Assignment 5

TI2206 Software Engineering Methods (2015-2016 Q1)
Date: 23/10/2015
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

Table of Contents

Exercise 1 - 20-Time, Revolutions
    1. Requirements
        Crucial
        Non-Crucial
    2. Analysis and Design
        User Story
Exercise 2 - Design Patterns
    Design Pattern I - Factory
        Description
        Class Diagram
        Sequence Diagram
    Design Pattern II - State
        Description
        Class Diagram
        Sequence Diagram
Exercise 3 - Wrap-up Reflection

Exercise 1 - 20-Time, Revolutions

1. Requirements

The planned extension for our Fishy game is a movement speed boost power-up. The requirements for this extension are defined below. Each requirement for the planned extension is divided into subcategories based on their priority, where requirements marked "Crucial" are crucial for the extension's implementation to be considered a success, and those marked "Non-crucial" are important but will only be implemented if time permits.

Crucial

- The game shall spawn a power-up every minute.
- The power-up shall have a unique sprite which will be displayed when spawned.
- The game shall remove the power-up when the player consumes it.
- The game shall start the power-up timer when the player consumes it.
- The power-up timer shall countdown from 10 seconds, after it has been consumed.
- Once the power-up timer has reached 0 it will expire, and the power-up will no longer be active.
- The game shall increase the movement speed by 100%, only if the power-up is active.
- The game shall change the movement speed back to the original speed, when the power-up is no longer active.

Non-Crucial

- The game shall spawn the power-up at a random location on the screen.
- The game shall despawn the power-up if it has not been picked up after 5 seconds.
- The power-up sprite shall be clearly visible with a bright yellow color.
- The game shall display a timer indicating the current time left for power-up, once consumed.

2. Analysis and Design

The analysis and design documentation produced based on the planned extension, of a movement speed boost power-up, are given below.

User Story

- As a player, I want the ability to consume a power-up which accelerates my movement speed by 100% when active.
- As a player, I want the power-up to last only 10 seconds before it becomes inactive.
- As a player, I want the power-up to spawn every 1 minutes.

- As a player, I want the power-up to be clearly visible.

Design

We plan on implementing it by mixing the Bubble and Enemy Entities and BubbleSpawner and EnemySpawner.

First we need to add the new “speedup” to all the current enums. So we need to add “speedup” to GameEntities, GameSprites and GameAnimations and add the right configuration options.

Then we need to create sprites/SpeedupSprite.java, entities/powerups/Speedup.java and entityspawner/SpeedupSpawner. This classes will be based on the current Bubble and Enemy classes. Most methods and code are partly the same so there will be a lot of inheritance and extending. Just like the current Bubble and Enemy classes we take a good look at what actions this object is responsible for and what information he needs to share.

We need to implement a way of tracking the current duration of the powerup. The collision/consuming between the player and the powerup can be handled just like the current enemys; the “consume(PowerUp powerUp)” method. If we handle it through the consume method it will be easy to adjust the movespeed through the Player class.

After this has been finished it is important that the classes/methods needs to be called by the rest of the software. That is quite easy with our current code layout, as this can all be done in the singleplayergames/ClassicGame.java class

This is our current idea of the implementation, changes can always occur during the development because we work iteratively. We will use pull-request code review with this implementation to ensure good and clean code.

