Assignment 5

Group 66, Bust-a-move

Exercise 1 - Anonymous peer suggestions

Last week you were asked to understand, read, and analyze the code of another anonymous group. In addition, you had to propose meaningful enhancements to the other's group codebase, for a total of 30 points. 1. Ask your TA for the peer feedback that he received about your group and implement the proposed enhancements. Your TA will make sure that all the enhancements are relevant to you current codebase.

The feedback provided by the anonymous group was extensive, bellow a paraphrased assignment is created which is implemented in our codebase.

Allot of responsibilities are combined together in a few classes. The most prominent is the BallManager class. This class is almost 700 lines and does not only manage balls, but also game functionality related functionality. The functionality needs to be split up among classes as the BallManager now is a God Class, update functionality should be moved to the balls themselves. Also within the BallManager methods with too much responsibility/complexity exists such as the updateBalls method at over 120 lines and many loops. Make the functionalities as atomic as possible.

There were other suggestions which were related to the asset loader and key initialisations. The asset loader will be redesigned in assignment 2, so it is left out of this assignment. The key initialisations are left out for this assignment and kept as a issue to improve upon in the future, as it's less related to the rest of the assignment and the current setup assignment already introduces a lot of work and is deemed more important.

This assignment removes the BallManager as a God Class and as can be seen in assignment 2, this is also an issue reported by inCode which we need to fix. So this assignment will also encompass the work done for assignment 2.2.1.

The main gripe of the assignment was the elimination of the BallManager as a God class. The suggestion is to split functionality into separate classes and make the methods smaller. This is implemented and will be discussed shortly. The suggestion that the balls themselves should know how to update regarding their placement in the grid was not implemented in the suggested way. The balls are still managed by separate classes, but these classes only manage balls and don't do game related management. The reason why this is done, is because the grouping of balls which need to be managed is more practical and logical for the management then moving this functionality into the balls themselves as a managing class will be needed to have the “overall” picture. Putting allot of this in the balls would unnecessarily complicate matters and we have the opinion that the balls should just be balls, and they should not know how they are used, just wat to do.

Implementation changes

The BallManager started as a place where the balls were managed, this grew however to a God class. To solve this the BallManager is split up into the following classes (shown within a CRC table):

(Here “game” relates to a single instance of the actual play actions and graphics. For split screen for example multiple “game” instances are ran simultaneously)

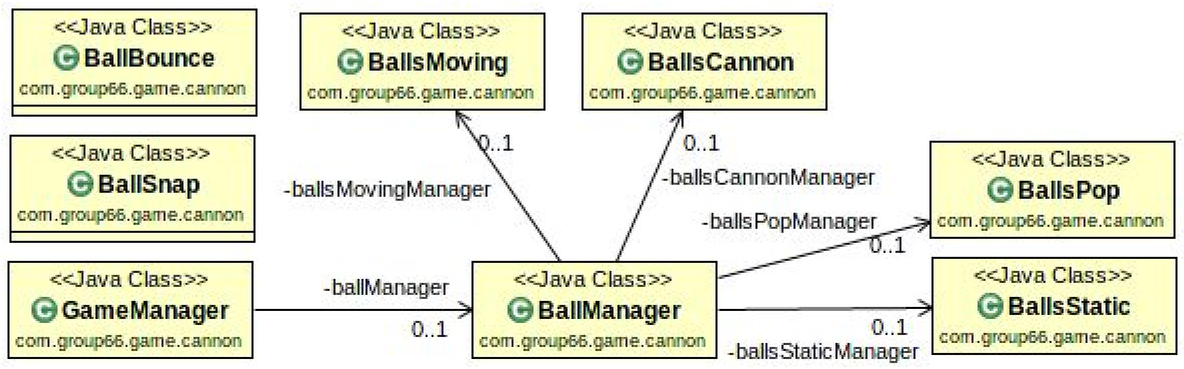
Class	**Responsibility**	**Collaborators**
GameManager	Manages the game related decisions and rules.	BallManager BallsStatic
BallManager	Manages the balls in the game. The collaborating classes are used to manage all the ball functionalities. The BallManager itself acts as a layer between the game rules in the GameManager and the actual managed balls present in BallsMoving, BallsStatic and BallsCannon. The BallsPop, BallSnap and BallBounce helper classes are used here to provide other ball functionality.	BallsMoving BallsStatic BallsCannon BallsPop BallSnap BallBounce
BallsMoving	Manages the balls in the game which are moving. These moving balls are the balls which are shot from the cannon but not yet reached the static balls.	
BallsStatic	Manages the balls in the game which are static in the “grid”.	
BallsCannon	Manages the balls that are fed into the cannon to be shot.	
BallsPop	Manages the popping state and animation of the balls which are popped. This will ensure that the ball pop animation is played.	

