System design of Sheepdog Trials

The MVC model is adopted to organize the program functions, control the logic and data flow. It is the main skeleton of the program and the classes under MVC model are put at the same “game” packages. Controller Class mediates between the model and view class, as well as access variables and methods under other classes.

The second package “Objects” is created to store all components required for the game, including the dog, sheep, bush and the field etc. The components are created as child classes that depend on an abstract class “FieldObject”.

The third package “util” includes the FieldSettings Class that stores the game settings parameter, and the InputUtil Class used for accepting user input.

Details of all classes:

Package “game”:

1. Main Class is executed to create a controller object.
2. SheepEnvironment Class (Model under MVC) has all the data that the program needs, including the field, the dogs and sheep etc. It creates and stores the sheep, dogs and other components from the “Objects” packages, based on the game settings from the FieldSetting Class. It controls and updates the data, for example, moving the sheep to the appropriate position on the field. Its methods are initiated by the Controller Class.
3. TextView Class (View under MVC) defines the text display, for example, the welcome message and the field. It received the data from SheepEnvironment Class to generate the display of the field. Its methods are initiated by the Controller Class.
4. GuiFrame and MainPanel Class (View under MVC) are used to create the GUI view of the game. They are implemented by the Controller Class. They receive the data from SheepEnvironment Class to generate a GUI display of the field. Another Class, FieldInputGUI, receives the input from the user on the GuiFrame and redirect the data to the Controller Class
5. Controller Class contains the game logic, which controls the game session to start, proceed and end. It also controls the data stores in SheepEnvironment Class, including initiating the movement of sheep and the dog, and calls the methods at TextView and GuiFrame Class to create a display for the user. It implements the game setting stored at FieldSettings Class and receives the user input from InputUtil Class, which are then executed and stored at the “SheepEnvironment” Class.

Package “Object”:

All classes under this package are implemented and stored by SheepEnvironment Class.

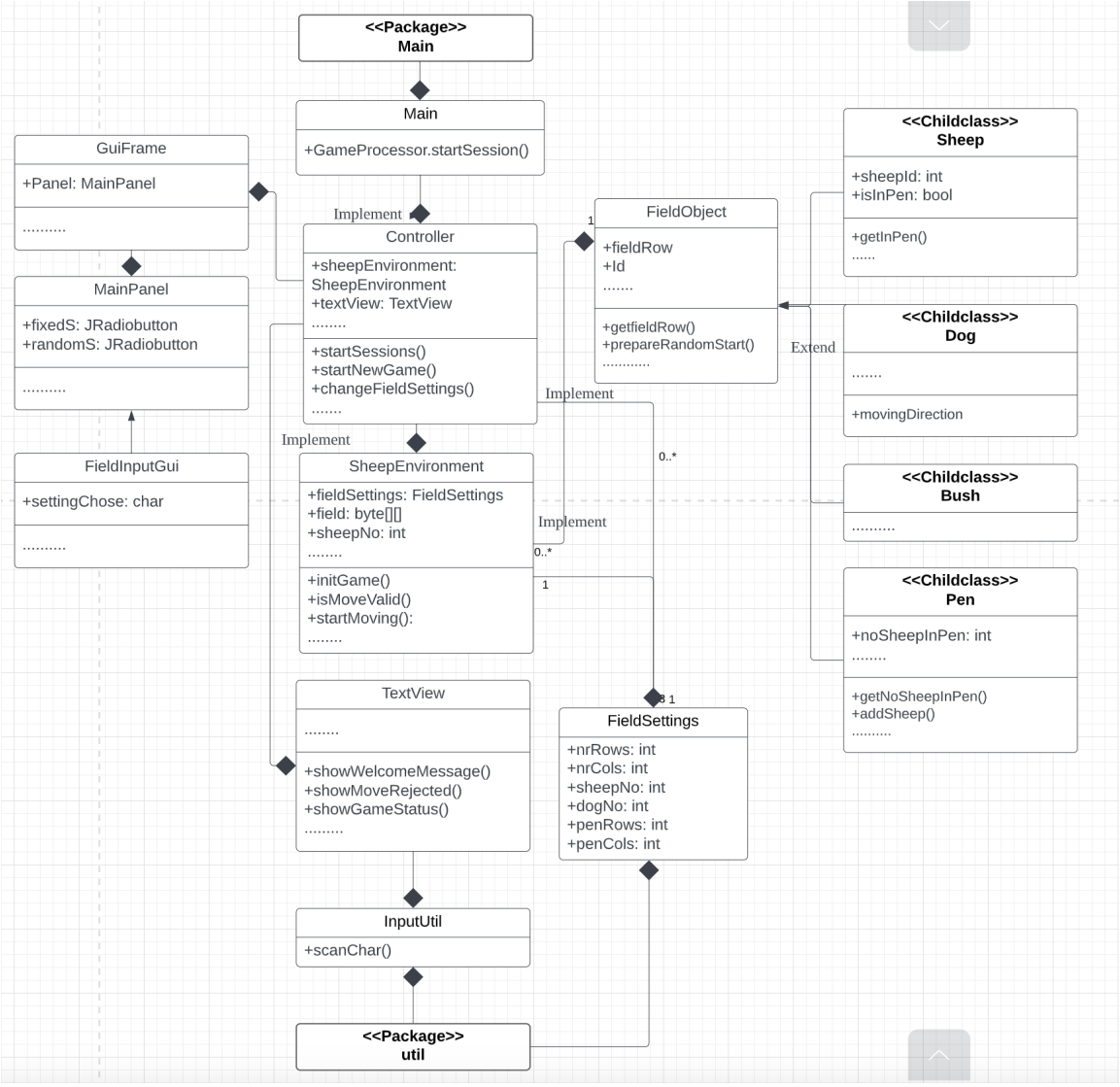
6. FieldObject Class is the abstract class defining the variables and methods that all game components should inherit, e.g. stores its own symbol shown on the field, prepare for start at a fixed or random setting, and store its own position on the field.
7. Dog Class is the child class extending from the FieldObject Class. It represents the dog moving on the field. It contains a specific method to receive the input from the player.