Exercise 2 - Design Patterns

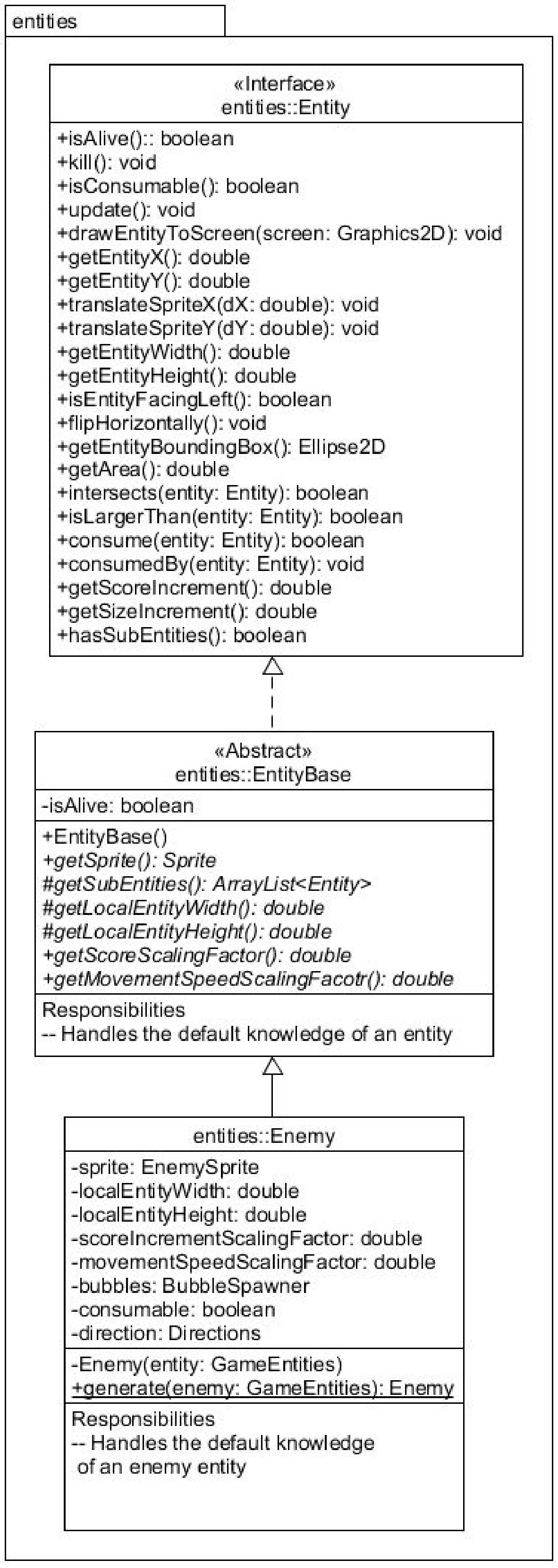
Design Pattern I - Factory

Description

Our game could benefit from implementing the factory design pattern to generate enemies. We had debated implementing the decorator design pattern to make customising the different enemy types easier, by allowing us to specialise certain methods for each enemy type, for example the movement methods This would mean the enemy classes would have to implement or override certain methods in the Enemy class, for example some enemy classes may be implemented to have both horizontal and vertical movement. However we decided to keep the generic enemy instead, which is customised by the information in the GameEntities enumeration. This means that we could implement the factory design pattern to generate enemies based on the GameEntities enumeration, instead of having multiple classes.

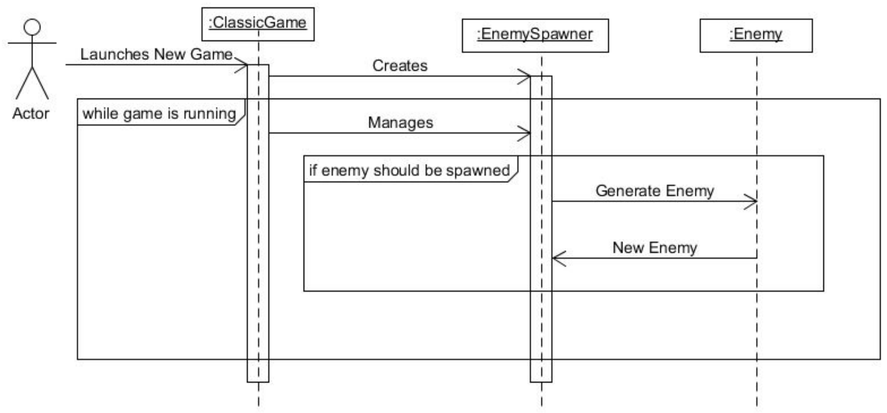
Class Diagram

Below is a class diagram which gives an overview of how the factory design pattern has been implemented to generate new enemies:


entities
«Interface»
entities::Entity
+isAlive():: boolean
+kill(): void
+isConsumable(): boolean
+update(): void
+drawEntityToScreen(screen: Graphics2D): void
+getEntityX(): double
+getEntityY(): double
+translateSpriteX(dX: double): void
+translateSpriteY(dY: double): void
+getEntityWidth(): double
+getEntityHeight(): double
+isEntityFacingLeft(): boolean
+flipHorizontally(): void
+getEntityBoundingBox(): Ellipse2D
+getArea(): double
+intersects(entity: Entity): boolean
+isLargerThan(entity: Entity): boolean
+consume(entity: Entity): boolean
+consumedBy(entity: Entity): void
+getScoreIncrement(): double
+getSizeIncrement(): double
+hasSubEntities(): boolean
«Abstract»
entities::EntityBase
-isAlive: boolean
+EntityBase()
+getSprite(): Sprite
#getSubEntities(): ArrayList<Entity>
#getLocalEntityWidth(): double
#getLocalEntityHeight(): double
+getScoreScalingFactor(): double
+getMovementSpeedScalingFacotr(): double
Responsibilities
-- Handles the default knowledge of an entity
entities::Enemy
-sprite: EnemySprite
-localEntityWidth: double
-localEntityHeight: double
-scoreIncrementScalingFactor: double
-movementSpeedScalingFactor: double
-bubbles: BubbleSpawner
-consumable: boolean
-direction: Directions
-Enemy(entity: GameEntities)
+generate(enemy: GameEntities): Enemy
Responsibilities
-- Handles the default knowledge
of an enemy entity

