Bitcode Assignment 1

1 The Core

1.1 Derive Classes

For deriving the the right classes that we can implement in the software we firstly look for noun phrases in the requirements document[1]. Secondly we try to refine the list of phrases and group them by using given guidelines.

Noun phrases In table 1 a list of noun phrases is presented that was derived from the functional requirements in the requirements document.

Noun Phrases	Requirement
game, board, grid	The game board will consist of a 10x10 square grid
tile	The game will have six different tiles with which the board will be filled.
filled board	The game will start with a filled board.
mouse	A tile must be able to move horizontal or vertical by using the mouse.
row, column	If one tile is moved, the whole row or column will move along with it. The tiles that get past the edge will reappear at the opposite edge.
...	A row or column of 3 or more of the same tile (independent of the white outline), will mean that these tiles get removed from the game.
...	The tiles above empty tiles will move down one position, the remaining empty tiles shall be filled randomly.
player, move	The game will end when the player runs out of possible moves.
...	The player should be able to start a new game.
...	The player should be able to stop a game in progress.
...	The game shall end when the player loses or stops the game, or clears all of the white outlining.
turn, cell	The game will end in a set amount of turns. The amount is based upon the amount of cells which are outlined. (For example 1 outlined cell gives the player five moves).
white outline	Some cells will have a white outline, moving the tile which rests on this cell will not affect the white outline.

[1]https://github.com/mkhattat/bitcode-SEM/blob/master/docs/requirements.pdf

...	The white outlining of a cell will be removed once a tile in that cell is removed.
pattern	The patterning of white tiles should be preprogrammed.
...	The player loses when there are no possible moves left, or if the player has run out of moves.
...	The player wins when all white outlined cells are cleared.
level, difficulty system	The game could have a level or difficulty based system.
scoring system	The game could have a scoring system based on the level or difficulty system.
score	The players score could be shown during the game.

Table 1: list of derived nouns

Refine Candidates We can refine the list of nouns by sorting them based on the groups of obvious, uncertain or nonsense class candidates. We also define the type of candidate classes such that it can be a physical object, conceptual entity, categories of classes an interface or values. In table 2 a list of candidate classes is shown.

Candidate Class	Group	Class Type
Game	obvious	conceptual entity
Board	obvious	interface
Tile	obvious	conceptual entity
Mouse	obvious	physical object
Player	obvious	physical object
Level	obvious	conceptual entity
Grid	uncertain	value
Move	uncertain	conceptual entity
Pattern	uncertain	value
ScoringSystem	uncertain	conceptual entity
Score	uncertain	conceptual entity
FilledBoard	nonsense	conceptual entity
Row, Column	nonsense	conceptual entity
Turn	nonsense	conceptual entity
WhiteOutline	nonsense	conceptual entity

Table 2: list of candidate classes

Class-Responsibility-Collaboration Cards After we refined the list of candidate classes we can create so called "class-responsibility-collaboration Cards" or CRC cards. These cards are used to get an overview of the responsibility of the classes and which classes are collaborating together. In the figure below the CRC cards are presented.

Game	
Supperclass(es): ...	
Subclasses: ...	
Create game window	...
Create board	Board
Create player	Player

Board	
Supperclass(es): ...	
Subclasses: ...	
Read level	Level
Create grid	Grid
Draw board	Tile
Move Tiles	Move

Tile	
Supperclass(es): ...	
Subclasses: ...	
Load Image	...
Draw Tile	...

Move	
Supperclass(es): ...	
Subclasses: ...	
Check for movements	EventHandler
Move animation	Tile, Grid

EventHandler	
Supperclass(es): ...	
Subclasses: ...	
Check for mouse events	MouseEvent Handler
Check for button events	ButtonEvent Handler

Player	
Supperclass(es): ...	
Subclasses: ...	
keep track of score	ScoringSystem

MouseEventHandler	
Supperclass(es): ...	
Subclasses: ...	
Capture and handle mouse events	...

ButtonEventHandler	
Supperclass(es): ...	
Subclasses: ...	
Capture and handle button events	...

ScoringSystem	
Supperclass(es): ...	
Subclasses: ...	
Keep track of scoring	Move

Level	
Supperclass(es): ...	
Subclasses: ...	
Read Level from file	...

Comparison with the implementation If we look at classes that were integrated into the initial implementation of the game[2] we can spot some differences. Namely, there are a couple of classes missing. This is mostly due to the fact that not all requirements where implemented in the initial version. For

[2]https://github.com/mkhattat/bitcode-SEM/releases

example, the Player class and the ScoringSystem class is absence from the code because scoring is not implemented. There is also not a Level class because there exists only one level that is randomly generated. Furthermore, the Game class is replaced by the Launcher class and Move class is replaced by the Animation class.

