Documentation

Assignment 4 - Software Engineering Methods

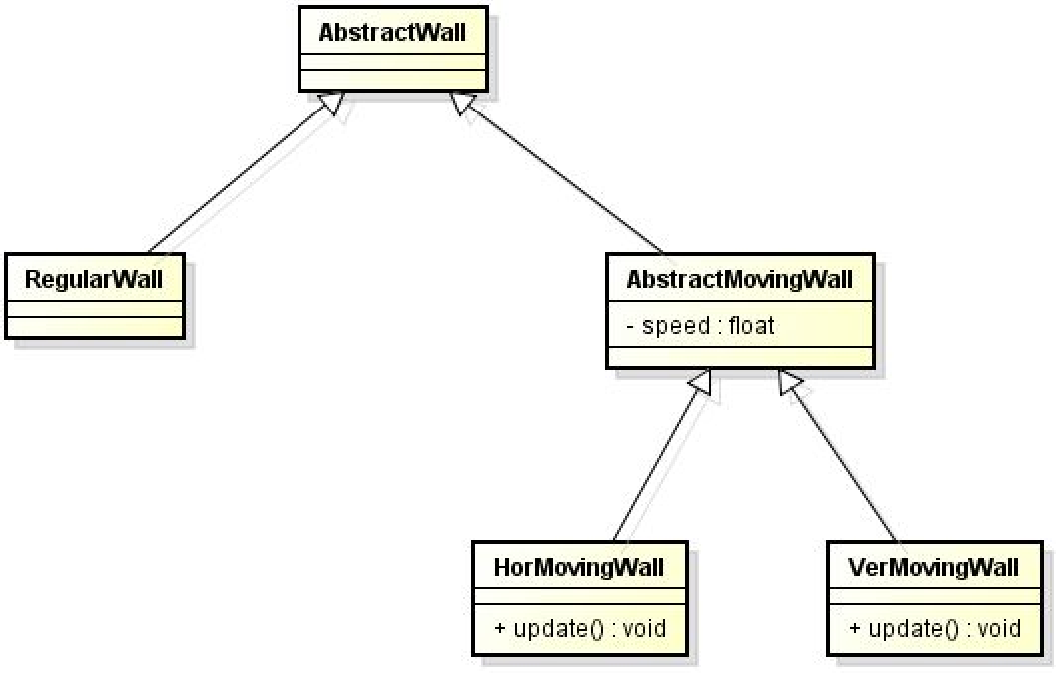
Game: Bubble Trouble
TA: Maiko Goudriaan

Exercise 1

1

In our sprint plan it says we were going to refactor Weapon and Pickup, by splitting the Weapon class into a Pickup object and a Weapon object. However, after putting some more thought into this, we didn’t really see any benefit to this, so we decided to leave it as is. Instead, we decided to refactor other things that were needed more. Those are listed below.

Moving walls



We implemented moving walls in our game. In the picture above this text, you can see the class diagram for the walls.

AbstractWall extends from AbstractEnvironmentObject (which is left out in the diagram for brevity) which handles the location and the image from the wall. The Wall as it was, was just a static image in the game, so it didn’t need any implementation, because its location and image are handled in a superclass.

The RegularWall still behaves like this. However, the AbstractMovingWall represent a moving wall. It has speed, which represents the number of pixels the wall will move per update. It also has an abstract update method, which needs to be implemented by its subclasses.

HorMovingWall and VerMovingWall are the two concrete implementations of the moving wall, where HorMovingWall moves horizontally (the x location is updated every cycle) and VerMovingWall moves vertically (the y location is updated every cycle).

When a wall has a frontal collision with another wall, the wall changes direction. But only the wall for which it was a frontal collision changes direction. So for example, when a horizontally moving wall collides with its front to the side of vertically moving wall, the vertically moving wall will keep going in the same direction, while the horizontally moving wall will change direction.

The requirements document can be found on the Git repo at the following path. `assignment04/RequirementsMovingWalls.pdf`

Resources

Our resources have completely been refactored. What we did, was loading every file that we needed manually. We changed that drastically. We made four folders (animation, img, music and sound) which contain files for the objects we create through our resources. For each folder, we do the following:

- We loop through all the files in the folder, and create an object with each file.
- This object is stored in a hashmap, where the relative path is the key to that object. For example, a key for an image could be: “img/someImage.png”. However, when there are subfolders, it also works.

