Software Engineering Methods - TI2206

Assignment 1

Group 47

Course Manager: Alberto Bacchelli
TA: Aaron Ang

Lillian Lehmann 4009037
Karin van Garderen 4144384
Christian Maulany 4097238
Rogier Vrooman 4001257
Fieke Miedema 4141369
Bas Böck 4366174

Exercise 1.1.

Classes

1. **Look for noun phrases in ‘must haves’ requirements**
    a. Game
    b. Main menu
    c. Board
    d. Layout element
    e. Wall
    f. Floor
    g. Ceiling
    h. Dragon (player)
    i. Keyboard
    j. Bubble
    k. Enemy
    l. Collision
    m. Starting position
    n. Level
2. **Refine to a list of candidate classes**
    a. **Model *physical objects***
        i. Board
        ii. Layout element
        iii. Wall
        iv. Floor
        v. Ceiling
        vi. Dragon
        vii. Bubble
        viii. Enemy
    b. **Model *conceptual entities***
        i. Game
        ii. Main menu
        iii. Collision
        iv. Starting position
        v. Level
    c. **Choose one word for one concept**
        i. Element
        ii. Wall **element**
        iii. Floor **element**
        iv. Ceiling **element**
        v. Entity
        vi. Dragon **entity**
        vii. Enemy **entity**

viii. Bubble **entity**
ix. State
x. Main menu **state**
xi. Level **state**

d. Be wary of adjectives
i. Wall, floor & ceiling element all have the same functionality. Therefore it is not necessary to create three different classes, but one ‘layout element’ class will be sufficient.

e. Be wary of missing or misleading subjects
i. OK.

f. Model categories of classes
i. Entity → Dragon, Enemy, Bubble
ii. Game state → Main menu, level
iii. Board consists of layout elements
iv. CollisionMap to coordinate collisions and positions

g. Model interfaces
i. Entity
ii. Game state

3. Create classes
a. Entity
b. Dragon entity
c. Enemy entity
d. Bubble entity
e. Game state
f. Main menu state
g. Level state
h. Board
i. Layout element
j. Game
k. CollisionMap

4. Create CRC cards for each class → discuss within team

One extra class is needed to keep track of the current game state. The level state class will be different for each level, so it should be named level1state for now.

Responsibilities

1. **Highlight verbs and determine which represent responsibilities**
    a. show (main menu/board)
    b. press (key)
    c. shall consist of (square bounded by walls)
    d. (entities) fall (down)
    e. fire (bubbles)
    f. appear (at top of the board)
    g. move (upwards)
    h. walk (left/right)
    i. (bubbles) trap (enemies)
    j. die
    k. collide
2. **Perform a walkthrough of the system**
    a. Player has to **press** start in the main menu (or exit or help).
    b. After pressing start, the first level will **show** itself on the screen.
    c. The level screen **consists** of layout elements, which are loaded according to a map file.
    d. Enemies and players are **placed** upon the board.
    e. Player can **move** around on the board (**walk** left & right, **jump** and **fall** down) by **pressing** keys on the keyboard.
    f. Enemies **move** around randomly.
    g. They player and enemies can't go through walls (except when jumping). Collision with a wall therefore leads to the entities staying at their current position.
    h. Player can **shoot** bubbles to **catch** enemies.
        i. Once the enemies are caught by the bubbles, they move up to the ceiling of the board.
        ii. Player can **collide** with these caught bubbles to **earn** points.
    i. When the player collides with an enemy, the player **dies**.
    j. When the player has died a certain amount of times, the game is over. A game over screen will **show**.
    k. Once the player killed and catched all the enemies, the player will continue to the next level. A new level will **show**.