1.2 Main Classes

tbd

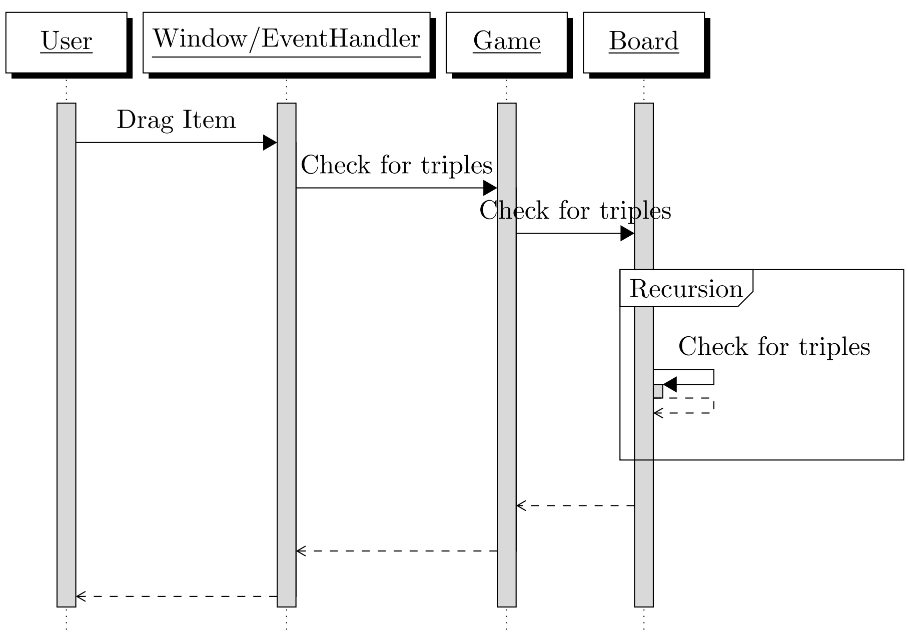
1.3 Reflect on main class decisions

tbd

1.4 The Class diagram

tbd

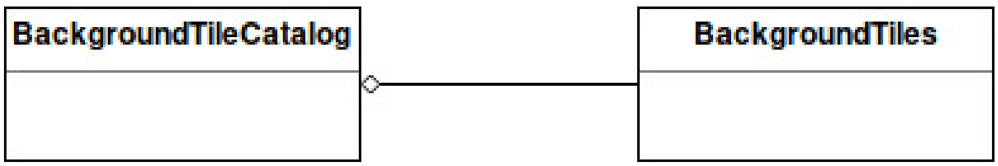
1.5 The Sequence Diagram



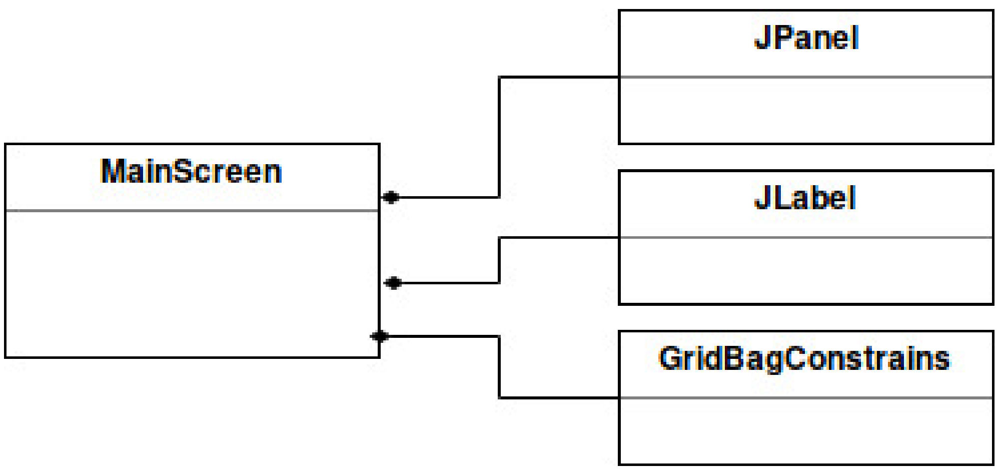
2 UML in Practice

tbd

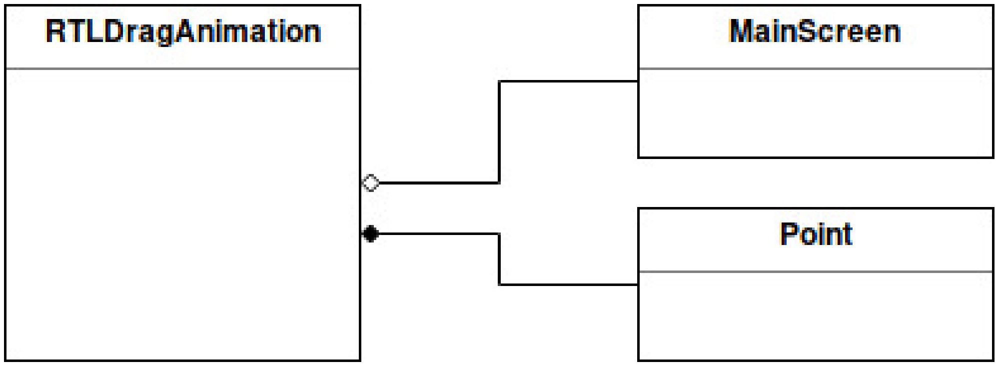
2.1 Composition and Aggregation



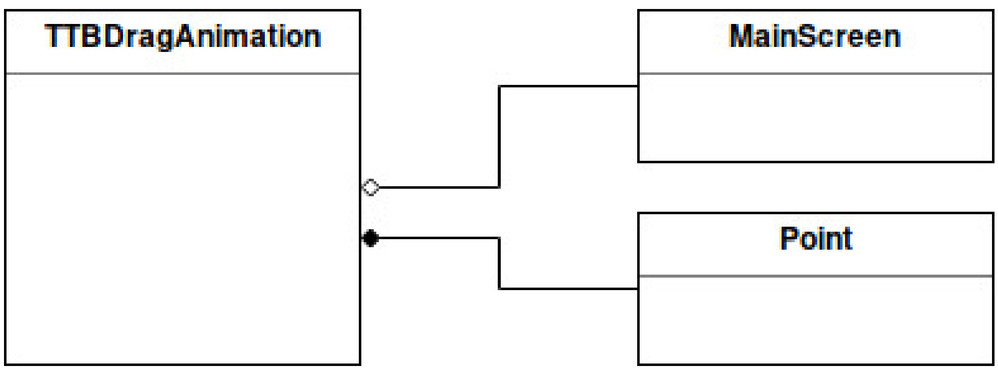
An instance of BackgroundTileCatalog contains a reference to the BackgroundTiles class but they are independent and they have their own lifetime, So BackgroundTileCatalog aggregates BackgroundTiles.