For the animation it was a little more complicated though, because Animations consist of multiple Images. A restriction on the Animations is that all the images of one Animation should be stored in one folder. Also, the names of these images should be “1.png”, “2.png”, etc.

We also made sure that it works on both Windows and Linux systems.

Bubbles refactored

We refactored the bubbles (again). Now, instead of passing a BubbleFactory to a Bubble, which can then create two new Bubbles when it splits, we pass a list of Bubbles to it that need to appear when the Bubble is split. This makes it easier to make a Bubble that splits into four instead of 2 Bubbles (for example).

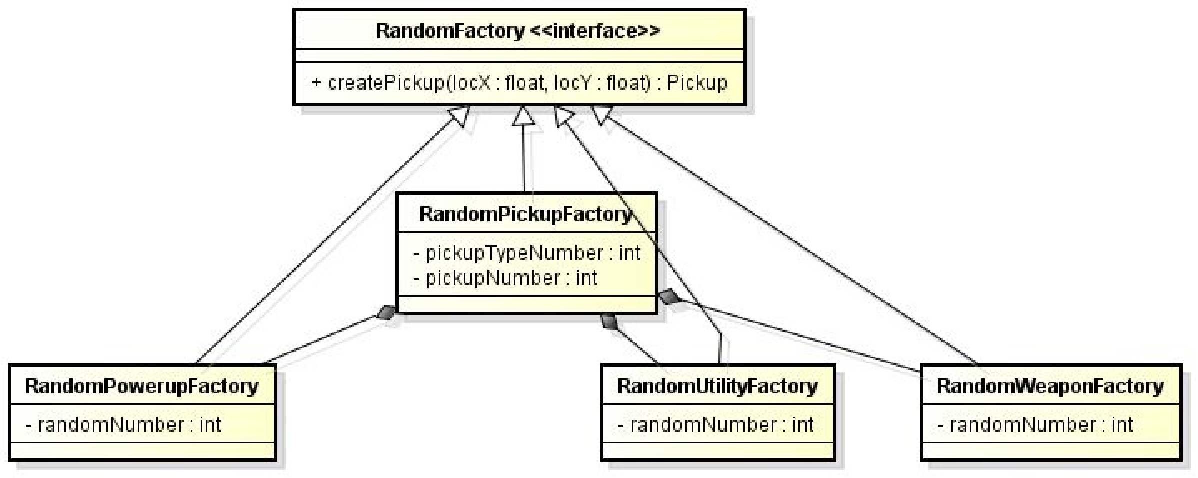
In AbstractBubble, we created two abstract methods: createNextBubbles() and initMaxVerticalSpeed(). These methods obligate concrete implementations of Bubble to return a list of Bubbles that will appear when the bubble splits and to initialize the maximum vertical speed of a bubble, respectively.

We didn’t make the split method final, because that would break our tests. Some tests use a mocked Bubble, and when the split method is made final, it cannot be altered and will have its regular implementation in a mock (which is not desirable).

We also kept the BubbleFactories. We felt that, while their usefulness may be not as high as we would have hoped, they still have some use. When a Bubble needs to be created, the creator doesn’t have to worry about which Image to use, the factory takes care of this. And when the creator doesn’t care about the location, this will be taken care of by the factory too.

We did adapt the factory however, by overloading the method createBubble(). It can now be called with two parameters, which represent the starting x and y location of the bubble. This way, a bubble can also be initialized with a x and y location different than 0, which is more convenient than first creating the bubble and the setting x and y location.

Pickup factories



We made factories to create random pickups. Above you can see the class diagram for this. Each of the factories implements the RandomFactory interface, which obligates them to implement a createPickup method.

The RandomPickupFactory creates two random numbers when it is created, “pickupNumber” and “pickupTypeNumber”. When createPickup is called, the first number determines whether the pickupfactory will actually create a pickup. If not, it will return null. The second number determines what kind of Pickup will be created (Weapon, Utility or Powerup). Then, one of the other random factories is used to actually create a new Pickup (it depends on the number which factory is used).

The other three factories all generate one random integer when they are created, within a range that is defined within the factory. When createPickup is called, this random number is used to determine which object will be created, and this object will be returned.

GameState

For the class GameState, an extra class is made, GameStateController. This class handles all the logic that was in GameState. This way, GameState really handles the view, while the logic is handled somewhere else.
Tests have also been written for both GameState and GameStateController.