3. **Study the candidate classes**
    a. **(what role --> what responsibility)**
        i. Entity (&dragon, enemy, projectile) - This class and its subclasses together model behaviour and location of elements in the board. Specific behaviour (like movement and bubbleshooting) should be modeled by subclasses, but location is general enough to be handled by the Entity class.
        ii. Game State (&menu state, level state) - This class and its subclasses together describe the different states. Transitions can be handled by another class, named GameStateManager, but the behaviour of the application should be contained in the subclasses. e.g. the reaction to input and the layout of the frame.
        iii. Board - This class holds the knowledge of board layout and must be able to generate the different levels from files.
        iv. Layout Element - The board consists of layout element.
        v. Game - This class holds the main game loop.
        vi. CollisionMap - This class handles collisions between different entities and the Board.
    b. **(responsibilities on class cards)**
        i. The gameStateManager keeps track of current game state and handles transitions between game states
4. **Study the relationships between classes**
    a. **'Is kind of'-relation**
        i. Dragon, Bubble and Enemy Entities are all a kind of Entity
        ii. Main Menu and Level state are a kind of Game State
    b. **'Is-analogous-to'-relation**
        i. none.
    c. **'Is-part-of'-relation**
        i. Layout Element is a part of Board
    d. **'Has-knowledge-of'-relation**
        i. Game has knowledge of the current Game State
        ii. The Board has knowledge of Layout Elements

Collaborations

1. **For each responsibility - Can the class fulfill the responsibility by itself?**
2. **For each class - What does this class know?**
3. **For each class - What other classes need its information or results?**
4. **Classes that do not interact with others should be discarded.**

CRC cards finished

Entity	
Super: /	
Sub: Dragon entity, Enemy entity, Bubble entity	
Track movement around the board	CollisionMap

Dragon entity	
Super: Entity	
Sub: /	
Die	
Shoot bubbles	Bubble Entity

Enemy entity	
Super: Entity	
Sub: /	
Movement (random)	Entity
Give damage to dragon	Dragon Entity, CollisionMap

Bubble entity	
Super: Entity	
Sub: /	
Move according to bubble behaviour	
Trap enemies	CollisionMap, Enemy Entity

Game state	
Super: /	
Sub: Main menu state, level state	
Handle state transitions	

Main menu state	
Super: Game state	
Sub: /	
Show main menu	
Handle input from player	

Level state	
Super: Game state	
Sub: /	
Show level situation	Layout Element
Handle input from player	

Board	
Super: /	
Sub: /	
Generate board from file	
Has knowledge of layout elements	CollisionMap

Layout element	
Super: /	
Sub: /	
Show element in the level	Level State
Collide with entities	Entity, CollisionMap

Game	
Super: /	
Sub: /	
Run the main game loop	
Has knowledge of current game state	Game State

CollisionMap	
Super: /	
Sub: /	
Determine collisions between entities and board	Entity, Board

Comparison with our implementation

a. Missing classes

There are no missing classes in our implementation, though naming has changed to a more general description.

b. Additional classes

Our implementation has an additional Game State Manager which keeps track of the state transitions, current state and available states. Game State is just an abstract class which has knowledge of the GSM and required methods for subclasses.

Also, we have implemented a GamePanel which handles input and output and passes the input on to the GSM. It also holds the main game loop, leaving the Game class to only generate a new window and GamePanel.

Exercise 1.2.

For exercise 1.2. and 1.3. we only consider the classes which were implemented in the first week.

The main classes are:

- **Game Panel** - which generates output screen and receives input. It acts as the front interface of the game. It collaborates with the GameStateManager to receive this output and to handle input further.
- **Game State Manager** - holds knowledge of the current state and can therefore redirect the commands from Game Panel to the right Game State.
- **Game State + subclasses** - these classes hold the actual implementation of input and output and work together with MapObjects, the HUD and tileMap to incorporate input and output with game logic.
- **Map Object + subclasses** - the mapobject holds all the knowledge of location, appearance and collision of objects in the game. Furthermore the subclasses hold information on the specific movement and functions the entity has. (for instance shooting bubbles by the player, move randomly by the enemies and move upwards after catching an enemy by the bubbles).
- **TileMap** - tileMap holds the knowledge of all the tiles on the board. It can generate different levels and draw them.