BallSnap	A static helper class, this class does not manage data but only contains the helper methods used to snap to a location on either another ball or the roof.	
BallBounce	A static helper class, this class does not manage data but only contains the helper methods used to bounce of the game edges to stay in the playing field.	

As can be seen in the CRC table, the functionality of the BallManager is now split up into 8 classes each responsible of a part of the functionality. This eliminates the BallManager (or any of the other created classes) from being a God class. In the overall bust-a-move codebase the BallManager instances are replaced by GameManager instances which provides the top layer of the functionality previously provided by the BallManager. The new BallManager manages, as stated in the CRC, the balls in the playing field but also uses newly created classes to aid this management.

Regarding the big methods, there are split into separate methods to improve readability and updatability. Two big functions remain which are the cleanStaticDead() and addStaticBalls() methods within the new BallManager class. It was decided to not split these methods up any further as it would reduce the ability to understand the method in our opinion. All the other methods were shortened accordingly.

Below a simplified UML class diagram is depicted without any of the methodsshown. In the same folder as in which this document is contained a larger UML diagram containing all the functions is also present named *uml-game-manager-complete.png*



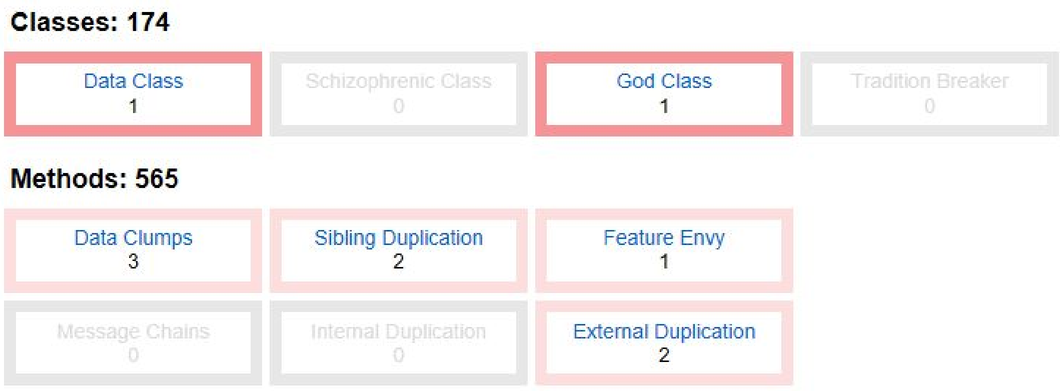
Exercise 2 - Software Metrics (45 pts)

2.1) The inCode tool uses software metrics to detect a number of design flaws. In this exercise you will use it to have guidelines to improve your implementation, from a code quality perspective. Use inCode3 to compute software metrics on your project, then upload the resulting analysis file4 to your git repository. Write in the explanation PDF file where the analysis file is located (3 pts).

The results files are attached as binaries to release tagged v1.4-incode1.
Attached binaries include:
1- bustamove_1477297895294.result: The inCode snapshot of the analysis of the full project (includes test code).
2- core_1477305205142.result: The inCode snapshot of the analysis of the source only (includes core only).

