Assignment 4

SEM Group Project, sprint 4, week 1.7.

Published on 16 October 2015.

TU Delft Software Engineering Methods 2015-2016 (TI2206)
Supervisor: Dr. A. Bacchelli
Teaching Assistant: A.W.Z. Ang

Group 12

David Alderliesten	4368703	J.W.D.Alderliesten@student.tudelft.nl
Jesse Tilro	4368142	J.Tilro@student.tudelft.nl
Jeroen Meijer	4382498	J.Meijer-5@student.tudelft.nl
Floris Doolaard	4362748	F.P.Doolaard@student.tudelft.nl
Niels Warnars	4372069	N.Warnars@student.tudelft.nl

Table of contents

1 Requirements for Sprint 4

1.1 Functional Requirements

1.1.1 Must haves

1. At game start-up a default keybinding shall be used.
2. The user shall be able to associate different keyboard keys with different player control actions, i.e. alter the keybinding, in the settings screen of the Graphical User Interface.
3. A user shall be able to select each of the available players individually in the settings menu and apply a different keybinding to that specific player.
4. When a key assigned to a current action (of any player) is consequently assigned to a new action (of any player), the assignment of the key to the current action shall be removed leaving the action 'undefined'. The key will be assigned to the new action.
5. When a particular action is 'undefined', this means that no key is assigned to this action resulting in the fact that the action cannot be performed.
6. A user shall be able to change the keybindings during the game by opening the settings menu from within the game.

1.1.2 Should haves

1. When the player has killed all the enemies in a level, the game should wait five second with advancing to the next level, granting the user the opportunity to collect any remaining pickups.
2. When the user has finished the final level, he/she should see a game completion dialogue in a dedicated screen of the Graphical User Interface with an overview of their score and an option to go to the main menu.

1.1.3 Could haves

1. Keybindings could be saved to disk after a user has assigned custom keys to the movement and shooting actions.

1.2 Non-Functional Requirements

2. The total production codebase, excluding code for the Graphical User Interface implementation, shall have a line test coverage of at least 75%.
3. No CheckStyle/FindBugs/PMD warnings shall be left unfixed at the moment of release on October 16th 2015.
4. All the issues being open as of October 13th 2015 should be fixed before October 16th 2015.
5. At least three design flaws indicated by InCode shall be resolved or analyzed in detail in case of a design flaw detection being a false positive.

2 Your Wish Is My Command: Keybinding

Motivation

The feature requested for this week's sprint by the client is the ability to bind different keys to different actions via the graphical user interface. This required the reengineering of the way keybindings were defined and called.

CRC design

The CRC design was created with simplicity in mind. This so that it would be easy to implement without and extend without having to mangle with an overly complex design for a relatively straightforward feature.

The final result consisted of only three cards:

- Keybinding Settings
- Keybinding Container
- Keybinding

The Keybinding Settings card represents the view presented to the user. It manages all the user interactions and is the place where users can rebind their keys.

Then there is the Keybinding Container card that holds all the keybindings of the user and has the ability to update these Keybindings.

Finally the Keybinding itself holds the actual data and manages the constraints of a keybinding.

The way the Keybinding Settings is implemented is nearly identical to the way the Logger Settings is implemented. Meaning that the Settings Menu can either display the Logger Settings or the Keybinding Settings, this can be selected by clicking two tabs at the top of the Settings Menu view.

