Assignment 1

TI2206 Software Engineering Methods (2015-2016 Q1)
Date: 18/9/2015
Group Number: 0
Group Members:
Thomas Baars tcbaars@gmail.com
Lars Ysla yslars@gmail.com
Boris Schrijver boris@radialcontext.nl
Adriaan de Vos adriaan.devos@gmail.com
Dylan Straub d.r.straub@student.tudelft.nl

Exercise 1

1. Starting with the requirements, we derive the Classes, Responsibilities, and Collaborations for our game. Our first step is to determine a set of candidate classes.

   Candidate classes:
   - GameController
     i. Responsibilities:
        1. Keep track of game state (start/pause/playing/over, current score)
        2. Display Playing Field
        3. Display relevant Screen (title/instructions/game over)
        4. Update Score
        5. Call Fish Controller
        6. Handles key inputs
        7. Passes key inputs to Fish Controller
     ii. Collaborations:
        1. Fish Controller
        2. Screen
        3. Playing Field
   - Player's Fish
     i. Responsibilities:
        1. Store its own state (size, location, orientation, speed, alive/dead)
        2. Handles Create/Read/Update/Destroy requests from Fish Controller
     ii. Collaborations:
        1. Fish Controller
   - Enemy Fish
     i. Responsibilities:

            1. Store their own state (size, locations, orientation, speed, alive/dead)
            2. Handles Create/Read/Update/Destroy requests from Fish Controller
        ii. Collaborations:
            1. Fish Controller
    - Fish Controller
        i. Responsibilities:
            1. Calls Player's Fish to Create/Read/Update/Destroy Player's Fish
            2. Calls Enemy Fish to Create/Read/Update/Destroy Enemy Fish
            3. Handle Collisions between fish
            4. Call GameController to update game state (win/lose)
        ii. Collaborations:
            1. Player's Fish
            2. Enemy Fish
            3. GameController
    - Playing Field
        i. Responsibilities:
            1. Display Player's Fish
            2. Display Enemy Fish
        ii. Collaborations:
            1. Fish Controller
            2. Player's Fish
            3. Enemy Fish
    - Screen
        i. Responsibilities:
            1. Display Game state (playing field / title screen / instructions / game over)
        ii. Collaborations:
            1. Game Controller
            2. Playing Field

In comparing this to our actual implementation, there are a few notable differences:

- Tests
    a. Both the required use of junit for testing and the requirement that there must be 75% meaningful test coverage both fall under the category of non-functional requirements. In the set of classes discussed above, we focused primarily on the functional requirements.
- application.Launcher and gui.MainFrame
    a. When comparing these classes from our implementation to the above CRC model, we may conclude that the contents of these two classes might better have been combined. The responsibilities of these two

classes fall under the responsibilities of the GameController class in our CRC model: namely displaying the relevant screens

- Layers
  a. In our implementation, we make use of a variety of layers, each of which is an extension of the Layer class. These are used to display the various game states (title screen, pause, game over, game area). For the static screens (title, pause, game over, and instructions), it may, in retrospect have been more efficient to have a single “modal layer”, to draw the appropriate screen, as much of the code in these static layers is duplicated. The responsibilities currently handled by the various layers would seem to fall under the responsibilities of the Screen class in the CRC model
- Animations
  a. Unlike our CRC model, our implementation also has an Animation class to handle the mechanics of animating the images in the game, such as the fish. Potentially, this could have been combined into the Entity class, in our implementation, as only the entities (i.e. the fish) in the game are animated
- Enumerations
  a. Our implementation contains a Key and a Direction class, which our CRC model does not. These are enumerations, which serve to simplify the conversion of keystrokes into events in the game. In our CRC model, these responsibilities are handled by the GameController.
- Handlers
  a. In contrast to our CRC model, our implementation makes use of a variety of handlers, which provide the functionality that is relegated to the GameController and Fish Controller in our CRC model. One exception is our implementation’s FontOutlineHandler, which could probably be combined with the suggested “model layer”, since the functionality of this class is really only needed in the display of the text screens.
- Entities
  a. Although the CRC model only refers to fish (enemy fish and the player’s fish), our implementation also contains an Entity class, which was created for the purpose of being able to extend the game to contain other entities such as obstacles, etc. in the future.

2. The main classes which we implemented in our project, in terms of responsibilities and collaborations are:
   - GameLayer
     i. Responsibilities:
        1. Draws the relevant screen (paused, game over, playing field)
        2. Passes keystrokes to the GameHandler
     ii. Collaborations:
        1. SinglePlayerGameHandler

2.

