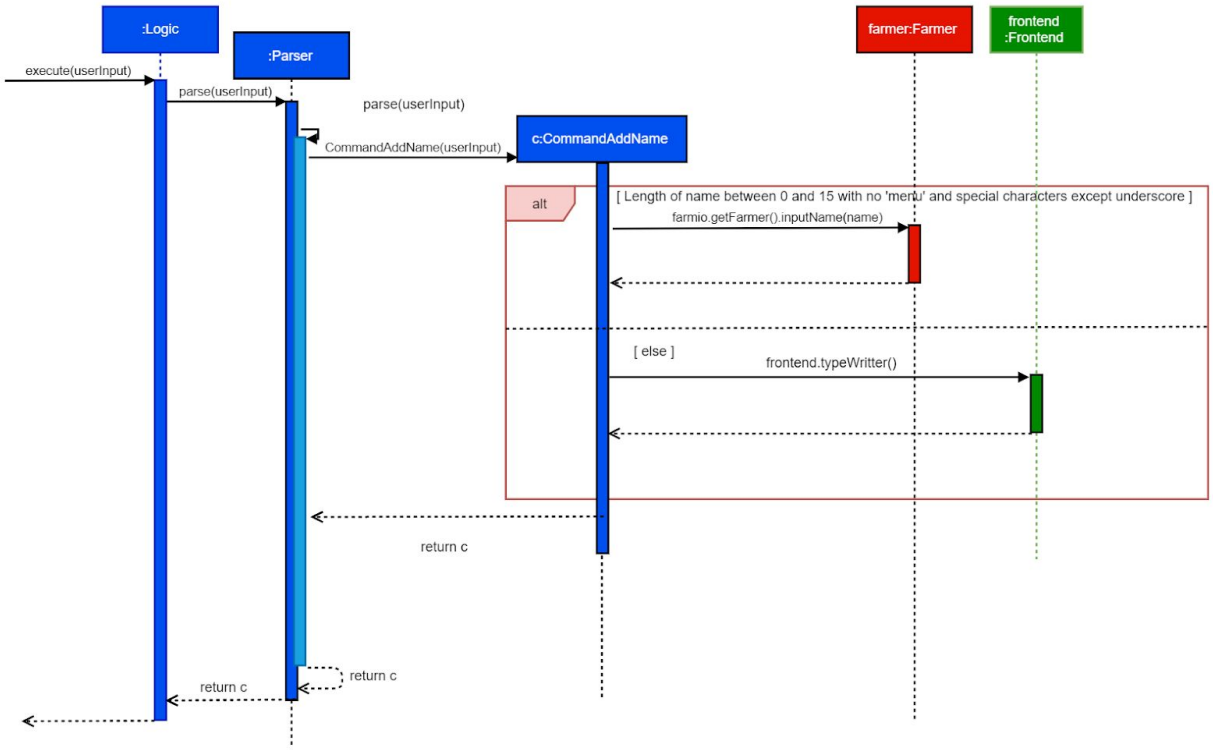


Figure 4.2.1.1. Sequence Diagram for Name Add Mechanism from a New Game.

Step 5: Checks for farmer's name from the save file. `Farmer#isValidName(loadName)` will check whether the name has an error when it is loaded by constructor `Farmer#Farmer(JSONObject jsonObject)`. This error will show if users try and tamper with the name within the save file.

Step 6: If there are errors, `Farmer#isValidName(loadName)` will throw FarmioException and would cause the save file to be corrupted. Users will not be able to load the game and they will resume at the current level as seen in Figure 4.2.1.2.

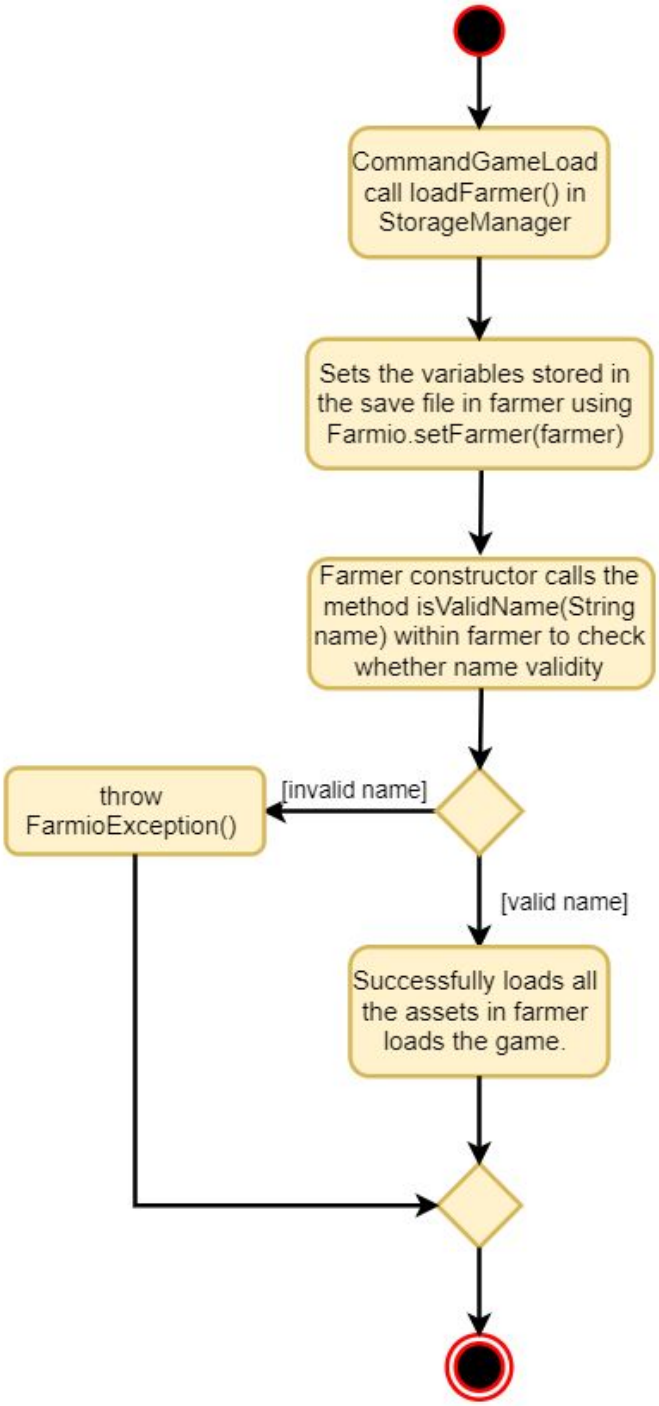


Figure 4.2.1.2. Check Valid Name from a Save File.

4.2.2. Design Considerations

Aspect: Validation of userInput for name

- Alternative 1 (current choice): Implement it using if-else statements to check the valid conditions within `Command#CommandAddName(String userInput)`
    - Pros: Improves maintainability of code, the conditions for a valid name are stated clearly within the if blocks making the restrictions easy to edit and change.
    - Cons: If there are too many conditions in the future, code will be harder to organise and understand as it gets longer.
- Alternative 2: Implement it within the `Farmer#inputName()` method
    - Pros: Easy to implement
    - Cons: Poor cohesion, the Farmer class is in-charge of many methods from other commands and putting the conditions within `Farmer#inputName()` method may cause the Farmer class to be overloaded.

4.3. 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`.

4.3.1. `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.

4.3.1.1. Implementation

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

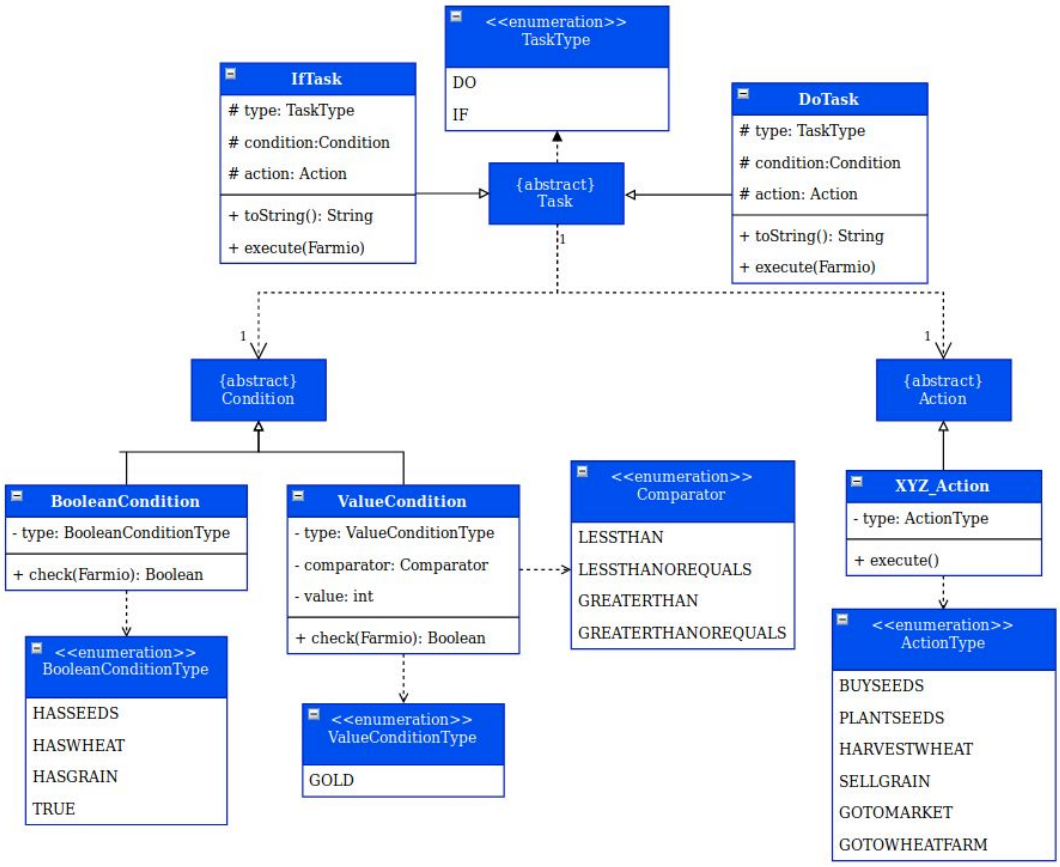


Figure 4.3.1. Class Diagram of `Task`, `Condition` and `Action` Classes

4.3.1.2. Design Considerations

Aspect 1: Implementation of conditions and actions

- Alternative 1 (current choice): Implement them as their own classes
    - Pros: Improves cohesion and maintainability as the check and execute methods are implemented inside `Condition` and `Action` classes
    - Cons: Difficult to design and implement
- Alternative 2: Implement them as enums or strings inside `Task`
    - Pros: Easy to implement
    - Cons: Poor maintainability, as any changes in the available conditions and actions would affect `Task`
    - Cons: Difficult to implement customizability of value related conditions

Aspect 2: Implementation of `Condition`

- Alternative 1 (current choice): Implement `BooleanCondition` and `ValueCondition` classes, and implement conditions as enums under them
    - Pros: Easy to implement

    - Cons: Poor cohesion, as the checking of the individual conditions are all in the same class
- Alternative 2: Implement each condition as a new class inherited from `BooleanCondition` or `ValueCondition` with their own check methods
    - Pros: Improves maintainability
    - Cons: Higher complexity, as a new class would have to be created for each condition when the check method is very simple

Aspect 3: Implementation of `Action`

- Alternative 1 (current choice): Implement each action as its own class inherited from `Action`
    - Pros: Increases maintainability, as action execution is more complex
    - Cons: Increased complexity as many new classes need to be created
- Alternative 2: Implement each action as an enum
    - Pros: Easy to implement
    - Cons: Poor cohesion and maintainability, as each action execution method is considerably long and complex

4.3.2. Creating a new `Task`

This feature facilitates creation of new tasks.

4.3.2.1. 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 return false, 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.

