https://github.com/bpl714/DIKUGames/tree/BreakoutExam

Assignment 11G

Department of Computer Science
University of Copenhagen

Helga Rykov Ibsen <MCV462>, Natasja Juul <QRG456>, Silas Lundin <BPL714>,
Agnes Galster <SNH917>

June 2022

https://github.com/bpl714/DIKUGames/tree/BreakoutExam

1 Introduction

The purpose of this technical report is to discuss an implementation of the game `Breakout`. The implementation covers the following sub-tasks : creating the levels of the game based on .txt files, creating block entities, a moveable player, a moveable ball, a statemachine, a points system, two special types of blocks, a GameOver state, player lives and power-up effects for the blocks of the game. The implementation draws heavily on the `DIKUArcade` directory.

`DIKUArcade` is the directory which contains all the background functionality that makes the game appear and work as it does. Using `DIKUArcade` to implement `Breakout` helps the overall design of the program become more flexible, maintainable and easy to understand.

The program which is the foundation of this report has been created for the course "Softwareudvikling" at DIKU.

2 Background

The original Breakout game was developed in 1976 by Atari. For this assignment's purposes, `Breakout` is the directory in `DIKUGames` which contains the classes and interfaces which are used to concretely implement the `Breakout` game. Thus `Breakout` contains `Ball, Game, BreakoutBus, Player` and `Program` classes as well as the directories `BlockFuncitons`, `GameStates` and `LevelManagement`. `DIKUArcade` is another directory in `DIKUGames` which contains all the background functionality that makes the `Breakout` appear and work as it does. It is a good idea to use `DIKUArcade` to implement `Breakout` because this makes the overall design of the program more maintainable and funcitonal.

3 Analysis

3.1 The overall problem

In order to implement the `Breakout` game, it will be necessary to load the level-files, to create a player, a ball, a block, and special block entities as well as to implement gamestates and a points system. Thus the overall goal of implementing the `Breakout` game can be divided into a number of sub-tasks which will be listed here.

3.2 Sub-problem : Block Functions

Also the issue of creating blocks for the `Breakout` game can be broken down into smaller sub-tasks : a) making the blocks be `DIKUArcade` entities, b) rendering the blocks to the screen, c) giving the blocks a "health status", d) making the blocks damageable, e) making the blocks deletable. Thus it is a goal of this sub-task to create a `Block` class which inherits from `DIKUArcade`'s `Entity` class.

In order to create the special blocks, `Hardened` and `Unbreakable`, a number of requirements must be adhered to : the hardened block must have twice the amount of health of a normal block, the unbreakable block must be unbreakable, the blocks must have some level of inheritance from the main block type, and the hardened block must change image when hit.

3.3 Sub-problem : Power-Ups

The implementation of the Power-Up effects involves the following sub-tasks : a) visually showing the power-up effect of a block, when it is destroyed, b) making sure that the visual element moves at a constant and negative vertical speed, c) making the visual element disappear when colliding with the player, d) making sure that the visual element is deleted when reaching a position that is less than the lower window boundary.

3.4 Sub-problem : Ball

The implementation of the ball involves the following sub-tasks : a) the ball may only leave the screen at the bottom, b) the ball must be deleted when leaving the screen, c) collision with a block will be registered, d) balls must move at same speed, e) ball unable to get stuck in same trajectory, f) the ball is always launched in an upwards, always more vertical than horizontal, direction, g) use DIKUArcade's AABB algorithm to detect collisions.

It is the goal of this sub-task to create restrictions for the ball's upwards, left and right movement, to associate leaving the screen's limits with being deleted, to set an immutable speed, to implement the AABB algorithm as part of the game's collision detection, and to instantiate some change to the ball's direction when it hits a block, the paddle or the walls.

3.5 Sub-problem : Player

In creating a `Player` entity one must : a) make the `Player` a `DIKUArcade` entity, b) render the `Player` to the screen, c) make the `Player` moveable, d) restrict the `Player`'s movement to the limits of the screen.