2.2) Consider the 'System summary' view regarding the analysis of your project. You can see the design flaws that seem affect your system. Pick the first three design flaws (in order of severity) that affect your software, and for each flaw complete the following points:
a) Explain the design choices or errors leading to the detected design flaw (4 pts).
b) Fix the design flaw or extensively and precisely explain why this detected flaw is not an error and, thus, should not be fixed (10 pts).

Analysis of the source code for the main module shows the following System Summary and Flawed Classes respectively:

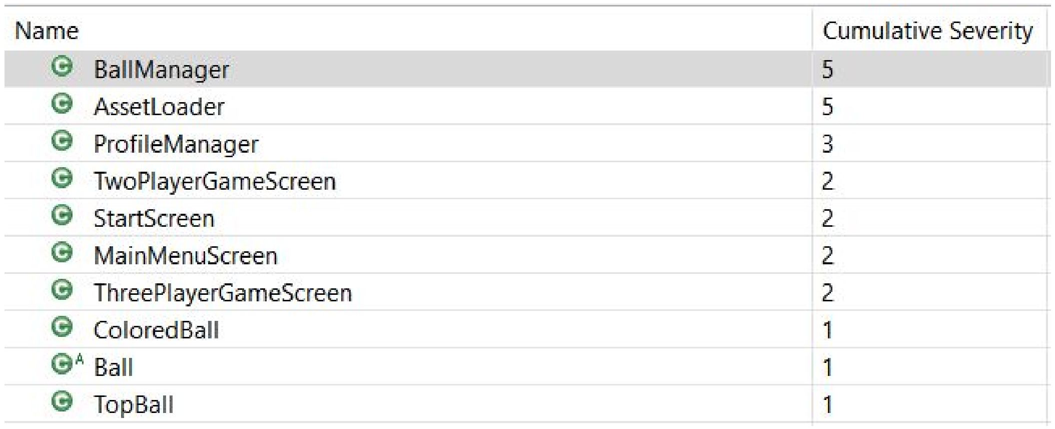


Classes: 174

Data Class 1	Schizophrenic Class 0	God Class 1	Tradition Breaker 0

Methods: 565

Data Clumps 3	Sibling Duplication 2	Feature Envy 1
Message Chains 0	Internal Duplication 0	External Duplication 2



Name	Cumulative Severity
BallManager	5
AssetLoader	5
ProfileManager	3
TwoPlayerGameScreen	2
StartScreen	2
MainMenuScreen	2
ThreePlayerGameScreen	2
ColoredBall	1
Ball	1
TopBall	1

Mainly,the three major flaws that InCode show are:

1. The BallManager class appears to be a God class.
2. AssetLoader class appears to be a Data Class.
3. ProfileManager is Feature Envy: it heavily uses from one or more external classes.

Regarding the ProfileManager code flaw: When a player changes something that should be saved for later e.g. he bought a powerup, a method of this class objects is called to save this information to a JSON file. This is also the case when a class needs to obtain some information saved in the JSON file too. This makes the job of saving/retrieving the player profile to/from a JSON file concise into this class. The method that is used to write data to the JSON file was tagged as Feature Envy, because it heavily uses data that are not in the same class.
Keeping this class this way might be better for future extension. The final goal of it is to manage his levels, things he bought in the shop, his scores, and any extra features or statistics that can be added or stored after extending this game. Currently, it only stores and retrieves his name, and the powerups he purchased at the shop, so the InCode

suggests moving the writeData method to the DymanicSettings class. This might be a valid suggestion at this moment. However, with game extension, things to be stored might be so fragmented around the code that sprinkling JSON writing and reading methods around will make the code unnecessarily complex. Furthermore, having an online multiplayer is one of the highly probable future extensions of this game. This extension would be rather easier and less complex code-wise taking the current ProfileManager structure into account, as the second player might know what he needs from the first player just by peeking at his profile; which will exist in profile manager in its current shape and form.