Sequence Diagram

Below is a sequence diagram which shows how the implementation of the factory design pattern has been implemented:



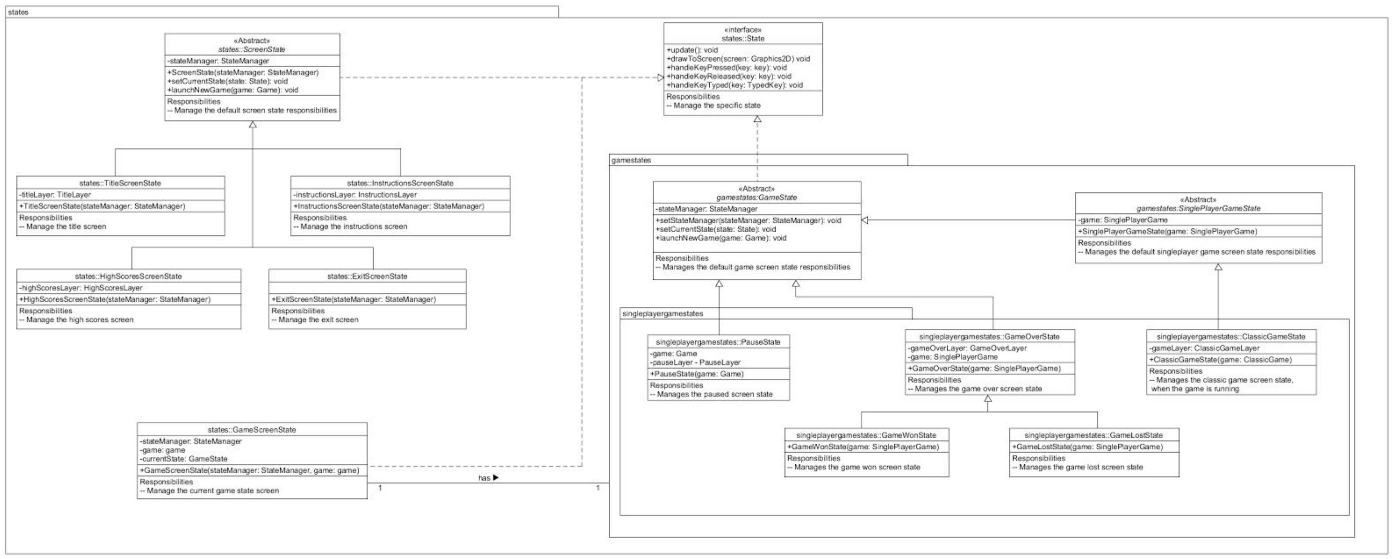
Design Pattern II - State

Description

Our game has multiple states, mainly the Title Screen state, the Game Running state, the Paused state, the Game Over state, the Instructions Screen state, and the High Score Screen state. Therefore we implemented the state design pattern, to facilitate changing between these states.