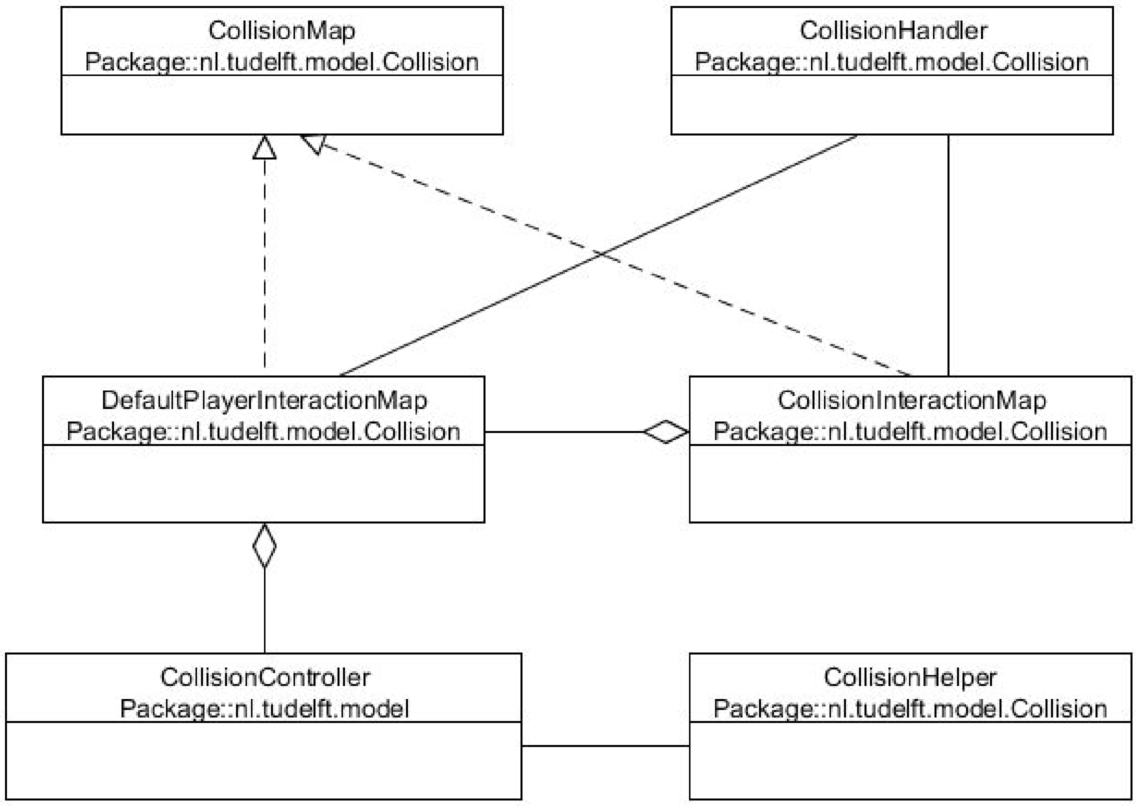
ShopState, Dashboard, GameEndedState and Pausescreen, KeyBindState

These classes underwent similar changes to GameState. The long update and render methods have been split up into shorter private methods for better visibility and maintainability.
ShopState, Dashboard and GameEndedState each have acquired a controller each to handle the logic of each class, this in order to conform to responsibility driven design.

Each class has gained tests to test the update and render methods, as well tests for the controllers, so the logic of each state.
The init of GameEndedState and ShopState however cannot be easily tested, because slick has problems creating objects with mocks. So every time a nullpointer exception will occur. Therefore we have decided to test that part by hand and write documentation about it:

The init method of ShopState has only 1 purpose, to initialize all mouseOverArea's of the state, which means all elements which need to be clicked, are clickable. The elements are the continue and buy button, the player indicators (for selecting a player) and all items, so you can select an item and buy it. The only way to test this is by starting up the game, giving each player $1000 and buying each item, for both players, just to be sure. When the right amount of money is withdrawn from the player when the buy button is clicked, you can determine if right item is selected. But this isn't accurate enough, since some items have the same price, so logs were implements which logged something if a MouseOverArea was selected, which is foolproof.