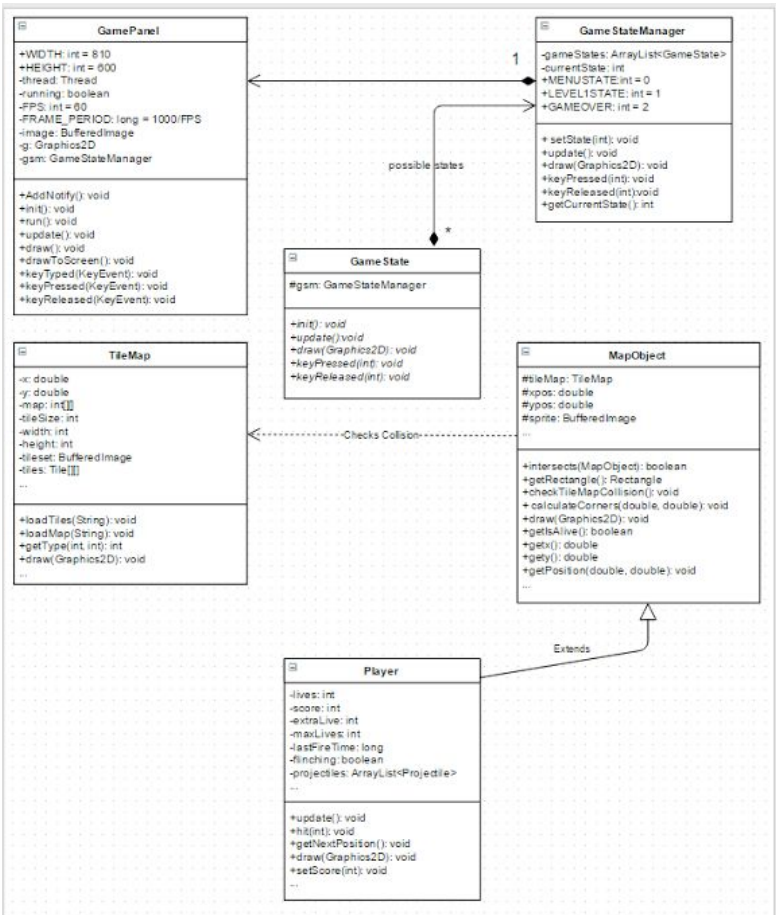
Exercise 1.3.

Less essential classes are:

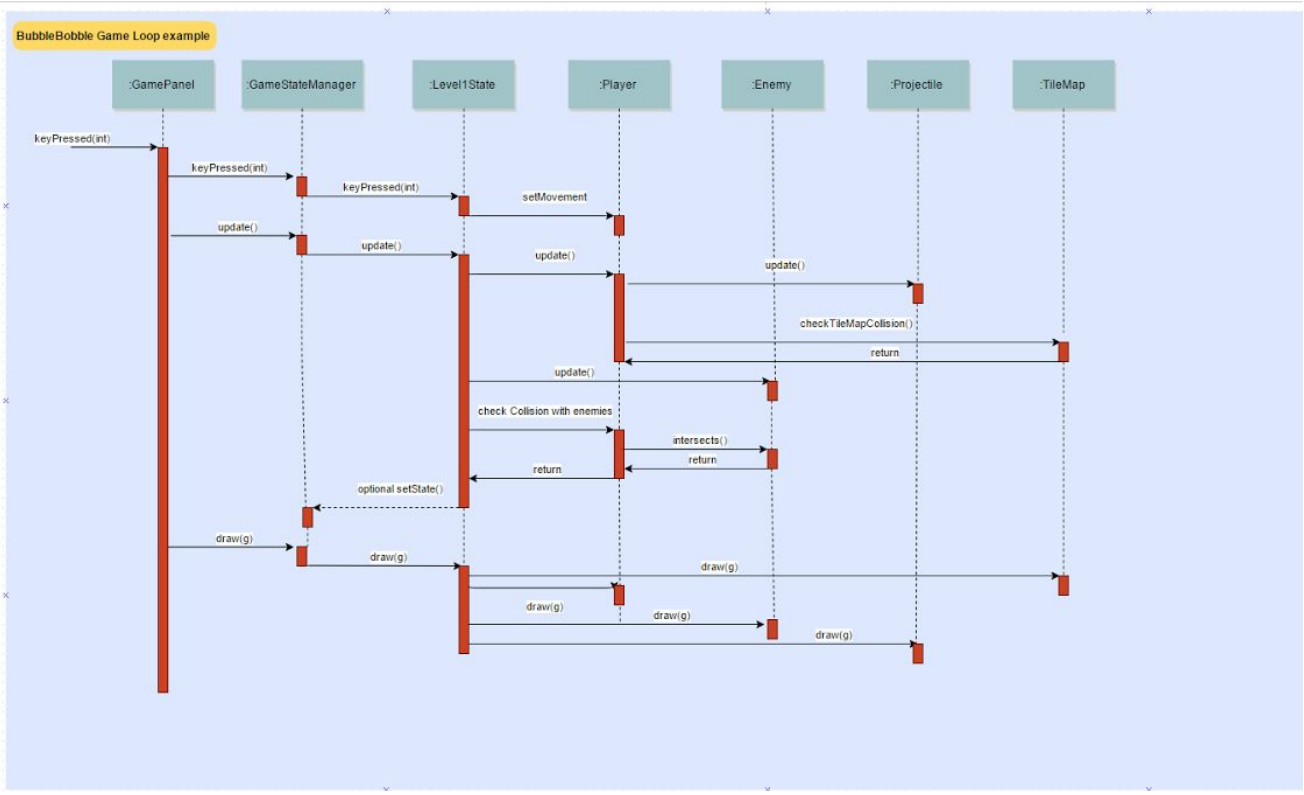
- **Game** - The only thing the Game class does is make a window and run the GamePanel. Not so much responsibility and this might be incorporated in GamePanel. However, keeping it as a seperate class may have its advantages, such as reduced size of classes and therefore readability.
- **Tile** - This class holds the detail of every single tile, which is its type, image and collision properties. As there is a limited amount of possibilities and type and image are correlated, this might be incorporated in TileMap as a property in a list of tiles. Keeping it as a class, however, has the advantage of extendability.
- **HUD** - This class generates the HUD. It has only a small responsibility, but keeping it as a class makes the entire system more readable. If you want to add something to the HUD, this is the easy place to find it.

There are some classes which are less important, since their content could be implemented by other classes. We decided not to change this in our game, since it increases the readability and it keeps the classes as small and specific as possible.

Exercise 1.4.



Exercise 1.5.



Exercise 2.1.

Aggregation and composition are specific kinds of association. Both show that an object of one class is part of an object of another class. Aggregation is used for the part-whole relationship. With an aggregation you show that an object is a part of a 'whole' other object. It is possible for object to be part of more than one object simultaneously. The notation of an aggregation is one with an open diamond at the end of the whole part (see figure 1.1). Multiplicities can be notated as usual with associations.

Figure 1.1 An aggregation

Composition is also used for a part-whole relationship, but with composition the whole strongly owns its parts. This means that if the whole object is copied or deleted, its parts are copied or deleted with it. Because of this, a part cannot be part of more than one whole, which leads to the
multiplicity at the whole and always has to be 1 or 0..1. A composition is notated with a closed

diamond at the end of the whole (see figure 1.2)

