



Final Exam

Software Development 2022 Department of Computer Science University of Copenhagen

Aske Lundsgaard <pwg445@alumni.ku.dk>
Henrik Christensen <djl587@alumni.ku.dk>
Mads Vestergaard <ndh409@alumni.ku.dk>

Repository: <https://github.com/henrikdchristensen/DIKUGames>

Wednesday, Jun 8, 15:00

Table of Contents

1	Introduction	1
2	Background	1
3	Analysis	2
4	Design	4
5	Implementation	12
6	Evaluation	13
7	Conclusion	15
	Appendices	18

1 Introduction

This report is about the implementation of the block-breaker game *Breakout*, inspired from the old Atari 2600 game of the same name. The solution is written in the Object-Oriented Programming (OOP) language C# (with .NET Core 6.0). The repository can be found at <https://github.com/henrikdchristensen/DIKUGames>. To clone the project run the following commands in a terminal:

```
git clone https://github.com/henrikdchristensen/DIKUGames.git
git submodule update -init -recursive
```

Checkout to latest tag by:

- List all tags: `git tag -l`
- Checkout latest *Breakout*-tag: `checkout tags/BreakoutExam_v(x).(x)`

In addition, an user guide of how to run the game and/or execute the tests, can be seen in Appendix A.

The process of developing the game has been an iterative process, and where the course's many different topics have contributed to the result.

2 Background

For a better understanding in the further reading of this report, some key software design principles used during the development needs to be addressed.

2.1 State Design Pattern

The State pattern, a behavioral pattern, enables the object to change behavior when its internal state changes during run-time [4]. Each state should be defined in a separate class. This pattern is useful for games which often have multiple states, e.g. running and paused.

2.2 Singleton Design Pattern

The Singleton pattern, a creational pattern, is to ensure that only one instance can exist of the same class [4]. This pattern is for example suitable for managing the states of the game. This also ensures that changes to a state is preserved, when another state is temporarily activated.

2.3 Observer Design Pattern

The Observer pattern can be used to prevent objects from constantly checking other object's states [4]. This is a common problem in game-development, where many events occur and different objects need to react upon them. This is avoided by the use of the Observer pattern, which maintains a list of subscribers, and will notify all objects, when events occur.

2.4 Mediator Design Pattern

The Mediator pattern, a behavioral pattern, that lets you reduce chaotic dependencies between objects. This pattern forces the objects to only communicate via an Mediator [4]. This is for example suitable in a game, where many objects need to communicate with each other, to reduce coupling and segregate responsibilities.

3 Analysis

In order to be able to make the right design choices, it is important to understand the actual problem first. The main goal for us is to make a new version of the old Atari game *Breakout* [1], which is described below.

3.1 Description of the Atari game *Breakout*

The old Atari game *Breakout*, see Figure 1, contains a number of blocks in the upper part of the screen, where the goal of the player is to destroy all blocks by bouncing a ball on a paddle that the player controls. The ball can also bounce on the walls, which are at the top and on the left and right sides of the screen. If the ball passes the paddle at the bottom of the screen, the player loses a life. The player has 3 lives to complete two levels. Each type of block gives a certain number of points.



Figure 1: The Atari game *Breakout* [1].

Two special events can occur:

- When the last red row at the top is destroyed and the ball hits the top wall, the ball size is reduced.
- The speed of the ball increases at specific intervals.

3.2 Our goal

Our goal is to make a new gaming experience of the Atari game *Breakout* by introducing other special events. For example, the player also loses if some specific level time expires or introducing so called powerups which emerges and descends by destroying a secret block. These powerups can be picked up by the player who, e.g. get extra life, double the size of the paddle, etc.

From the two previous Sections 3.1 and 3.2, it appears that the game contains many contributing factors, each of which contribute to the overall new game experience. These contributing factors are described in the following sections.

3.3 Levels

To define where the blocks are at a certain level, a definition of where to place them are needed. Furthermore, to be able to keep track of how long the player have to complete a level, it seems obvious to include this information here. Also powerups are mentioned, which are secretly located in a block in the game. Since the blocks is located in the level, it makes sense to define the powerups here as well. The information kept here is thus divided into two parts: A map of the game and some metadata such as time.

As mentioned, the game should contain different levels. These levels should vary in difficulty in terms of different types of blocks as well as where they are placed in the game. It should therefore be possible to load different levels and to put into context next time. In order not to change the code after finishing

the implementation, it would be smart to load a level externally. One simple solution would be to have the files located locally on the hard drive, where new levels can be placed. The disadvantages of having the dynamic loading of level files is that they still have to follow a certain template, as they have to be match the fixed game size and objects should be placed in a certain location on the map.

3.4 Game states

In order to be able to e.g. pause the game, the game should be able to change its state. When the game starts, it would be convenient for the player to start in a menu where the game afterwards can be started. It would also be suitable for the player, to be able to pause the game and resume later. Furthermore, once the game is over, the player should be able to see the overall score, as the player does not necessarily keep an eye on this during the game. A game can only be in one state at a time, so there must be mechanism to manage state changes as well.

3.5 Player

In order to play the game, a player object is needed. In this game the player should only be able to move left and right at the bottom of the game. To control the player the user of the game, should be able to interact with it. Thus the game must be able to read inputs. Since most people have a keyboard for their computer, it is obvious to make use of this. When the ball passes the paddle at the bottom of the screen, the player should lose a life, and therefore needs a life property.

3.6 Ball

Another central part of the game is the ball. The ball should be launched at the start of the game. To ensure that the player is prepared it would be appropriate to launch the ball in a upwards direction. To keep it simple, the ball should move with an constant speed. The ball should be able to bounce off the player's paddle as well as off the walls around the edge of the game window. If the ball hits a block, then it should also bounce away. When the ball hits a wall or a block, it should by the laws of nature follow the rules of entry angle = exit angle. To make the game more engaging, friction should be introduced such that when the ball is hit by the paddle in motion, then the direction of the ball should be affected by the player's acceleration. In addition to the above, adding several balls to the game could make it more challenging to play.

3.7 Block

The square blocks in the game are the ones that the player aims to destroy in order to complete a level. As mentioned in Section 3.3, their location are specified in the level file. When the player destroys certain types of blocks, the player gets rewarded with points equal to the reward value of the block. The blocks can be of different types in terms of how much it takes for them to be destroyed.

3.8 Wall

To keep the ball within the the game's frame, walls are needed around the edge of the game except from the bottom of the screen where the ball can exit and the player loses a life. The ball should be able to bounce off these walls. The walls must be present throughout the lifetime of the game.

3.9 Powerups

To make the game a bit more immersive than the original game, a series of powerups could be added. These powerups could be spawned when a specific block is destroyed. The different powerup types, should be clear and consistent, so the player knows what it does and which to pick. Different types of powerups should be able to exist in the game for various amount of time. They must therefore be able to be activated and deactivated as needed.

In the game we will try to implement the following powerups:

- Extra life: The player gets an additional life.
- Player speed: The player gain extra speed.
- Double speed: Doubles the speed of all balls.
- Double size: Doubles the size of all balls.
- Wide: Increases the size of the player.
- Split: Each ball are split into three new balls.
- Hard ball: The ball breaks everything it hits.

3.10 Collision detection

As mentioned in Sections 3.6, 3.7 and 3.9, collision is a central part of the game. To decide what kind of entities that can collide, how they will collide and when it happens, a collision detector is needed. Since multiple entities in the game can collide, a generic implementation of a collision handler would be preferred.

