# CHC6186 Advanced Object-Oriented Programming

# Coursework

For this coursework, you will produce in Java two versions of the game **Weaver**. 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.

## Learning Outcomes

This coursework will assess the following learning outcomes.

* Create a software artefact by applying the methodologies of advanced object-oriented programming to a requirements specification
* Consult on-line code libraries to find the classes and methods most appropriate for solving a problem
* Create appropriate documentation that clearly communicates the intended behaviour of a program

This coursework is worth 50% of your module mark; the remaining 50% comes from your exam.

## How to Play Weaver Weaver is a word ladder game where players must transform one word into another by changing one letter at a time, with each intermediate step also being a valid word.[[1]](#footnote-1) Players start with a given starting word and aim to reach the target word in as few steps as possible. Each step must result in a valid word of the same length as the starting and target words. See figures 1, 2 and 3.

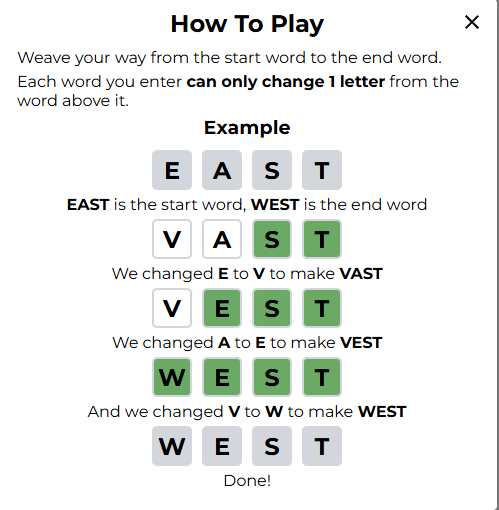


Figure 1: How to play weaver

For this coursework:

1. The starting word and target word will be fixed at four letters.
2. Players will have unlimited attempts to reach the target word.
3. Each attempt must produce a valid intermediate word.
4. If the player successfully reaches the target word, they win.
5. Feedback is provided for each word entered (see figure 3):

* Letters in the correct position are highlighted in green.
* Letters not in the target word are shown in grey.

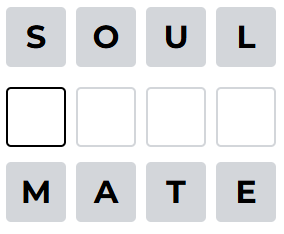


Figure 2: Start state

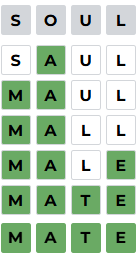


Figure 3: End state

You must implement the game in **Java**. The website is implemented in Javascript. Any attempt to submit Javascript will receive a **mark of zero** and any Java based on the website’s Javascript will be treated as **plagiarism** in the normal way. The website colours may be used.

## Functional Requirements

For greater clarity, the description of the GUI and the CLI versions of the game can be summarised in the following list of functional requirements.

|  |  |
| --- | --- |
| FR1 | For the GUI version, a confirmatory message or a message box should be displayed to indicate whether the player has won (transformed the starting word into the target word), even though the game status is clear from the tile colouring on the last filled row. |
| FR2 | For the CLI version, a confirmatory message indicating the player has won is required. |
| FR3 | The behaviour of the program shall be controlled by three flags:   * One flag should, if set to true, cause an error message to be displayed if the word is not a valid intermediate word. If set to false, no error message should be displayed. * Another flag should, if set to true, display the path (not necessarily the best path) from the start word to the end word for testing purposes. * A third flag should, if set to true, cause the starting and target words to be randomly selected from a dictionary (dictionary.txt). If set to false, these words will be fixed, as shown in figure 2. |
| FR4 | Both GUI and CLI versions of the program should allow players to input their intermediate words. |
| FR5 | The Model should load a list of valid words from a fixed location (from one provided file, dictionary.txt). This list will serve as the potential valid intermediate words for the player. |
| FR6 | The graphical user interface (GUI) should present a keyboard (refer to figure 4), with the letters displayed in dark grey. There is no need to alter the colour of the letters as was done in the game. However, it is necessary to update the colour of the end word in the same manner as the game does.  When using the GUI version of the program, the user can enter words either by typing on the physical keyboard or by clicking on the virtual keyboard with the mouse. Both methods of input should be supported for user convenience.  The CLI should indicate the start word, the end word, and any intermediate words, if applicable. |
| FR7 | The GUI version should have a button to request a reset of the game (return to the original state), which will be enabled only after the first valid input has been made. The GUI version should also have a button to request a new game, regardless of whether input has been made. This is not required for the CLI version. |

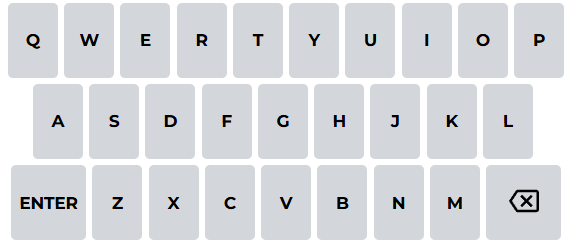


Figure 4: The keyboard of the game weaver

## Non-functional Requirements

The following non-functional requirements also apply

|  |  |
| --- | --- |
| 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. 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; nor should it define a new View or Controller. |
| NFR4 | The code must be documented with assert statements, unit testing, class diagrams, docstrings and comments as described below. |
| NFR5 | The code must be of good quality as described in the marking scheme below. |
| NFR6 | The flags mentioned in FR3 should be in the Model. It is necessary for them to be changeable at runtime. (e.g., using three buttons or other controls). |
| NFR7 | The flags mentioned in FR3 should be in the Model. It is not necessary for them to be changeable at runtime. |