As fas as the GameEndedState init method goes, not only mouseOverAreas are being initialized, but also textFields and fonts. The same principles applies though, you can only check things so much by starting up the game. With logs, the size of the mouseOverArea's and TextFields are compared to make sure they match. When you can type into the textFields, you can be sure the fonts are initialized. This way everything in the init methods is tested.


CollisionMap
Package::nl.tudelft.model.Collision
CollisionHandler
Package::nl.tudelft.model.Collision
DefaultPlayerInteractionMap
Package::nl.tudelft.model.Collision
CollisionInteractionMap
Package::nl.tudelft.model.Collision
CollisionController
Package::nl.tudelft.model
CollisionHelper
Package::nl.tudelft.model.Collision

Collision Map

The collision has been changed a lot, we now use a collision mapping. The collision controller takes care of all the collisions, it consists of a DefaultPlayerInteractionMap which in turn consists of a CollisionInteractionMap. The DefaultPlayerInteractionMap takes care of looping over every collision, while the helper checks if there is a collision and the CollisionHandler actually handles the collision.

So the functionality hasn't changed much, but the collision checking no longer uses instanceof checks. Also the code has become a lot cleaner.

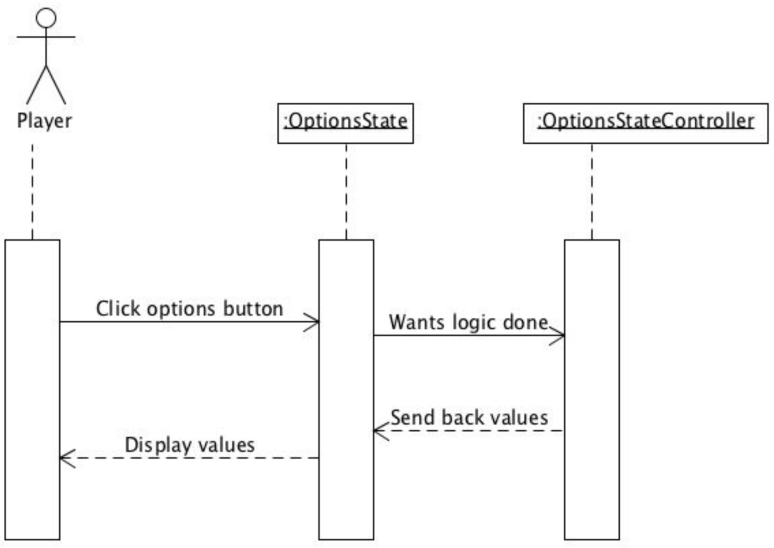
2

All the below states have been refactored. The main goals were to

- provide the ability to test the instructions.
- make the code more readable and therefore maintainable.
- Split logic (controller) from GUI (view).

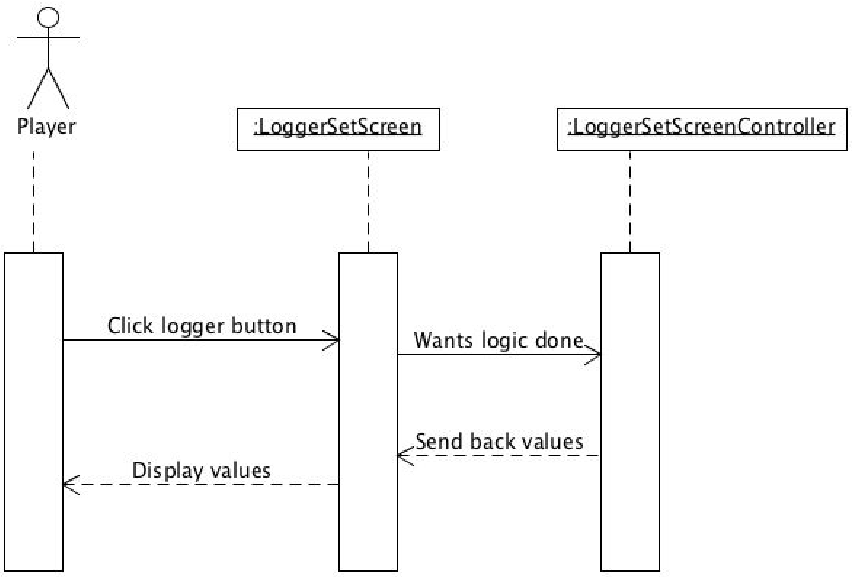
OptionsState

OptionsState has been completely reworked. It now consists of OptionsState and OptionsStateController. All logic which was performed in OptionsState is now performed by its controller. This was necessary to test them. The line coverage for both OptionsState and OptionsStateController is now 100%.