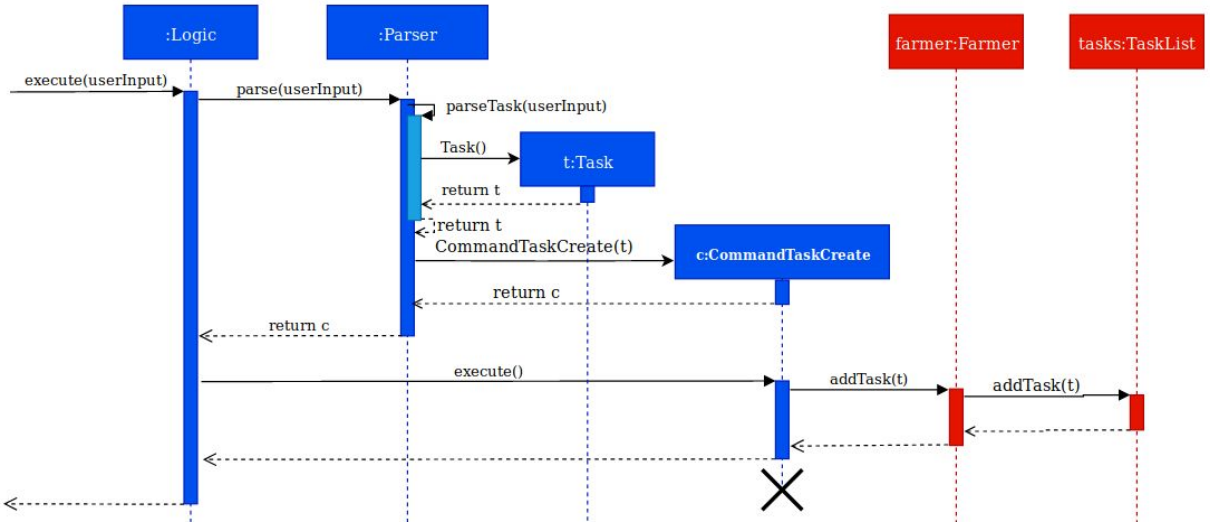


Figure 4.3.2.1.1. Sequence Diagram for Creating and Executing `CommandTaskCreate`

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`.

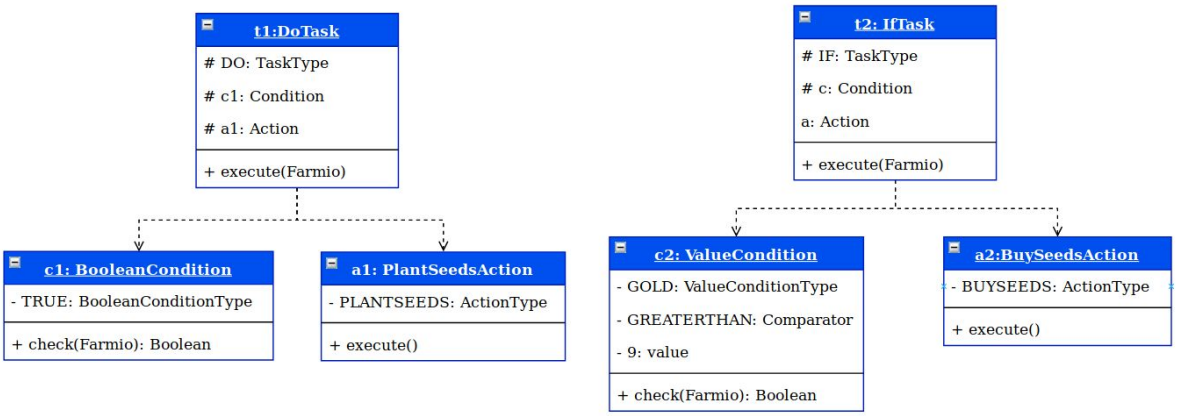


Figure 4.3.2.1.2. Class Diagram of Two Different `Task` Objects, `t1` and `t2`

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

4.3.2.2. Design Considerations

Aspect: Validation of conditions and actions

- Alternative 1 (current choice): Implement it as a static method in `Condition` and `Action` classes
    - Pros: Improves maintainability of code, as validation of conditions and actions are abstracted, making the various methods easier to understand and modify
    - Pros: Improves cohesion of code, as the methods for validating the conditions and actions are in the `Condition` and `Action` classes
    - 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
    - Cons: Poor cohesion, and the `Parser#parseTask()` method may become buggy and difficult to understand as it gets longer

4.3.3. Inserting a new `Task`

This feature is used when the user wants to insert a new task between existing tasks currently on the tasklist.

4.3.3.1. Implementation

This feature is similar to the previous feature for creating a new `Task` object, and reuses the following method:

- `Parser#parseTask()` - Returns a `Task` created from user input

Additionally, it implements a new method:

- `Parser#parseTaskInsert()`- Returns a `Command` that inserts a new `Task` in the `TaskList`

The steps below describe the high level behaviour for the task insertion mechanism for the example where the user inputs the command `insert 1 do goToMarket`.

Step 1. The user enters the command `insert 1 do goToMarket`. `Logic` will then invoke `Parser#parse()` which then calls `Parser#parseTaskInsert(insert 1 do goToMarket)`.

Step 2. `Parser#parseTaskInsert()` will then extract the substring that specifies the position to insert the new task at, as well as the substring that describes the new task.

Step 3. `Parser#parseTaskInsert()` will then utilize the `Parser#parseTask()` method to create the `Task` object to be inserted at the specified position. A `CommandTaskInsert` object is then created and returned to `Logic`.

Step 4. `Logic` then calls the execute method of the `CommandTaskInsert` object. It first validates that the user specified position is indeed a valid position in the current `TaskList` and then adds the new `Task` into the `TaskList`.

Similar to task creation, no tasks can be inserted once the task list has 18 elements

The sequence diagram below shows the high-level behaviour of the task insertion mechanism:

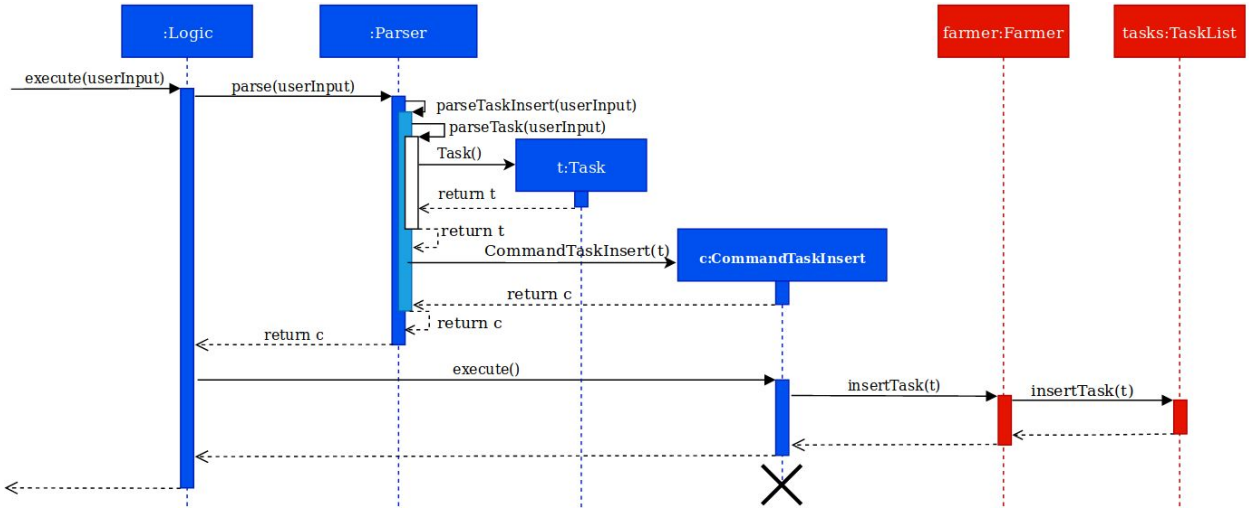


Figure 4.3.3.1.1. Sequence Diagram for Task Insertion Mechanism

4.3.3.2. Design Considerations

Aspect: Validation of user specified position

- Alternative 1 (current choice): Validate in command execution
    - Pros: `Parser` class does not access `TaskList`, improving coupling
    - Cons: `Command` has to do input validation, poor cohesion
- Alternative 2: Validate inside `Parser` class
    - Pros: Better cohesion as all validation of input is done in `Parser`
    - Cons: Poor coupling, as `Parser` would need to access game assets

4.3.4. Editing an Existing `Task`

This feature allows users to edit existing tasks in their tasklist by replacing it with a new task.

4.3.4.1. Implementation

The editing mechanism replaces a current `Task` in the `TaskList`, and is facilitated by the set method of the `ArrayList`. Its implementation is similar to that of the task insertion feature. It reuses the following method:

- `Parser#parseTask()` - Returns a `Task` created from user input

Additionally, it implements a new method:

- `Parser#parseTaskEdit()` - Returns a `Command` that edits an existing `Task`

The following steps describe the high-level behaviour of the task editing mechanism for the scenario where the user inputs the command `edit 2 do sellGrain`.

Step 1. The user inputs `edit 2 do sellGrain` command. `Logic` will then invoke `Parser#parse()` which then calls `Parser#parseTaskEdit(edit 2 do sellGrain)`.

Step 2. `Parser#parseTaskEdit()` will then extract the substring that specifies the index of the task to replace, as well as the substring that describes the new task.

Step 3. `Parser#parseTaskEdit()` will then utilize the `Parser#parseTask()` method to create the `Task` object to replace the current `Task` at the specified index. A `CommandTaskEdit` object is then created and returned to `Logic`.

Step 4. `Logic` then calls the execute method of the `CommandTaskEdit` object first validates that the user specified index is indeed a valid index in the current `TaskList` and then replaces the current `Task` with the new `Task` in the `TaskList`.

The object diagrams below illustrate an example of how the editing mechanism modifies the `TaskList`.

Before the user enters the edit command:

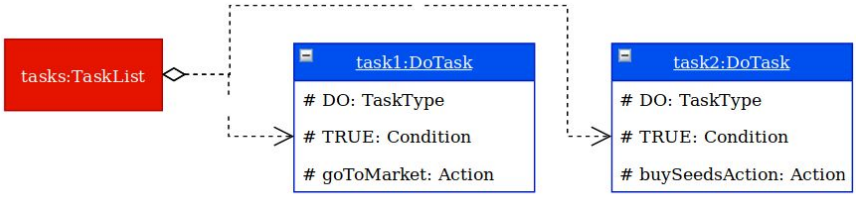


Figure 4.3.4.1.1. Object Diagram of an Example `TaskList`

After user enters edit command `edit 2 do sellGrain`:

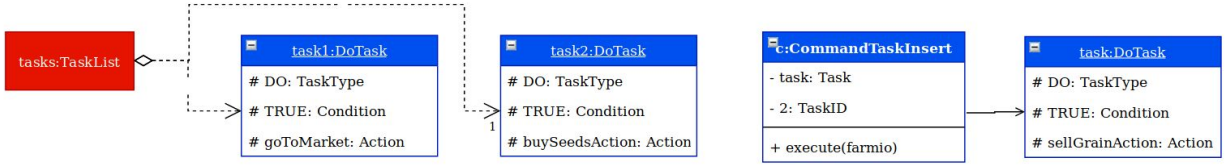


Figure 4.3.4.1.2. Object Diagram of the Example `TaskList` and `CommandTaskEdit`

After `CommandTaskEdit` is executed:

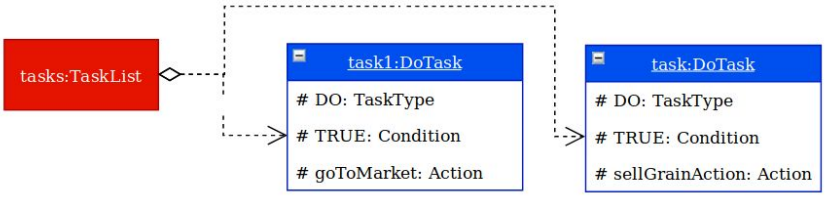


Figure 4.3.4.1.3. Object Diagram of Example `TaskList` after it is Edited

4.3.4.2. Design Considerations

Aspect: Editing Method

- Alternative 1 (current choice): A new `Task` replaces the old `Task`
  - Pros: Easy to implement.
  - Cons: User has to input long command including the task description.
- Alternative 2: Replace either `Condition` or `Action` of the `Task`
  - Pros: User is able to edit a specific part of the `Task` easily.
  - Cons: Difficult to implement.

4.3.5. Deleting `Task` objects

The user can delete a task or all tasks using the delete command.

4.3.5.1. Implementation

To improve user experience, this feature allows users to delete a single task, or delete their entire tasklist if they wish to start over entirely. As such, this feature is facilitated through two `Command` objects:

- `CommandTaskDelete` - Removes one `Task` from the `TaskList`
- `CommandTaskDeleteAll` - Clears the `TaskList`

It also implements a new method:

- `Parser#parseTaskDelete()` - Parse user commands for deleting one object

Deleting a Single Task:
The steps below describe the mechanism for deleting a single task:

Step 1. The user inputs the command `delete 1`. `Logic` will then invoke `Parser#parse()` which then calls `Parser#parseTaskDelete(delete 1)`.

Step 2. `Parser#parseTaskDelete()` will then extract the substring that represents the index of the `Task` the user wishes to delete. It will then create the `CommandTaskDelete` object which is returned to `Logic`.

Step 3. `Logic` executes the `CommandTaskDelete` object which then removes the specified `Task` from the `TaskList` using the remove method of `ArrayList`.