3.6 Sub-problem : Player Lives

The implementation of the Player Lives involves the following sub-tasks : a) making sure that the value of player lives is never negative, b) ensuring that the player loses a life when there are no more balls on the screen, c) giving the player a new ball, when all balls have gone and the player has lost a life, d) ensuring that when the player has no more lives, the game changes to GameOver state.

3.7 Sub-problem : Game States

The creation of the `State machine` involves the following jobs : a) making sure that only one state is active at a time, b) making sure that states use the

`IGameState` interface, c) having a state for before, during and after the game is running, d) having running, paused, mainmenu game states, e) having the possibility of level changing, f) providing the possibility of incrementing levels, g) returning to mainmenu when all levels have been played.

The implementation of the individual states, e.g. GameOver, involves the following sub-tasks : a) creating the state class, b) incorporating the state class into the state machine - thus making sure that the state changes appropriately (e.g. when the game is won/lost), c) making sure that the game begins afresh once lost or won.

3.8 Sub-problem : Level Management

The task of loading levels from .txt files, can be split into a number of sub-tasks : a) loading the .txt files, b) extracting map, meta and legend data separately, c) translating the elements of the map into blocks using the images stored in the Assets directory, d) storing the map as list or array of strings, e) storing the converted map in an entity container, f) clearing this container between level-changes. Thus it is a goal of the level loading sub-task to perform these sub-sub-tasks.

3.9 Sub-problem : Score

The implementation of the points system of `Breakout` involves making sure that a) points are not negative, b) points system is not publicly available in the code, c) points are awarded when blocks are destroyed, d) points are displayed while the game is running.

4 Design

4.1 Overview

To get an overview of the concepts of this assignment, as well as their responsibilities, associations and attributes, the reader might consider the updated UML-diagram of the `Breakout` classes and relations between them - see figure 1.

4.2 Block Functions

The `Block` class will not only contain the standard block type, but also the special block types `Hardened`, `Unbreakable` and `Upgrade`. The `Block` class is responsible for updating and checking whether the block has been hit or granted a power-up, as well as for rendering the block in its standard, hardened or power-up state. If the block is of the Upgrade type, it must randomly select one of the five types of upgrades, and visually represent that.