An example of different kinds of collisions is between a ball and a block and a ball and a player. If the ball collides with a block, then its direction should change according to the law of physics: $entry-angle = exit-angle$. However, if the ball collides with a player then it should be affected by the acceleration of the player. Therefore, the ball needs to know which entity it has collided with as well as data of e.g. the player's acceleration.

4 Design

From previous section, it appears that there are several concepts in the game. These are treated as being classes from this point. The key classes is described after presenting the various responsibilities, associations and attributes. Subsequently, the design goals we strive to follow is discussed and at the end of this section an UML class diagram of the entire program is presented.

4.1 Responsibilities, associations and attributes

From the analysis, we have identified different responsibilities that must be imposed the individual classes as well as the associations between them. In addition, the classes different attributes, e.g. player's health. These three are summarized in the tables below:

Responsibility description	Concept name
Loads the level file	LevelLoader
Create level by adding entities according to level file.	Level
Manage state switching between: MainMenu, GameRunning, GamePaused, GameOver.	StateMachine
State where a new game can be started or quitting.	MainMenu
State where the game is in action.	GameRunning
State where the game is paused.	GamePaused
State where the game is over and score should is shown.	GameOver
Manage player's movement, amount of life and render player.	Player
Manage ball's direction according to type of collision and render ball.	Ball
Manage block's health if collision with a ball occurs and render block.	Block
Notify ball that it has collided with a wall.	Wall
Activate/deactivate power-up and render the power-up.	Power-up
Collision detection of player, ball, blocks, walls and power-ups.	CollisionHandler

Table 1: Responsibility table

Concept pair	Association description	Association name
LevelLoader ↔ Level	LevelLoader loads the level file and creates a new Level	Parses
StateMachine ↔ MainMenu	StateMachine uses MainMenu in active state	Receives
StateMachine ↔ GameRunning	StateMachine uses GameRunning in active state	Receives
StateMachine ↔ GamePaused	StateMachine uses GamePaused in active state	Receives
StateMachine ↔ GameOver	StateMachine uses GameOver in active state	Receives
GameRunning ↔ Player	GameRunning contains Player and call some methods	Contains
GameRunning ↔ CollisionHandler	GameRunning contains CollisionHandler and call the Update method to check for new collisions.	Contains
CollisionHandler ↔ Player	CollisionHandler contains Player to detect collisions between Player and other objects.	Receives
CollisionHandler ↔ Ball	CollisionHandler contains Ball to detect collisions between Ball and other objects.	Receives
CollisionHandler ↔ Block	CollisionHandler contains Block to detect collisions between Block and other objects.	Receives
CollisionHandler ↔ Wall	CollisionHandler contains Wall to detect collisions between Wall and other objects.	Receives
CollisionHandler ↔ Power-up	CollisionHandler contains Power-up to detect collisions between Power-up and other objects.	Receives

Table 2: Association table

Concept	Attribute	Attribute description
LevelLoader		
Level	Map	List of blocks
	Meta	List of meta data (e.g. time, name of level, etc.)
StateMachine	ActiveState	Currently active state
MainMenu		
GameRunning	Player	Player of the game
	CollisionHandler	Continuously checking for collisions
	CurrentLevel	Currently active level
GamePaused		
GameOver		
Player	Life	Amount of life left
	Position	Position in the game window
Ball	Position	Position in the game window
	Direction	Current direction of the ball
Block	Position	Position in the game window
	Health	Amount of health left
	PointReward	Rewarding points when destroyed
Wall	Position	Position in the game window
Power-up	Duration	How long the powerup is activated
CollisionHandler		

Table 3: Attribute table

4.2 Design goals

In addition to solving the main problem of creating a new version of Breakout, it is important for us to create a good design which follows some core software design principles:

- The design (and code) should strive to follow SOLID, GRASP, DRY.
- Should make use of design patterns, where it benefits the solution and comprehensibility.
- The code should be consistent, follow DIKU-DK coding convention and have good comments where it benefits understanding.
- The overall program structure, should be easily comprehensible.
- The framework DIKUArcade should be used as much as possible where it fits into the design, as it will accelerate the development [2].
- Good tests of the method and the whole solution, should be made, based on the requirements and specifications stated in Section 3, which is summarized in Appendices B and C.

4.3 Game entities

All game entities (ball, block, player etc.), should have a similar behaviour and state. For example, all should be able to move, and collide with each other. To define this, the abstract class called `GameObject` is created and inherits from DIKUArcade's `Entity`-class. The class makes it possible to store all `GameObjects` of a level in one list, and due to polymorphic operations each `GameObject` are responsible for defining how they should be drawn, get updated, and what to do when they "die". This is in line with the polymorphic GRASP principle. It also makes it easier to implement the collision handling - discussed further in Section 4.6.

4.4 Handling game events

Many different objects need to be notified when specific events occur. This is especially the case for the power up functionality. To avoid a high coupling and reduce chaos, the Mediator pattern could be used, as an intermediary between objects. However, this can over time evolve into a 'god' class, that contains a lot of game-logic to distribute events. Another alternative is to use the Observer pattern to notify subscribed objects, when a certain state change happens. Of course this can also evolve into a very complex Observer class, that needs to keep track of many object states.

A solution is to 'combine' elements from both patterns, where objects can subscribe to different event-types. In addition, every object should be able to register/publish events on the bus to notify all subscribed objects of that event-type. We use DIKUArcade's `GameEventBus` wrapped in a Singleton to create a global access point. This event-bus, makes it possible for subscribers to react on events, without modifying other classes Open-Closed Principle (OCP).

4.5 Level loading

From the analysis in 3.3, the level loading needs to implement and handle the following responsibility: (1) loading the level text files, (2) validating them, (3) and parsing the loaded text into meaningful data-structures.

Responsibility (3) could be facilitated by utilizing the Builder pattern. This could be done by creating a method, that loop through each line of text, and identifies what sort of level-data is being passed, and then sending it to a builder class. The builder class then builds each component of the level step-by-step. Thus, the level does not need a new level-constructor for each variation, which is compliant with OCP. This solution also decouples the level parsing and the level building from each other, which is in accordance to the Single-Responsibility Principle (SRP).

One problem with this solution is that the overall complexity of the code increases since the pattern requires the creation of multiple new classes and would be more time consuming to implement. For that reason it is decided not to follow this pattern and instead create the classes shown in the Figure below:

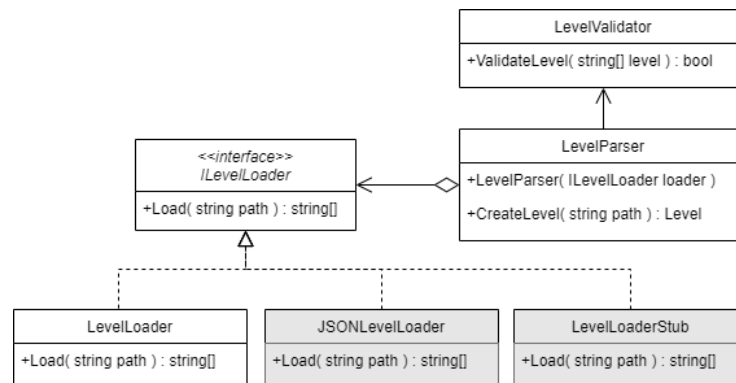


Figure 2: UML-diagram of Level loader classes, where *JSONLevelLoader* and *LevelLoaderStub* is not implemented, but for illustrative purposes

In the design, we have used the Strategy pattern to decouple the loading of levels from the level parsing. This is done by introducing the *ILevelLoader*-interface as an extension point for different *LevelLoaders*. In this way, a *LevelLoader* could be created for different file-formats - e.g. *JSON*-, *XML*-files etc. Thus, the use of the Strategy pattern follows the OCP. Furthermore, a *LevelLoaderStub* could also easily be created in order to better do unit testing and integration testing for the *LevelParser* class. In general, we have designed the classes in mind of the SRP. For instance, the *LevelParser*-class should not also be responsible for validating the level-file, but put in a separate class, *LevelValidator*.