8. Sheep Class is the child class extending from the FieldObject Class. It represents the sheep moving on the field. It contains its specific variable, e.g. “isInPen” variable indicates if the sheep has already moved into the Pen.
9. Bush Class is the child class extending from the FieldObject Class. It represents the bush on the field.
10. Pen Class is the child class extending from the FieldObject Class. It represents the pen on the field. It contains its specific variable, e.g. “noSheepInPen” variable indicates the number of sheep in the pen.

Package “Util”:

11. FieldSettings Class stores the game settings for the program, including the field and pen size, the number of sheep and bush etc. It is implemented by the Controller Class.
12. InputUtil Class accepts and verifies a valid input from the user. It is implemented by the Controller Class.

Below is a class diagram.



Reasons for the design:

Reasons for using MVC:

1. It provides loose coupling that allows easy planning and testing. It divides the program into three main parts that gives me an overview to organize the game flow as actual codes. And the program is well structured that it is easier to debug from separated parts of the program, instead of the entire program as a whole.
2. It can provide multiple views, including the text view and GUI view, from the same set of data and game logic. Because it separates the display from the model and provides limited code duplication.
3. It allows easier enhancement for a particular part of the program, which changes do not affect the entire program. Because it provides sufficient decoupling that any changes of one part, such as adding new components under “Object” package, e.g. chicken on the field, do not affect the game logic and the existing data.

Reasons for using bridge, the structural design pattern:

1. Bridge allows us to split the entire game from a large class into a set of classes into 2 hierarchies, which enhance decoupling so that we can develop them independently.
2. First example, the game is divided into 2 dimensions, including abstraction and implementation. Under abstractions, TextView and GuiFrame Class display the game content, which delegate the work from the Controller class. The Controller class is the implementation that governs the underlying game logic. These two parts are independent, the game logic at the controller class can be shown at different displays, whereas the display can be used for a different controller with another game logic.
3. Second example, the Sheep Environment class (Model of MVC) is separated into 2 hierarchies, 1) the number of components on the field is defined by the FieldSetting class, 2) the behavior of the component is determined by the FieldObject class. The parts can be created separately.

Reasons for using inheritance hierarchy for game component, e.g. dog, sheep, instead of interface:

All game components share similar features, including having positions on the field, starting with random position on the field. Thus, either interface or inheritance could be used to provide re-usable codes. They both reduce duplication of the codes, which avoid building two classes with the exact same methods, and allow code simplification at client classes, SheepEnvironment. Inheritance hierarchy is used because:

1. Interface does not contain the implementation details of the methods. But some methods, e.g. getting field position, can be shared among all components. Thus, interface cannot fulfill this purpose.
2. Therefore, inheritance hierarchy with the use of abstract class was used.
3. Abstract class was used, instead of super class, because the abstract class cannot be instantiated and is used for define general characteristics. So I am sure no ambiguous object from the FieldObject class was created on the field.
4. The inheritance hierarchy ensure the classes under Package “Objects" can be treated uniformly so the polymorphic method call treat every object appropriately, e.g.

we can put the objects under the same array list, and client class can iterate all subclasses at once

Adopting a template method, the behavioral design pattern:
The abstract class is also a template method, which is a behavioral design pattern. The FieldObject class defines the logic and behavior in its own superclass but allows the child class to override some of the behavior while not amending the structure of superclass. Template methods, such as getting and setting components' locations on the field, are added. The child class, e.g. sheep, can choose to implement the methods, and override other methods if needed, for example, the methods for setting the sheep in the pen.

Reasons for using singleton for ActionPerformer under FieldInputGui Class:

1. Ensure the instance can only be created within the Class. No other instance will exist.
2. It saves memory as only one object is created, which benefits the future enhancement of the program.
3. Make sure only one return instance for the game setting is created after the GUI button is clicked to avoid confusing data flow within the program.