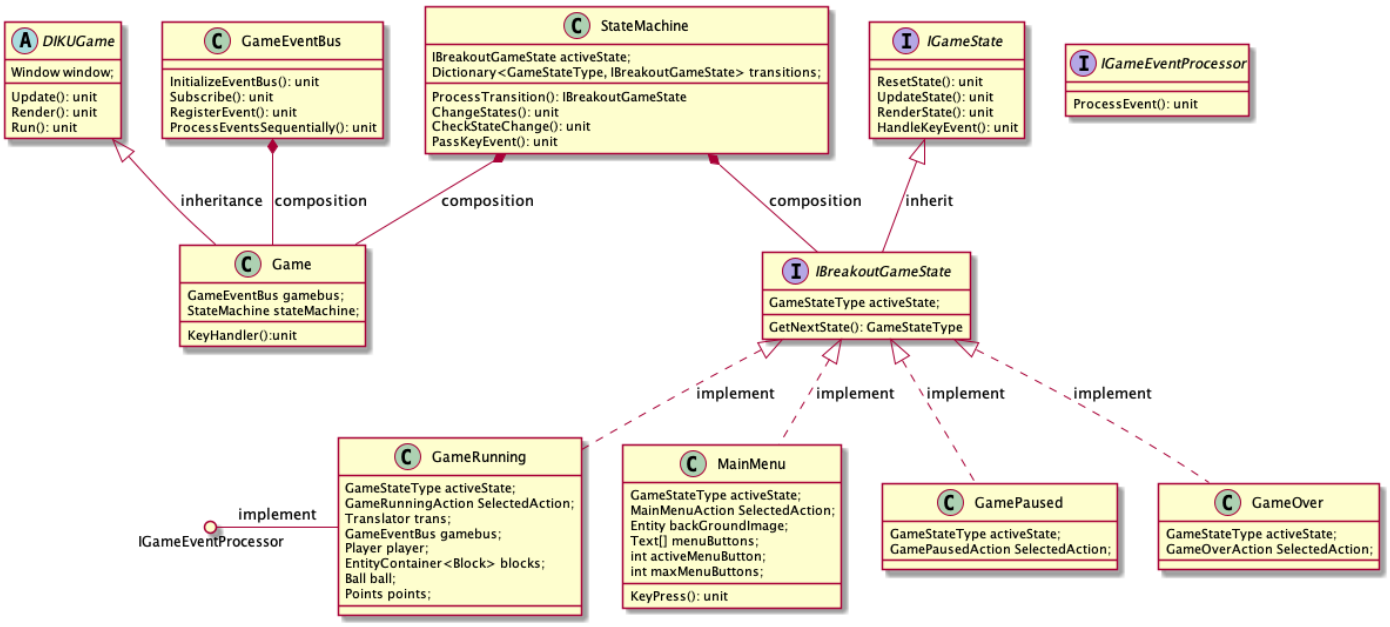

Figure 1: The UML of Game components

4.3 Power-Ups

In the design of power-ups, one might let the power-ups be types, which can then be part of a block's identity. In this way, the type of a given block will determine what effect the player gets when he hits it with the ball. There should also be Power-Up entities that are created and falls down the screen as the powerup blocks are destroyed. These need to store their specific Upgrade Type, change their image accordingly, and grant the correct effect when they collide with the Player Paddle.

4.4 Ball Class

The `Ball` class is responsible for rendering the ball to the screen, for making it moveable, and making its direction "updatable", so that it can change direction, e.g. when it hits a wall. The specifics of how the ball moves is taken care of elsewhere.

In the design of collision between the ball and blocks, we check if there is a collision between block and ball, and depending on where the collision happens, the ball's direction is changed and a new Event "HIT-BLOCK" is registered.

Finally, it is essential to mention that some of `Ball`'s responsibility has been handed over to `GameRunning`, where the whole collision-part of the code is located.

4.5 Player Class

R.1 (The player must start in the horizontal center of the screen) and R.6 (The player must default to a rectangular shape) are addressed when a new player is instantiated. The former's specification meets R.1 and the latter's specification meets R.6 (cf. Player.cs).
R.2 (The player should be, upon pressing left or right / a or d, capable of moving horizontally) is met by the `SetMoveLeft` and `SetMoveRight` specifications in `Player`.
R.3 (The player should not be able to exit either side of the screen and movement should be prohibited if it results in the player moving outside of the screen) and R.4 (The player shape must not 'stutter' when attempting to move outside the screen) state that `Player` should not be able to exit either side of the screen and the player should not "stutter" when attempting to move outside the screen. Those two requirements have been met through determining the position.X interval or simply the bounds, within which the player is permitted to move (cf. the specification in the code snippet below).
Besides, `Move` also addresses R.7 (The player must exist in the bottom half of the screen) and secures that the player never moves up on the screen.
Finally, R.5 (The player must be a DIKUArcade Entity) is addressed in the following specification, according to which a `Player` *inherits* from the `Entity` class — ergo, it is a `DIKUArcade` Entity.

4.6 Player Lives

In designing the PlayerLives, the class is given a number of methods : RenderLives(), ChangePlayerLives() and ProcessEvent(). Thus the PlayerLives class is responsible for rendering the player's number of lives to the screen, for making it possible to subtract or add to the player's lives and for making sure that the player's lives are indeed changed, when certain events occur.