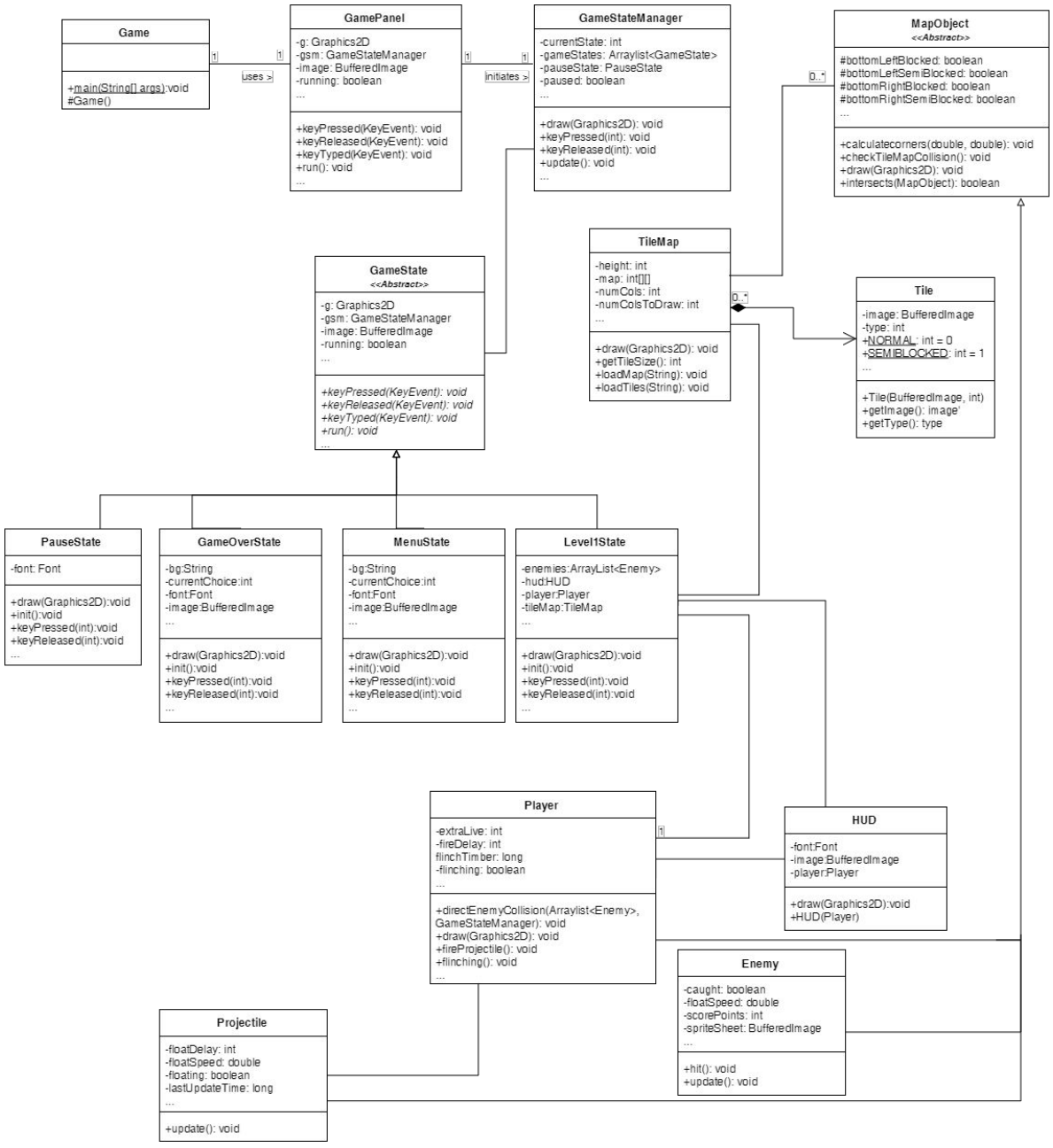
Figure 1.2 A composition

Where are composition and aggregation used in your project? Describe the classes and explain how these associations work.There is one associations between classes that can be defined as a composition, namely the association between Tile and TileMap. A TileMap uses different types of Tiles to construct the map for the game. Here the tiles work as the squares that make up the board of for example 'Tic-Tac-Toe'. Each Tile is part of exactly one TileMap, and when you would copy or delete the TileMap, you would copy or delete the Tiles with the map.There are more associations that could be described as a composition, but as Stevens and Pooley explain in their book 'Using UML – Software Engineering with Objects and Components' (2006): "In our experience people new to object-oriented modeling use aggregation and composition far too often. Remember that both are kinds of association, so whenever an aggregation or composition is correct, so is a plain association. If in doubt, use a plain association.". This is why we chose to use associations instead of compositions/aggregations for for example Game – GamePanel and GameSate – GameStateManager.

Exercise 2.2.

Though we currently do not use any parameterized classes, they have many valuable uses. They
allow programmers to implement more generic classes and algorithms that work on collections of
different types which can be customized. Parameterized classes are generally more type safe and
easier to read. Parameterization enforces type checks at compile time, preventing the programmer from causing runtime errors. Furthermore this eliminates the need for many type casts, improving readability and preventing faulty use of the class by both the programmer and others who are less familiar with the class.