Based on this, the following flaws will be fixed:
1- The BallManager
2- AssetLoad
3- The two External Duplication method flaws in classes StartScreen, MainMenuScreen.

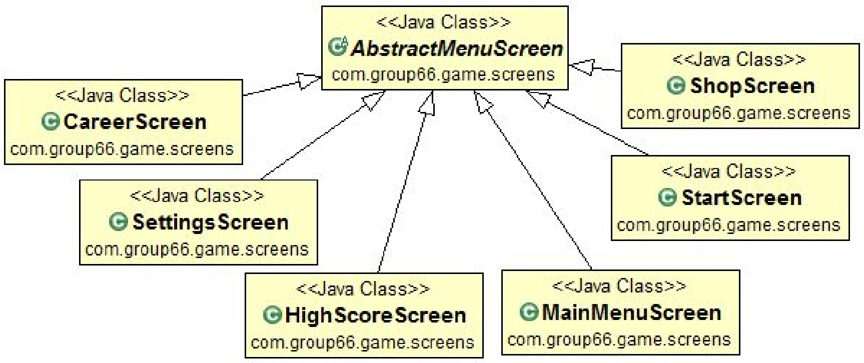
1. **The Ball Manager Class:**

The problem with the BallManager class is that was a God class. This means that the class had a lot of functionality and managed a lot of separate classes. This problem was also given by and resolved in assignment 1. Because of this the resulting changes and a detailed description of the problem and the implemented solutions can be found in the answer given to assignment 1.
After all the changes specified in assignment 1 where implemented, the BallManager (or any of of newly created classes) was no longer a God class. Moreover the severity of the BallManager is now zero (non-existent). This is also the case for the newly created classes.

2. **External Method Duplication:**

Cause of the error:
Up to this previous submission, the screens were separate classes and do not extend any specific super screen class, although they all share the same functionality. This was a design flaw caused by the rapid expansion of the game features from a week to another. With each screen, almost the same things should be generated (buttons, fonts, etc..). This caused a duplication in the code from a screen to another. The two most similar screen classes were StartMenu and MainMenu, which inCode detected the duplication in them.
Fixing the error:

All game screens already extend AbstractGameScreen. Winning screens and losing screens extend AbstractWinScreen and AbstractLoseScreen already. The 6 remaining normal screens will be refactored to extend a newly created AbstractMenuScreen. Currently, those classes use specific Skin objects (used to write fonts on the screen) or specific Stage objects (used to create buttons). These operations will be moved along with sprite-related calls to a method that handles all the graphic load operations of that screen.This will solve the detected code flaw, and will further improve the code quality and readability. As will be seen later, this also helped in solving the class flaw in AssetLoader class, as in this case, the abstract class and its children almost share the same graphical functionality.



2. **AssetLoader Class:**

Cause of the error:

This appears to be classified as a Data Class. InCode describes Data Classes as classes is exposing a significant amount of data in its public interface, either via public attributes or public accessor methods. Furthermore, other websites elaborates saying that a data class refers to a class that contains only fields and crude methods for accessing them (getters and setters). These are simply containers for data used by other classes. These classes do not contain any additional functionality and cannot independently operate on the data that they own.

The goal of having an AssetLoader class in the beginning was to create a unified place for initializing textures, texture regions, texture animations, and all sprite related functionalities that are repeatedly used by few existing classes in the project. With the expansion of the game and the addition of many other features, more sprites had to be included. With that, two problems rose:

- With each screen instantiation, the AssetLoader was called. This caused the loading of all sprites -including those unrelated to that specific screen.
- Splitting the AssetLoader would've been an equally ugly practice, as screens were organized in a bad way e.g. GameScreen, MultiplayerScreen and so on were all separate classes that do not extend a specific AbstractGameScreen.

Splitting would've caused copy-pasting long chunks of code repetitively inside each screen class.

However, with the latest code refactoring explained above, screen classes now have an inheritance hierarchy that makes splitting AssetLoader a rather logical option. This would decrease inter-class coupling, and unify the location of data existence and data usage.

