**Marking grid is attached at the end of this assignment.**

**ASSIGNMENT IN DETAIL: INTRODUCTION**

|  |
| --- |
| For this coursework, you will produce two versions of the game Hotels, described below. One version will have a Graphical User Interface (GUI) and the other version will have a command-line interface (CLI). The GUI version will be constructed according to the principles of Model View Controller, and the CLI version will use the same model. The two versions will from now on be called the GUI version and the CLI version. |

**ASSIGNMENT IN DETAIL: HOW TO PLAY HOTELS**

|  |
| --- |
| There are two players. Each player starts at GO on the board shown below and moves a coloured clockwise around the board by throwing a twelve-sided die and moving the appropriate number of places. Each non-blank square other than GO contains a hotel with an alphanumeric name eg A1. When a player lands on a square with a hotel they can buy it (if neither player owns it), be forced to pay an overnight fee as a guest (if the other player owns it) or pay money to upgrade the hotel (if they own it). |

**ASSIGNMENT IN DETAIL: FUNCTIONAL REQUIREMENTS**

|  |  |
| --- | --- |
| FR1 | The game will show whose turn it is and how much money each player has. |
| FR2 | Each square should display the counter (or counters) on it, the star-rating of the hotel (between 0 and 5) if there is one and who owns it, in addition to the name and price shown in the diagram above. |
| FR3 | If a player lands on a hotel owned by neither player, they should be offered the chance to buy it if the player can afford it. The cost of buying a hotel is the purchase price shown next to it on the square. |
| FR4 | If the player lands on a hotel owned by the other player, an overnight fee should be deducted from the first player (guest) and given to the second player (owner). The overnight fee for a hotel is 10% of the purchase price multiplied by the square of the star-rating. However, the fee is doubled if the owner owns both of the other two hotels in the letter group eg A3 and A2 as well as A1 and is halved if the guest owns either or both of the other two hotels in the letter group. |
| FR5 | If a player lands on a hotel that they own or have just purchased a hotel, they should be given the opportunity to increase its star rating if they can afford it. A hotel has a star-rating of zero when purchased (so the overnight fee of a guest is £0). The cost of increasing the star rating by one is 50% of the purchase price. The player can increase the star rating as many times as they like on a turn but the maximum star rating is five. |
| FR6 | Each player starts with £2000. If a player ends up with less than £0, the game will stop and the other player will be declared the winner. |
| FR7 | For ease of testing, the game should have a “cheat mode” in which the player can decide which of the twelve squares in front of their current position they are going to land on. |

**ASSIGNMENT IN DETAIL: NON-FUNCTIONAL REQUIREMENTS**

|  |  |
| --- | --- |
| NFR1 | The GUI version and CLI version should be two separate programs i.e. there should be two files each with a main method in them and which file is run determines which version is activated. |
| NFR2 | The GUI version must be constructed according to the principles of MVC, as restated below. Because of this requirement, code that belongs in the View but is placed in the Model will usually not be counted towards the marks for the View. Similar rules will apply for other misplaced code. |
| NFR3 | The CLI version will use the Model part of the GUI version directly without using the View or Controller; it should not define a new view or controller. |
| NFR4 | The code must be documented with asserts, unit testing, class diagram, comments as described in the marking scheme below. |
| NFR5 | The code must be of good quality as described in the marking scheme below. |

**FORMATIVE FEEDBACK OPPORTUNITIES**

|  |
| --- |
| We are giving you the opportunity to receive feedback on the design of your program. To receive this feedback, you need to upload a detailed UML class diagram of your code to your GitHub repository before 1pm on Friday March 3. As this is a formative feedback deadline, it will not be possible for you to seek deadline extensions. You will be given a few sentences of written feedback on your design within a week and a face-to-face explanation in the Week 6 practical. The Week 5 teaching session on Thursday March 2 will go through a worked example in order to help you produce the class diagram.  The class diagram should have all methods and attributes showing. A class diagram with insufficient detail or syntactically nonsensical or not realisable as an actual Java program will make it more difficult for us to give you feedback and will receive a low mark if submitted with the final report.  Further advice will be given in the module leader’s or practical tutor’s office hours or at the end of practicals or lectures. Your commits so far to the GitHub repository will be used to give advice. |