4.7 Game States

Th design of our `StateMachine` adheres to the Closed/Open SOLID Principle, so that the class at hand is open for extension (i.e. adding new types of game states), but is closed for modification (see a detailed description in Implementation section 5.6). As far as requirements are concerned:

R.1 (Only one state can ever be active at a time) is met in that `StateMachine` ensures that the active state is always set to the current state:
R.2 (States should, at some level of inheritance, use the IGameState interface) is also met in that all states implement the `IBreakoutGameState` interface, which, in turn, inherit `IGameState`.
R.3 (You must have a state for; before the game starts, when the game is running and when the game is paused) is met as well in that the game includes such game states as `MainMenu`, `GameRunning`,

In specific, the `GameOver` state is designed so that the it is reached in two cases: 1) `PlayerLives` is below zero; or 2) the time is up. When `GameOver` is

active, the final score, the `GameOver` title and the two option buttons: `MainMenu` and `Exit Game` are displayed on the screen.

4.8 Level Management

In loading the level-files and extracting their information, the class `Translator` has 4 methods - one responsible for extracting the level map, one for extracting the meta data and one for extracting the legend data. This information is stored as raw strings in arrays. Through a number of loops, the methods go through each element of each relevant line in the level-files until the desired information is found. This information is compared to the information stored in the two arrays containing information about the level Legend, and the level metadata. This comparison allows the function to create block entities with the proper image files from the `Assets` directory.

A small subfunction is also added that reads the level data and returns the amount of indestructible blocks in the given level.

4.9 Score

The `Points` class is responsible for rendering the points to the screen when the game is running, making sure that points can be incremented, and making sure that this happens whenever the `GameEvent.Message` "BLOCK-DESTROYED" takes place. When a standard block, or a upgrade block is destroyed, the player receives 1 point, and when a hardened block is destroyed, the player receives 2 points.

5 Implementation

Implementing the required functionalities of the `Breakout` game, this assignment has taken its "Responsibilities", "Associations" and "Attributes" tables as guidelines. These tables can be seen in Appendix 4.

5.1 Block Functions

Using the Level Loading functionality, the blocks are created by substituting each "block" in the .txt files with its relevant .png image. The blocks are created using the "new" keyword, the `Entity` constructor as well as the .png files obtained through the legend part of the .txt file.

The block entity itself is created using shape, image and type. This provides the block with a a standard image, an image for when it has been damaged, but not yet destroyed as well as a type - which might indicate e.g. that the block is a power-up. Apart from this, the block class yields methods that render the blocks to the screen, that delete the blocks or grant them power-ups effects, that update the blocks' healthpoints, and that create power-up effects.

```
public Block(DynamicShape shape, IBaseImage image1, IBaseImage damagedImage1, BlockType typeInput)
        : base(shape, image1) {
    entity = new Entity(shape, image1);
    hitpoints = 10;
    BreakoutBus.GetBus().Subscribe(GameEventType.ControlEvent, this);
    type = typeInput;
    image = image1;
    damagedImage = damagedImage1;
    if (typeInput == BlockType.PowerUp) {
        powerUp = rollPowerUp();
        generatePowerUpIcon(powerUp);
...
```

5.2 Power-Ups

An enum is created for containing the list of power-up types. A separate class is created for registering the power-ups as events. This makes it possible for other parts of the game to react to the power-ups in appropriate ways. Finally, the `Block` class is changed, so that the block type is part of the constructor.

```
public Block(DynamicShape shape, IBaseImage image1, IBaseImage damagedImage1,
                                BlockType typeInput) : base(shape, image1) {
    entity = new Entity(shape, image1);
    hitpoints = 10;
    BreakoutBus.GetBus().Subscribe(GameEventType.ControlEvent, this);
    type = typeInput;
    image = image1;
    damagedImage = damagedImage1;
    if (typeInput == BlockType.PowerUp) {
        powerUp = rollPowerUp();
        generatePowerUpIcon(powerUp);
...
```

5.3 Ball

In itself, the implementation of the ball - as an entity - is fairly simple. A `Ball` class is created - inheriting from `Entity` - and via its constructor an entity is created, and default values are set for the properties `playing` and `direction`. Then the methods, `Move()`, `Render()` and `Update()` are created. The `Move()` method determines the ball's x- and y-coordinates as the sum of the ball's previous position and the cos and sin values of the ball's direction in radians. The `Render()` method renders the ball to the screen, and the `Update()` method updates the ball's position, i.e. movement, and its direction continuously.