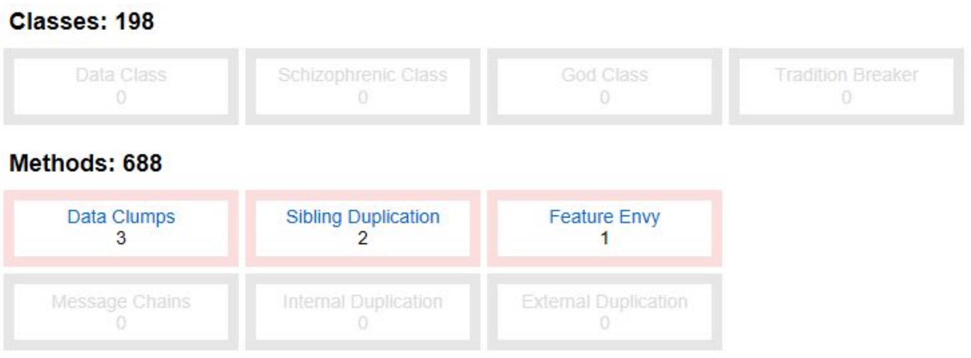
Fixing the error:

Currently, about 40 objects are initialized within this class. Each object refers to a texture of a texture region for a sprite file that exists in the assets folder. All these will be redistributed, and will be managed and called in a method inside their relative class or abstract class. This will diminish the existence of a data class.

After applying the fix, the following improvements to the code were applied:

- InCode shows no data classes exist in the project anymore.
- Sprite-related TextureRegions are no longer static. Getter methods were implemented to get the TextureRegions when needed.
- After splitting the functionality, sprites related to ball animations only have been left in the AssetLoader. This class have been renamed to BallAnimationLoader, moved to the cannon package, and can be later further improved to remove staticness and make it better testable.

After fixing all the errors, inCode output looks as expected, with all target flaws fixed.



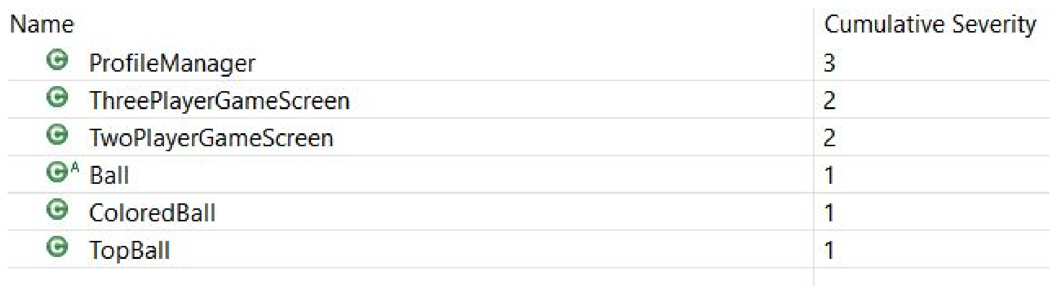
Classes: 198

Data Class	Schizophrenic Class	God Class	Tradition Breaker
0	0	0	0

Methods: 688

Data Clumps	Sibling Duplication	Feature Envy
3	2	1

Message Chains	Internal Duplication	External Duplication
0	0	0



Name	Cumulative Severity
ProfileManager	3
ThreePlayerGameScreen	2
TwoPlayerGameScreen	2
Ball	1
ColoredBall	1
TopBall	1

Exercise 3 - Teaming up (15 pts)

1. In this exercise, you decide together with your TA, during the group meeting on Monday, new game features to add to your game.6 After you decide on the tasks, write a requirements document, which will be evaluated in the same way as for the requirements document of the initial version. Afterwards you must implement the requirements. (11 pts).
Available in a separate file

2. During the analysis and design phases of this extension use responsibility driven design and UML (push to the repository the single PDF file including all the documents produced) (4 pts).
Available in a separate file