- SinglePlayerGameHandler:
    i. Responsibilities
        1. Handles game states
        2. Handles collisions (via CollisionHandler)
    ii. Collaborations
        1.
- OptionsHandler
    i. Responsibilities
        1. Stores game parameters
    ii. Collaborations
        1.
- Enemy
    i. Responsibilites
        1. Displays enemy fish
    ii. Collaborations
        1.
- Player
    i. Responsibilities
        1.
    ii. Collaborations

A few classes in our implementation which could probably be merged are:

- HeadLayer + KeyHandler
- SInglePlayerGameHandler + CollisionHandler
- SinglePlayerGameHandler + EnemyHandler
- Launcher + MainFrame + HeadLayer
- Trout + Enemy
- TitleLayer/InstructionsLayer/FinishLayer/PauseLayer → “Modal Layer”
- FontOutlineHandler + Layer (in which case the HUD should be moved out of SinglePlayerGameHandler)
- GameHandler + SinglePlayerGameHandler

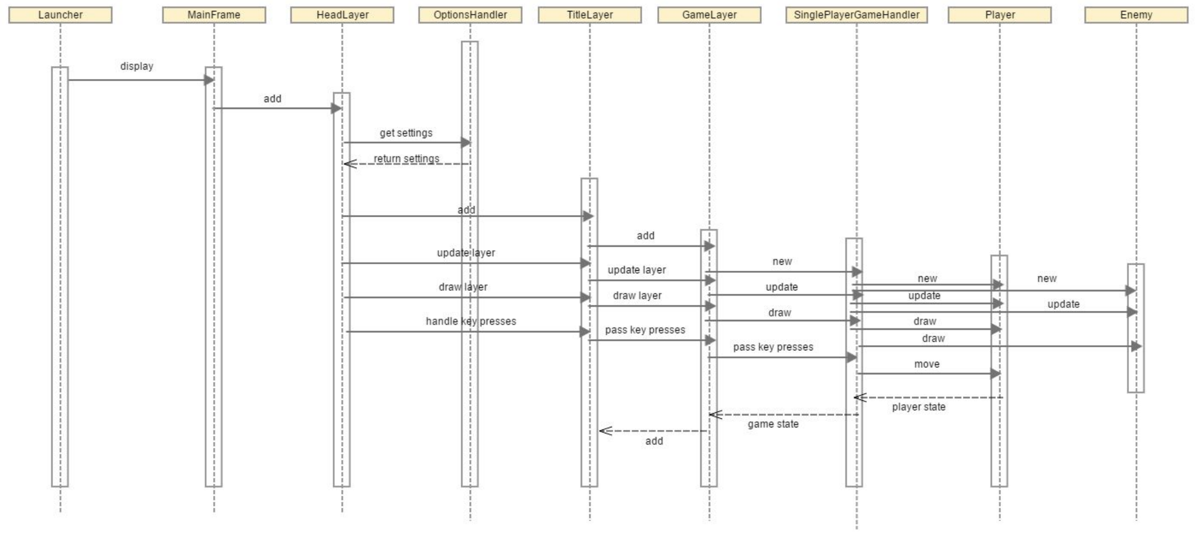
A few classes in our current implementation which could probably be removed are:

- HighScoresHandler

3. In our implementation, there are quite a few classes that were created as a consequence of the non-functional requirement that the game be built in Java. Launcher.java, Mainframe.java, Layer.java, and some of the subsequent sublayers are implemented primarily because of the way in which Java renders graphic components. Had the game been written in a different language, for a different system, these classes might be unnecessary, or at least quite different. Additionally, there are a few classes such as the two enumerations classes: Direction and Key, which serve mainly as “helper” classes to