The code which checks for collisions between the ball and the walls, blocks and player is somewhat more complicated. This happens in the GameRunning class. Here, methods exist for the ball's collision with the player-paddle,

with the blocks, and with the walls. Each of these methods ensures that the ball gains a new direction when it collides with elements of the game. As an example, see the code snippet from the paddle-collision method :

```
1  private void CollisionPaddle(bool redirect, Ball inputBall) {
2      if (CollisionDetection.Aabb(inputBall.Shape.AsDynamicShape(),
3          player.Shape).Collision == true) {
4
5          switch (inputBall.direction) {
6              case >270:
7                  inputBall.direction = (360-inputBall.direction) + (int) Math.Round((
8                  (inputBall.Shape.Position.X-(player.Shape.Position.X+0.08f)*(1.0f))*400.0f)*(-1.0f));
9
10                 if (inputBall.direction < 5)
11                     {inputBall.direction = 5;}
12                 break;
13 ...
```

This specific implementation either amplifies the angle following collision with the paddle, or reduces it, based on where the paddle is hit. This value is gradually increasing, and thus if the paddle is hit directly in the center, the resulting angle following collison will be "normal". Otherwise, if the ball approaches the paddle from the left side, and hits the far left of the paddle, the angle will be "increased" to the left, and if it hits the right side, it will be reduced. As an extra precaution, the final if statement ensures that this angle can never be amplified below 5 degrees, as this could either make the ball fly through the paddle, or hit the wall with such a low angle that it can bounce between the left and the right wall almost infinitely.

5.4 Player

In our design, `Player` is implemented in `GameRunning`) with `DynamicShape` that takes a `Position` vector and an `Extent`. In creating the `Player` entity for the `Breakout` game, the main task was to render the player and to make it moveable. In order to render the player, the `Entity` class' `Render()` method was used. In order to make the `Player` moveable, two move methods - one for moving left and one for moving right - as well as an `UpdateDirection()` method were created. These methods then work in collaboration with the `BreakoutBus` in a way similar to that described with respect to the `Block` class.

5.5 PlayerLives

The `PlayerLives` is implemented by creating a class for the player lives : this has a method for rendering the five hearts on the screen, displaying the amount of lives, and it has a method for changing the number of player lives. The visual effect of the lives change as an internal integer value representing lives change. This relationship was hardcoded, stating which images should be used for the cases 1 ... 10+ lives. Finally, the class yields a method which uses the

`ChangePlayerLives()` method to reduce the number of player lives when all balls have gone off the screen, or to increase the number of player lives, when the player is granted an additional life.

5.6 Game States

Our `StateMachine` has been implemented so that the class is closed to modification, but open to extension. We have therefore refactored `SwitchState()` that used to switch directly between the states in our `Galaga` implementation. In the present implementation the class `StateMachine` holds a property `transitions` of the type `Dictionary` of `GameStateTypes` and the instances of those types:

```
public struct StateAlignments {
    public static Dictionary<GameStateType, IBreakoutGameState> Transitions { get; }
        = new Dictionary<GameStateType, IBreakoutGameState> {
            {GameStateType.MainMenu, new MainMenu()},
            {GameStateType.GameRunning, new GameRunning()},
            {GameStateType.GamePaused, new GamePaused()},
            {GameStateType.GameOver, new GameOver()}
        };

}
```

When `StateMachine` receives a message from any game state that it desires to change a state (that is done by constantly updating the active state and setting it to `activesState.GetNextState()`,

```
public void CheckStateChange() {
    var stateCheck = activeState.GetNextState();
    ChangeStates(stateCheck);
}
```