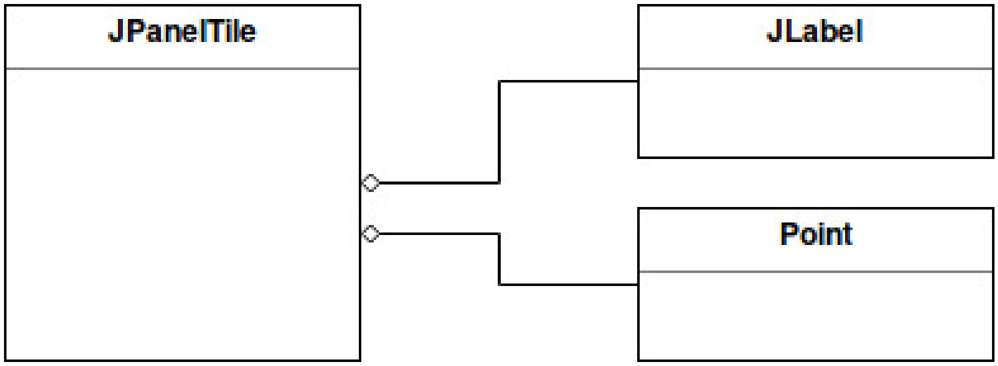
An instance of MainScreen class has a strong relation, or in other words owns, the JPanel, JLabel, GridBagConstrains classes. There is no reason for these classes to exist without MainScreen class. So it's composition.



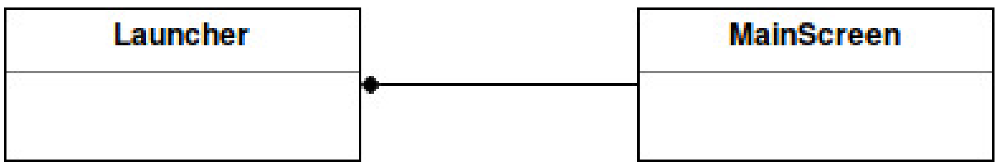
An instance of RTLDragAnimation class has a relationship with MainScreen and it's aggregation because MainScreen can exist without RTLDragAnimation but at the same time this class owns a Point which cannot exist without RTLDragAnimation, so it's composition.