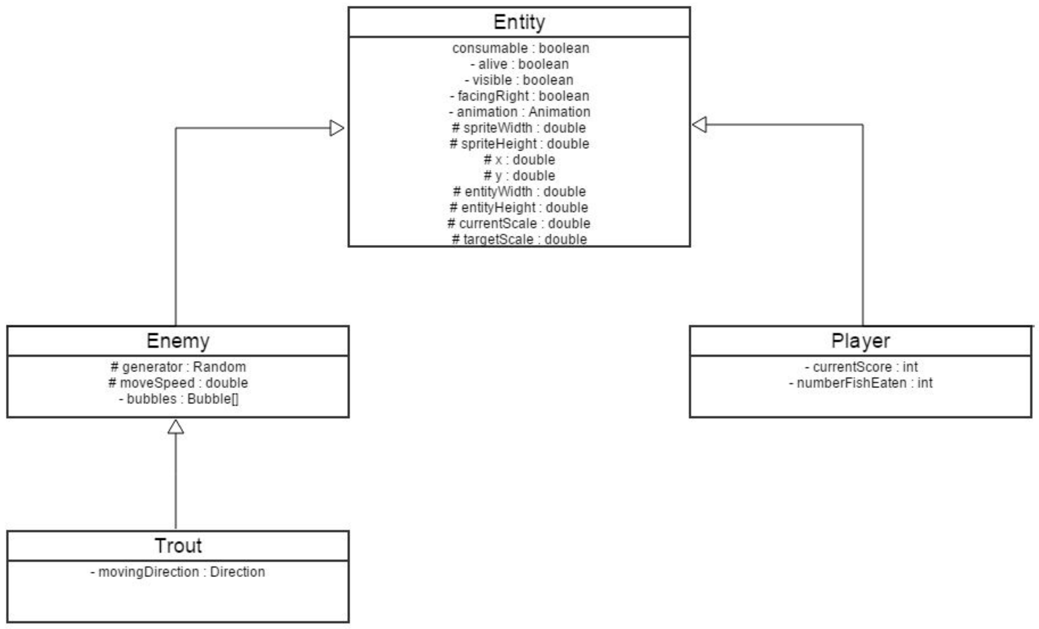
eliminate some code duplication. As such, these are not considered important enough to be among the “main” classes.

4. Class Diagram
5. Sequence Diagram:



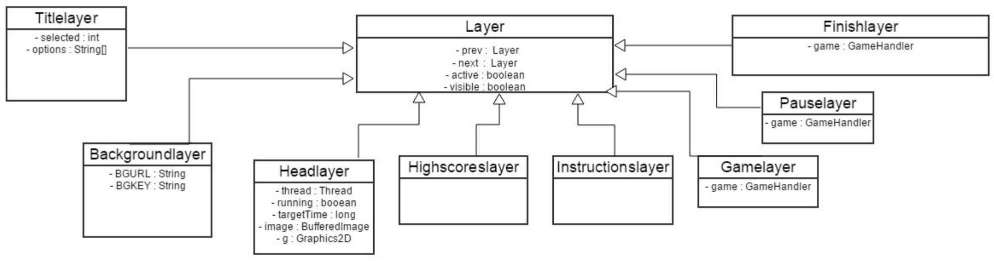
Exercise 2

1. Aggregation and composition are both member objects, they are a part of something bigger, but aggregation is used when the object can also have meaning on its own and composition is used when the object is only important for the bigger object. When a bigger object wouldn’t exist the composition object would no longer be useful but an aggregation object can still be useful. We use aggregation with the Enemy and the Enemyhandler the Enemy is an object itself and doesn’t need the Enemyhandler to exist in the game. Composition is used between the Animation and The Entity. The Entity can’t exist without an Animation.
2. The use of a parameterized class is handy when multiple objects would benefit from such a template but all these objects have different parameters. This will help because errors can be detected in compilation time instead of the run time. In our game there are not a lot objects or classes that benefit from this. Therefore it is not necessary to have a parameterized version for this.

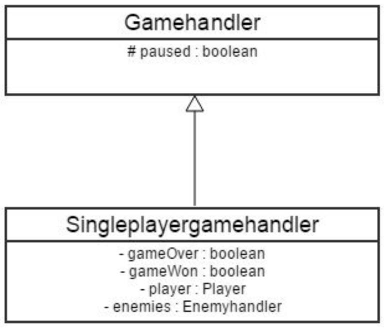


3.

Enemy and Player both inherit from Entity. this is polymorphism because an Entity can be anything. These hierarchies are created because there are a lot of methods that are shared between these entities and if we want to develop added features this allows for a quicker implementation of different entities. The Trout and Enemy have a Is-a inheritance because the Trout is a specialization of the Enemy.



All the different layers have an inheritance from Layer. We use inheritance her because the layer all have basic operations that can be used for all of them. This is polymorphism as they can all be treated uniformly.



the Singleplayergamhandler is a specialization of the Gamehandler. This is used for when or if we decide to implement multiplayer. there are no hierarchies that need to be removed since it is not to look pretty or for the reuse of code. We implemented them because we think they help in making a better structure for the game.

Exercise 3

1. Implementation done.
2. The logger is implemented in a single static class. It supports multiple loglevels and multiple outputs. Either file, console or both can be chosen as output. With each their own loglevel. Using a single static class means we it can be called from anywhere in the code by only using the appropriate function. No instantiation of a logger needed per class.