## Marking Scheme (See rubric as well).

|  |  |
| --- | --- |
| Model. 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 two 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. File reading should also be done in the Model. A high mark will be earned for a Model that implements all the required functionality and respects all these constraints. A pass mark will be earned for a Model that implements only some of the required functionality or fails to respect these constraints. | 20% |
| Controller. 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 buttons as described above in the functional requirements. It must have no GUI code, though it may send messages to the View. A high mark will be given to a controller that respects all these constraints and a pass mark will be given to a controller that respects only some of them | 10% |
| View of GUI version using the Swing framework. 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. A high mark will be given to a view that implements all the requirements and a pass mark will be given to a view that implements only some of them. | 10% |
| CLI version of the program, using the Model. | 10% |
| Specification of Model with asserts. This should include invariants for the class as well as pre and post conditions for each public method in the model. This will be marked according to how many of the relevant conditions are included and whether the ones that are included are correct. Partial credit will be available for describing them in English. A high mark will be given to a specification that includes all the relevant constraints. A pass mark will be given to a specification that includes only a few of them. | 10% |
| Unit testing of the Model in JUnit. There should be three tests, significantly different from each other. You should explain the scenario in comments, i.e., the situation you are testing for. You should write (and then call) methods for the Model that set it into the desired state for the test. It should be easy to determine the state of the Model by reading the code for the unit tests. A high mark will be awarded for significantly different tests that are easy for the marker to interpret. A passing mark will be given to unoriginal second or third tests, or to three tests that are difficult to understand. Your Model may use a separate Board class, but the testing should focus on the Model class, and the specification should also apply to that class. | 10% |
| Code quality. 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. A high mark will be awarded if all the practices are observed and a pass mark will be awarded if only some of them are. | 10% |
| Class diagram. 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 reward. It will be marked according to its accuracy as a representation of the program. A high mark will be awarded for an accurate diagram and a pass mark will be awarded for a less accurate diagram. | 10% |
| Video presentation that shows you displaying the code and 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. | 10% |

## Coursework Management

Students are required to regularly back up all electronic files related to their coursework. Ensure that backups are organized and not solely dependent on a single source of storage, as the loss of files could severely impact your progress. Computer crashes and disk failures will not be accepted as valid excuses for missing or late submissions. Students must use an industry-recognized version control system, such as Git, to track versions of their software development work throughout the module. Additionally, the repository must remain private, both before and after submission. Keep the repository confidential to protect your work.

## Submission

**Requirements**

1. Your submission should contain **three files** (.pdf, .zip, and .mp4).
2. The first file is a **PDF**document containing the **design**(represented by a **UML**class diagram), screenshots of the **testing**, and printed copies of the Java **implementation**code‌.
3. The second file is a .zip file with the Java project.
4. The third file is a .mp4 video that is less than **1 GB**. If the video is not viewable, it will not receive marks. The video must be a maximum of **five minutes** long during which you must display most of the relevant functionality and refer to your code. Any recording software can be used so long as it captures your screen and your voice.
5. Additionally, you are required to regularly upload your code to a repository (**Note: such as GitHub, and you must set it to private.**) as per the university counterpart's requirement . Provide the link and screenshots of the repository and the record of the commits in the PDF document. To ensure against plagiarism, **frequent** and **early** **commits** are required. For example, work submitted as one commit on the day of the deadline will be penalized.
6. The PDF document is the version that will be marked, but the .zip and .mp4 are requested so that we may run the code.

**File Naming Convention**

You must save the files with the following names:

* + {YourStudentNumber}-coursework.pdf
  + {YourStudentNumber}-coursework.zip
  + {YourStudentNumber}-coursework.mp4

For example:

* + 202107081314-coursework.pdf
  + 202107081314-coursework.zip
  + 202107081314-coursework.mp4

**Submission Deadline:**

You must upload from the student website (student.zy.cdut.edu.cn) before **TBD**

Some students will be selected to give a Zoom presentation, after the exam period. If you are asked to give a Zoom presentation then you must do so.

## Formative Feedback

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 student website before **TBD**. As this is a formative feedback deadline, it will not be possible for you to seek deadline extensions. You will be given a short amount of written feedback on your design within a week. The Week 5 teaching session 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. In addition, you should indicate which methods call which other methods. 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.

## Academic Conduct

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. Refer to the Module Handbook for further information on this. If you need help you can always ask for advice and guidance from the module leader and tutor by email; online sessions can be arranged for further clarification.

# Rubric The work shall be marked according to the following rubric.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | D | C | B | A |
| Model | 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 | 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 | 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 | 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 the FRs, using same Model as the GUI version, but no Controller and is demonstrated on the video |
| Specification of  Model with asserts | a few pre/postconditions 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 | 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 | most code quality practices not observed | some code quality practices observed but many not | 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 | Inadequate class diagram with serious mistakes in attributes, methods and relationships between classes | Adequate class diagram with mistakes in attributes, methods and relationships between classes | Good class diagram with only a few mistakes in attributes, methods, visibility or relationships between classes | Excellent class diagram with all attributes and methods indicated with correct visibilities and correct relationships between classes all shown |
| Video Presentation | 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, referencing the code, well presented and within time limit |

1. <https://weavergame.org/> [↑](#footnote-ref-1)