Deleting all Tasks:

Step 1. The user inputs the command `delete all`. `Logic` will then invoke `Parser#parse()` which returns the `CommandTaskDeleteAll` object to `Logic`.

Step 2. `Logic` will then execute the `CommandTaskDeleteAll` object, which removes all elements inside the `TaskList` using the clear method of `ArrayList`.

4.3.5.2. Design Considerations

Aspect: Validation of user specified index for deleting one task

- Alternative 1 (current choice): Validate when executing the `Command`
    - Pros: `Parser` class does not need to access the `TaskList` to check the number of elements
    - Cons: Poor cohesion as the `Command` has to do user input validation
- Alternative 2: Validate inside the `Parser` class
    - Pros: Improves cohesion as the `Command` would not have to do any input validation
    - Cons: `Parser` class would need to access the `TaskList`, increasing coupling

4.4. Running User Created Tasks Feature

All Tasks added by the user will be stored in a `TaskList`. This feature is used when the user is done with his input and starts running the code.

4.4.1. Implementation

This feature goes through the task list after the user has completed creating tasks, and executes the user created tasks one by one.
The running user created tasks feature consists of the following methods:

- `Command#CommandDayStart()` - Shows and sets the start of a new day.
- `Command#CommandTasksRun()` - Runs the `TaskList` and prepare to check if the level's objectives are met.
- `Command#CommandCheckObjectives()` - Checks if the objectives are met and sets the corresponding stage according to the results.
- `Command#CommandDayEnd()` - Sets and shows the end of the day and sets the next day.

Given below is an example usage scenario of how the running tasks feature behaves for a tasklist where the user has created only one task that plants seeds.

Step 1: The User will be prompted to input `start` for the game to run their code. The parser will then initialize `Command#CommandDayStart()` object. The purpose of `Command#CommandDayStart()` is to change the stage in Farmio class from `DAY_START` to `DAY_RUNNINF.`

Step 2: The parser will check which stage is the game at. In this case the current stage is `DAY_RUNNINF.` Upon checking, the parser will call the respective game state command. In this case, the parser will create `Command#CommandTasksRun()` and `Logic` will execute it.

Step 3: `Command#CommandTasksRun()` will call the `Farmer#startDay()` method in `Farmer`. The `Farmer#startDay()` method will execute all the tasks in TaskList which consists of all the tasks added by the user. In this example, the action to be executed in the `TaskList` will be `plantSeedAction`.

Step 4: The abstract `Action` class will receive the task to be executed. If the condition in the task is an ‘if’ condition, `IfTask` will check whether the condition is satisfied from the `Condition#check()` method in the ConditionChecker class. In this case, the task type is a ‘do’ task and since ‘plantSeeds’ is a valid action, `Action#PlantSeedAction()` will be called.

Step 5: Inside the class `Action#PlantSeedAction` which extends the abstract `Action` class, the class checks whether the conditions are met. For example, the farmer can only plant seeds if the farmer is in the WheatFarm and has enough seeds. If these conditions are not met, the program will return an error message to the users. If the conditions are met, the simulation will be animated through the simulation feature which will be elaborated on below and we will call the method `WheatFarm#plantSeeds()` in the class WheatFarm through `Farmer#farmer.plantSeeds()` to change the value of seeds and seedlings the user has.

Step 6: After executing all the tasks in the `TaskList`, `Command#CommandTasksRun()` will change the game state to `CHECK_OBJECTIVES`. The parser will then call the `Command#CommandCheckObjectives()` method in the CommandCheckObjective Class to check whether the objectives are met.

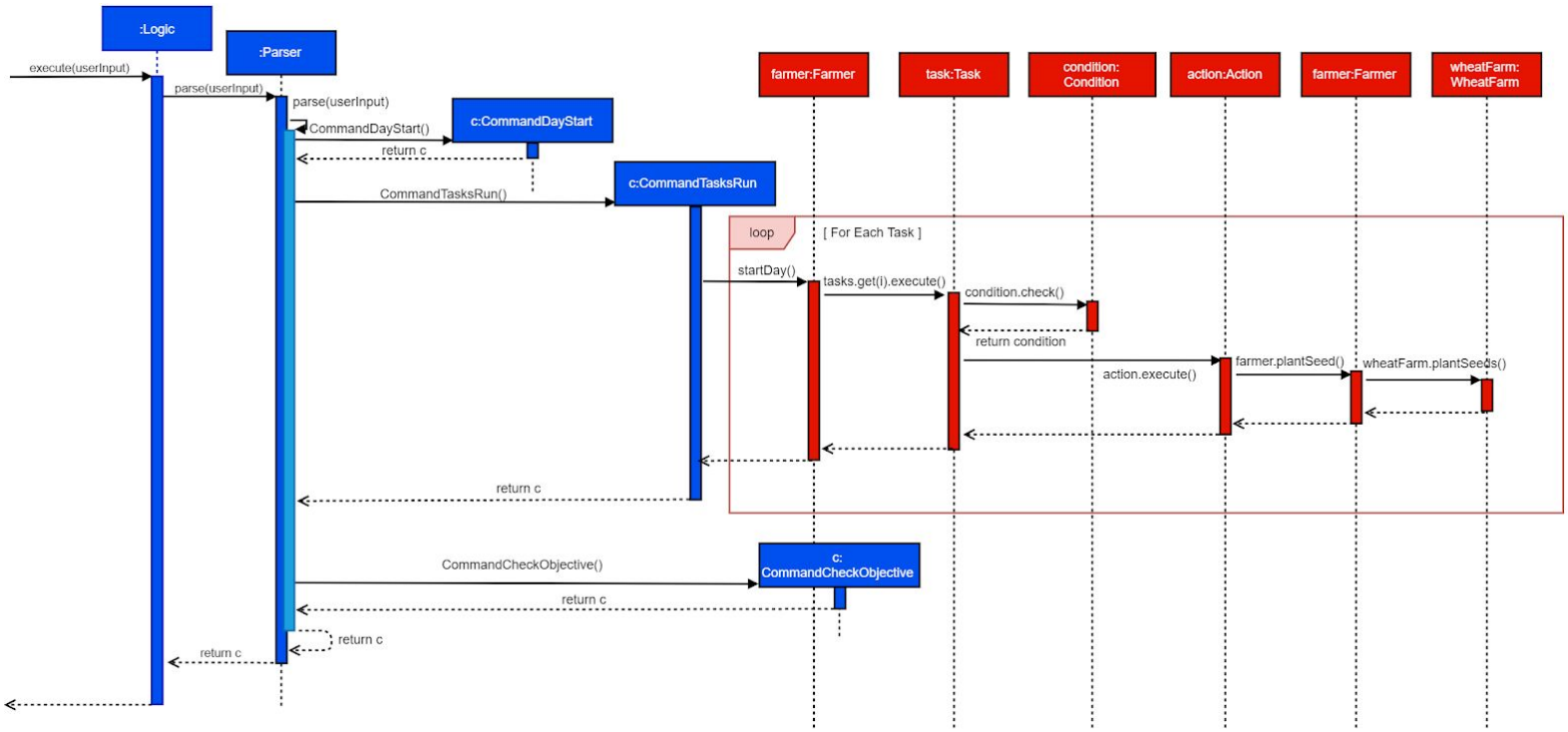


Figure 4.4.1.1. Sequence Diagram for Task Running Mechanism.

4.4.2. Design Considerations

Aspect: Execution of tasks in the task list

- Alternative 1 (current choice): Use the execute method in the `Task` object
    - Pros: Improves abstraction, improves maintainability of the `Farmer#startDay()` method.
    - Cons: Increases complexity of `Task` object, as it has to do condition validation instead of simply being a container for a condition and action.
- Alternative 2: `TaskList` checks the `Condition` contained in the `Task` and calls the execute method of the `Action`.
    - Pros: The `Task` class will be much simpler and easy to maintain, as it would just be a container
    - Cons: Poor maintainability for `TaskList`, and poor cohesion as it would then become responsible for the low level steps of task execution instead of simply calling the execute method of the `Task` object.

4.5. Task Visualisation Feature

The Task visualisation feature allows users to better understand what happens during task execution. With this feature, the user can easily see which task is currently executing, asset changes as well as goals accomplished.

4.5.1. Implementation

At its core, it utilises `GameConsole` to create a single output frame that formats and highlights changes in important variables from `Game Assets`. `Simulation` is then used to show one or more output frames in sequence which creates an animation.

This feature is facilitated by `GameConsole` which consist of the following key methods:

- `GameConsole#formatAndHighlightCode()` - Highlights current task.
- `GameConsole#formatGoals()` - Highlights completed goals.
- `GameConsole#formatAssets()` - Highlights changes in assets.
- `GameConsole#fullConsole()` - Compiles the entire console and creates a single output frame.

The method for showing one or more output frames created by `GameConsole#fullConsole()` is exposed in the `Frontend` class as `Frontend#simulate()`.

`Frontend#simulate()` is an overloaded method. As such, it is able to either show a single output frame of `GameConsole#fullConsole()` or multiple output frames as an animation.

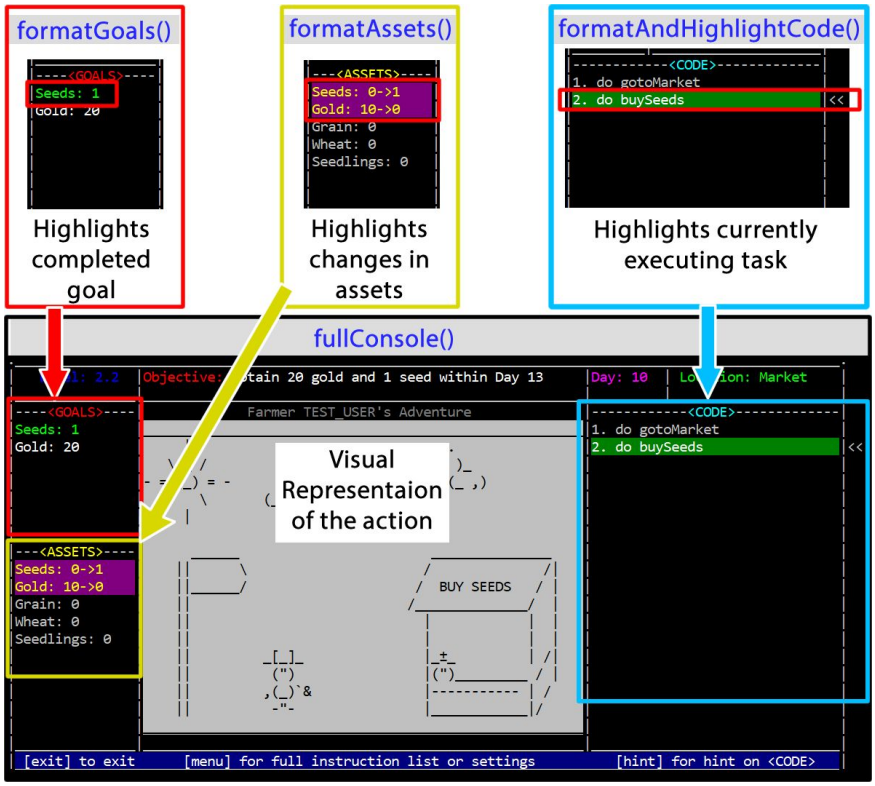


Figure 4.5.1.1. Diagram showing how an Output Frame is created

Given below is an example usage scenario. Let task be a `DoTask` which consist of an action of type `BuySeedsAction`. For this example, we shall assume that the task executes successfully. (Section 4.6. describes what happens if an error occurs.)

Step 1: The task calls the execute method of the action.

Step 2: This would in turn call `Action#checkActionCriteria()`. For this example, we assume that the corresponding `Action` criterias are met and thus this method does not produce feedback.

Step 3: At this point, `Frontend#simulate()` is called which in turn calls `GameConsole#fullConsole()` repeatedly to show a visual representation action in the center of the console and highlights the current task in green.

Step 4: Next, the method `Farmer#buySeeds()` and `Farmer#spendGold()` would decrease gold and increase the number of seeds by 1 respectively.

Step 5: `Frontend#simulate()` will then be called again. This time, the change in assets and a completed goal will be highlighted with a result similar to that in Figure 4.5.1.1.

The sequence diagram below shows the high-level behaviour of the task visualisation mechanism for the example above.