4.6 Collision Handler

One way of handling and detecting collisions is to use the Observer pattern, where each game-object subscribes to a `CollisionHandler`, that maintains a list of subscribers (`GameObjects`). The `CollisionHandler` then checks in each game-loop if any of the subscribed entities has collided and publishes the information to the collided objects.

Another alternative is to handle all collisions inside the main game-loop. However this would add extra responsibility (lowers cohesion) and it needs to know all the different entities which would increase coupling.

One problem with the `CollisionHandler` is, that it does not know the types of the objects that has collided. It only knows that two `GameObjects` has collided due to Polymorphism. One way of solving this, is to use the Visitor pattern. By utilizing this pattern, the `CollisionHandler` will ask each object to send a message to the other object, to inform what kind of collision it is (*who it is visiting*). By utilizing the Visitor pattern, one can easily add new behaviour to an existing class, by implementing one of the collision-methods. In this way, the solution follows the OCP. However, this is not entirely true, since adding a new class, would probably imply a change in the *game-objects* (visitors) there are concerned about the new class.

4.7 Managing states

For managing states, the State design pattern can be used. This pattern makes it possible to choose different behaviour at runtime depending on the active state. This means, a class for each state should be created, where responsibilities are segregated to each class. This also adhere with the Liskov Substitution Principle (LSP), which makes change of behaviour possible.

An alternative to this method, is to create a single class to handle all different states. This way of managing the states would violate the SRP, since it is responsible for running each state, and switching between them. Therefore the first solution is chosen.

4.8 Powerups

Powerups alters the gameplay in some way and thus could introduce an extra overhead of complexity if not implemented carefully. A central question is, where this responsibility for managing the active powerups should be placed.

One way of implementing the powerups is having each of the `GameObjects` to manage the active powerups themselves. However, this approach would probably introduce a tight coupling between the `GameObjects` and the powerups.

Another way of approaching this problem is by making each powerup a singleton. Each powerup could then control whether it is active or not, refresh durations and communicate through the Eventbus. Logic for this could be inherited from the powerup super class.

However, introducing numerous singletons makes it difficult to test the individual powerups as the state of the objects would live on through all the unit tests. This might make it necessary to execute unit tests in a certain order, which is not desirable.

In our solution, in order to decouple the powerups from the `GameObject`, we used the Eventbus and implemented a singleton `PowerupContainer` to control the activation and deactivation of the powerups. By moving this logic into a single separate `PowerupContainer` we are gathering all the knowledge of how to handle powerups in one class, thus ensuring a High Cohesion and avoiding making each powerup responsible for handling this (SRP). However, this approach also implies a rather high chain of communication, see Figure 3.

We have decided to utilize the Eventbus and its timed-events, to handle the duration of the powerups. Initially, we thought that only one instance of the `PowerupContainer` should exist throughout the game. However, each level could have its own `PowerupContainer` thus encouraging all powerups to be disabled when reaching a new level. Omitting the use of a singleton would make the powerups much easier to test as the unit tests would not be affected by previous states. The sequence of actions and messages exchanged when a player collides with a powerup is illustrated in Figure 3 below:

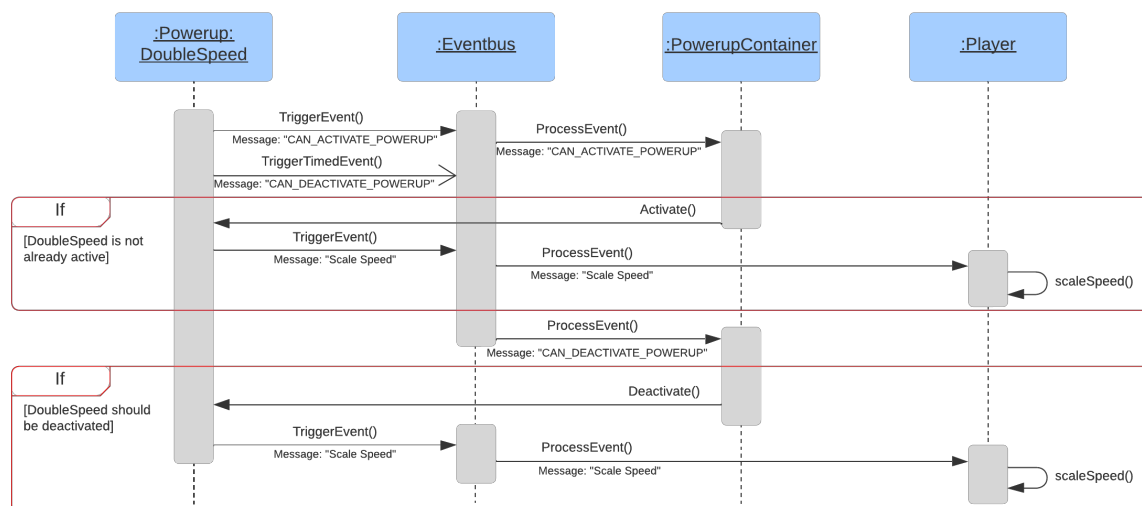


Figure 3: Sequence diagram of the actions performed when a Player collides with the DoubleSpeed power-up

4.9 Strengths and weaknesses

The workload is spread across many different classes which increases the overall compliance with the SRP. Implementing the `GameObject` to extend the behaviour of the `Entity` class introduces a rather deep level of inheritance which is a weakness. All `GameObjects` inherits the same methods and must implement the same interfaces even though they are rarely used together thus compromising the Interface Segregation Principle (ISP). In the overall design, we have strived to create a Low Coupling, which from the UML-diagram 4.11 looks decent. These points are elaborated in the section below and will be discussed further in the evaluation section 6.

4.10 SOLID

In the design we strived to comply with the SOLID principles as well as we could. This is done to ensure a high code quality, that is easy to maintain and extend in the future. If SOLID are followed, one change in the code base will be relatively isolated, such that it does not trigger errors in other parts of the code base.

4.10.1 SRP

Generally we comply with the SRP by separating responsibilities into different classes and thus increases cohesion. An example of this includes the *LevelContainer*. This class only contains methods strictly related to handling levels like `initializeLevels()` and `NextLevel()`.

4.10.2 OCP

To decouple the loading of levels from the level parsing we introduced the `ILevelLoader`-interface, as an extension point for different `LevelLoaders`. In this way, a `LevelLoader` could be created for different file-formats. Thus, the use of the strategy pattern follows the OCP. Also the `CollisionHandler` follows this principle as we can introduce new subscriber classes without having to change the publisher's (`CollisionHandler.cs`) code

4.10.3 LSP

Generally, the LSP is not followed much. The reason why, is that we do not make extensive use of inheritance and interfaces. However, in our `Block` design we violate this principle. Every type of special block in the game, e.g. `Unbreakable`, is inheriting from the class `Block` and overrides the method `Hit()`. However the post-condition in the two types of blocks' `Hit()`-methods does not match since `Unbreakable` should not loose a life when hit. Therefore the abstraction of having `Unbreakable` inherit from `Block` is not an ideal one. Instead we should create a new abstract class or interface that makes the right abstraction to extend from.