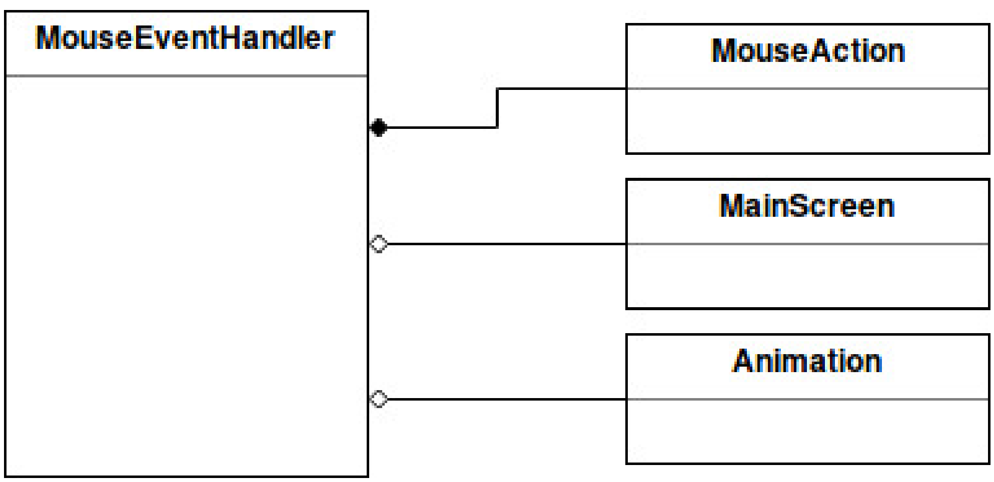
An instance of TTBDragAnimation class has a relationship with MainScreen and it's aggregation because MainScreen can exist without TTBDragAnimation but at the same time this class owns a Point which cannot exist without TTBDragAnimation, so it's composition.



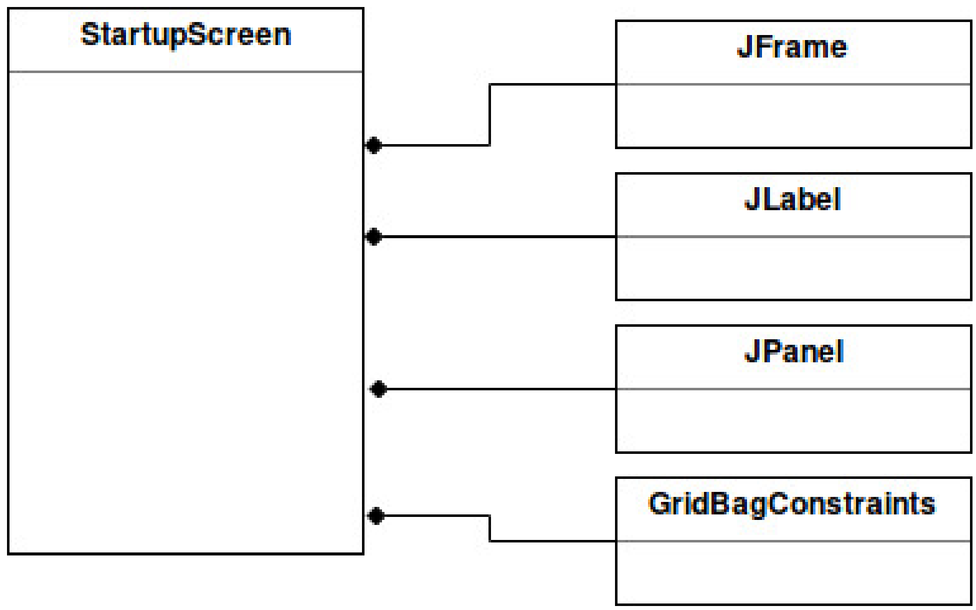
An instance of JPanelTile class has a relationship with JLabel and Point. It's aggregation because these two classes can exist without JPanelTile.



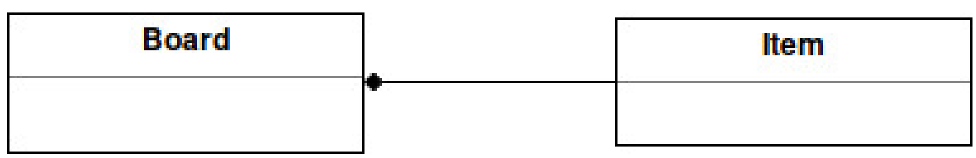
A Launcher class owns a MainScreen class because if the Launcher class is killed the MainScreen class will also be killed, so a composition.