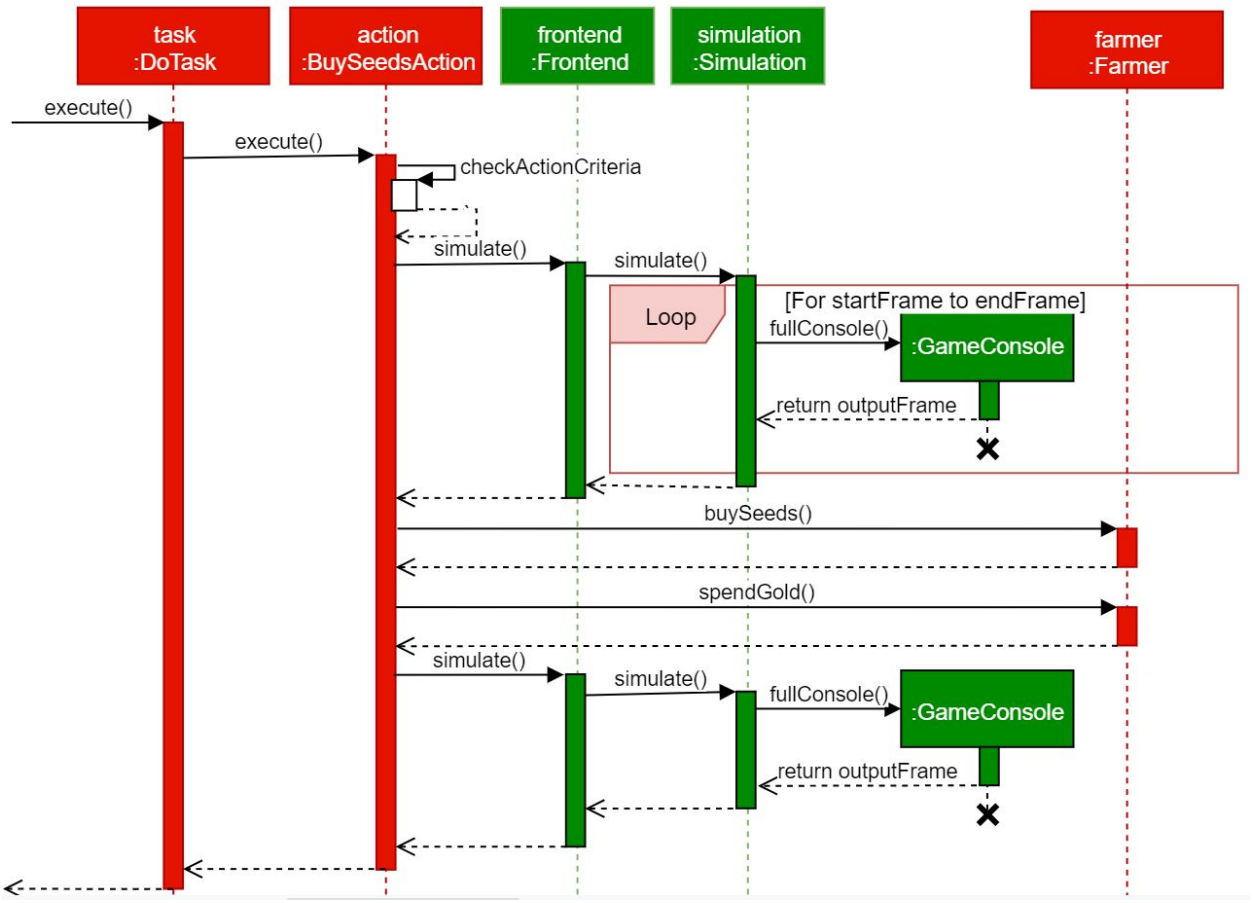


Figure 4.5.1.2. Sequence diagram for Task Visualisation Mechanism.

4.5.2. Design considerations

Aspect: Interface used for task visualisation feature

- Alternative 1 (current choice): On Command Line Interface.
    - Pros: Lower system requirements and less memory used.
    - Cons: The feature will be less dynamic. For example, users are unable to get immediate feedback when issuing commands during task visualisation.
- Alternative 2: On Graphical User Interface using javafx.
    - Pros: Better graphics may improve the aesthetics of this feature.
    - Cons: More complex. Higher system requirements and difficult to implement

4.6. Task Error Detection Feature

The Task error detection feature identifies and highlights tasks with errors and states which criteria of the action is not met.

4.6.1. Implementation

This error detection mechanism is facilitated by the `Action` abstract class and it's child classes. `Frontend` is used to format and show the error effectively. This feature is implemented through the following methods.

- `Action#execute()` - Executes the action.
- `Action#checkActionCriteria()` - Checks if the action criteria is met.
- `Farmer#setTaskFailed()` - Marks the current running task as failed.
- `Frontend#simulate()` - Highlights the task that triggered the error.
- `Frontend#show()` - Shows the reason for the error.

Below describes the high level behaviour of the error detection mechanism at each step when the task `do buySeeds` is executed but the action's criteria is not met. For example, the Farmer's location is not at the Market.

Step 1. In this case, the `DoTask#execute()` method will call the `BuySeedsAction#execute()` method to execute the action.

Step 2. However, as the Farmer being at the market is a criteria to buy seeds, `Action#checkActionCriteria()` would call `Farmer#setTaskFailed()` to mark that the current task has failed.

Step 3. `Frontend#simulate()` will then be called, triggering a cascade of method calls. Similar to section 4.5., `GameConsole#formatAndHighlightCode()` is eventually called.

Step 4. `GameConsole#formatAndHighlightCode()` is responsible for detecting that the currently running task is marked as failed and thus, highlighting it in red instead of green in the GameConsole.

Step 5. `Action#checkActionCriteria()` will then call `Frontend#show()` to show the reason why the task has failed (the farmer not being at the market in this case).

If more than one criteria for an action is not met, this feature will ensure that each explanation for every unmet criteria is generated by `Frontend#show()`.

Step 6. Lastly, a Farmio exception is thrown to signal the change in the stage of `Farmio` class to `LEVEL_FAILED` and the level is reset.

The sequence diagram below shows the behaviour of the task error detection mechanism. This diagram is similar to Figure 4.5.1.2. above but with more focus on `Action#checkActionCriteria()`.

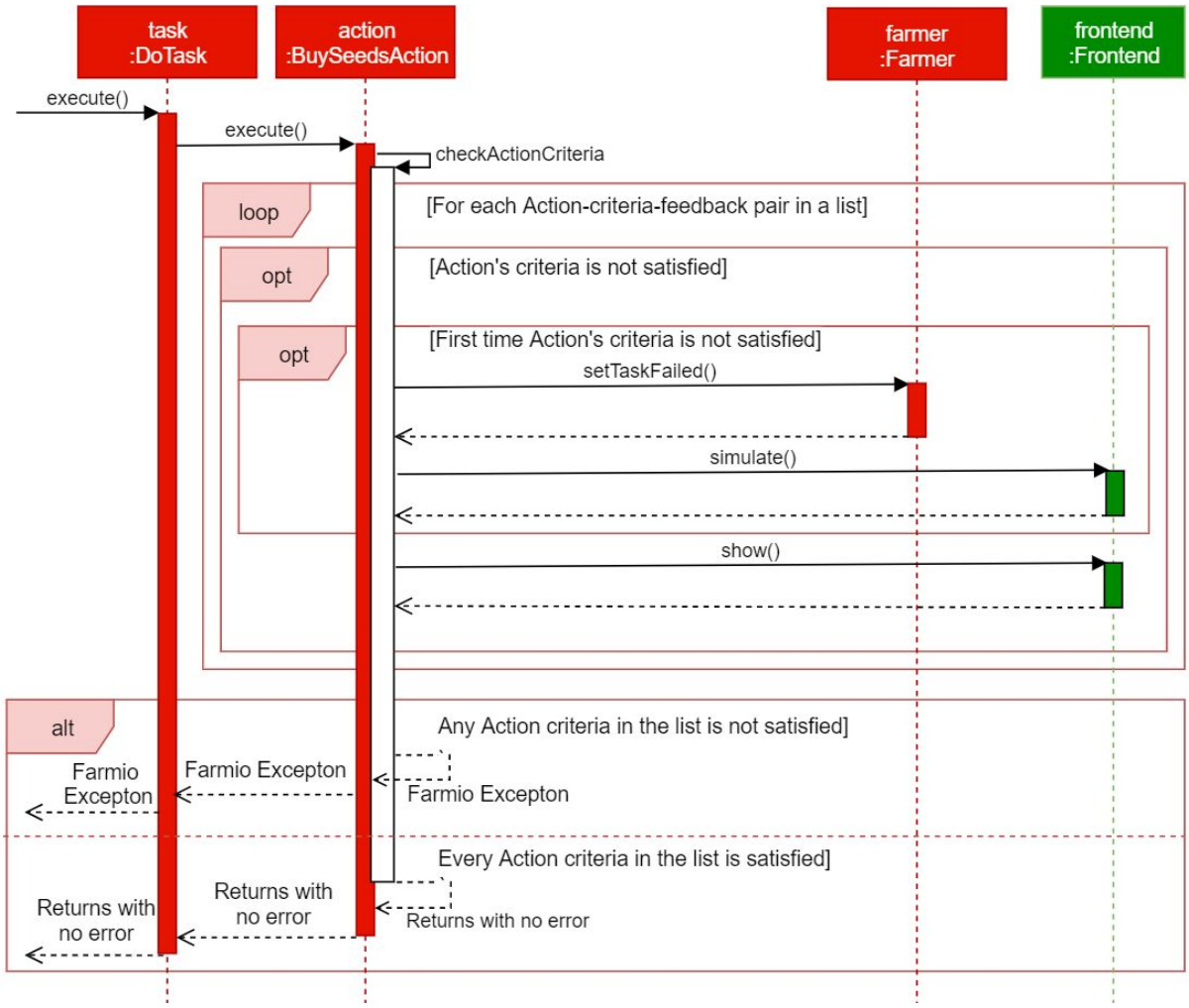


Figure 4.6.1.1 Sequence Diagram for Task Error Detection Mechanism

4.6.2. Design considerations

Aspect: Implementation of the error detection feature.

- Alternative 1 (current choice): Implemented the error detection method in `Action` abstract class.
    - Pros: Increased cohesion. Classes using the same methods would not need to define them separately. By localising the method, it is also easier to modify them.
    - Cons: Decreased coupling. Action child classes are more reliant on each other.
- Alternative 2: Implement individual checking methods in each execute method of the `Action` child class.
    - Pros: Increase coupling. Classes would depend less on one another. If the developer chooses to change the mechanics of one method, it will have less effect on the rest.
    - Cons: Decreased cohesion and increased complexity.

4.7. Level Progression and Reset feature

The level progression and reset feature is responsible for allowing the game to progress to the next level or whether to reset to the previous level.

4.7.1. Level Progression

The level progression feature allows the user to progress to the next level of the game if the user successfully completes the objectives for the current level.

4.7.1.1. Implementation

The level progression feature is facilitated by the `Command#CommandCheckObjectives()`, `Command#CommandLevelEnd()` methods and the `Level` class.

The `Command` class also utilises these methods from the `Level` class to check whether the objectives for the level have been met before attempting to progress onto the next level.

- `Level#checkAnswer()` - is used to check on the state of the state of the level ,whether the user has completed the task successfully (DONE), have yet to complete the task but is within the allocated deadline (NOT_DONE),encountered an error during code execution(INVALID) or exceeded the stipulated deadline (FAILED) for the level.
- `Level#checkDeadlineExceeded()` - checks whether the user has exceeded the deadline.
- `Level#allDone()` - checks whether the user has completed all the tasks.

The steps below explain the behaviour of the level progression feature

Step 1: At the end of each day, the `Command#CommandCheckObjectives()` method will call the `Level#checkAnswer()` method in the `Level` class. The `Level#checkAnswer()` method will then invoke the `Level#checkDeadlineExceeded()` and `Level#allDone()` method to check whether the objectives for the level has been met within the preallocated deadline and will return either one of these `Level.ObjectiveResult` state - (NOT_DONE/ DONE/ FAILED/ INVALID)

Step 2 : If the objectives has been met for the level by the user. The `Command#CommandCheckObjectives()` method will then change the `STAGE` in the `Farmio` class to LEVEL_END. The Parser class will then call the `Command#CommandLevelEnd()` method.

Step 3 : The `Command#CommandLevelEnd()` method obtains the Farmer object. It updates the current level of the Farmer object to the next level before creating a new `Level` object instance. The `Level` class will then obtain values from a `JSON file` object to initialise and set the objectives for the next game level.

4.7.1.2 Design Considerations

Aspect:Setting of Farmio.Stage to the various States

- Alternative 1 (current choice): Set Stage once the checkAnswer Method is returned
    - Pros: Reduces coupling since setting of stage is no longer dependent on checkAnswer
    - Cons: Reduces Cohesion since the checkAnswer method should set the Stage
- Alternative 2: Set Farmio.Stage inside checkAnswer()
    - Pros: Increases Cohesion
    - Cons: More complex and increases couplinF.