4.10.4 ISP

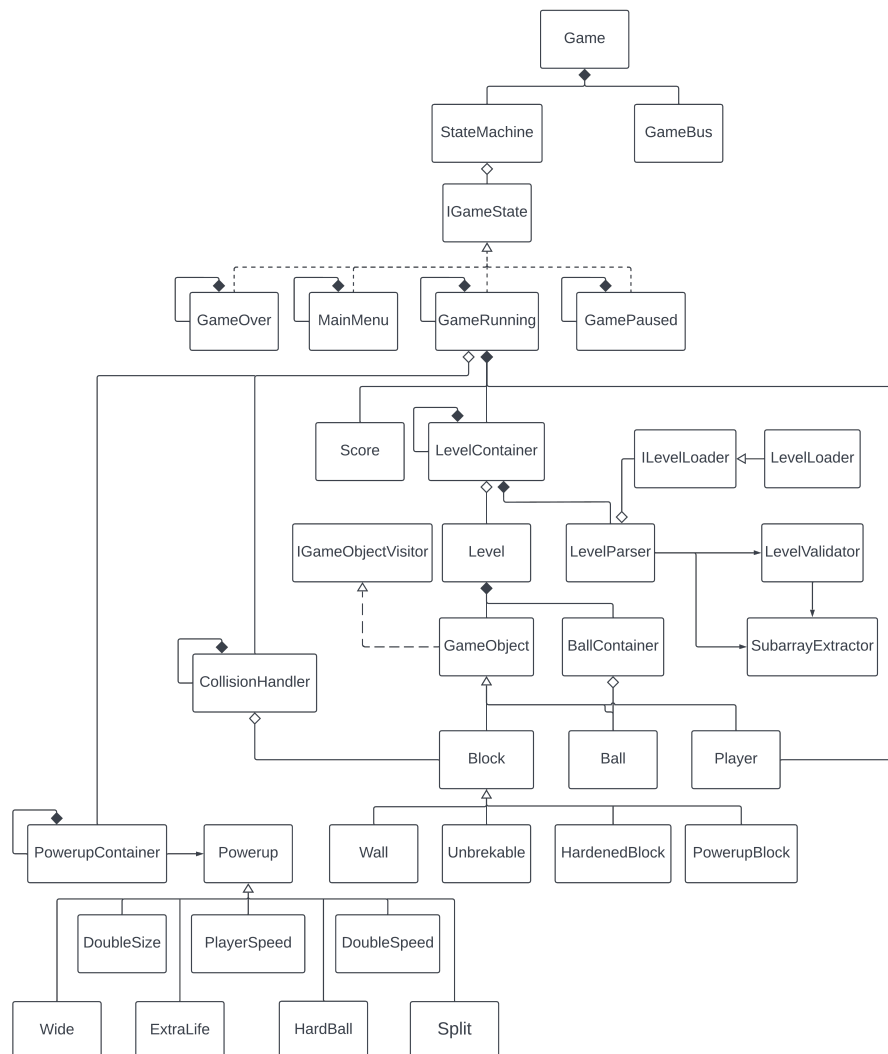
One example of compliance with this principle is the `LevelLoader` that implements the `ILevelLoader`-interface. `ILevelLoader` is a small and specific interface which allows clients to only use methods that are of actual interest to them. An example that is not fully compliant with this principle is the version of the Visitor pattern in our `CollisionHandler`. All `GameObject`'s implements the `IGameObjectVisitor` pattern, however, a block should not be able to collide with a `Player` and therefore should not implement this method. Due to the nature of the Visitor pattern, this is difficult to avoid, as each `accept()` method is depending on all the methods in `IGameObjectVisitor` to exist in order to call its methods on the different visitors.

4.10.5 Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. This helps to ensure Low Coupling between classes. An example of this principle is the `LevelParser` class, that only depends on the `ILevelLoader`-interface. Thus it relies on an abstraction and not a concrete implementation. This is also the case for the `CollisionHandler`, since it checks collisions between the abstract `GameObjects`, and does not care about the concrete classes (e.g. `Ball`, `Player` etc.).

4.11 UML class diagram

A simplified UML class diagram without methods for better overview is shown below:



5 Implementation

This section describes the key features of the implemented classes. It does not document all features or methods, but only those that are most central or difficult to comprehend.

5.1 LevelParser

The `LevelParser` has the method `CreateLevel(string path)`, which should return a `Level`-instance containing the data from the text-file with the given *path*. This method calls three helper methods, which has been created to delegate the `CreateLevel`-method's responsibility (SRP), and since some are called more than once (DRY). These methods are:

string[] extractData(string[] lines, string start, string end): the method searches for the two strings *start* and *end*, and returns all lines in between. E.g. the start- and end-keyword could be "Map:" and "Map/". The pre-condition of the method is, that both keywords exist, and appear in the correct order.

Dictionary<string, string> linesToDict(string[] lines, char delimiter): the method loops through the array of lines, and for each line, splits it up into two string separated by the delimiter. It then adds a key-value pair to a dictionary data-structure, where the key is the first string, and the value is the second string. E.g if you have method-call: `extractData(new string[]{"Name: LEVEL 2", ':'})` would return a dictionary with the pair: `{"Name", "LEVEL 2"}`. In short this method converts an array of lines into a dictionary data-structure.

char[,] LinesToChar2DArray(string[] lines): as parameter the method takes an array of lines, and converts the lines into a multidimensional char array. This is done by looping through every char in each line.

The method `CreateLevel` uses these three methods to convert a level file into representative data-structures: *map*: a 2D char array, *meta*: a dictionary, *legend*: a dictionary. These are used to instantiate a `Level`-object, which is returned by the method.

5.2 Collision detection

The collision detection is handled by the `CollisionHandler`. It has the field, `collidableList` (`List<GameObject>`), which contains every object that has subscribed to the handler. In each game-loop its `Update()`-method is called, where it first removes all objects marked for deletion. This is checked by calling the method `GameObject.IsDeleted()`. Subsequently all collision-combinations between two `GameObject`-objects are checked by using the `CollisionDetection.Aabb()` method. If two objects are colliding, then each need to react on the collision. This is done for each `GameObject` by calling the `GameObject.Accept(IGameobjectVisitor visitor)`, which then calls the correct visitor method - e.g. `visitor.BallCollision`. Then each `GameObject` (visitor) is responsible for defining how they should react on the collision, which complies to the Information Expert principle.

5.3 Ball movement

The balls movement from frame to frame, is defined by the *Shape.Direction*-vector. This vector should of course change direction, when colliding with other game objects. This is done by finding the normal-vector of the collision point, and reflecting the ball's direction-vector in this. For the ball not to get stuck on the same trajectory, the new direction is also dependent on the colliding objects direction-vector. This makes it possible for the player to affect the ball.

5.4 Powerups

Every Powerup inherits from the Powerup-class, and has the two abstract methods: `activate()` and `deactivate()`, in which they should implement. These defines what should happen, when they get activated/deactivated, which is controlled by the PowerupContainer. All powerups uses the GameBus to send event to the effected objects. For instance the PlayerSpeed-event sends a ScaleSpeed-event to the player. The Powerup extension point makes it easy to add new Powerups, and is in adherence to OCP.

6 Evaluation

To have a safe development environment with high *correctness* and *robustness* of the code, a test environment with automated unit-tests has been set up. This is done using the unit-testing framework NUnit (v.3.13.3). Here unit-tests can be performed repeatedly during development. The aim is to test against the specification stated in Appendix C.

Three well-known strategies for testing has been used, black-box testing, white-box testing and integration testing. The overall aim for these tests is to have high code coverage (ideally 100% C0 or C1 coverage).