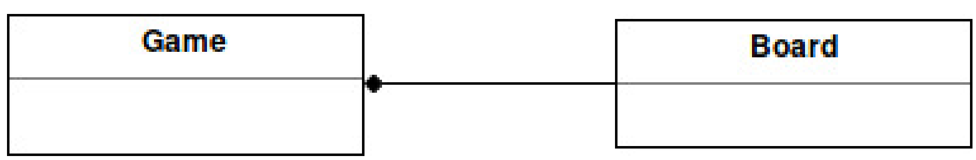
MouseEventHandler aggregates MainScreen and Animation classes. It has a relation with these two classes but they are independent of each other, or in other words an aggregation. But it has a stronger relation with MouseAction, means when MouseEventHandler goes out of scope, then MouseAction also goes out of scope, so in this case a composition.



StartupScreen class owns some classes namely: JFrame, JLabel, JPanel and GridBagConstraints. These classes are dependent on the StartupScreen, means if it dies these classes will also die. So it is a composition.



Board class owns an Item class. This is a composition.

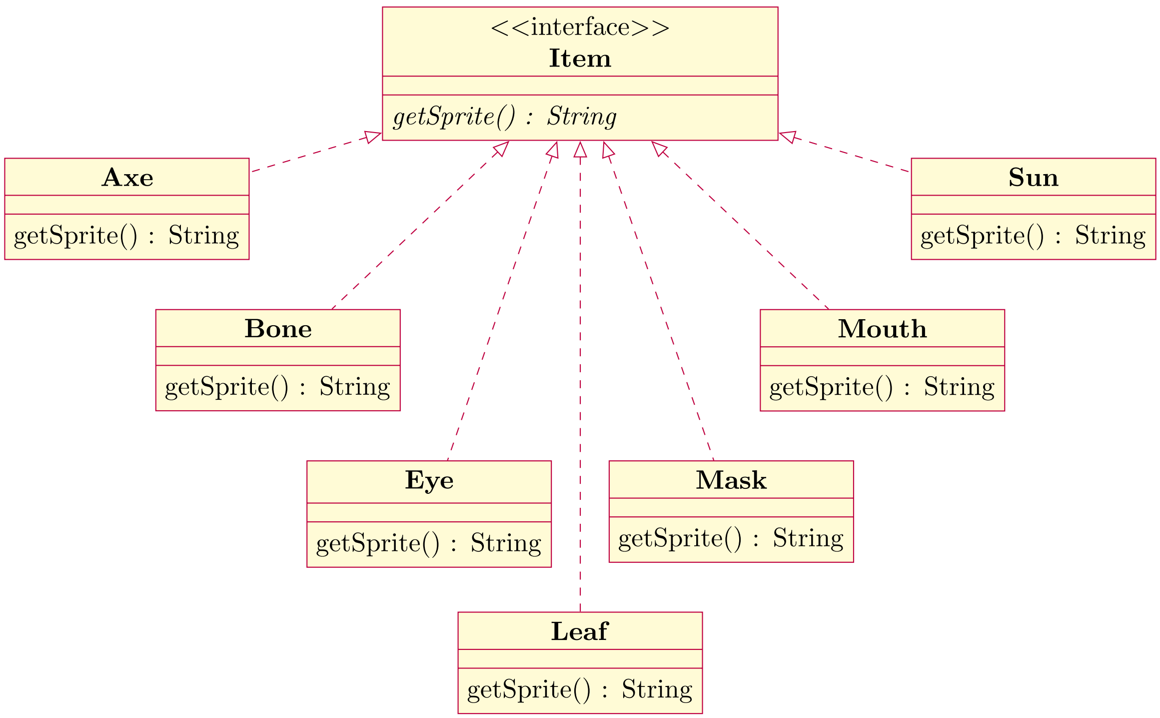


Game class owns a Board class. This is a composition.

2.2 Parametrized Classes

We didn't create any parametrized class in our source code but we use some parametrized classes from standard java library namely ArrayList and LinkedList. Java is a typed language and the concept of parametrized classes is mainly useful for working with collections. Parametrized classes allow us to derive the type of a class. We write the body of a parametrized class, we may invoke some operation on the parameter. Later when the class is bounded with a parameter the compiler tries to ensure this parameter supports operations required by the template.

2.3 Hierarchy Class Diagrams



Items are produced by an item factory, which can create each of the seven types of items. Each item implements the item interface, as a result the board can contain every type of item and request its sprite. A similar functionality could be implemented using an item class with an id attribute, however such an implementation would make further expanding each item individually much more complicated and inconvenient.