Exercise 2.3.

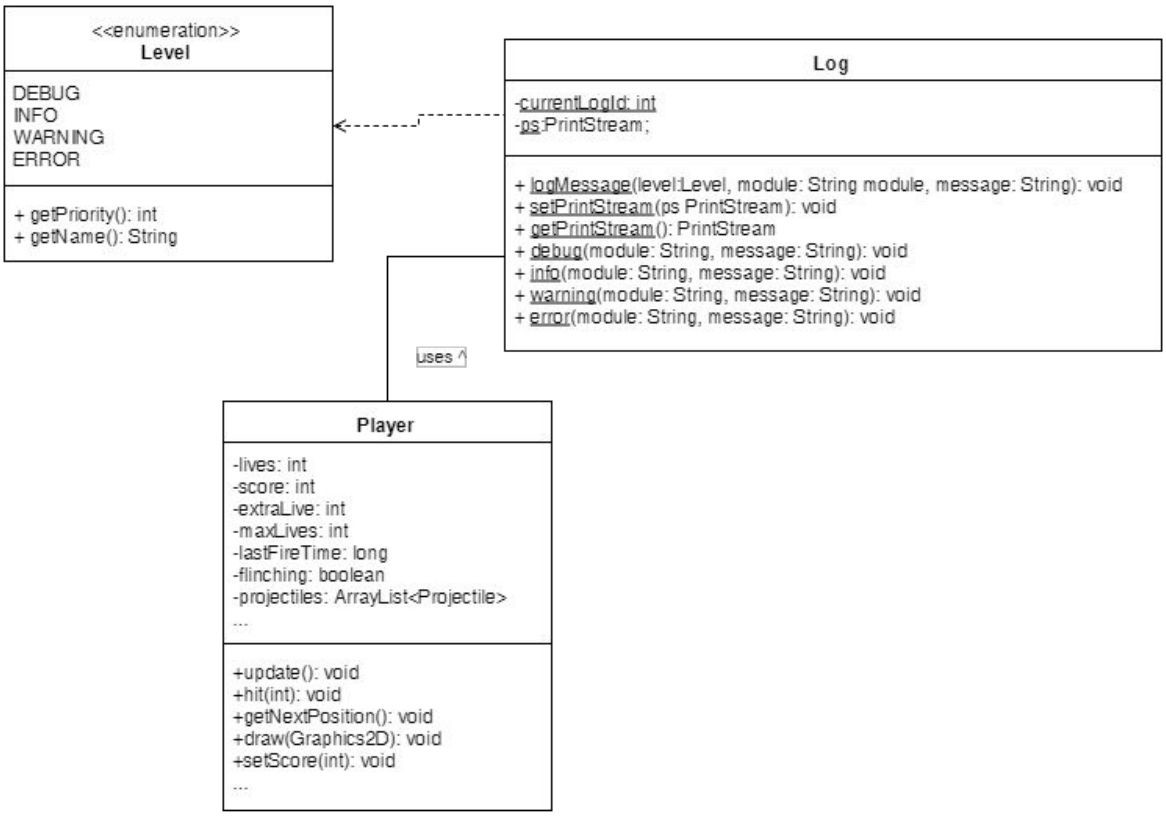


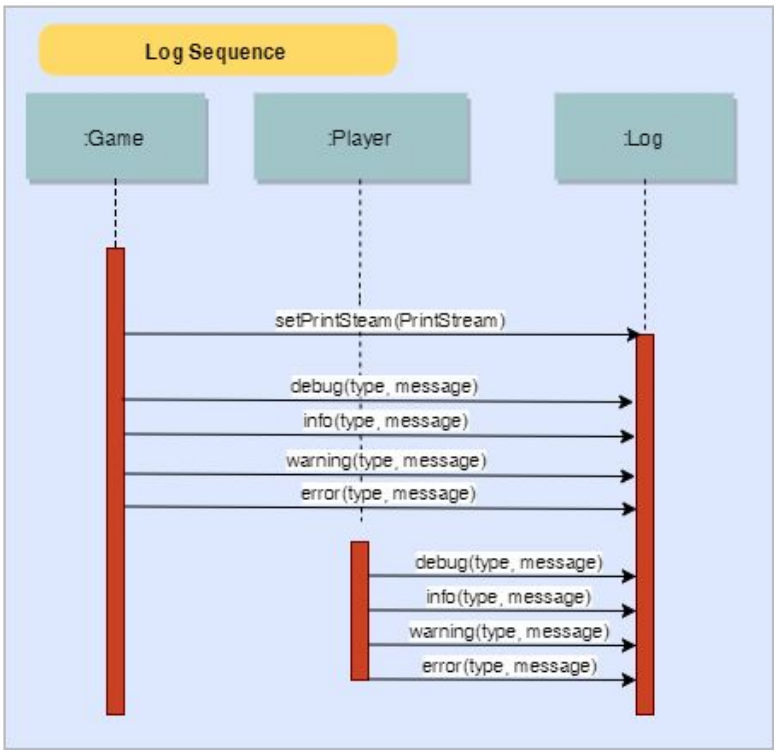
The project already contains quite some hierarchy. For example all moveable objects are subclasses of the MapObject and all the GameStates have one common abstract class. Therefore we did not remove any hierarchy. Furthermore there migh even be a lack of hierarchy. In order to achieve working results quickly at the moment we have a Level[n]State object for each level. This causes a lot of duplicate code, a strong indicator that we might need a common superclass. Because there are only two levels the lack of hierarchy is still manageable, but as soon as we are going to implement more, this abstraction will be needed.

Exercise 3.1.

1. **Data storage:**
   - 1.1. At launch, a text file will be created where all to be logged activities will be logged.
   - 1.2. The file will have a filename, that is distinct for each runned session: a timestamp will be used.
   - 1.3. Within the repository, a folder must be created where up to 10 files are stored.
   - 1.4. When more files are created (so after >10 runs) the oldest files must be deleted automatically.
2. **Logging:**
   - 2.1. At the top of the document (header) some basic information will be placed:
     - 2.1.1. Name of the program
     - 2.1.2. Version of the program
     - 2.1.3. Date and time
   - 2.2. All actions of the player must be logged in file. A new line will be created…:
     - 2.2.1. When the player makes a movement
     - 2.2.2. When the player shoots a bubble
     - 2.2.3. When the player collides with an enemy
     - 2.2.4. When the player collides with a caught enemy
