Assignment 4

By group 30

1.1 & 1.2 can be found in the Documents map on Github.

Requirement Motivation

Only 2 players, because more would make it too chaotic and would bring too much downtime for a player, as he has to wait for animations and others' moves to finish when he or she is too late.

Players will battle it out on 1 identical board to give more sense of a competition, as they battle to be faster than the other.

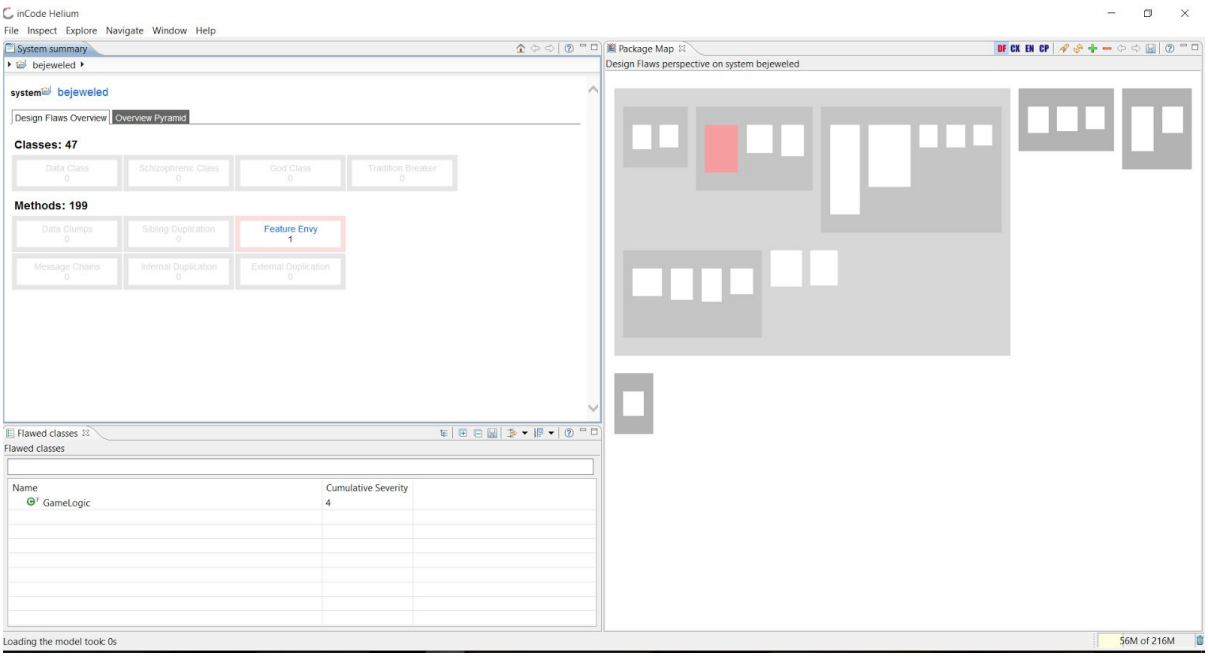
The game will have a fixed time to see which player has the most points at the end, again, to create a greater sense of competition and a sense of urgency.

2.1

We put the analysis file in Documents-->InCode-->Assignmenr4-2.1.

2.2

InCode could only find one flaw in our project. This is an envy flaw in GameLogic. So we can only answer 2.2.a and 2.2.b for one flaw. Below is a screenshot of our program runned by inCode.



a. The first flaw is a feature envy flaw, which was found by inCode. This flaw occurs in GameLogic in the method HandleMouseClicked. This method calls the Board class too many times. We chose to put the method in GameLogic because Board already had a lot of functionalities and we agreed to handle user input in the GameLogic class. Our mistake was to not pass the board altering responsibilities to the Board class.

b. We have passed the board-altering responsibilities from GameLogic onto the Board class. We now have a method call in the original mouseclick handler method in GameLogic to

Board, which now contains a new method to alter the board in the way that was done from GameLogic originally. Since there were conditions made from board attributes and used to alter GameLogic data, we let the Board class method give back a boolean, to simulate the conditional branching of the original method in GameLogic.

Alternative 2.2

There are certain factors in our project, which result in only having one flaw. First of all, a good practice is steady heartbeat. This is the biggest factor. We come together every week with a working release to discuss the project with a stakeholder (Bastiaan Reijm). The regular input/feedback ensures that we stay on the right track in order to satisfy the wish of the customer. This also helps us as developers to properly evaluate the product, which further improves the structure of our application. For example, after one of the first meetings we decided to not do our project in Java Swing, but in JavaFX, to make the animations easier and smoother.

Next, a second good practice is release-based working on a single application. Our project only has to be audible for Windows 7 or higher. So for each release the project has to work for only one application (Windows).

A third good practice is an experienced team. We have a fixed team of five people who were equally involved from the start of the project. This prevents having to brief new teammates and keeps team interaction constant, clear and smooth, which further prevents miscommunication and conflict.

These three good practices translate into a fourth good practice: Scrum. We use scrum and make a sprintplan each week, which we always review a week later. This way, we keep a good overview on what went good or what went wrong. We come together in meetings to properly discuss and propose changes. Discussions have led to a clear, intuitive and natural approach to Bejeweled in terms of objects modeled, which, for example, prevented data clumps.

Besides al these good practices there are more factors which can lead to a good project. One of which is UML, due to the use of UML our system becomes clear and easy to understand and see design flaws.

Another factor is responsibility driven design. We used responsibility driven design thoroughly in our project. We made requirements each week to clarify the adaptations we were going to make to the system. Furthermore, with use of CRC cards, we identified issues in our system. For example, classes with too many responsibilities or that were superfluous. We try to follow the single responsibility to a certain extent, so this made us decide to make new classes to properly distribute responsibilities from overloaded classes, preventing god

classes for example, and delete superfluous classes, which prevented data classes. A concrete example would be that during the first assignment, we made CRC cards, which made us decide to split up the Game class into GameLogic and GameScene. What we also identified from the CRC card for Board, is that it had too many responsibilities, which luckily hasn’t given any problems yet, but we are working on distributing them.