All of our tests can be found in the test suite `BreakoutTests`. All tests is passed, see Appendix E. We strived to make our tests as dynamic as possible to make our results possible to reproduce with different inputs. Thus, in some test classes, we had to make a public getter for some properties in the classes. Furthermore, we also tried to only test one unit at a time during blackbox- and whitebox-testing. This sometimes required, the creation of a test-double to mimic the dependencies behavior. E.g. we created the `GameObjectSpy`, that inherits from `GameObject` to test if the correct methods are called from the `CollisionHandler`.

Black-box test Has been made for most non-trivial methods. They were made early in the development phase from the specifications, where both positive and negative tests were created (testing intended and unintended functionality), as we strived to do Test Driven Development.

White-box test Has been made for those classes that has methods with branches. In this way, the more complex units were tested, to ensure correctness. They can first be made after implementation of the method, since they test internal parts of the code, i.e. statements and branches. In our white-box tests we es-

pecially focused on the the following and more complex classes: `LevelParser`, `LevelValidator`, `CollisionHandler` and `Player`.

Integration test Integration tests has been made to test parts of the program, where complex interaction between units exist. Therefore, we primarily focused on testing the powerup- and level loading functionality as a whole. For the level loader, this was primarily done with a top-down approach where top-modules were tested with stubs. E.g. the `LevelLoader` class could easily be replaced by a stub, as the `LevelParser` class is dependent on the `ILevelLoader`-interface. These stubs provide canned answers to calls made during the test.

6.1 Playtesting and bugs

The game has also been tested by playing the game, to test if the game is balanced, and does not contain any unintended visual or behavioural bugs, that does not get caught during unit testing.

Overall the game functions as intended. However, we discovered the following bugs / problems that can be improved:

- The ball moves a little too slow, and it can be hard to complete some of the levels in the required time.
- When the ball hits multiple blocks at once, the ball changes direction twice, and will get launched back the same way it came from. However, this bug does not occur very often, and is not fatal.
- It can be difficult for the user to understand, what the different numbers in the top means (points and time left).

6.2 Evaluation of test

To evaluate how well our test covers the requirements (i.e. how well the tests cover the actual code), we have used `tdotnet-reportgenerator-globaltool` [9]. This tool shows that there is a total of 73,2% line coverage and 70,7% branch coverage, see Appendix F. 20 classes out of 38 have over 90% line coverage. As can be seen in the analysis of the test, some of the classes have low code coverage, this is partly due to the fact that no tests have been made for methods that make trivial method calls to others.

Overall, decent tests have been made for all the methods where it provides value and where specifications must be tested. This was especially the case for the level loading, collision and power-up functionality, where all has complete or very high coverage. In further work, more time can of course be spent on getting a higher code coverage.

One problem with some of the unit tests were that it was difficult to isolate the unit without testing any other concerns/dependencies. This was often the case, when methods used the singleton `GameBus`, where we had to process the events, to check if the correct event was generated.

In general the singleton classes were difficult to test, since it is not possible to create a new instance from test to test. Instead we had to reset the state ourselves. This can possibly could have violated the integrity of some tests.

6.3 Code Metrics

To analyze code quality, code metrics are ideal for this purpose. To do this, we took advantage of built-in Visual Studio tool called `Calculate Code Metrics` [8]. This analyzes the solution in the form of white-box metrics, and gives concrete figures on code quality, i.e. identify code smells or anti-patterns [3] and how complex the code is and how easy/difficult it is to maintain.

The code metrics shows that the main part of the solution have fairly good metrics, see Appendix G. The following are some key facts from the analysis:

- The solution is easy to maintain. All members are in the good area concerning the `Maintainability index`.
- According to the Cyclomatic Complexity (CC), the solution contains simple methods. Only 1 out of 177 methods is a bit complex, where the rest are in the good area.
- The solution has a number of classes with a rather deep level of inheritance, but not critical. Unfortunately, this highers the risk of base class modifications to result in a breaking change.
- The solution has a number of methods with a little too much coupling, but not critical. 16 out of 177 methods are medium connected, where the rest are in the good area. This lowers maintainability of the code.
- The solution contains mainly methods with small amount of Source Lines of Code (SLOC), where 11 methods out of 177 are in the critical area, and the rest are in the good area.

In future versions of the project, the depth of inheritance should be reduced (or remain at same depth) as well as making the coupling more low even though this has been in focus throughout the project. Although the deep inheritance is also partly due to the fact, that we were not allowed to edit DIKUArcade. Furthermore, the number of SLOC should also be considered. However, in general, most classes and methods have a high maintainability and simple structure with a fairly low coupling.

7 Conclusion

In this implementation of the game, we have implemented a new version of the game Breakout with the game engine DIKUArcade in C# successfully. As stated in the requirements, the solution consists of a menu, where a new game can be started from, and the game loads all the levels from the given text files. The game is overall fairly well-structured, where we have strived to follow the SOLID and GRASP principles. This was among others, achieved by using several design patterns in the game design, which improves the overall design structure. However, compliance with these principles could be improved. Furthermore all game-units have been tested, and everything seems to work as intended according to the specification, besides the few bugs found during play-testing. According to the calculated code metrics, most units are maintainable with a Low Coupling, but of course can be improved.

References

- [1] *Breakout - Atari 2600*. RetroGames. URL: https://www.retrogames.cz/play_222-Atari2600.php (visited on 06/06/2022).
- [2] Boris Düdder. *Frameworks and Libraries [Slides]*. University of Copenhagen - Department of Computer Science. May 24, 2022. URL: <https://absalon.ku.dk/courses/56602>.
- [3] Martin Fowler and Kent Beck. *Refactoring - Improving the Design of Existing Code*. 10th ed. Addison-Wesley, 2022.
- [4] Erich Gamma et al. *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994.
- [5] Gary McLean Hall. *Adaptive Code. Agile Coding with Design Patterns and SOLID Principles*. 2nd ed. Microsoft Press, 2017.
- [6] Stephen Kan. *Metrics and Models in Software Quality Engineering*. 2nd ed. Addison-Wesley, 2003.
- [7] Craig Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and the Unified Process*. 2nd ed. Prentice Hall.
- [8] Microsoft. *Calculate code metrics - Visual Studio (Windows)*. URL: <https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022> (visited on 06/03/2022).
- [9] Microsoft. *Unit test code coverage*. URL: <https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-code-coverage> (visited on 06/06/2022).
- [10] Manikanta Pattigulla. *Measure Your Code Using Code Metrics*. URL: <https://www.c-sharpcorner.com/article/measure-your-code-using-code-metrics/> (visited on 06/03/2022).
- [11] David Thomas and Andrew Hunt. *The Pragmatic Programmer. Your Journey To Mastery, 20th Anniversary Edition*. 2nd ed. Addison-Wesley Professional, 2019.

Acronyms

- OOP** Object-Oriented Programming. 1
- SRP** Single-Responsibility Principle. 7–10, 12
- OCP** Open-Closed Principle. 6–8, 10, 13
- LSP** Liskov Substitution Principle. 8, 10
- ISP** Interface Segregation Principle. 9, 10
- DIP** Dependency Inversion Principle. 11
- CC** Cyclomatic Complexity. 15, 27
- SLOC** Source Lines of Code. 15, 27

Glossary

- SOLID** Five design principles to make object-oriented designs more understandable and maintainable [5].. 6, 10, 15
- GRASP** Five patterns to make a proper design with a methodical approach. Objects can have mainly two types of responsibilities: *Doing* and *Knowing* [7].. 6, 15
- Low Coupling** Increase reuse and reduce the impact of change, i.e. the responsibilities should be assigned to those to remain the coupling low. [7].. 9, 11, 15
- High Cohesion** How closely (strongly) all responsibilities are related - the aim is to have high cohesion. "*Do one thing and do it well*" [7].. 9
- Information Expert** The responsibility must be given to the Information Expert, i.e. the class who has the information to be able to fulfill the responsibility [7].. 12
- Polymorphism** Assign the responsibility to those which behavior varies by type. [7].. 8
- DRY** Don't Repeat Yourself: "*Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*" [11].. 6

Appendices

A User's guide

This guides shows how to run, test and play the game *Breakout*.