`StateMachine` then changes the state by setting the active state to the `IBreakoutGameState` by using `ProcessTransition()` that returns the game state (value) that corresponds to the desired `GameStateType` (key)

```
public IBreakoutGameState ProcessTransition(GameStateType key) {
    try {
        return transitions[key];
    } catch (ArgumentException) {
        throw new Exception("Illegal state transition.");
    }
}

public void ChangeStates(GameStateType state) {
```

```
10      activeState = ProcessTransition(state);
11  }
```

In other words, in our implementation, all game states are responsible themselves for changing. `StateMachine's` task is just to ask them: "Do you want to change?" If yes, then it changes the current state by returning the `IBreakoutGameState` that corresponds to the `GameStateType`, which the current state wants to be changed into. Consider an example of the possible states the `MainMenu` might desire to change into:

```
1   public GameStateType GetNextState() {
2
3       if (SelectedAction == MainMenuAction.StartGame) return GameStateType.GameRunning;
4       if (SelectedAction == MainMenuAction.ExitGame) return GameStateType.GameOver;
5
6       return activeState;
7   }
```

5.7 Level Management

One of the main challenges of the Level Loading was to figure out, how one could load the file as a whole but read its parts separately. This problem was solved by using the `System.IO.File.ReadAllLines` to read the text file and converting this into a list of strings. Then certain key words or characters of the text file - such as "Map", "/Map", ":", "Legend" - were used to dictate where to slice these lines of raw text. These sliced strings were added to new string arrays formatted properly, so that the legend symbol always predates the image filename by 1 index in the array.

```
1   public string[] getLegend(string filename) {
2     string[] levelStr =
3       System.IO.File.ReadAllLines(Path.Combine("Assets", "Levels", filename));
4
5     string[] getLegend(string filename) {
6       var path = FileIO.GetProjectPath();
7
8       string[] levelStr =
9           System.IO.File.ReadAllLines(Path.Combine(@path, "Assets", "Levels", filename));
10
11      string[] legendStr = { };
12
13      for (var i = 0; i < levelStr.Length - 1; i++) {
14
15          if (levelStr[i] == "Legend:") {
16
17            for (var j = i + 1; j < levelStr.Length - 1; j++) {
18
```

```
19              if (levelStr[j][1].ToString() == ")"
20              && (levelStr[j].IndexOf(".png") != -1
21              || levelStr[j].IndexOf(".jpeg")   != -1)) {
22
23                legendStr = legendStr.Append(levelStr[j].Substring(0, 1)).ToArray();
24                legendStr = legendStr.Append(levelStr[j].Substring(3, levelStr[j].Length - 3)).ToArray();
25  ...
```

5.8 Score

In implementing the score system, a `Points` class is created, together with a method for adding points, for rendering points and for processing the event of a block being destroyed. In the latter method, a conditional statement ensures that different rewards are given for destroying a standard block and for destroying a hardened block.

6 User's Guide

You play `Breakout` by navigating to the `Breakout` directory and by typing "dotnet run" into the terminal. This opens the main menu where the player can use the up and down arrow keys to choose between "New Game" and "Quit". If "Quit" is chosen, the window will shut down. If "New Game" is chosen, the game will begin. From here, the player can move the player paddle using the keyboard's left and right arrow keys. The ball is set in motion by pressing on the "space" key. It is the player's mission to move the paddle in such a way that the ball hits and destroys the blocks in the game. If the player destroys a Block with a PowerUp image on it, a powerup will appear and fall in the direction of the player. If these are caught, the player is granted the specific powerup. If the player receives the rocket PowerUp, it can be launched by pressing the "space" key while the ball is in play. If all breakable blocks on the screen is destroyed, the player will advance to the next level. If the player does not succeed, and loses all their lives, or runs out of time, the game is over. At GameOver, the player will be able to enter their 3-character name in the event that they make the top 10 high score; then the player will see their name in a list of players and their scores. The player is here also presented with the choice of going back to the main menu or exiting the game.