**SUMMATIVE FEEDBACK DELIVERABLES**

|  |  |
| --- | --- |
| **Deliverable description and instructions**  1. Your code will be uploaded to the GitHub repository as regular commits a long time in advance of the deadline. In addition  2. Upload a Word document or PDF which must:   * begin with the URL of your video   + uploaded to Google Drive   + permissions set so that anybody in Google@Brookes with the URL can view it * contain the class diagram * contain the code as text not images * be submitted to the Turnitin link provided.   3. You may be asked to attend a demonstration on Zoom to answer questions about the video and if you are then you must do so. | **Weighting out of 100%** |
| **Code for Model** (in repository and report)**:**  This should have an interface designed to be convenient for the Controller, View and JUnit class to use with no superfluous public methods, no references to those three classes and contain no GUI code. It may consist of several classes but there must be a class called Model or similar that provides the interface and this class should extend Observable. | **15%** |
| **Code for Controller** (in repository and report)**:**  This should forward only valid requests to the Model, querying the Model if necessary to find out if the request is valid, and must also enable / disable any buttons needed to implement the above functional requirements. It must have no GUI code, though it may send messages to the View. | **5%** |
| **View using the Swing framework** (in repository and report)**:**  It should implement Observer and therefore have an update method that is called when the Model changes. This will be marked according to how many of the functional requirements have been met. | **10%** |
| **CLI version of program, using the Model** (in repository and report)**:**  A single class using the Model which when run implements the same functional requirements as the GUI except that all of the information about each square should be recorded in a separate line of text rather than mimicking the ring structure shown in the diagram. | **10%** |
| **Specification of Model with asserts** (in repository and report)**:**  This should include invariants for the class as well as pre and post conditions for each public method in the model and should preferably be in logic with partial credit for descriptions in English. | **10%** |
| **Unit testing of the Model in JUnit** (in repository and report)**:**  There should be three tests, significantly different from each other. You should explain in the comments the scenario i.e. the situation you are testing for. You should use write (and then call) methods for the Model that set it into the state desired for the test. It should be easy to see what state the Model is being set to by reading the code for the unit tests. Your Model may use a separate Board class but the testing should be of the Model class and the specification should be applied to that class also. | **10%** |
| **Code quality** (in repository and report)**:**  Use of the code quality practices described in Lecture 1, plus the additional practices of light relevant commenting and correct formatting. Short elegant programs are preferred, and code smells are to be avoided. Note that high marks for this category will only be possible if the GUI fulfils most of the requirements. | **10%** |
| **Class diagram** (in report)**:**  This should show how the Model, View and Controller are related to each other, as well as how they interact with library classes such as Observable. Simplicity and clarity will be rewarded. It will be marked according to its accuracy as a representation of the program. | **10%** |
| **Video** (URL in report)**:**  Five-minute video that shows you using the program. It will be marked according to timing, presentation and how well you show that you have met the FRs and NFRs in both versions. If the marker has any questions about the video, then you will be invited to a Zoom presentation. | **10%** |
| **GitHub Commits** (in repository)**:**  You must make regular commits to the GitHub repository that you will be given. | **10%** |

This is an individual piece of work and you will have to work on your own and submit your own original attempt at the assignment. Any code that has been copied from any source (e.g. Stack Overflow, online tutorial, textbooks, other students etc.) must be properly referenced to avoid any suspicion of plagiarism. If you need help you can always ask for advice and guidance from the module leader by email; online sessions can be arranged for further clarification.

**The coursework will be marked according to the following rubric:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | F | P | B/C | A+ |
| Model  15% | only basic functionality implemented or slightly more than basic but references to View or Controller or superfluous methods | no superfluous methods and no references to View or Controller but only the basics of functionality implemented | no superfluous methods and no references to View or Controller but only the basics of functionality implemented | convenient to use with no superfluous methods, all required functionality and no references to View or Controller, extends Observable, calls setChanged and notifyObservers |
| Controller  5% | zero of the requirements: only valid requests, querying Model first, enables/disables buttons without GUI code | one out of only valid requests, querying Model first, enables/disables buttons without GUI code | two out of only valid requests, querying Model first, enables/disables buttons without GUI code | only valid requests, has references to both Model and View, converting UI interactions into methods to change the Model, querying Model first, enables/disables buttons without GUI code |
| GUI View  10% | no view update method or update method implementing very few of the FRs | update method in view implementing some of the FRs | update method in view implementing most of the FRs | update method in view implementing all the FRs, uses Swing, has Model and Controller as attributes, displays board and allows Controller to change the view e.g. enable/disable options, implements Observer and calls addObserver |
| CLI class  10% | CLI version implementing very few of the FRs | CLI version implementing some of the FRs | CLI version implementing most of the FRs | CLI version implementing all of the FRs, using same Model as the GUI version, but no Controller and is demonstrated on the video |
| Specification of Model with asserts  10% | a few pre/post conditions described in English | suitable pre/post conditions for most public methods but in English | suitable pre/post conditions for most public methods expressed in some logic | suitable pre/post conditions for all public methods and class invariants all expressed as statements of formal logic |
| Unit testing of Model with JUnit  10% | one test with the scenario poorly described or not at all | tests all essentially similar or only one or two or scenario being tested poorly described | third test not significantly different or scenario being tested not described with sufficient care | three significantly different tests of the model with all scenarios exactly described and with all inputs satisfying the preconditions |
| Code quality practices  10% | most code quality practices not observed or program is too small to demonstrate good code quality practices | some code quality practices observed but many not or program is too small to demonstrate good code quality practices | most code quality practices observed but some clearly not | all code quality practices observed including light correct commenting, suitable identifier names (constants, methods, classes etc) in appropriate cases, indentation, lack of code smells (long methods, repeated code, lack of modularity) |
| Class diagram  10% | Inadequate class diagram with serious mistakes in attributes and relationships between classes | Adequate class diagram with mistakes in both attributes and relationships between classes | Good class diagram with only a few mistakes in attributes, visibility or relationships between classes | Excellent class diagram with all attributes indicated with correct visibilities and correct relationships between classes all shown |
| Video Presentation  10% | Very poor presentation with insufficient coverage of FRs and NFRs, poorly presented and overly long | Passable presentation covering FRs or NFRs or well-presented or at least appropriate length | Quite good presentation but missing some details of FRs and NFRs or poorly presented or overly long | Excellent presentation with full explanation of most FRs and NFRs, well presented and within time limit |
| GitHub 10% | Some minimal code in the repository | Updates to the repository but mostly for the last few days before submission | Evidence of regular updates to the repository over a period of several weeks | Clear pattern of regular updates over a period of two months according a clear version policy with informative commit messages |