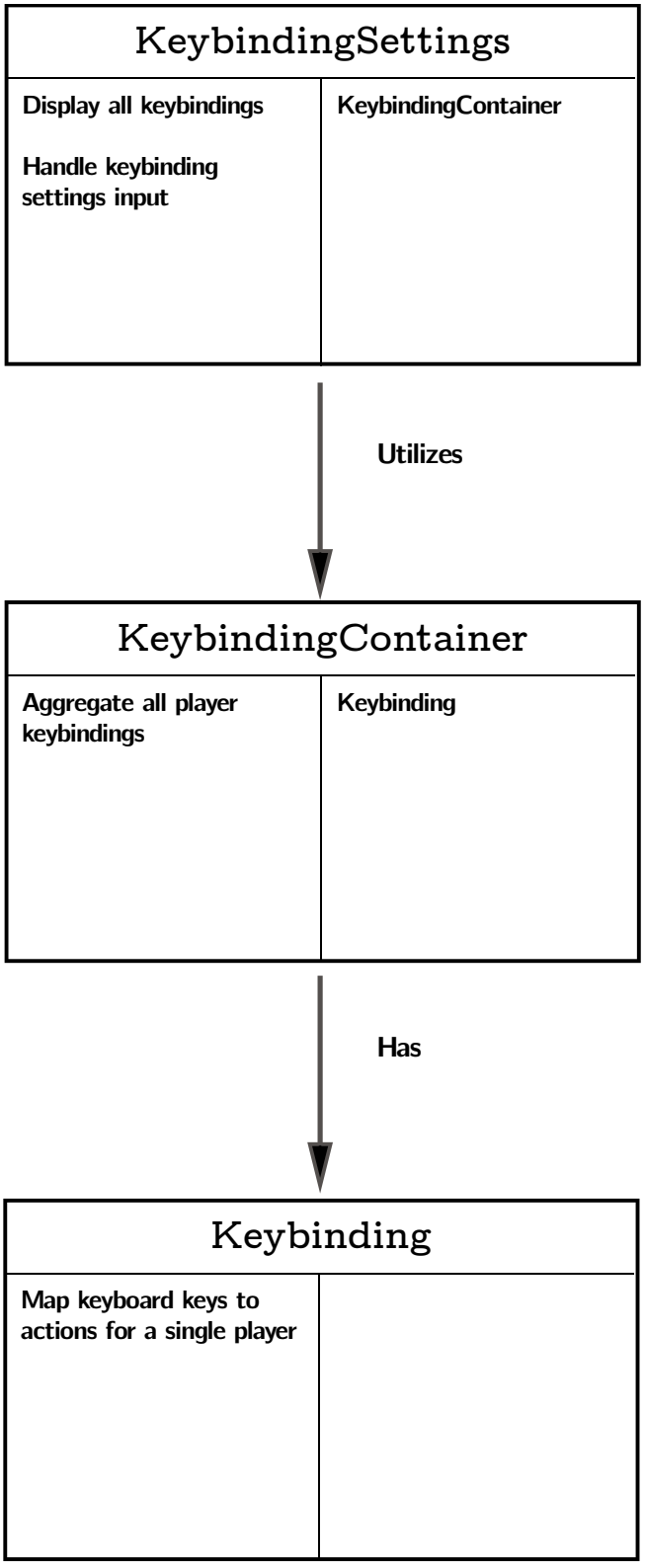

Figure 1: A diagram of the CRC Cards derived through Responsibility Driven Design for the implementation of this sprint's updated Keybinding Mechanism.

3 Software Metrics

3.1 InCode Analysis Result

The file containing the inCode analysis can be found in: documents/sprint-4/inCode.result.

3.2 InCode Analysis Design Flaws

3.2.1 Feature Envy Design Flaw (severity factor 2)

3.2.1.a Cause of the Design Flaw

The feature envy flaw occurs when a method within a class uses data heavily from external classes. The method in question is the detectEnemyBubble() method in the ColissionsLevelModifier, which is responsible for handling player and enemy bubble collisions. The reason this method is flagged is the fact that the method requires more external variables than other methods. This is due to the fact that the method requires the player position and active input to track the button direction and velocity (which together is at least five calls per cycle per check), requires the tracking of bubbles (which can grow to a large amount, currently being restricted at approximately 30 bubbles), and it requires tracking of all the enemies, their position, and their currently headed direction (which is also at least five calls per cycle per check). These collisions could be done in another other way, but the beauty of the current implementation is that all the components that are currently implemented by the level are handled at this high level, allowing easy access to all their data and variables from a central location.

3.2.1.b Solution to the Design Flaw

In order to fix the issue of feature envy, we are advised by inCode to remove the major external capsule (which is our Constants data class), and to re-organize data and operations that are taking place within this method that are making all of these calls. Thus, the class would require separate methods that handle the getting and setting of each of these variables, which in turn would (likely) create more greedy methods. Thus, we fix one problem by creating another, which would also bloat the code and severely drop the code quality. For example, we currently have the following snippet of code within the collisions class under evaluation, as displayed on the next page.

```
if (bubbles.size() > 0 && enemies.size() > 0) {
     for (int i = 0; i < enemies.size(); i++) {

      for (int j = 0; j < bubbles.size(); j++) {
         if (!(bubbles.get(j).hasNPC()) && enemies.size() != i
             && enemies.get(i).inBoxRangeOf(bubbles.get(j),
Constants.COLLISION_RADIUS)
             && new Collision(bubbles.get(j), enemies.get(i),
delta).colliding()) {

            enemies.remove(i);
            enemyBubbles.add(bubbles.get(j));
            bubbles.get(j).setHasNPC(true);
            bubbles.get(j).setLifetime(1.5 * Constants.BUBBLE_LIFETIME);
```

Figure 2: The code under analysis for the Feature Envy design flaw by inCode.