7 Evaluation

7.1 Overview of tests

The directory `BreakoutTests/EntityTests` contained the following tests:

- `CollisionTest`
- `TestBall`
- `TestBlock`

- `TestBlockTypeTransformer`
- `TestPlayer`

Furthermore, this directory also contained a sub-directory `PowerUpTests`:

- `PowerUpTests`
- `TestPowerUpDistributor`
- `TestRocket`

The directory `BreakoutTests/LevelLoadingTests` contained the following tests:

- `TestCreateMap`
- `TestGetLegend`
- `TestGetMeta`

The directory `BreakoutTests/OtherTests` contained the following tests:

- `TestGame`
- `TestPlayerLives`
- `TestPoints`
- `TestTimer`

The directory `BreakoutTests/StateTests` contained the following tests:

- `TestGameOver`
- `TestGamePaused`
- `TestGameRunning`
- `TestGameStates`
- `TestMainMenu`

7.2 General reflections on testing the project

The aim of the testing endeavours of this assignment, was to do White Box Testing with c0 coverage - i.e. a coverage that ensures that all control flow **nodes** are executed at least once. However, due to the time constrains of the assignment, only the essential functions of each class are tested - and these functions are not tested as thoroughly as they should have been for c0 coverage. The reason for this is simply poor time management and poor communication on the group's part.

Overall, two central issues were encountered in testing the `Breakout` game: one related to the `Singleton` principle, and one related to the security of the code. First of all, in maintaining the `Singleton` principle, e.g. with respect to the `BreakoutBus`, it proved problematic that all tests had to access a single `BreakoutBus`. Since so many of the `Breakout` classes rely on the `BreakoutBus`, the test classes become interdependent on each other - which is far from ideal. Secondly, in trying to keep methods and properties private, testing proved immensely difficult. It seems impossible to achieve code security and testability simultaneously. In order to be able to perform the required tests, it has therefore been decided open up a number of properties and methods - either via the `public` keyword or via a public `get` function.

7.3 What do the test results tell us ?

`CollisionTest` checks that a collision is only registered when objects actually collide, and not e.g. when they are merely placed within the same game window.

`BallTest` checks that when a ball is initiated, its default state is to not be "in play". The tests further show us that the ball's `Move()`, `Update` and `SetPosition()` functions actually impact the ball's movement as expected. The `insideScreenTest()` checks that a ball is deleted when it leaves the screen at the bottom.

`TestBlock` shows us if the different block types lose the appropriate amount of hitpoints when they are hit by the ball.

In general, unfortunately, the `Entity` tests did not work. The problem seems to be the that the entities do not move when the `Move()` function is called. However, since this is not an issue when playing the game itself, the failing tests might perhaps rather be explained by the fact that the `registerEvent()` method does not work appropriately within the tests directory.

`TestBlockTypeTransformer` tests if the `TransformStringToBlock` method does indeed transform a string into a block type.

`TestCreateMap` tests that our code can handle different kinds of maps. `TestGetLegend` tests that our Translator method can handle varying kinds of .txt files and that it reads their legend contents in the correct order. `TestGetMeta` does the same but with respect to meta data in the .txt files.

In `playerLivesTest` it is tested whether the player has the correct amount of lives at the beginning of a game, and it is tested if a life is actually being subtracted when the ball's y-position goes below 0.

In `pointsTest` it is tested if points are 0 when no blocks have been hit by the ball, and it is tested if points are incremented when a block is hit by the ball.

`TestGameOver` checks that the `ChangeStates` method can change the `GameStateType` from `GameOver` to `MainMenu`. `TestGamePaused` checks that the `ChangeStates` method can change the `GameStateType` from `GamePaused` to `GameRunning`. `TestGameRunning` checks that the `ChangeStates` method can change the `GameStateType` from `GameRunning` to `GamePaused`. `TestMainMenu` checks that the `ChangeStates` method can change the `GameStateType` from `MeinMenu` to `GameRunning`.