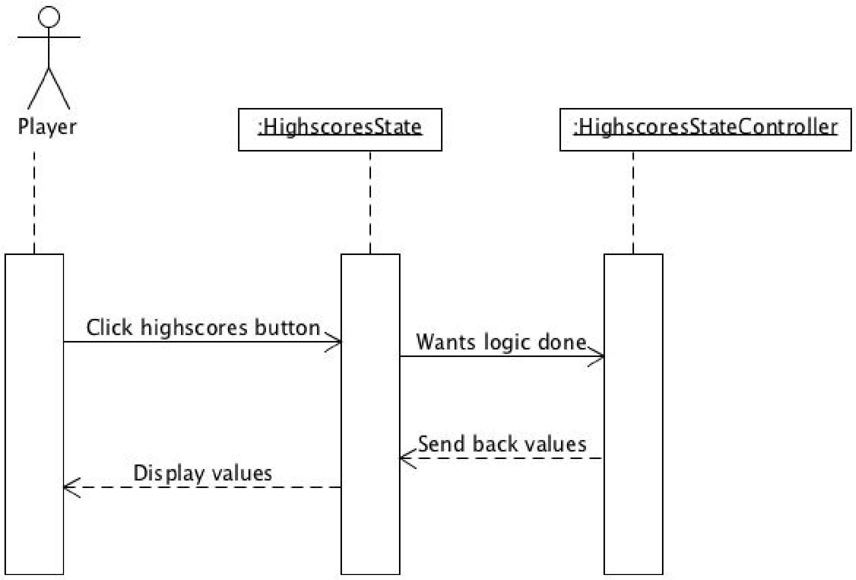
LoggerSetScreen

LoggerSetScreen has been completely reworked. It now consists of LoggerSetScreen and LoggerSetScreenController. All logic which was performed in LoggerSetScreen is now performed by its controller. This was necessary to test them. The line coverage for both LoggerSetScreen and LoggerSetScreenController is now 100%.



HighscoresState

HighscoresState has been completely reworked. It now consists of HighscoresState and HighscoresStateController. All logic which was performed in HighscoresState is now performed by its controller. This was necessary to test them. The line coverage for both HighscoresState and HighscoresStateController ar now 100% and 86%, respectively.



StartScreenState

Like the above, StartScreenState got its own controller, which contains all logic that the State must make.
The buttons are separated in the State itself and can be separately tested. They are contained in the state rather than the controller, because the StartScreen deals with many state switches, and the state has the proper API for this, which is inherited from the Slick2D BasicGameState class.

Exercise 2

1

The analysis file is located on the Git repository at the following path:
`assignment04/inCodeAnalysisCheckout1.3.result`

The analysis was run on the main folder of the maven project, thus excluding the test files. This is because we did not want the test code (which is ⅔ or more of the project) to influence the analysis, in terms of LOC / coupling / etc calculation.

Running it on the whole src gave less design flaws, so we picked the hard way out.

2

The summary shows to type of Design Flaws:

- Schizophrenic class (twice)
- God Class (once)

Schizophrenic Class

a

The Schizophrenic class flaw is caused by lack of adherence to the single-responsibility principle. This means that the class effectively performs two completely separate jobs (responsibilities). Because of this, half the methods are only used by some classes, while the other half is only used by some other classes, causing the schizophrenic appearance.

The Settings class was schizophrenic because it had two roles:

1. Storing and modifying the PlayerInput settings
2. Converting the objects in settings to JSON and vice versa.

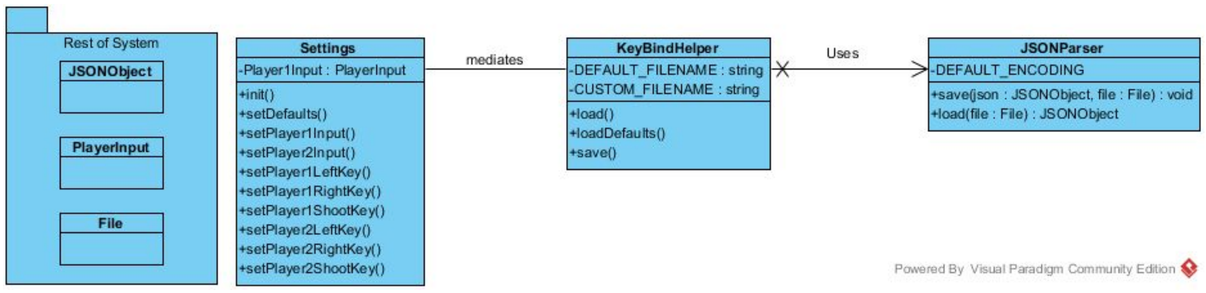
This was caused by the lack of responsibility in the KeybindHelper, which should have used the json and 'prepared' it into correct objects for the Settings class.