Software requirements

The game is expected to work on the following operating systems: Windows, macOS and Linux.

1. Download and install *.NET 6.0 version SDK 6.0.200*¹
2. Download and install *Visual Studio Code (VS Code)*²
3. Install the VS Code Extension C# (powered by OmniSharp)³ via Marketplace.

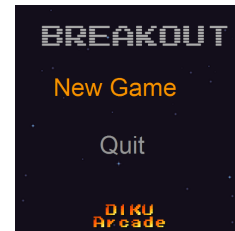
How to run and play the game?

1. Open a terminal and navigate into the project folder *Breakout*.
2. Execute the .NET command *dotnet run*, which will compile and run the program.

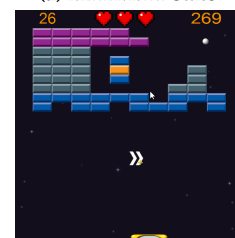
When the game is instantiated, the *MainMenu* is shown, see Figure 4a. The player can here either start a *New Game* or *Quit* by using , and to choose. If *New Game* is chosen the player together with some blocks is shown, see Figure 4c below. The player can move left and right with the keyboard keys and . The game can be paused, see Figure 4b by pressing and can be *Continued* again and it is also possible to *Quit* the game. If the game is lost or won then the score will be shown and it is possible to start a new game or go back to Main Menu, see Figure 4d.

How to execute the tests

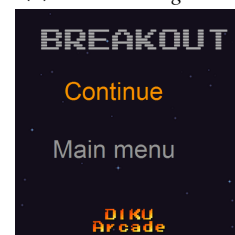
1. Open a terminal and navigate into the project folder *BreakoutTests*.
2. Execute the .NET command *dotnet test*, which will compile and run all the tests.



(a) MainMenu state.



(b) GameRunning state.



(c) GamePaused state.



(d) GameOver state.

Figure 4: The *Breakout* game.

¹<https://dotnet.microsoft.com/en-us/download/dotnet/6.0>

²<https://code.visualstudio.com/>

³Extension ID: ms-dotnettools.csharp

B Requirements



B.1 Levels

- R.1 Load levels locally from the computer hard drive.
- R.2 Validate if the contents of the level file match predefined requirements (template).
- R.3 If (R.2) is fulfilled, the content should be stored in data structures.
- R.4 Responsibility must be distributed to the respective entities.

B.2 Game (-states)

- R.1 The game should have 4 states:
 - Main menu: Start a game when the player is ready or quit.
 - Game Running: Run the game.
 - Game Paused: Pause the game.
 - Game Over: The game is over when all available levels has been completed or the player is dead - show score.
- R.2 The game should only be in one state at a time.

B.3 Player

- R.1 The player should be a rectangular shape.
- R.2 The player should be, upon pressing  or  (only) / a or d, capable of moving horizontally within the screen and at the bottom half of the screen.
- R.3 The player should be able to pause and start a new game.
- R.4 When the last remaining ball leaves the screen the player loses a life. If any lives are remaining the player will gain a new ball to launch. If zero remaining lives then then the game is over.

B.4 Ball

- R.1 The ball should have a round shape.
- R.2 The ball must only be able to leave the screen at the bottom. When a ball leaves the screen.
- R.3 If the ball collides with a block or a wall, the ball should change direction.
- R.4 If the ball collides with the player, the ball should change direction in a opposite vertical direction.
- R.5 Possibility to add more ball to the game.
- R.6 All balls must at all times move at the same constant speed.
- R.7 A ball must always be launched with a positive vertical speed (upwards direction).

B.5 Block

- R.1 Every block should have a rectangular shape.
- R.2 Every block should have a rewarding point-value if it is hit and a certain number of lives (health). When health is zero block should be removed
- R.3 Blocks should be placed according to the level files.

B.6 Wall

- R.1 There should be walls around the edge of the game except from the bottom at the screen.
- R.2 The ball should be able to bounce on the walls.

B.7 Powerups

- R.1 When a power-up block is destroyed it will spawn a visual representation of the power-up effect that it will yield.
- R.2 The power-up must move at a constant negative vertical speed (downwards direction).
- R.3 If the power-up collides with the player the power-up will be considered activated and it should disappear from the screen.
- R.4 Powerups should be able to be activated/deactivated when needed.

B.8 Collision detection

- R.1 Detect collision between:
 - player and ball,
 - player and powerup,
 - ball and block,
 - ball and wall.

B.9 General design requirements

In addition to the above requirements, we will strive to fulfill the following design requirements:

- R.1 The game should utilize the *DIKUArcade* framework, to make the development process easier and faster.
- R.2 The implementation should follow the SOLID-, GRASP principles and should take advantage of design patterns where appreciated.
- R.3 Many objects needs to communicate, use therefore the same generic way to communicate.

C Specifications

From the requirements in Appendix B, we have created the following specifications - only the most important classes are shown.

C.1 LevelLoader

```
1 string[] Load (string path)
2     //precondition: path points to an existing text-file
3     //postcondition: returns a string array with each line as an entry
```

C.2 LevelParser

```
1 Level CreateLevel (string path)
2     //precondition:
3     //     path points to an existing text-file
4     //     the text-file has the correct level-format (LevelValidator.validate return true)
5     //postcondition:
6     //     returns a level-instance containing the data of the text-file
```

C.3 LevelValidator

```
1 bool ValidateLevel (string[] level)
2     //precondition:
3     //     level is not null
4     //postcondition:
5     //     returns true if the following conditions is meet
6     //         The groups; Map, Meta and Legend exist
7     //         The groups cannot overlap
8     //         All meta data is seperated by a ':'
9     //         All legend data is seperated by a ')'
10    //         No repetition of data-entries
11    //         The map has a size of 25 * 12
12    //         All tiles in the map is represented as a legend field
13    //         All images in legend should exist
14    //         All tiles described in meta-data should exist in legend-data
15    //     returns otherwise false
```

C.4 CollisionHandler

```
1 var collidableList //the subscribed gameobjects
2 void Update ()
3     //precondition:
4     //     none
5     //postcondition:
6     //     collidableList does not contain any gameobject marked for deletion
7     //     notifies all Gameobjects in collidableList, that will collide in the next frame
```

C.5 Block

```
1  var health
2  var deleted
3
4  constructor()//
5      //precondition:
6      //    none
7      //postcondition:
8      //    health = 3
9
10 void Hit ()
11     //precondition:
12     //    the block is hit by ball
13     //postcondition:
14     //    health' = health - 1
15     //    if health <= 0 then deleted = true
```

C.6 Player

```
1  Vec2F position, acceleration, extent
2
3  //object invariants:
4  //    position.y + extent.y / 2 < 0,5
5  //    position.x >= 0 && position.x + extent.x <= 1
6
7  constructor()
8      //precondition:
9      //    none
10     //postcondition:
11     //    position.x + extent.x / 2 == 0,5
12
13 void Update ()
14     //precondition:
15     //    the block is hit by ball
16     //postcondition:
17     //    position.x' == position.x + acceleration.x
18     //    position.x' == 0
19     //    position.x' == position.x - extent.x
```

