Project Portfolio Page

About the project

The purpose of this project is to create a simulation game that teaches children, age between 11 and 14, basic computing methodology. With this game, teachers can get their students interested in computing by allowing their students to play the game and learn computational thinking skill at the same time. With each level, students playing the game can learn basic programming concepts such as “if-else”, “for” and “while” loops a progressive manner. Students will need to put their knowledge to the test in order to advance to the next level in the game. The image below shows the design of the game.

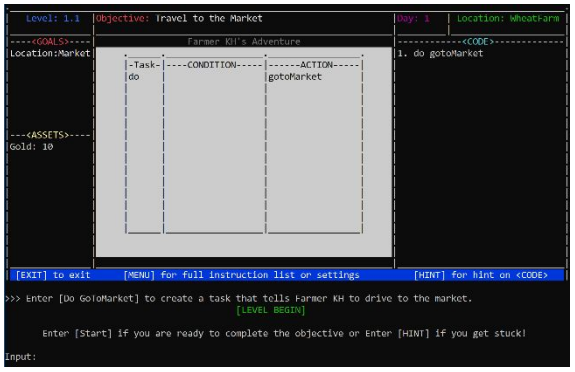

Figure 1. Farmio User Interface

The role I was tasked with in this project was to design and develop the storage feature in the project. Detailed description of my contributions to the project will be in the next few sections.

This document will use the following symbols and formatting standards:

	This symbol indicates additional information from the section above.
`save game`	A grey highlight indicates that this is a command that can be typed into the command line and executed by the game.
`Storage`	A grey highlight with blue text indicates a component, class or object in the application.

Summary of contributions

This section list and describe some of my contribution in this project.

Enhancement added: I added the feature to save and load game.

- What it does:

  The command save game enable users to save current session of the game to be continued in a later time. To resume the previous game session, the user can use the command load game to continue from the last saved game session.

- Justification:

  With the load and save game feature, users can manage their time better without having to complete the game in a single session.

- Highlights:

  The feature is compatible with existing features and features to be added in the future. Experimenting alternate storage methods was essential to achieve a balance of user readability, data formatting standards and scalability for future development. The decision on the use of a data format was challenging as all formatting standards have their merits. The data formats in consideration were Extensible Markup Language (XML), JavaScript Object Notation (JSON) and custom data format.

Code contributed: [Functional Code] [Test Code]

Other contributions:

- Project management:
  - Allocate tasks to be done for the milestone.
- Implementation of features:
  - Fetch level data from storage to allow level information to be store in resources folder providing a clear separation between data and code. (Pull request #91)
  - Reset level function to allow level reset in case the level cannot be completed or loaded. (Pull request #192)
  - Load frame function to fetch and format frames data in data files to be used for user interface animation. (Pull request #117)
  - Logging feature to write activity logs of the program for developers debugging purpose. (Pull request #270)
- Enhancements to existing features:
  - Updated program skeleton to improve program readability and logic flow. (Pull request #85)
  - Update user interface frames to enhance visual aid experience for the user. (Pull request #212)
- Document
  - Added the following sections in User Guide (UG): Getting Started, Starting a New Game, Loading a Saved Game and Saving Your Game.
  - Added the following sections in Developers Guide (DG): Setting Up, Storage Component, Save/Load Game Feature, Logging.

- Community
    - Reviewed pull request: #43, #65, #110, #117
- Tools:
    - Integrated a third party library (JSON.simple) to the project (#110)

Contributions to the User Guide

The User Guide requires an inclusion of additional instructions for features that were added to the project. The following are excerpts of our User Guide to show the changes I made in the User Guide for save game and load game commands.

Save Game

Format: `save game`

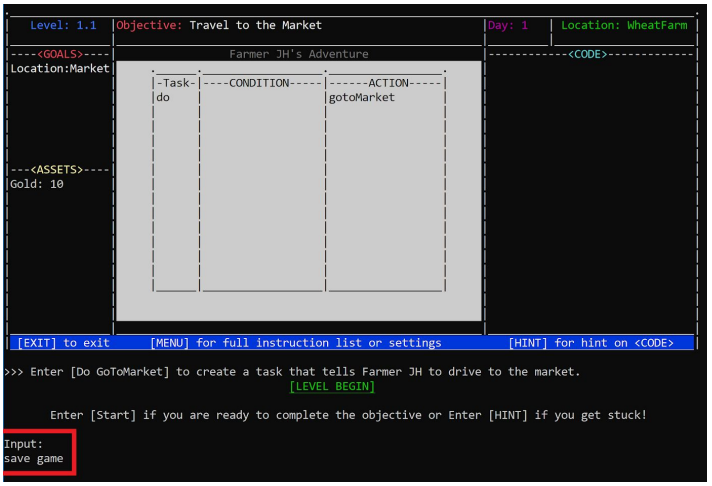

Figure 2. Save Game Command

Upon successful game save, you will see a success message and the save file location on the screen.