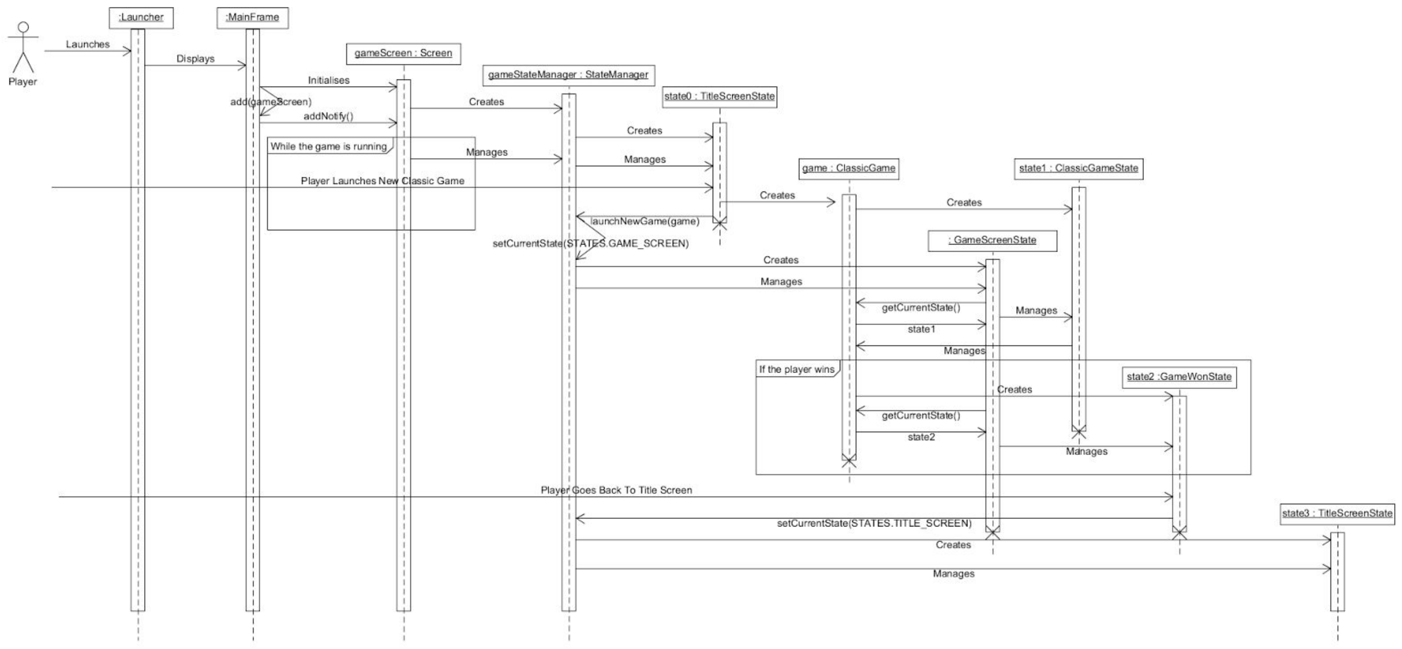
Class Diagram

Below is a class diagram which gives an overview of how the state design pattern has been implemented in our code:


states
«Abstract»
states::ScreenState
-stateManager: StateManager
+ScreenState(stateManager: StateManager)
+setCurrentState(state: State): void
+launchNewGame(game: Game): void
Responsibilities
-- Manage the default screen state responsibilities
states::TitleScreenState
-titleLayer: TitleLayer
+TitleScreenState(stateManager: StateManager)
Responsibilities
-- Manage the title screen
states::InstructionsScreenState
-instructionsLayer: InstructionsLayer
+InstructionsScreenState(stateManager: StateManager)
Responsibilities
-- Manage the instructions screen
states::HighScoresScreenState
-highScoresLayer: HighScoresLayer
+HighScoresScreenState(stateManager: StateManager)
Responsibilities
-- Manage the high scores screen
states::ExitScreenState
+ExitScreenState(stateManager: StateManager)
Responsibilities
-- Manage the exit screen
states::GameScreenState
-stateManager: StateManager
-game: game
-currentState: GameState
+GameScreenState(stateManager: StateManager, game: game)
Responsibilities
-- Manage the current game state screen
has ▶
1
1
«interface»
states::State
+update(): void
+drawToScreen(screen: Graphics2D) void
+handleKeyPressed(key: key): void
+handleKeyReleased(key: key): void
+handleKeyTyped(key: TypedKey): void
Responsibilities
-- Manage the specific state
gamestates
«Abstract»
gamestates:GameState
-stateManager: StateManager
+setStateManager(stateManager: StateManager): void
+setCurrentState(state: State): void
+launchNewGame(game: Game): void
Responsibilities
-- Manages the default game screen state responsibilities
«Abstract»
gamestates:SinglePlayerGameState
-game: SinglePlayerGame
+SinglePlayerGameState(game: SinglePlayerGame)
Responsibilities
-- Manages the default singleplayer game screen state responsibilities
singleplayergamestates
singleplayergamestates::PauseState
-game: Game
-pauseLayer - PauseLayer
+PauseState(game: Game)
Responsibilities
-- Manages the paused screen state
singleplayergamestates::GameOverState
-gameOverLayer: GameOverLayer
-game: SinglePlayerGame
+GameOverState(game: SinglePlayerGame)
Responsibilities
-- Manages the game over screen state
singleplayergamestates::ClassicGameState
-gameLayer: ClassicGameLayer
+ClassicGameState(game: ClassicGame)
Responsibilities
-- Manages the classic game screen state, when the game is running
singleplayergamestates::GameWonState
+GameWonState(game: SinglePlayerGame)
Responsibilities
-- Manages the game won screen state
singleplayergamestates::GameLostState
+GameLostState(game: SinglePlayerGame)
Responsibilities
-- Manages the game lost screen state

Sequence Diagram

Below is a sequence diagram which gives a general overview of how the state design pattern is used in our game:



The above sequence diagram depicts how the state design pattern is implemented, using a scenario where the player launches the application, then launches a new classic game, then wins the game, and then returns to the title screen.

