Bitcode Assignment 2

1 Design Patterns

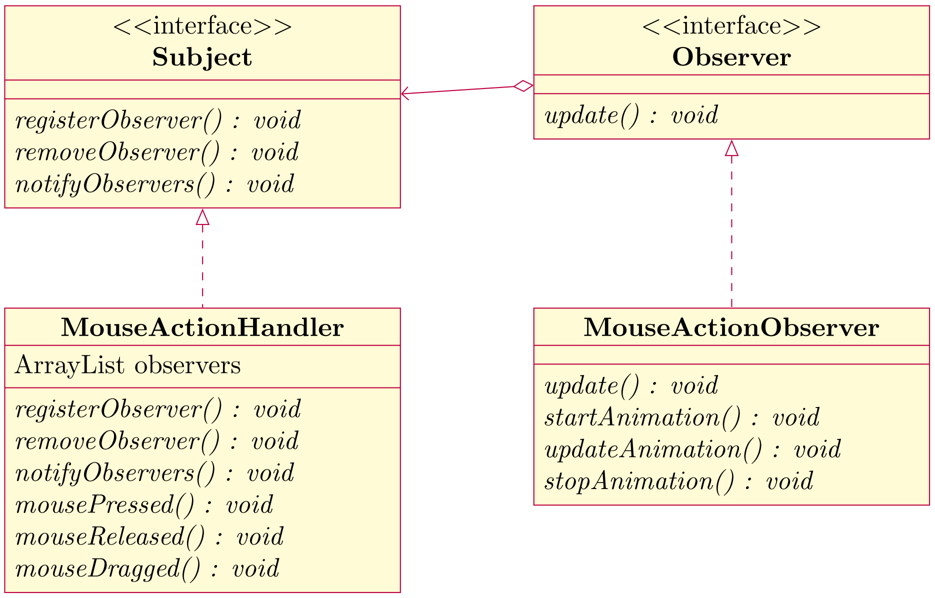
Using design patterns in software project is a good practice. It helps to make your software understandable, sustainable and expendable. We have chosen two design patterns and implemented them in our existing code, the observer pattern and the factory pattern.

1.1 The Observer Design Pattern

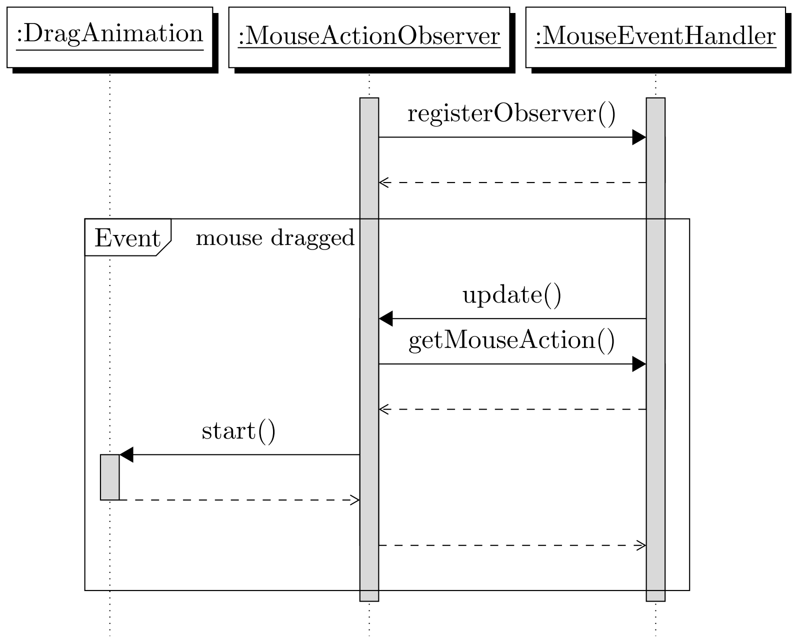
The observer pattern is implemented between the MouseActionHandler class (subject) and the DragAnimation class (observer). The MouseActionHandler listens for mouse input in a given window. A object of the DragAnimation class is created when a mouse drag is registered and updated when the position of the mouse pointer changes.

We have chosen this pattern on this location because it enables us to create more observers for the MouseActionHandler in the future. Therefore it will save time to implement new features that need the MouseActionHandler class.