b

This flaw was fixed by redesigning the KeyBindHelper and Settings class, as well as adding a JSONParser class.
The responsibility of these classes can be described as follows.

- Settings is responsible for storing all settings (in this case only PlayerInputs are stored). It also must provide an API, mainly through getters and setters, to change the settings by external objects.
- KeyBindHelper is responsible for the translation process between Settings and its objects and a JSON in which it is stored. The helper knows the file locations and the structure of the objects and JSON. It is thus able to convert between these two, and uses the Settings API to update the Settings.

- JSONParser is a parser that knows specifically how to load and write JSON files. It provides a JSONObject when loading a file, and saves a JSONObject in a file, creating it and performing all actions needed during this write.



God Class

a

The God Class design flaw is caused by a class having too many responsibilities, and also pulling too much system intelligence to itself, such as using a lot of external data. It often is a bloated class with many non-cohesive methods and functionality.

The Game class was a god class because it is a big object that contains a lot of other objects (literally the rest of the game). While this is not a problem in itself, it also was responsible for finding and handling collisions. this means it had to:

1. Fill the QuadTree used for collision finding
2. Check all objects against each other using the quadTree
3. Call the collisionHandler when it detects a collision.

These tasks should be done by a dedicated class, but our choice to put it simply in Game, because it is a very top-level object, caused this design flaw.

N.B. Because of its God Class characteristics, it was also flagged as a schizophrenic class.

b

We changed it by making a class dedicated to performing the aforementioned actions. This class is contained in the GameStateController, and is updated every cycle. All the methods that had to do with collisions have been removed from Game, making it a class that is a lot cleaner than before.
We also removed the Modifiable interface from Game, because there was no use for it anymore. All objects that need to be deleted or added, are in Level.

Avoided Design Flaw: Tradition Breaker

The Tradition Breaker design flaw is caused by wrongly implemented interfaces or wrongly extended objects. It describes the act of implementing/extending, but only using half the functionality of the super class/interface. The rest of the functionality is then oftentimes implemented with empty methods.

We managed to avoid this design flaw by elaborate adherence to interface segregation. In the first design phases of our project, we decided that all game objects should have an update and a render method. However, we soon found out that the Weapon class is an example of an object that should not be rendered, only updated. We therefore created two interfaces: Updateable and Renderable. These were used to describe the above situation. We implemented Updateable for Weapon, but not Renderable. Had we not separated these interfaces, we would have had to implement an empty render method.

Result After Refactoring

After this iteration's changes, the above mentioned design flaws are cleared.
Sadly, a few new flaws were introduced.
The new analysis file is located on the Git repository at the following path:
`assignment04/inCodeAnalysisCheckout1.4.result`

The introduced flaws are *Data Class* and *Feature Envy* on Dashboard and Dashboard Controller respectively.

These flaws have low priority, and arguably are not flaws whatsoever.
Dashboard is a part of the GUI (view in MVC) that purely displays data.
It is therefore not surprising to note that it does not provide functionality for the attributes (the data) that it stores, as the description of the *Data Class* design flaw tells us.

Feature Envy is a design flaw on the setPlayerInfo() method in DashboardController.
This flaw and the *Data Class* flaw are closely related: When one class is a data class, another class often manipulates this data. The DashboardController is a Controller in the MVC pattern and is meant purely for this reason: to apply logic to the fields of the Dashboard.

Therefore, these flaws are not really flaws.
However, inCode does detect them, so there is some sort of code smell going on. This is because the dashboard - controller interaction is reversed.
Instead of dashboard asking its controller for help, the controller pushes modifications onto the dashboard.
Reversing this will most likely resolve the trigger of this design flaw (if it really adds to code quality is another question altogether).