Thus, overall, it can be said, that if all of our tests had worked, they would have verified that our code works as it should - that each class lives up to its responsibility. However, since not all tests ended up working, only a partial verification has been achieved.

7.4 Code Quality

In general, it can be said that although this project lives up to the vast majority of the requirements declared in the assignment texts, it has not been possible to *verify* this completely - due to the team running out of time.

Furthermore, although the code violates the SOLID principles in multiple ways - which will be discussed in the exam - the principles are not being com-

https://github.com/bpl714/DIKUGames/tree/BreakoutExam

pletely ignored. In rendering a block and implementing its functionality, for examples, the single-responsibility principle is upheld. The open-closed principle is also partly upheld in the `Entity` class which provides a framework for creating a multitude of varying entities. Thus one can simply extend the code with a new class with a new entity without changing the original code and its functionality at all. Regarding the Liskov Substitution Principle, the `Entity` and the `Block` classes illustrate this. The `Entity` class' constructor - taking only a shape and an image as parameters - is implemented by the `Block` class in a way which makes the `Block` behave perfectly like an `Entity`; thus adhering to the Liskov Substitution Principle. Thus all in all, the code of this project is not impossible to maintain.

8 Conclusion

The project at the heart of this report has implemented the game `Breakout`. It has done so by loading .txt files containing the levels of the games, by converting these text-based level descriptions to .png files, by creating standard as well as special blocks, by creating a player entity, by creating a ball entity, by creating a state machine, by creating a points system, by creating player lives and by creating power-up effects. Although these features have all been implemented fairly successfully, they have not been tested thoroughly - this is due to the fact that the team behind the code ran out of time and therefore decided to focus only on testing the most essential parts of the program. Furthermore, several design principles are violated throughout the code - examples of this will be presented at the oral exam.

Appendix

Responsibilities Table

Responsibility Description	Type	Concept Name
create block and its functionalities	D	block-creator
convert information from level-files into concrete blocks	C	convertor
create ball and its basic functionality	D	ball-creator
create player and make it moveable	D	player-creator
handle collision between ball and blocks, player, walls	D/C	collision-handler
initialise functioning program with a player, blocks and a ball	D/C	program-initiator
handle the game's timer	D/C	time handler
handle the gamestates of running, paused, over and main menu	C	gamestate-handler
handle the awarding of points to the player	D/C	point-tracker
handle the player's lives	D/C	playerLives-tracker

Associations Table

Concept Pair	Association Description	Association Name
block-creator, convertor	convertor links level-file with block entity	translates
ball-, player- and block-creators, collision-handler	the collision-handler handles the interaction between the ball and the walls, blocks and player	handles collisions
program-initiator, collision-handler	program-initiator initiates game and thereby the interaction between objects	catalyses game and entity interaction
gamestate-handler, all	when the game is running all concepts are activated	changes game state
point-tracker, GameRunning	points are awarded at collision, and collision is in GameRunning	tracks points
playerLives-tracker, GameRunning	Lives are awarded at game start, and are updated as the game is running	tracks PlayerLives

Attributes Table

Concept	Attributes	Attribute Description
block-creator	elements constituting the block	textttentity, image, damagedImage, type
convertor	collection of blocks	blocks produced according to level-files
ball-creator	elements constituting the ball and its behaviour	entity, playing, direction
player-creator	entity containing elements of player and its behaviour	moveLeft, moveRight, MOVEMENTSPEED, entity
collision-handler	elements required for game to be playable	trans, gamebus, blocks, player, ball, firstRun, cooldown
gamestate-handler	the different gamestates	GameOver, GamePaused, GameRunning, MainMenu
points-tracker	the Player's points rendered to the screen	points rendered as text in the game
playerLives-tracker	the entities and images constituting the lives	hearts showing how many lives the player has left