Exercise 3 - Wrap-up Reflection

The Evolution of Our Source Code

Initial Version

The initial version of our software was characterized by its primary underlying motivation: namely to produce a simple, working version of our game. The functionality and basic playability of the game where the main focuses of concern. And, as intended, the game was indeed playable on some basic level. But, even so, it was clear that there was room for improvement in the overall user experience. We needed to do some extreme programming in 2 weeks so the game code was a little bit messy and undeveloped due to time constraints. We have refactored and fixed it a lot in the following iterations.

Evolution

With each sprint, our software was presented with new challenges, Not only was there new functionality to implement: e.g. features such as high scores, sound, music, etc., but there was reason to rethink and restructure the existing code in order to improve its maintainability and scalability. Indeed, virtually the entire project was restructured in order to ensure that each class would have only one responsibility. Furthermore, we were able to reduce the number of design flaws such as the existence of God classes and schizophrenic classes. The addition of a logger improved the ease with which the code could be debugged.

Final Version

The final version is a dream to behold. Perhaps this is because it exists somewhere in the future. Or perhaps it is because, in the paraphrased words of Erik Meijer, software is never truly finished. Can we really speak of a final version? Perhaps it would be more constructive to then consider the current version of the project.

Current Version

Compared to the initial version, it can be said that the focus of development has shifted. Whereas the initial version was mainly concerned with the basic functionality of the game, the current version reflects a focus on maintainability, scalability, and testability. In a certain sense, the initial version was built for the user, while the current version is built for the developer. Although it must be said that the current version also works better for the user than the initial version, it can almost be seen as a side-effect of the fact that the code has become more maintainable and scalable. These latter aspects are the qualities that allow improvements to the user experience to be made more easily and reliably.

Issues Encountered

Test Coverage

We started late with testing our classes. When we had a test coverage of about 30 percent we discovered that our structure of the project was not fully optimized and desired to change this. The side effect was that our tests were not in sync with the new structure of the project which lead to having 0 percent testing on our project. Afterwards we had many tasks on increasing test coverage. But because of setbacks we could not implement many of these tasks as they were less important than the other tasks. When we came towards the end of the project the test coverage was so low that it became a serious problem that could no longer be a secondary task but had to become the main focus of the project.

Missed Deadlines

We scheduled so many tasks each sprint that it was hard implementing them before the deadline. Some weeks there were also setbacks such as illness which we did not account for. The project also suffered from the other projects our team members were involved in. Because we did not discuss the deadlines of other projects together we did not consider these in the sprint plan. Overall the main tasks were finished and only the documentation and testing were pushed back to a later date which lead to an increase of tasks in later sprints.

What We Have Learned

Process

Perhaps the overarching lesson of this course was that software is really defined by the process by which it was created. Rather than being some static product that meets a fixed set of stipulations that are specified in a requirements document, the software product seems to be a by-product of the process. Insofar as that process incorporates best practices, the resulting product will commensurately better. This course was not so much about creating a game as it was about mastering a process of which the game that we built was a product. The game could have been built any number of ways, but it was the process that resulted in a game that is built as much for developers as it is for end-users. This process consisted of several components: the Scrum methodology, design pattern implementation, code review, versioning/integration tools such as Travis CI and GitHub, and building/testing tools such as FindBugs, Maven, JUnit, and Mockito/JMockit.

Scrum

The use of the Scrum methodology proved to be an excellent way to ensure that the quality of the resulting software would be improved each week. Aside from receiving feedback on the code, which helped us to identify the necessary changes that had to be made, we found Scrum to also be useful in improving the process itself. It allowed us to identify changes that had to be made in the way that we communicated and prioritized out tasks. Because the software is, as mentioned previously, a product of this process, it is absolutely essential that this process contains a mechanism by which it can improve itself.

Tools

Of the various tools that we used, we might conclude that GitHub provided us with the most value as a team. Not only did it provide an excellent way of enabling collaboration, through branches and pull requests, it also served as a communication platform for code review. We would certainly use it again in future projects.

Conclusion

In conclusion, our journey in this course has been one of evolution. Not only has our software evolved over many iterations, but we have evolved from software developers to software engineers. Rather than allowing ourselves to be content merely with a product that works, we have learned to become masters of a process by which production-quality software can be delivered.