4.7.2. Level Reset

The Level Reset feature will reset the level of the game back to its initial state.There are 3 instances which would lead to the resetting of a level.
1. The first is if the user decides to initiate the reset manually when prompted with the option at the end of each day.
2. The user fails to complete the level within the allocated deadline.
3. There is an error encountered during code execution

4.7.2.1. Implementation

The Level Reset feature is facilitated by the following methods

- `Command#CommandCheckObjectives()` - checks whether the user has exceeded the deadline, completed the objectives or whether they encountered an error during task execution.
- `Command#CommandLevelReset()` - initiates the reset for the level.
- `Parser#parseDayEnd()` - checks whether the user wants to reset at the end of each day.

The steps below show the behaviour of the level reset feature:

Step 1: At the end of each day or level , the `Command#CommandCheckObjectives()` method will be invoked. It will check whether the objectives of the level has been met and whether the user has exceeded the deadline. It follows the same implementation for the checking objectives as level progression.

Step 2.1.1 : If the user has exceeded the deadline or if the user encountered an error during task execution, `Farmio.Stage` will be set to the `LEVEL_FAILED` state. `Parser#parse()` will then invoke the `Command#CommandLevelReset()` method.

Step 2.1.2 : If the user has not exceeded the deadline and have yet to complete the task , `Farmio.Stage` will be set to the `DAY_END` state. `Parser#parseDayEnd()` method will then check whether the user had decided to key in “reset”. If yes the `Command#CommandLevelReset()` method will be invoked

Step 2.3 : `Command#CommandLevelReset()` method will then reset the game level back to its initial state

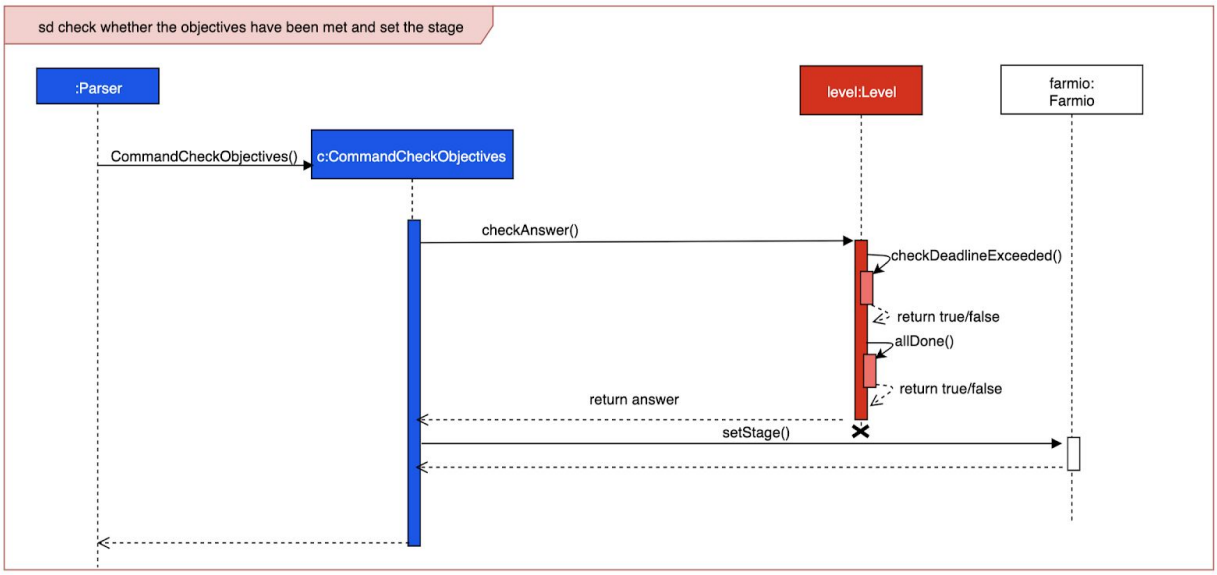


Figure 4.7.2.1.1. Sequence Diagram for Checking Objectives and Setting of Stage

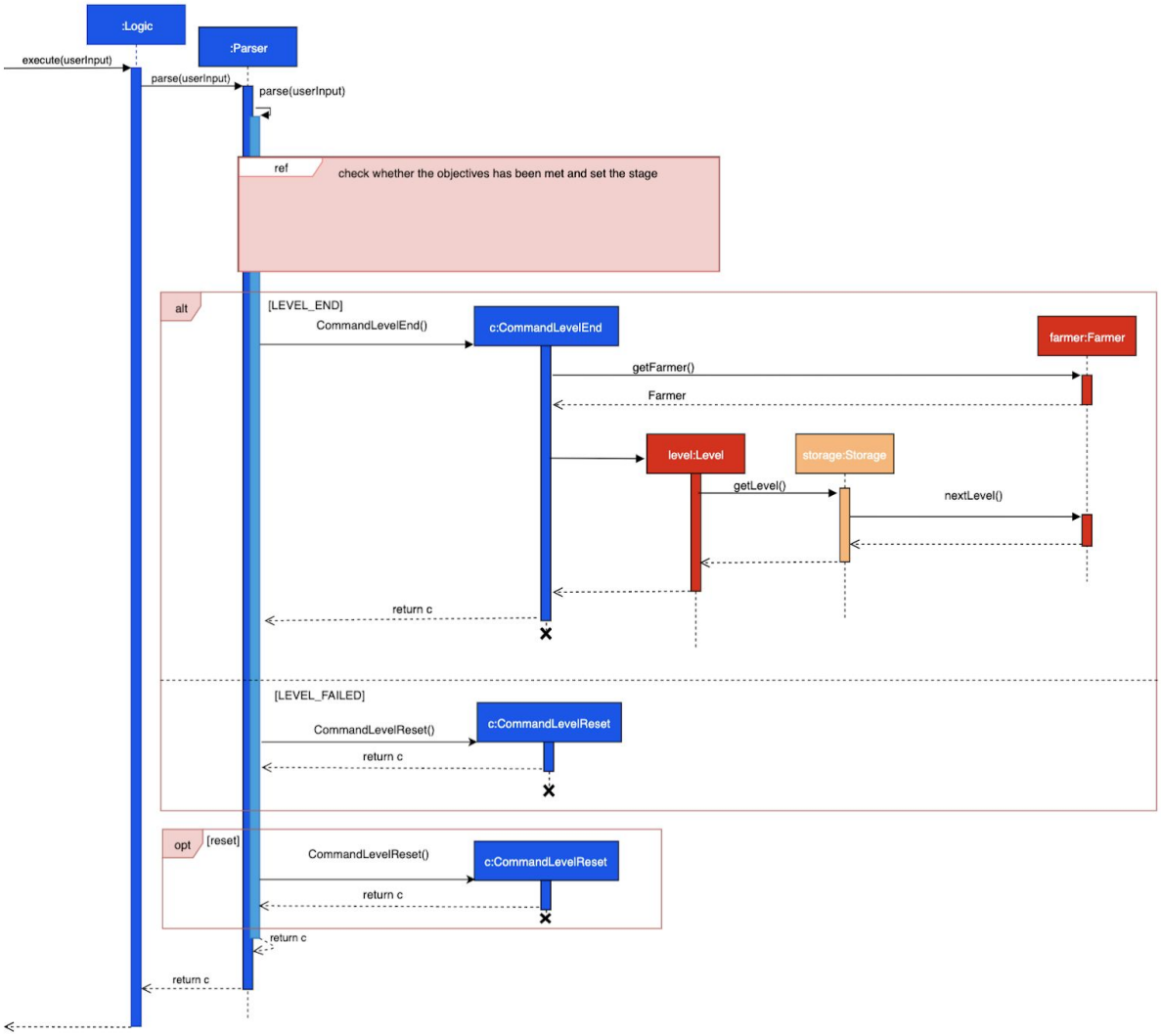


Figure 4.7.2.1.2 Sequence diagram for Level Progression and Reset Mechanism

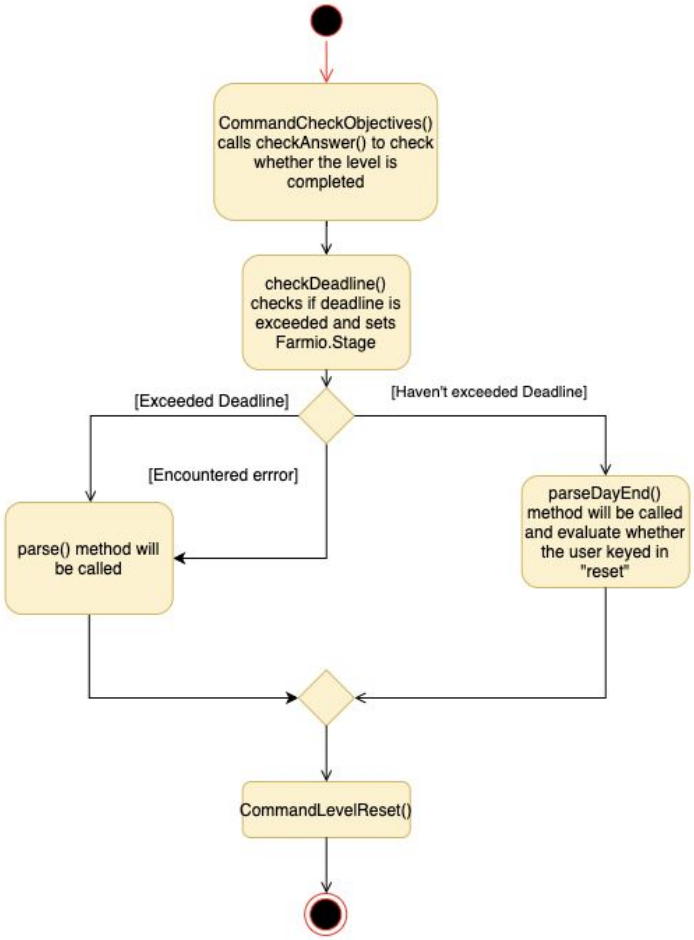


Figure 4.7.2.1.3.Activity Diagram for Reset Mechanism

4.7.2.2. Design Considerations

Aspect: Resetting Level

- Alternative 1 (current choice): Reset function called under CommandReset method
    - Pros: Reduces coupling and Increases Cohesion
    - Cons: More complex implementation
- Alternative 2: Resets without whenever desired without the Command Reset method
    - Pros: Easier to trace
    - Cons: Increases Coupling since the reset function is now dependent on multiple functions.

4.8 Feedback feature

4.8.1 Implementation

The feedback feature allows user to see their performance for the level. Feedback will be provided when the user has successfully completed or unsuccessfully completed the task. Feedback is carried out when `Command#CommandCheckObjectives()` is invoked. Feedback utilises `Frontend` to display feedback to the user.This feature is facilitated by the following methods.

- `Level#getFeedback()` - is used to obtain feedback
- `Level#getSuccessfulFeedback()` - feedback when the user passes the level
- `Level#getDetailedFeedback()` - feedback when the user fails the level
- `Level#getPermutationFeedback()` - provides feedback on whether the user has the required number of tasks and if the user does have, how close it is to the model answer for the level
- `Farmio#getLevel()` - returns the current level of the user
- `Level#checkAnswer()` - checks whether tasks has been accomplished or failed, returns the `ObjectiveResult` state

Steps for obtaining Feedback

Step 1 : When `Command#CommandCheckObjectives()` is invoked, it will call first call `Farmio#getLevel()` method to obtain the current level after which `Level#checkAnswer()` - will be called to determine the `Level.ObjectiveResult` state. The `Level#getFeedback()` method will then be called, which will assess the `Level.ObjectiveResult` state to determine the correct type of feedback to be provided to the user.

Step 2.1: If the `Level.ObjectiveResult` state was set to `DONE`, the `Level#getSuccessfulFeedback()` will be called. This method will then retrieve the appropriate information on feedback from the `Level` class instance and return the feedback.

Step 2.2: If however the `Level.ObjectiveResult` state was set to `FAILED`, the `Level#getDetailedFeedback()` will be called instead. This method will then call `Level#getPermutationFeedback()` to obtain feedback for the different permutations of tasks which was executed during the level, before returning the desired feedback.

Step 2.3:The `Command#CommandCheckObjectives()` will then utilise `Frontend` to display the feedback to the user.