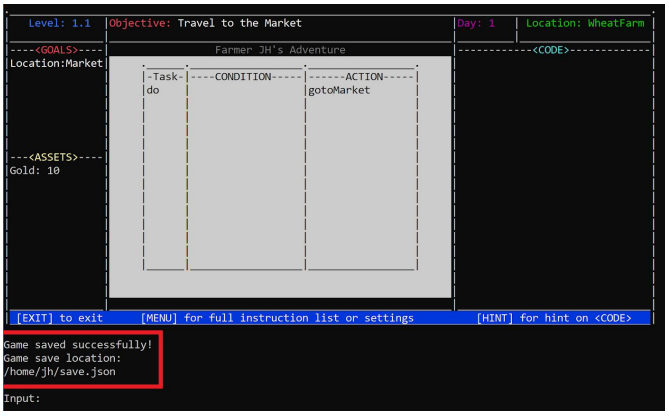

Figure 3. Save Game Message

You can only use the save game command in the interactive segment

If you are in the story segment please use the skip command to enter the interactive segment to save your game.

If you are unable to save your game, you may be executing the game in a permission restrictive path. Please ensure that you are in a non-permission restrictive path before running the game.

Loading a Saved Game

Step 1: Open terminal console, and type `java -jar` `followed` by the location of the game file that ends with ‘jar’ file extension.

```
java -jar your/file/path/farmio.jar
```

Figure 4. Launch Farmio Command

Step 2: Upon the successful game launch, you will see a welcome message.

Figure 5. Welcome Screen

Step 3: Press `Enter` to proceed.

Step 4: Next you will see the game menu.

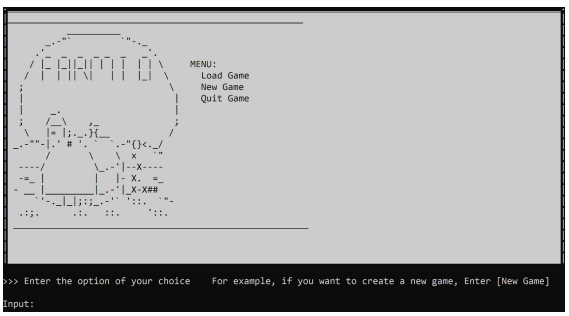

Figure 6. Start Game Menu

Step 5: Type `load game`, and press `Enter` to start a previously saved game session.

> `load game` command will only function properly if the game save file is in the same folder as the folder the game is running from and if the save file is not corrupted.
>
> You can enter the `load game` command in the welcome screen as well.

To view the full UG documentation click here.

Contributions to the Developer Guide

The next section will contain excerpts from our DG that shows my additions for save and load game features in the document.

Save Game Feature

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`.

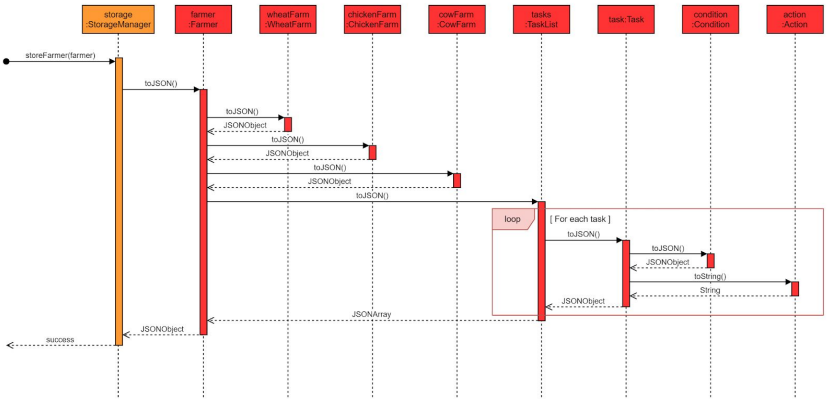

Figure 7. Generate JSON Behaviour

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()`.

Load Game Feature

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.

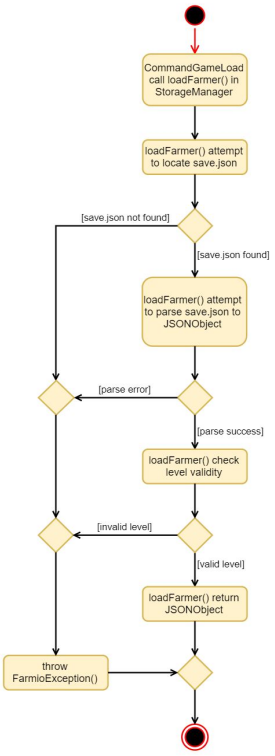

Figure 8. Fetch save.json Process

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 7.

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.

To view the full DG documentation click here.

Design Considerations

There are decisions I made on the design of load and save game function which I think is best suited for the project. The following is an overview of my analysis and justification for the decision made.

Aspect	Alternative 1 (current choice)	Alternative 2
Execution of load and save game operation.	Split conversion of `Farmer` object responsibility to respective data objects. Pros: Introduces better cohesion in the code. Each object is responsible for the data contained in it. Cons: Every non primitive class need to implement 2 extra methods to convert data both ways.	Localise all conversion operation in to `Storage` object. Pros: Easier to introduce partial conversion since all data is handled inside `Storage`. Cons: Poorer code cohesion since variables may be difficult to access in Storage.
File data structure to be supported by load and save game features.	Use JSON formatting standard for game session data. Pros: JSON is a well known standard, as it has online support to edit JSON using basic text editor. Cons: JSON has limited variable type support to store data directly into JSON format.	Use object serialization to store game session Pros: Can be implemented easily and can efficiently store game session. Cons: Saved data is not easily editable by the user.