In order to get rid of the feature enviness of this method, we would have to implement at least five additional class methods and their associated variables in order to track this. For starters, this would bloat the class even more (it is already one of our largest classes with the greatest complexity in terms of functioning), and it would make the class coupling and complexity even worse. This would result in a "fixed" method below that is not any improvement at all.

```
if (bubblesInGame.getBubbleSize() > 0 && enemiesInGame.amountOfEnemies()
> 0) {
      for (int i = 0; i < enemies.size(); i++) {

       for (int j = 0; j < bubblesInGame.getBubblesize(); j++) {
          if (!(bubbles.getBubble(j).hasALinkTo(fetchAnNPC())))
          && enemiesInLevel.size() != i
              &&
enemiesInLevel.getSpecificEnemy(i).inBoxRangeOf(bubbles.get(j),
Constants.COLLISION_RADIUS)
              && new Collision(bubblesIInLevel.getCurrentBubble(j),
enemiesInLevel.getCertainEnemy(i),
delta.desiredDelta).collidingWithEnemy()) {
```

Figure 3: Hypothetical solution to the Feature Envy design flaw.

This does not seem like an improvement at all, and only makes the code more difficult to follow. It also requires duplicate methods and variables from other classes, as the collisions do not store current enemies, bubbles, and other level elements.

Thus, due to the complexity of the required fixes, and the lack of actual improvement (due to identical bloatedness of the method and fixing design flaws by creating others), the decision was made to not fix this design flaw and to maintain the current implementation.

A possible way to have prevented the creation of this feature envy method was to have made collisions a general, implementable interface alongside internal class behavior of dynamic objects such as enemies, bubbles, and the player(s). However, the collision handling in the system cannot be generalized, as the way collisions between two elements are handled strongly depends on the types of elements and may therefore result in a variety of different behaviors. For example, a collision between a player and an enemy is radically different whether the enemy is in a bubble or not. If the enemy is not in a bubble, the collision would cause the player to die and the level/game to restart. If the enemy is encapsulated within a bubble, it (enemy and bubble) would be eliminated from the game.

3.2.2 Data Class Design Flaw (severity factor 1)

3.2.2.a Cause of the Design Flaw

The data class design flaw occurs when a class exposes a large amount of data to a public access, such as by having many getter methods. Our Level class was identified by inCode as being a data class. The design flaw error is triggered because the method contains six Array variables and the associated setters and getters, but does not perform any other functionality. This is due to the fact that the only method that actually performs a function, addElement(), only contains a large if-statement that acts as a type of factory and then references the LevelElement class to actually create and handle the artifact. Thus, no tasks are inherently performed within the class, causing it to be marked as a data class.

3.2.2.b Solution to the Design Flaw

In order to fix the data class design flaw issue, we could change the Level class sufficiently and implement functionality related to the levels within the class. However, in doing so, we would split the functionality for each level element among many classes, meaning that every level would be a composition of many classes, making a god class for the parsing and eventual display of the levels. Although the current Level class is, in its purest form, a data class, it takes many loose attributes and turns them into an essential, single-reference class. This takes care of many parsing and display issues, and allows the code be a bit easier to follow.

Another reason that a decision was made to not change the Level class was that it is integral to the code. As with the previous design flaw example, fixing this would make the class bloated and difficult to follow. It would also cause many repetitions of variables which would still require referencing to the original level element implementation. Thus, we would get redundant local variables and perhaps create other design flaws, such as another feature envy class/set of methods. Although the end user would not really notice any changes made to this class, the amount of effort and time require to make the class conform to the standards is too large for a project of this scope.

The only real way in which this could have been avoided was if the Level itself had access to all of the operations from within the class without having to depend on external variables. This implementation would, however, require the creation of an interface to handle level elements, which in turn would break certain design principles such as the Single Responsibility principle, as the level element classes in the current implementation also handle certain movement and behavior of the elements.

3.2.3 Message Chain as Possible Design Flaw

3.2.3.a Cause of the Design Flaw

A Message Chain design flaw is defined by the Intooitus inCode program as an operation that accesses a sequence of a data and allows it to move between multiple objects. This can cause problems when the data sequence changes, as it requires a change to the operations that are associated with this data sequence, which in turn can cause huge changes in the intermediate relationships between the objects.

3.2.3.a Solution to the Design Flaw

A situation in which this could have occurred within the the system is within the graphical user interface (GUI). The refactored user interface (from Sprint 3) makes use of a graphics context object which interacts with almost every active Level, Level Element, and Game object. If one of these objects changes, it requires a change to the entire graphics context. However, the User Interface itself also takes care of user input/key handling, meaning that it can change these values. Imagine if the user interface notices the player has completed the level, and it wishes to update the level, but there is still a fruit object within the level. The Game object would then change the fruit to be deleted before the next level is loaded, and then the user interface must show this change, but the Level must be loaded at that moment and not have a fruit object at the same location as the old one. This could cause a message chain and break certain elements in the user interface, not displaying them or displaying them incorrectly.

The reason the system does not have true message chaining is due to the fact that every cluster of essential objects/elements is generalized by a pseudo-god class. The level elements and modifiers are all generalized by the Level class. The Game score, positioning, and timing behavior are all generalized by the Game class. Although this could cause an issue as with the data class design flaw outlined above, but it does ensures that all classes include local instances of essential objects that are reset each cycle, resulting in a lack of true message chaining.