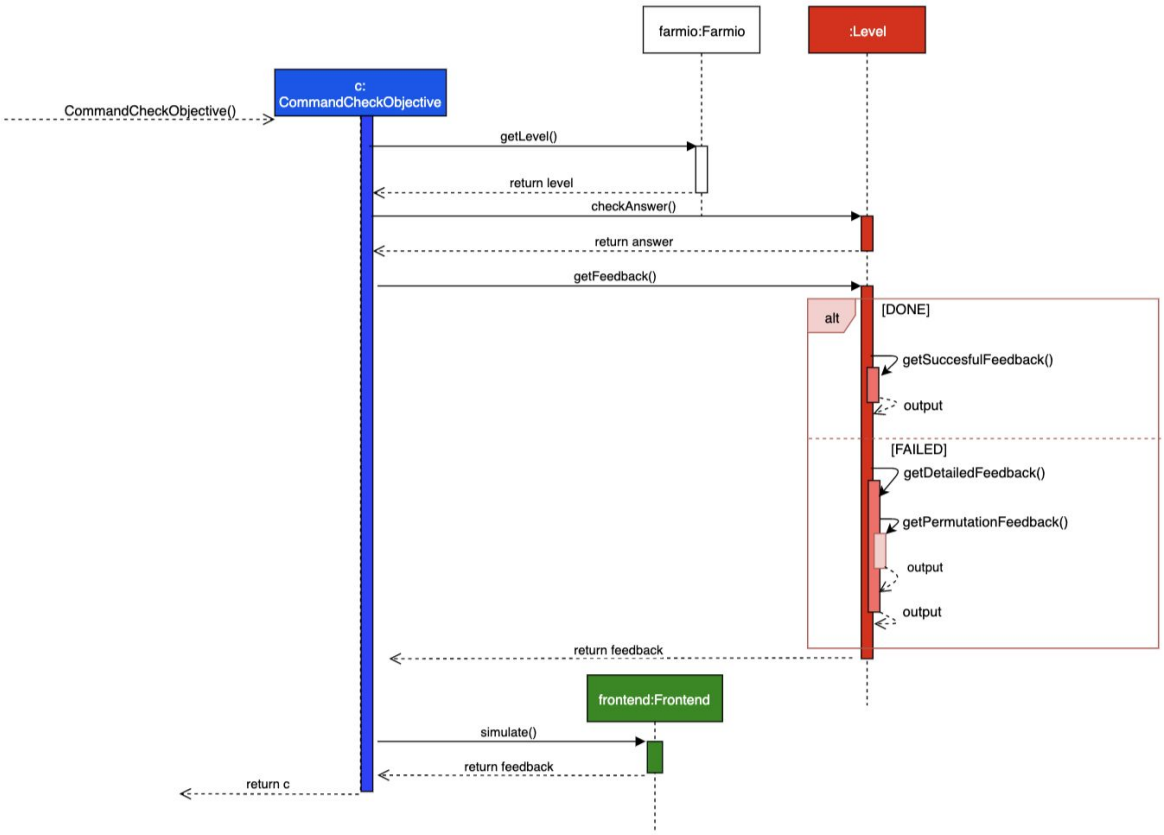


Figure 4.8.1.1. Sequence diagram for feedback mechanism

4.8.2. Design Considerations

Aspect: Fetching and displaying feedback to the User

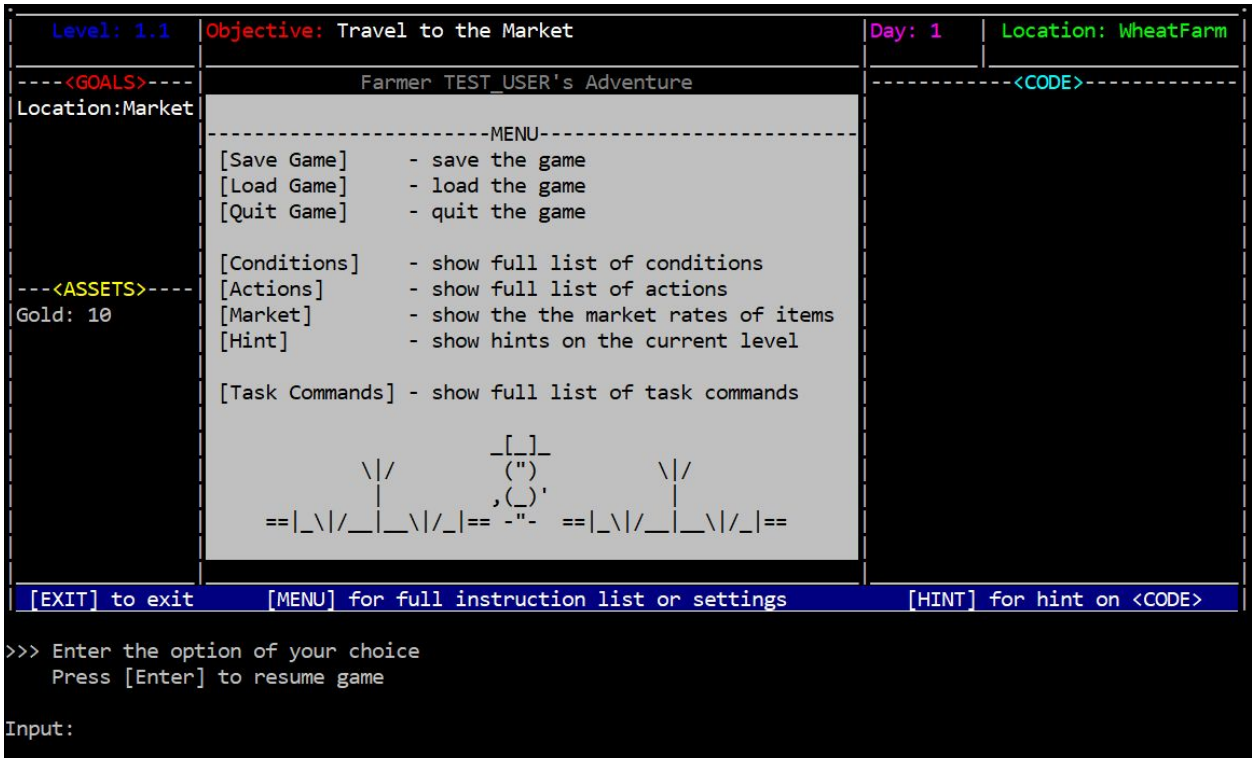
- Alternative 1 (current choice): feedback is displayed only after executing `Level#checkAnswer()`
    - Pros: Decreases coupling since the typewriter UI is not dependent on checkAnswer
- Alternative 2: feedback is displayed whilst `Level#checkAnswer()` is executed.
    - Pros: Increased Cohesion since checkAnswer displays the feedback to the user
    - Cons: Increased Coupling since user feedback now depends on the checkAnswer() method

4.9 In-Game Menu and ShowList Feature

The in-game menu feature brings the user to the in-game menu. The users will view a list of commands they are able to use from the menu. The in-game menu implements the `Command#CommandShowList()` Commands as well as the Save game and Load game commands.

4.9.1. In-Game Menu

The in-game menu consists of a simulated frame that includes all the commands the user can use to complete the level. The in-game menu stage is part of the TaskAdd stage. This means that user can still create, edit, insert or delete tasks as stated in section 4.3.



4.9.1.1 Implementation

The in-game menu mechanism is done through the `Logic` class, which uses `Parser` and `Command` to implement this feature. It shows a simulated frame to users which states the commands they can use to aid them in completing the level.

- `Command#CommandMenuInGame()` - Simulates a in-game menu frame.
- `FrontEnd#getFrontEnd()` - returns a new FrontEnd object.
- `Menu#showMenu()` - Determines whether the menu is a welcome menu or an in-game menu and simulate the respective frame.

Step 1: When the user inputs `menu` in the command line, the Parser will detect the command and returns a Command#CommandMenuInGame() method.

Step 2: The command will access the `Menu#showMenu()` method from the `FrontEnd` object by using the `FrontEnd#getFrontEnd()` Method.

Step 3: The `showMenu()` Method will access the `Menu#show()` method. The `Menu#show()` method consists of three arguments. The main difference between the starting menu and the in-game menu is the third argument. If `canResume` is true, it will simulate the in-game menu. You can refer to point 3.4 for a detailed analysis of the Simulation process.

Step 4: If the user would like to access any of the showList, he can just enter the commands as stated in the menu. For example, if the user would like to access the `ActionList`, the user can input `action`.

Step 5: if the user would like to return to the stage he left off, the user can just press `Enter`.

4.9.1.2 Design considerations

Aspect: Change in game stage

- Alternative 1 (current choice): The in-game menu stage is within the task add stage
    - Pros: Easy to implement as no changing of stages is required
    - Pros: User can easily type in commands directly to add tasks.
    - Cons: User
- Alternative 2: Create an independent stage for the in-game menu
    - Pros: Improves navigability for the user as the user can only use commands related to the menu
    - Cons: Difficult to implement which makes the menu quite buggy.

4.9.2 ShowList

The showList consists of a simulated frame that includes all the actions, conditions, the prices of assets and commands that the user can use to complete the level. The showList stage is part of the TaskAdd stage. This means that user can still create, edit, insert or delete tasks as stated in point 3.3.

4.9.2.1 Implementation

The ShowList feature consists of the `actionList`, `conditionList`, `marketList` and `taskcommandList`.
The showList mechanism is done through the `Logic` class, which uses `Parser` and `Command` to implement this feature. It shows a simulated frame to users which states the respective actions, conditions, the price of assets that can be bought or sold in the market and the edit and delete commands syntax.

The showList feature consists of the following methods:

- `Command#CommandShowList(String userInput)` - Simulates the respective showList based on the user input.
- `FrontEnd#getFrontEnd()` - returns a new FrontEnd object.
- `Simulate#simulate()` - shows the respective frame for the showList.
- `Farmer#getLevel()` - Gets the current level of the user.

Step 1: The ShowList refers to the combination of the actionList, conditionList, marketList and taskcommandList. When the user inputs action, condition, market or

task command, the Parser will detect the command and call the `Command#CommandShowList(userInput)`.

Step 2: The `Command#CommandShowList(userInput)` will save the user input as a string and gets a new FrontEnd object by using `farmio.getFrontend()` method.

Step 3: The showList differs from levels, thus we will attain the current level of the user using Farmer#getLevel() method.

Step 4: Based on the user Input, the respective frame will be simulated on the game console. This is done by using the Simulate#simulate() method.

Step 5: While accessing the showList pages, users are still at the TASK_ADD stage. Users will still be able to create, edit, insert or delete tasks as elaborated in section 4.3.2.

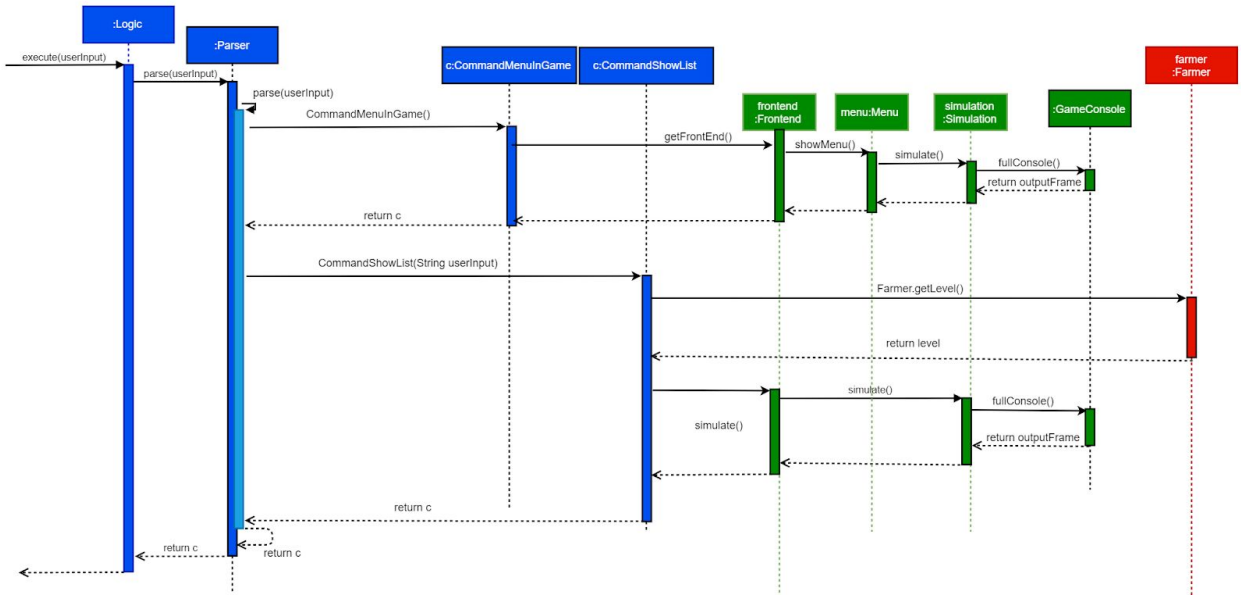


Figure 4.9.2.1.1. Sequence diagram for in-game menu and showList mechanism

4.9.2.2. Design considerations

Aspect: Setting different game stage

- Alternative 1 (current choice): The in-game menu stage is within the task add stage
    - Pros: Easy to implement
    - Pros: User can easily type in commands directly to add tasks.
    - Cons: User have to enter `menu` to access the menu again accessing the showList.
- Alternative 2: Create an independent stage for the in-game menu
    - Pros: Improves navigability for the user as the user can only use commands related to the menu
    - Cons: Difficult to implement which makes the menu quite buggy.