     - 2.2.5. When the player loses a life
     - 2.2.6. When the player receives a new life
     - 2.2.7. When the player's score increases
     - 2.2.8. When the player reaches a new level
     - 2.2.9. When the player dies
   - 2.3. All possible warnings the player can have should be logged
   - 2.4. All possible errors the player can have should be logged
3. **Notation:**
   - 3.1. On each line of the log-file, a report is written with the following information:
     - 3.1.1. Log-ID number
     - 3.1.2. Timestamp
     - 3.1.3. What kind of report it is: Information, Warning or Error
     - 3.1.4. Brief description of action of player / warning / error

Exercise 3.2.


<<enumeration>>
Level
DEBUG
INFO
WARNING
ERROR
+ getPriority(): int
+ getName(): String
Log
-currentLogId: int
-ps:PrintStream;
+ logMessage(level:Level, module: String module, message: String): void
+ setPrintStream(ps PrintStream): void
+ getPrintStream(): PrintStream
+ debug(module: String, message: String): void
+ info(module: String, message: String): void
+ warning(module: String, message: String): void
+ error(module: String, message: String): void
uses ^
Player
-lives: int
-score: int
-extraLive: int
-maxLives: int
-lastFireTime: long
-flinching: boolean
-projectiles: ArrayList<Projectile>
...
+update(): void
+hit(int): void
+getNextPosition(): void
+draw(Graphics2D): void
+setScore(int): void
...


Log Sequence
:Game
:Player
:Log
setPrintSteam(PrintStream)
debug(type, message)
info(type, message)
warning(type, message)
error(type, message)
debug(type, message)
info(type, message)
warning(type, message)
error(type, message)