1.1.1 Observer Class Diagram



1.1.2 Observer Sequence Diagram



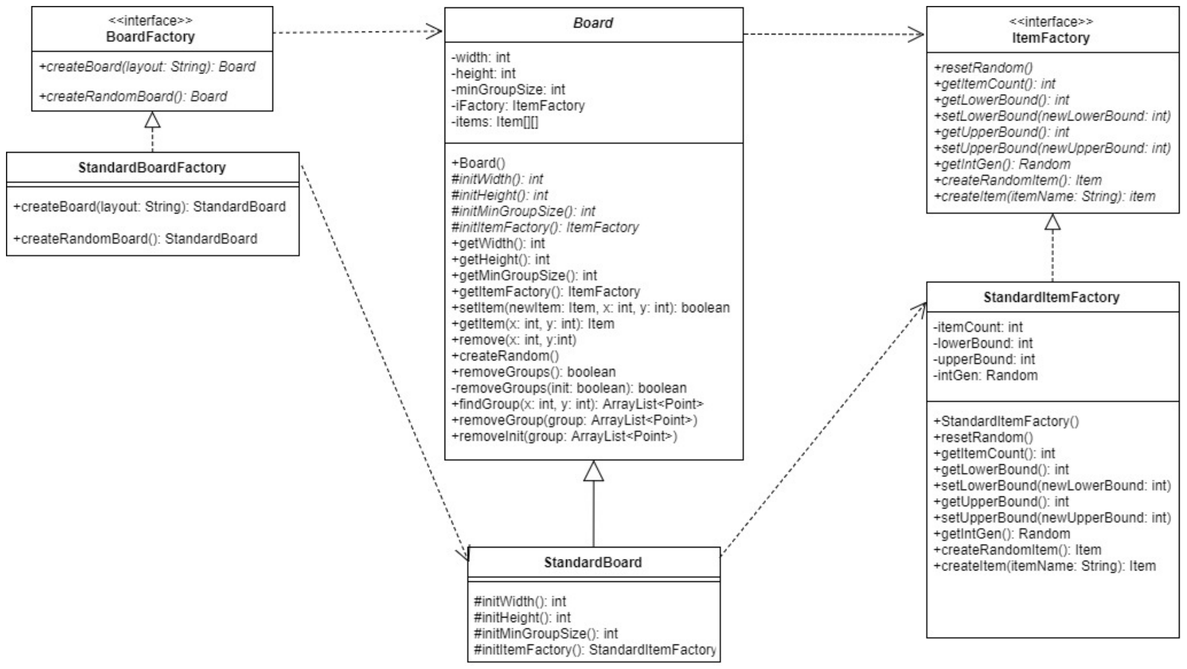
1.2 The Factory Design Pattern

The factory design patterns are implemented for the BoardFactory and the ItemFactory. On launch the board is created via the StandardBoardFactory which will construct a board using the StandardItemFactory.

The reason that we have implemented a factory pattern for the Board class is that we can use this pattern in the future for creating different types of board. For instance creating boards with different sizes based on difficulty.

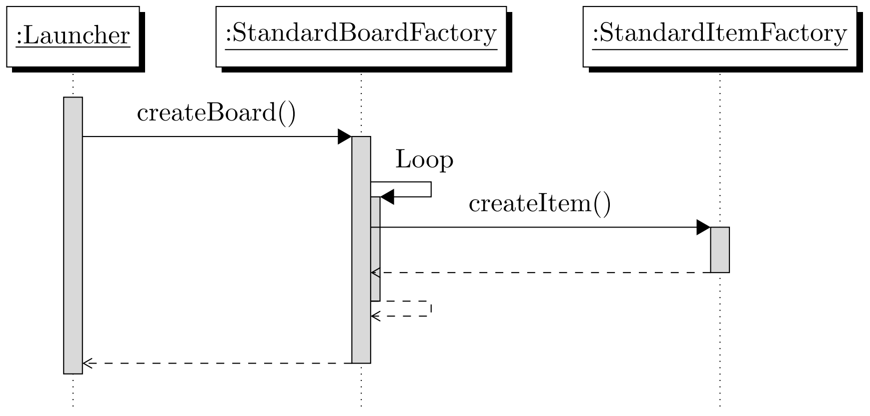
We have implemented the factory pattern for the Item class because that allows us to create different types of items in the board, such as special items that will remove all items from the board.

1.2.1 Factory Class Diagram



<<interface>>
BoardFactory
+createBoard(layout: String): Board
+createRandomBoard(): Board
StandardBoardFactory
+createBoard(layout: String): StandardBoard
+createRandomBoard(): StandardBoard
Board
-width: int
-height: int
-minGroupSize: int
-iFactory: ItemFactory
-items: Item[][]
+Board()
#initWidth(): int
#initHeight(): int
#initMinGroupSize(): int
#initItemFactory(): ItemFactory
+getWidth(): int
+getHeight(): int
+getMinGroupSize(): int
+getItemFactory(): ItemFactory
+setItem(newItem: Item, x: int, y: int): boolean
+getItem(x: int, y: int): Item
+remove(x: int, y:int)
+createRandom()
+removeGroups(): boolean
-removeGroups(init: boolean): boolean
+findGroup(x: int, y: int): ArrayList<Point>
+removeGroup(group: ArrayList<Point>)
+removeInit(group: ArrayList<Point>)
StandardBoard
#initWidth(): int
#initHeight(): int
#initMinGroupSize(): int
#initItemFactory(): StandardItemFactory
<<interface>>
ItemFactory
+resetRandom()
+getItemCount(): int
+getLowerBound(): int
+setLowerBound(newLowerBound: int)
+getUpperBound(): int
+setUpperBound(newUpperBound: int)
+getIntGen(): Random
+createRandomItem(): Item
+createItem(itemName: String): item
StandardItemFactory
-itemCount: int
-lowerBound: int
-upperBound: int
-intGen: Random
+StandardItemFactory()
+resetRandom()
+getItemCount(): int
+getLowerBound(): int
+setLowerBound(newLowerBound: int)
+getUpperBound(): int
+setUpperBound(newUpperBound: int)
+getIntGen(): Random
+createRandomItem(): Item
+createItem(itemName: String): Item

1.2.2 Factory Sequence Diagram



2 Your wish is my command

Our client wanted to have two new features implemented, which consist of a high score page and maximum amount of moves per level. For each each feature we have created new requirements and a software design.

2.1 High Score Page

tbd

2.1.1 Requirements

tbd

2.1.2 Software Design

tbd use UML

2.2 Maximum Moves per Level

tbd

2.2.1 Requirements

tbd

2.2.2 Software Design

tbd use UML

3 Turn-based Multiplayer

In exercise three we have been asked to implement a feature that we wanted to implement. We have chosen to implement turn-based Multiplayer. For this feature we also created new requirements and a software design.

3.1 Requirements

tbd

3.2 Software Design

tbd use UML