4.10. Game Logging Feature

The Logging feature allows the user to see their previous runtime code execution.

4.10.1. Populating and Clearing of the Log

The log is populated at the start of each day when the tasks are executed and is cleared whenever the user keys keys in “start” during the `TASK_ADD` stage.

4.10.1.1 Implementation

The populating and clearing of the log data will utilise the following methods

- `Command#CommandTaskRun()` - used to determine when to add to the log
- `Farmer#startDay()` - where the log is populated
- `Parser#parseTaskAdd()` - used to obtain user input and check when to start the day
- `Command#CommandDayStart()` - used to determine when to clear the log
- `Log#clearLogList()` - Clears the data in the log

4.10.1.2.1 Clearing of Log

Step 1: `Parser#parseTaskAdd()` will obtain user input.
Step 2: If `Parser#parseTaskAdd()` determines that the user keyed in “start”, it will invoke the `Command#CommandDayStart()` method.
Step 3: The `Command#CommandDayStart()` method will then check on the current `STAGE of Farmio.` If the current stage is set to `TASK_ADD` , the `Log#clearLogList()` method will be called and the contents inside the log will be emptied.

4.10.1.2.2 Adding to the Log

Steps 1: When `Command#CommandTaskRun()` method is invoked, it calls the `Farmer#startDay()` method.
Steps 2: The `Farmer#startDay()` method will then add the tasks that are being executed for that day.

4.10.1.2.3 Viewing the Log

The feature will allow the user to view their log files

The The View Logging feature will utilise the following methods

- `Parser#parseTaskLog()` - Parse the log command from the user
- `Command#CommandLog()` - executes the log command
- `Log#getLogTaskList()` - obtains the log details
- `Log#toStringSplitLogArray()`- modifies log data to appropriate format for UI

Steps 1: `Parser#parseTaskAdd()` will obtain user input. If the user enters `log [Position]` , the `Command#CommandLog()` method will be invoked.

Steps 2: The `Command#CommandLog()` method will obtain the log data when calling the `Log#getLogTaskList()` method. The `Log#toStringSplitLogArray()` method will then be used to modify the log data to a certain format.

Step 3: The log data will then be passed to `FrontEnd` to display log details to the user.

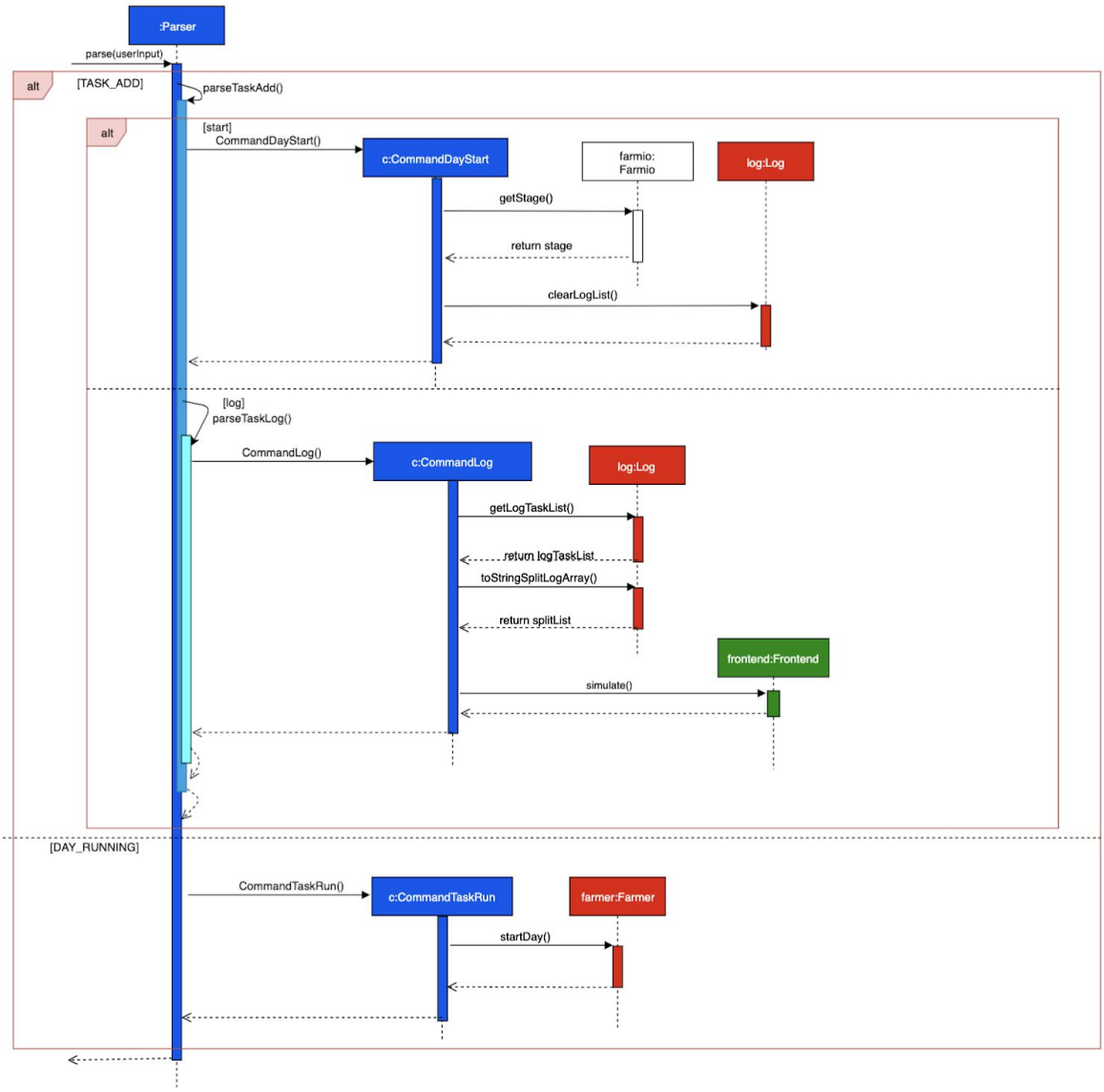


Figure 4.10.2.1 Sequence diagram for logging feature

4.10.1.2. Design Considerations

Aspect: Printing of Log

- Alternative 1 (current choice): Log displayed under a separate Command
    - Pros: Increases Cohesion
    - Cons: Increased complexity in design implementation. Increased coupling
- Alternative 2: Display Log by calling frontend immediately
    - Pros: Increased coupling
    - Cons: Decreased cohesion since is not related to log

4.11. Logging

This program uses java.util.logging package to log events within the game. Since the game uses the console for user interaction, all log events are saved in the farmio.log file.

- All log events are saved in the farmio.loF. Log messages do not shown on console.
- Log level to be saved are defined using Farmio#setupLogger().
- Logger can be obtained from any class using Logger.getLogger() with Logger.GLOBAL_LOGGER_NAME.

The following are the log levels used in this project.

- SEVERE : Intolerable error detected, program must terminate.
- WARNING : Tolerable error detected, program can recover without terminatinF.
- INFO: Information of the events that occured within the program runtime.

Appendix A: Product Scope

Target User Profile:

- age between 11 to 14
- prefers typing over other input methods
- has a need to learn computational thinking
- comfortable with CLI
- prefers visual aids for learning

Unique Value Proposition:

Farmio will look to facilitate teaching students who are struggling to understand programming by allowing them to visualise code in a much more intuitive manner

Appendix B: User Stories

Priority Level:

- `* * *` High (must have)
- `* *` Medium (nice to have)
- `*` Low (unlikely to have)

Priority	**As a ...**	**I want to …**	**So that ...**
`* * *`	Student	Run a visually appealing game	I will not get bored of the game
`* * *`	Game Player	Save my game session	I can continue at the point I left off when I'm free
`* * *`	Student	Learn the logic of if-statements	I will have a strong foundation in programminF.
`* *`	Student	Learn the logic of if-else statements	
`* *`	Student	Learn the logic of while-loops	
`* *`	Student	Learn the logic of for-loops	
`* *`	Game Player	Relate to the code to a story	I will be able to understand how the code runs.
`* * *`	Student	Get feedback on whether my code fulfilled the game objectives	I will know whether I have learnt all aspects the game has to teach
`* * *`	Student	Visualise my code	I will understand why it works or does not work.
`* *`	Game Player	Have a name for my farmer	I can be more engaged to the game
`* *`	Game Player	Have an overview of what I can do	I can have a bigger picture of the commands

* *	Game Player	Skip the tutorial for stages I am confident of solving	Move onto harder stages to learn more concepts
* *	Game Player	Keep a log on what I have done	Keep track of the commands that I have used

Appendix C: Use Cases

(For all use cases below, the System is `farmio` and the Actor is the `user`, unless specified otherwise)

Use case: Initial Game Launch

MSS

1. User starts the farmio.
2. Farmio display game menu.
3. User type “New Game” and press ENTER.
4. Farmio introduce user to background story.
5. User press ENTER to proceed.
6. Farmio display instructions to advance in the game.
7. User follow the instructions by typing into the CLI prompt.
8. Farmio advance to the next level and show the level requirements.
9. User enter commands that fulfills level requirements and “start” command to start the simulation.
10. Farmio executed commands in insertion order and checks for objective fulfillment then prints level advancement message.
11. User enter “Next Level”.

    Steps 6 - 11 are repeated for each level, and continues to repeat until Farmio runs out of levels or until user enters “Quit Game”.
12. Game show game over.
13. User agrees to exit program.

    Use case ends.

Extensions

2a. Farmio detect existing game save data.

2a1. Farmio display “Load Game Save” option.

Use case resumes from step 3.

3a. User enters “Load Game”.

3a1. Farmio show all game save.

3a2. User choose and types game saves index to be loaded.

3a3. Farmio load specified game save.

User case resume from step 9.

3b. User enters “Quit Game”.

Use case ends.

10a. Farmio detect objective(s) not met

10a1. Farmio show results and prompt user to try again.

User case resume from step 9.

*a. At any time, the user enters an invalid command.

*a1. Farmio display command invalid message.

Use case resumes from parent step.

*b. At any time, the user enters “Load Game”

Use case resumes from step 3a.

*c. At any time, the user enters “Quit Game”

Use case ends.

*d. At any time, the user enters “Save Game”

*d1. Farmio save game with a specified game save name if available into the game save file and show game saved message.

Use case resumes from parent step.

Appendix D: Non Functional Requirements

- Should work on supported OS operating system.
- Should be usable by young users with no prior experience in programming
- Should not include offensive language and images
- Should allow developers to easily make modifications in the future
- Should not crash during application runtime

Appendix E: Gloxarry

Supported OS

Windows, Mac OS X, Linux, Solaris. For more information click here.

Appendix F: Instructions for Manual Testing

Given below are instructions to test the features manually.

These instructions only provide a starting point, testers are expected to do more exploratory testing

F.1. Save and Load Game

F.1.1. Save Game

1. Saving a game while the farmer is workinF.
    i. Prerequisites: Start farmer day using `start` command. Only enter command when user is prompted for input.
    ii. Test case: `save game`
    Expected: Game will save successfully, but the assets, like gold and location, will not be updated. This is intentional, to prevent users from cheating using the save and load game feature.

F.1.2. Load Game

1. Handle corrupted save file.
    i. Prerequisites: Generate save file while in game using `save game`.
    ii. Test case: Delete the first character in save.json.
    Expected: Game save corruption detected. Game data will not be loaded and users can continue with the game if the user is in a game session. Otherwise, the user loads the game from the start menu then a new game session will automatically be selected.
    iii. Test case: Delete half of save.json data.
    Expected: Same as previous test case.
    iv. Test case: Delete all data in save.json.
    Expected: Same as previous test case.
    v. Test case: Delete anything from save.json.
    Expected: Same as previous test case. Unless the data deleted is valid like task from task list.
    vi. Test case: Change gold to 9.
    Expected: Gold is now 9.
    vii. Test case: Change gold to 9×10^{10}. $9 \times 10^{10} = 90000000000$.

Expected: Same expected result at test case ii. The game does not require players to obtain so much gold.

viii. Test case: Change gold to 9×10^{20}.
Expected: Same as previous test case.

ix. Test case: Change gold to -1.
Expected: Same as previous test case.

x. Test case: Change gold to abc.
Expected: Same as previous test case.

xi. Test case: Change values in fields other than gold.
Expected: Same expected result as test case ii. Unless the data deleted is valid like task from task list.

F.1.3. Auto Save Game

1. Close program unexpectedly.
    i. Prerequisites: Generate save data file while in game using save game.
    ii. Test case 1: Add 1 task, quit game, re-run farmio and load game.
    Expected: Added task will still exist.
    iii. Test case 2: Add 1 task, enter start, ctrl-c, re-run farmio and load game.
    Expected: Same as previous test case.
    iv. Test case 3: Add 1 task, ctrl-c, re-run farmio and load game.
    Expected: Task cannot be recovered since autosave does not trigger for every task manipulation.