C.7 Ball

```
1 Vec2F position, direction, extent
2 bool deleted
3
4 constructor()
5     //precondition:
6     //     none
7     //postcondition:
8     //     direction.x > direction.y
9     //     direction.x > 0
10    //     deleted = false
11
12 void ChangeDirection ()
13     //precondition:
14     //     the ball has hit another gameobject
15     //postcondition:
16     //     position is updated according to equation (1) to (4)
17
18 void Update()
19     //precondition:
20     //     none
21     //postcondition:
22     //     if direction.x < 0 then deleted = true
```

When the ball hits a block/object, it should change direction where the new direction-vector can be calculated in the following way:

$$\theta = \text{object.rotation} \quad (1)$$

$$n = \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \end{bmatrix} \quad (2)$$

$$\text{new_dir} = \text{dir} - 2(d \cdot n)n \quad (3)$$

$$\text{new_dir.X} = \text{new_dir.X} + \text{object.dir} \cdot 0.5 \quad (4)$$

In (2) the normal vector of the collided object are calculated. This can be used in (3) to calculate the reflection vector (the new balls new direction). This direction should also depend on the other objects directory (4). This is done to fulfill R4. In this way, the player should be able to control the ball by moving the padel at collision with the ball.

D SwitchState method in the StateMachine-class

```
1 private void SwitchState(GameStateType stateType) {  
2     switch (stateType) {  
3         case GameStateType.MainMenu:  
4             ActiveState = MainMenu.GetInstance();  
5             break;  
6     ...  
}
```

E List of tests

Test method	Passed
BallTest	10/10
TestBounceDynamic	2/2
TestBounceStationary	6/6
TestUnbreakableCollision	1/1
TestWallCollision	1/1
CollisionHandlerTest	5/5
UpdateWCTest	2/2
CollisionBallAndBlockTest	1/1
CollisionBallAndPlayerTest	1/1
CollisionWallAndPlayerTest	1/1
GamePausedTest	6/6
HandleKeyEventTest	6/6
LevelParserTest	8/8
TestCreateLevelInvalid	1/1
TestCreateLevelInvalidPath	1/1
TestCreateLevelValid	1/1
TestInvalidPath	1/1
TestLevel1	1/1
TestLevel2	1/1
TestLevel3	1/1
TestNoContent	1/1
LevelValidatorTest	18/18
TestAll	2/2
TestAllWhiteBox	2/2
TestLegend	5/5
TestMapSize	3/3
TestMeta	5/5
TestSimpleLevel	1/1
PlayerTest	9/9
TestMove	5/5
TestDecelerateAndStop	1/1
TestGameOver	1/1
TestMaxSpeed	1/1
TestMoveBoth	1/1
PointsTest	7/7
TestAddPoints	5/5
TestMaximumPoints	1/1
TestResetScore	1/1
PowerupTest	7/7
TestDoubleSize	1/1
TestDoubleSpeedBall	1/1
TestExtraLife	1/1
TestHardBall	1/1
TestPlayerSpeed	1/1
TestSplit	1/1
TestWideSize	1/1
StateMachineTest	4/4
TestInitialState	1/1
UnbreakableTest	1/1
TestUnbreakable	1/1
HardenedBlockTests	4/4
TestDieBlock	1/1
TestDieHardened	1/1
TestHealth	1/1
TestHit	1/1
Tests	2/2
TestDie	1/1
TestHit	1/1
PowerUpIntegrationTest	1/1
PowerUpSpawnTest	1/1
PowerUpTypeTransformer	2/2
TestStateToString	1/1
TestTransformStringToState	1/1

F Code Coverage report

Generated by use of the tool: `dotnet-reportgenerator-globaltool` [9]. List is sorted by line coverage in ascending order. Gray highlighted cells, are methods without branches.

Name	Covered lines	Line coverage	Branch coverage
Program	0/5	0%	
Game	0/24	0%	0%
LevelContainer	5/33	15.1%	33.3%
GameObject	3/10	30%	
MainMenu	15/44	34%	12.5%
GameRunning	33/76	43.4%	25%
GameOver	28/59	47.4%	30%
Level	87/158	55%	57.1%
BallContainer	45/64	70.3%	63.6%
MetaTransformer	6/8	75%	60%
Split	6/8	75%	
Powerup	20/26	76.9%	100%
GamePaused	38/47	80.8%	75%
StateTransformer	13/16	81.2%	76.9%
Block	18/22	81.8%	100%
Ball	72/85	84.7%	89.4%
ExtraLife	7/8	87.5%	
Score	22/25	88%	100%
Player	91/101	90%	94.4%
PowerupTransformer	18/20	90%	85%
SubarrayExtractor	12/13	92.3%	87.5%
CollisionHandler	35/35	100%	100%
CollisionHandlerData	6/6	100%	
GameBus	25/25	100%	100%
StateMachine	30/30	100%	100%
GameObjectSpy	5/5	100%	
HardenedBlock	13/13	100%	100%
DoubleSize	10/10	100%	
DoubleSpeed	10/10	100%	
HardBall	10/10	100%	
PlayerSpeed	10/10	100%	
PowerupBlock	5/5	100%	
PowerupContainer	29/29	100%	100%
Wide	11/11	100%	
Unbreakable	7/7	100%	
Wall	5/5	100%	
LevelLoader	7/7	100%	100%
LevelParser	34/34	100%	100%
LevelValidator	68/68	100%	100%
Total	859/1.172	73.2%	70.7%

G Code metrics analysis

This appendix includes an code metrics analysis performed in Visual Studio. The metrics is analyzed by the tables listed on [10], and with support from the literature [6, 8].

G.1 Evaluation tables

Evaluation tables taken from [10].

Maintainability Index	Evaluation
20-100	Good maintainability
10-19	Moderate maintainability
0-9	Low maintainability
CC	Evaluation
1-10	Simple procedure
11-20	Little complex
21-50	Complex
>50	Untestable
Depth of Inheritance	Evaluation
1-2	Good
3-4	Medium (still okay)
>4	Critical
Class Coupling	Evaluation
0-9	Low coupling
10-30 (on member level) & 10-80 (on type level)	Medium coupling
>30 (on member level) & >80 (on type level)	High coupling
SLOC	Evaluation
1-10	Good
11-20	Medium (still okay)
>20	Critical

G.2 Results

The following table is the code metrics analysis performed in Visual Studio. The metrics that are too high according to risk evaluation tables have the same color code as in the evaluation tables in G.1.

Class/Interface	Maintainability Index	CC	Depth of Inheritance	Class Coupling	SLOC
Ball			3	13	164
Ball()	83	1		5	7
Ball()	82	1		5	4
SetRandomDirection()	68	1		5	7
ResetPosition()	83	1		2	4
getDirFromCollisionVec()	85	6		2	17
Accept()	97	1		2	10
BlockCollision()	95	2		1	7
PlayerCollision()	100	1		1	6
WallCollision()	100	1		1	6
UnbreakableCollision()	100	1		1	6
changeDirection()	61	2		6	18
Update()	77	2		3	11
AtDeletion()	97	1		2	5
Clone()	79	1		4	5
Render()	100	1		0	4
ProcessEvent()	77	5		5	17
BallContainer			1	13	110
BallContainer()	100	1		0	3

AddBall()	69	2		6	15
addBall()	79	1		8	7
Render()	96	2		2	5
Update()	71	4		2	15
RemoveBall()	76	2		2	9
Destroy()	96	2		2	5
splitBalls()	69	3		2	13
ProcessEvent()	82	3		2	13
CountBalls()	95	1		2	3
Block			3	7	47
Block()	83	1		4	7
Hit()	85	2		0	6
AtDeletion()	86	1		2	6
Accept()	97	1		2	10
BallCollision()	100	1		1	6
CollisionHandler			1	9	50
GetInstance()	80	2		0	8
Subscribe()	96	1		2	5
removeDestroyed()	78	3		2	8
Update()	61	5		9	19
CollisionHandlerData			1	2	23
CollisionHandlerData()	85	1		2	4
DoubleSize			4	7	16
DoubleSize()	94	1		3	1
Activate()	94	1		3	4
Deactivate()	92	1		3	4
DoubleSpeed			4	7	16
DoubleSpeed()	94	1		3	1
Activate()	94	1		3	4
Deactivate()	92	1		3	4
ExtraLife			4	8	12
ExtraLife()	97	1		3	1
Activate()	87	1		4	5
Deactivate()	100	1		0	1
Game			2	15	49
Game()	70	1		13	10
Render()	95	1		2	4
Update()	86	1		3	5
keyHandler()	92	1		4	6
ProcessEvent()	82	4		3	13
GameBus			1	4	35
GetBus()	93	2		1	5
TriggerEvent()	65	1		3	13
TriggerTimedEvent()	64	1		4	9
GameObject			2	4	47
GameObject()	97	1		3	1
AtDeletion()	100	1		0	3
Update()	100	1		0	2
Accept()	100	1		1	4
BlockCollision()	100	1		1	4
BallCollision()	100	1		1	4
WallCollision()	100	1		1	4
PlayerCollision()	100	1		1	4
PowerUpCollision()	100	1		1	4
UnbreakableCollision()	100	1		1	4
GameObjectSpy			3	4	17
GameObjectSpy()	97	1		3	4
Accept()	96	1		2	6
GameOver			1	14	99
GetInstance()	76	2		0	9
InitializeGameState()	68	1		6	12
ResetState()	96	1		0	4
UpdateState()	100	1		0	2
RenderState()	66	3		2	13
HandleKeyEvent()	93	2		2	10
keyPressed()	73	5		6	20
ProcessEvent()	81	3		2	11
GamePaused			1	11	78
GetInstance()	76	2		0	9
InitializeGameState()	79	1		2	8
ResetState()	96	1		0	4
UpdateState()	100	1		0	2
RenderState()	74	3		2	10
HandleKeyEvent()	93	2		2	10
keyPressed()	73	5		6	21
GameRunning			1	25	134
GameRunning()	100	1		0	1
GetInstance()	76	2		0	9
InitializeGameState()	56	1		17	25
ResetState()	81	1		3	6
UpdateState()	80	1		4	6
RenderState()	80	1		4	6
HandleKeyEvent()	91	3		2	13
keyPress()	86	5		5	17

keyRelease()	87	4		5	15
ProcessEvent()	78	4		6	15
GameStateType			1	0	7
HardBall			4	7	16
HardBall()	94	1		3	1
Activate()	94	1		3	4
Deactivate()	94	1		3	4
HardenedBlock			4	4	31
HardenedBlock()	78	1		3	12
Hit()	73	3		2	12
ILevelLoader			0	0	8
Load()	100	1		0	4
Level			1	29	235
Level()	70	1		2	13
AddGameObject()	86	1		3	6
Activate()	59	3		16	19
Update()	71	3		4	14
DeleteMarkedEntity()	72	3		2	15
Render()	84	1		3	6
renderTime()	76	2		2	7
hasWon()	75	4		2	10
generateBlocks()	65	4		6	12
Deactivate()	79	2		7	7
DeleteBlock()	81	3		2	9
chooseBlock()	58	7		11	21
CountItems()	95	1		2	7
Equals()	67	10		4	16
ToString()	53	9		2	32
ProcessEvent()	82	3		3	9
LevelContainer			1	12	73
LevelContainer()	100	1		0	1
GetLevelContainer()	93	2		0	7
Reset()	81	2		1	6
initializeLevels()	75	1		6	15
NextLevel()	71	2		8	15
SetActiveLevel()	89	1		3	7
ResetLevelContainer()	100	1		0	6
LevelLoader			1	8	19
Load()	73	2		7	15
LevelParser()	96	1		1	7
CreateLevel()	64	2		7	33
linesToDict()	71	2		3	12
linesTo2DCharArr()	74	3		0	10
LevelValidator			1	11	131
ValidateLevel()	65	4		2	20
validateGroups()	69	3		3	15
validateLegend()	70	3		3	14
validateLegendPair()	84	4		3	11
validateMeta()	69	3		3	15
validateMetaPair()	85	4		3	11
validateMap()	68	7		2	18
MainMenu			1	11	75
GetInstance()	76	2		0	9
InitializeGameState()	79	1		2	8
ResetState()	100	1		0	2
UpdateState()	100	1		0	2
RenderState()	74	3		2	10
HandleKeyEvent()	93	2		2	10
keyPressed()	73	5		6	20
MetaTransformer			1	3	22
StateToString()	86	6		3	18
MetaType			1	0	7
Player			3	13	176
Player()	69	1		7	12
Render()	93	1		1	5
Update()	93	1		1	6
updateMovement()	57	6		3	25
resetDir()	78	1		2	6
setMoveLeft()	91	2		0	5
setMoveRight()	92	2		0	5
GetPosition()	87	1		2	5
loseLife()	85	1		1	5
addLife()	85	1		1	5
Reset()	84	1		1	5
gameOver()	97	1		2	4
ProcessEvent()	67	12		4	40
Accept()	97	1		2	9
PlayerSpeed			4	7	16
PlayerSpeed()	94	1		3	1
Activate()	94	1		3	4
Deactivate()	92	1		3	4
Powerup			3	15	78
CreateRandom()	67	1		9	26
Powerup()	70	1		5	8

Update()	100	1		1	4
Accept()	97	1		2	9
PlayerCollision()	79	2		3	10
Activate()	100	1		0	1
Deactivate()	100	1		0	1
PowerupBlock			4	9	16
PowerupBlock()	97	1		3	5
AtDeletion()	91	1		6	6
PowerupContainer			1	9	52
GetPowerupContainer()	93	2		0	3
PowerupContainer()	96	1		4	3
activate()	81	3		3	11
deactivate()	72	3		4	13
ProcessEvent()	88	3		2	10
PowerupTransformer			1	3	51
StringToState()	85	8		3	24
StateToString()	85	8		3	22
PowerupType			1	0	16
Score			1	3	46
Score()	76	1		3	8
AddPoints()	66	4		1	13
Render()	100	1		1	4
Reset()	84	1		1	5
GetScore()	100	1		0	5
StateMachine			1	12	54
StateMachine()	73	1		9	12
switchState()	85	5		6	18
ProcessEvent()	75	4		5	16
StateTransformer			1	3	44
StringToState()	86	6		3	18
StateToString()	86	6		3	18
Split			4	7	12
Split()	97	1		3	1
Activate()	97	1		3	4
Deactivate()	100	1		0	2
SubarrayExtractor			1	2	24
Extract()	67	5		2	24
Unbreakable			4	5	17
Unbreakable()	92	1		3	6
Accept()	97	1		2	6
Wall			3	7	26
Wall()	96	1		5	5
GetShape()	95	1		2	5
Accept()	97	1		2	10
Wide			4	8	17
Wide()	94	1		3	1
Activate()	86	1		4	5
Deactivate()	92	1		3	4
Min overall: 53		Max on member lvl: 12 & Max on type lvl: 4	Max on type lvl: 29	Max on member lvl: 17	Max on member lvl: 40

No. of Types (classes and interfaces)	41
No. of Methods	177