F.2. Add Name Feature

1. Adding a name
    a. Prerequisites: You are able to start a new game.
    b. Test case: `test`
    Expected: The farmer name would be set as ‘TEST’ throughout the game.
    c. Test case: `test_user`
    Expected: The farmer name would be set as ‘TEST_USER’ throughout the game.
    d. Test case: press `Enter`
    Expected: Error details shown as input cannot be empty. User will be given another attempt to enter a name
    e. Test case: `testtesttesttesttesttesttest`
    Expected: Error details shown as name cannot exceed 15 characters. User will be given another attempt to enter a name.
    f. Test case: `Test@123`
    Expected: Error details shown as name cannot contain special characters with the exception of underscores. User will be given another attempt to enter a name.
    g. Test case: `menu`
    Expected: Error details shown as name cannot contain some keywords. User will be given another attempt to enter a name.

F.3. Task manipulation features

F.3.1. Creating a Task

1. Creating a new task
    a. Prerequisites: You are in the interactive segment of the game, where users are allowed to create tasks
    b. Test Case: `do goToMarket`
    Expected: A new task with the description “do goToMarket” is shown in the task list
    c. Test Case: `if gold greater than or equals 10 do buyseeds`
    Expected: A new task with the description “if gold >= 10 do buySeeds” is shown in the task list
    d. Test Case: `do nothing`
    Expected: No new task is created, and error details will be shown on screen
    e. Other incorrect commands to try: `do go to market`, `do`, `if do plantSeeds`
2. Creating a new task when the task list is full
    a. Prerequisite: You are in the interactive segment of the game, and your task list has been populated with 18 tasks already
    b. Test Case: `do goToMarket`
    Expected: No task will be added to the task list, and error details will be shown on the screen.

F.3.2. Inserting a Task

1. Inserting a task in a populated task list
    a. Prerequisites: You are in the interactive segment of the game, you have populated your task list with 3 tasks
    b. Test Case: `insert 1 do gotoMarket`
    Expected: A new task with description “do goToMarket” is now at the top of the task list, and all other tasks are pushed down by one. Your task list should have 4 tasks now.
    c. Test Case: `insert 5 do gotoWheatFarm`
    Expected: A new task with the description “do goToWheatFarm” appears at the end of the task list.
    d. Test Case: `insert 9 do buySeeds`
    Expected: No new tasks are added, and error details are shown on screen
    e. Other incorrect commands to try: `insert 0 do buySeeds`, `insert 1`

2. Inserting a task into a full task list
    a. Prerequisite: You are in the interactive segment, and your task list has been populated with 18 tasks
    b. Test case: `insert 1 do gotoMarket`
    Expected: No new task is inserted into the task list, and error details are shown on the screen.

F.3.3. Editing a Task

1. Editing a task in a populated task list
    a. Prerequisites: You are in the interactive segment of the game, you have populated your task list with 3 tasks
    b. Test Case: `edit 1 do gotoMarket`
    Expected: The first task on your task list should have been replaced by a new task with description “do goToMarket”
    c. Test Case: `edit 3 do gotoWheatFarm`
    Expected: The task with index 3 should have been replaced by a task with description “do goToWheatFarm”
    d. Test Case: `edit 9 do buySeeds`
    Expected: No changes are made, and error details are shown on screen
    e. Other incorrect commands to try: `edit 0 do buySeeds`, `edit 1`

F.3.4. Deleting Tasks

1. Deleting a single task in a populated task list
    a. Prerequisites: You are in the interactive segment of the game, you have populated your task list with 3 tasks
    b. Test Case: `delete 1`
    Expected: The first task on your task list should have been removed
    c. Test Case: `delete 5`
    Expected: No change in your task list, and an error details are shown on screen
    d. Other incorrect commands to try: `delete 0`, `delete -1`
2. Deleting all tasks in a populated task list
    a. Prerequisites: You are in the interactive segment of the game, you have populated your task list with at least one task
    b. Test Case: `delete all`
    Expected: All tasks in your task list are removed

F.4. Running User Created Tasks Feature

F.4.1. Checking for change in asset

1. Assets will be changed when using some of the action commands.
    a. Prerequisites: You are in the interactive segment of the game and you have created at least one task.
    b. Test Case: add the following task with the following commands in the same sequence; `do gotoMarket`, `do buySeeds`, `do gotoWheatFarm`, `do plantSeeds` then enter `start`.
    Expected: At the end of the execution of `do buySeeds`, it can be seen that under the assets of the console, ‘Gold: 10->0’ and ‘Seeds: 0->1’ is printed.
    At the end of the execution of `do plantSeeds`, it can be seen that under the assets of the console, ‘Seeds: 1->0’ and ‘Seedlings: 0->1’ is printed.

F.4.2. Checking for change in location

1. Location will be changed when using some of the action commands.
    a. Prerequisites: You are in the interactive segment of the game and you have created at least one task.
    b. Test Case: add the following task with the following commands in the sequence do gotoMarket, start.
    Expected: At the end of the execution of do gotoMarket, there can be two outcomes. First, if the farmer is already at the Market, ‘You are already at the Market’ will be printed. If the farmer is at the WheatFarm, there will be a simulation of the farmer travelling to the market and the location box on the game console will change from WheatFarm to Market.

F.5. Task Visualisation Feature

Colours are only available for Unix-based terminal, such as Linux and Mac. In addition, to test this feature, ensure that `fast mode` has not been entered at anytime during the game. Also, ensure that the prerequisites are met before trying each test case.

F.5.1. Highlight current task

1. Current task would be marked with '<<' on the right and highlighted in green
    a. Prerequisites: You are in the interactive segment of the game, you have no tasks in your task list.
    b. Test Case: add 18 tasks with `do gotoMarket` and enter `start`.
    Expected: symbol '<<' will iterate through all 18 tasks.
    c. Test Case: add 3 tasks with `do gotoWheatFarm` and enter `start`.
    Expected: symbol '<<' will iterate through only 3 tasks.
    d. Test Case: Ensure you are at the market. Add 3 tasks with `if hasGrain do sellGrain` and enter `start`.
    Expected: symbol '<<' will still iterate through these 3 tasks (no tasks gets skipped). The tasks still execute although some of the actions may not.

F.5.2. Highlight change in game assets

1. A change in game assets will be emphasized with '->' and will be highlighted in magenta.
    a. Prerequisites: You are in the interactive segment of the game, you have no tasks in your task list. Your current location is the WheatFarm and you have 10 gold and no other assets. (level 1.1).
    b. Test Case: add the following task with the following commands in the same sequence; `do gotoMarket`, `do buySeeds`, `do gotoWheatFarm`, `do plantSeeds` then enter `start`.
    Expected: At the end of the execution of `do buySeeds`, it can be seen that under the assets of the console, 'Gold: 10->0' and 'Seeds: 0->1' is printed.
    At the end of the execution of `do plantSeeds`, it can be seen that under the assets of the console, 'Seeds: 1->0' and 'Seedlings: 0->1' is printed.

F.5.3. Highlight completed goal

1. If a goal in the Goal section of the console is reached, it turns green.

a. Prerequisites: Load the game with the following save files described in the test case where the load game is successful.
b. Test Case: Level 1.1, Location: Market.
   Expected: ‘Location:Market’ under the Goal section of the console is coloured in green.
c. Test Case: Level 1.1, Location: WheatFarm.
   Expected: ‘Location:Market’ under the Goal section of the console is uncoloured.

F.5.4. Visual representation of the action will occur when it is executed

1. If an action is executed with no error, an animation will play in the center of the console.
    a. Prerequisites: Enter the following task add commands in the interactive segment of the game. Your current location is the WheatFarm and you have 10 gold and no other assets. (level 1.1)
    b. Test Case: `do gotomarket` and enter `start`
       Expected: Visual representation of going to the market can be seen.
    c. Test Case: `if hasseeds do plantseeds` and enter `start`
       Expected: Visual representation of planting seeds will not happen. Instead, a message saying ‘Condition not fulfilled, not executing task!’ will be shown.

F.6. Task Error Detection Feature

F.6.1. Triggering an Error in task execution

1. Single action criteria not met.
    a. Prerequisites: You are in the interactive segment of the game, you have no tasks in your task list. Your current location is the WheatFarm and you have 10 gold and no other assets. (level 1.1)
    b. Test Case: add task with `do plantSeeds` and enter `start`
    Expected: after the day start animation, only the message “Error! you have attempted to plant seeds despite not having any seeds” is shown.
    c. Test Case: add task with `do buySeeds` and enter `start`
    Expected: after the day start animation, only the message “Error! you have attempted to buy seeds despite not being at the market” is shown.

2. More than one action criteria not met.
    a. Prerequisites: You are in the interactive segment of the game, you have no tasks in your task list. Your current location is the WheatFarm and you have 10 gold and no other assets. (level 1.1)
    b. Test Case: add task with do sellGrain and enter start
    Expected: after the day start animation, 2 messages are shown; “Error! you have attempted to sell grain despite not having any grain” and “Error! you have attempted to sell grain despite not being at the market”.

F.7. Level Progression and Reset Feature

F.7.1 Level Progression

1. Passing the Level
    a. Prerequisites: You are in the interactive segment of the game, Your current location is the WheatFarm and you have 10 gold and no other assets. (level 1.1)
    b. Test Case: enter do gotoMarket followed by Enter
    c. Expected: Task will be executed and level will be Passed and will move onto the next Level

F.7.2. Reset Level

1. Failing the Level
    a. Prerequisites: You are in the interactive segment of the game, Your current location is the WheatFarm and you have 10 gold and no other assets. (level 1.2)
    b. Test Case: enter do gotoMarket, do gotoMarket then start
    c. Expected: You will fail the level and the level reset feature will reset the level
2. Individually resetting the level
    a. Prerequisites: You are in the interactive segment of the game, Your current location is the WheatFarm and you have 1 seedling and no other assets(level 1.4)
    b. Test Case: enter do gotoMarket and start, after the day ends, enter reset
    c. Expected: You will carry out the tasks and end the day. You will then be prompted with the option to reset, the level feature reset will then reset the level.

F.8. Feedback Feature

F.8.1. Feedback

1. Successful Feedback
    a. Prerequisites: You are in the interactive segment of the game, Your current location is the WheatFarm and you have 10 gold and no other assets. (level 1.1)
    b. Test Case: enter do gotoMarket, then start
    c. Expected: You will pass the level and a success feedback will be provided
2. Unsuccessful Feedback
    a. Prerequisites: You are in the interactive segment of the game, Your current location is the WheatFarm and you have 10 gold and no other assets. (level 1.2)
    b. Test Case: enter do gotoMarket, do gotoMarket then start
    c. Expected: You will fail the level and unsuccessful feedback will be provided.

F.9. In-Game Menu and ShowListFeature

F.9.1. In-Game Menu

1. Using Commands from the in-game menu section.
    a. Prerequisites: You are in the interactive segment of the game.
    b. Test case: `menu`
    Expected: In-game menu frame will be shown.
    c. Test case: enter `menu` followed by adding a task do `gotomarket`
    Expected: a task will be added and user will move back to the game stage.
    d. Test case: enter `menu` followed by `Enter`
    Expected: Game will resume to narrative mode of the level.

F.9.2. ShowList

1. Using Commands from the showList Section.
    a. Prerequisites: You are in the interactive segment of the game.
    b. Test case: enter `action`
    Expected: ActionList frame will be simulated
    c. Test case: enter `action` followed by `do gotomarket`
    Expected: ActionList frame will be simulated followed by the task being added and the user will be in the game mode.
    d. Test case: enter `condition`
    Expected: ConditionList frame will be simulated.
    e. Test case: enter `condition` followed by `do gotomarket`
    Expected: ActionList frame will be simulated followed by the task being added and the user will be in the game mode.
    f. Test case: `market`
    Expected: MarketList frame will be simulated
    g. Test case: enter `market` followed by `do gotomarket`
    Expected: MarketList frame will be simulated followed by the task being added and the user will be in the game mode.
    h. Test case: enter `task command`
    Expected: Task Command List frame will be simulated.
    i. Test case: enter `task command` followed by `do gotomarket`
    Expected: Task Command List frame will be simulated followed by the task being added and the user will be in the game mode.

F.10. Logging Feature

F.10.1 Logging Tasks

1. Viewing a populated log file
2. Prerequisites: You are in the interactive segment of the game and you have already carried out tasks execution and have populated the log list with 15 task logs.
3. Test Case: log 1
4. Expected: Displays page 1 of the log
5. Test Case: log 2
6. Expected: throws an error since the log is not populated
7. Incorrect commands to try: `log -1, log 0`