Quiz Program

A teacher provides a Quiz Program which helps students to prepare their exams. The teacher provides a set of *Questions*, each *Question* having an *id* (e.g. 1) a question *text* (e.g. "Which of the following functions belong to the python built-ins? (a) id (b) ident (c) identity"), and one and only one *correct answer* (e.g. "b").

1. When the program starts, the list of questions is read from a text file *questions.txt*. The program has three modes/states *idle*, *quiz* and *review*. Initially the program enters the *idle* state.
2. When the program is in the *idle* state, the user can enter the *quiz* or *review* commands and the program enters the *quiz*, respectively the *review* state.
3. In *quiz* state, the program is configured to show a fixed number of questions (e.g. 3). The program will select the questions randomly. Immediately after entering in this state, the program shows the first question (only the text).
4. In *quiz* state, the user can navigate between questions using *next* and *prev* commands, and the program shows accordingly that question.
5. In *quiz* state, the user can answer to the current question using the command *answer option*, where option is a string. The program informs the user if that option is the correct one, and shows the number of correct answers/the number of answers given by the user. Only one answer may be given to a question.
6. In quiz state, when the user has answered to all questions, the program will automatically enter the *idle* state.
7. When the program exits the *quiz* state, the questions shown to the user together with his/her answers will be added to a *quizes.txt* file. This file will collect questions and answers from all quizes made by the user.
8. In *review* state, the program allows the user to see all questions and his/her answers (those saved in *quizes.txt*), including the correct answers. Immediately after entering in this state, the program shows the first question, answer, and correct answer.
9. In review state, the user can navigate between these items using *next* and *prev* commands. The user can exits this state using *exit*.

Non-functional requirements:

- Put UI, domain (including controllers/services), and repository elements in separate modules.
- Validate the type of the values entered by the user in UI.
- Validation (logical errors) will be defined within the domain – e.g. for a given question, only an answer may be given during a given quiz.
- Define validation algorithms in separate classes.
- Object validation will be applied/called by repositories and the errors will be reported via exceptions.
- Propagate exceptions from domain and repository to UI.
- Handle exceptions in UI.
- The program will not crash during execution when user enters invalid commands.
- The program will not crash during execution when the files are missing or corrupted.

Execution: 4.5p; Non-functional requirements: 4.5p;

Hangman

You have to implement a console-based variation of the classical Hangman game. The computer will select a Sentence that the user can attempt to guess, letter by letter. Each time the user guesses a correct letter, the computer will fill it in the sentence at the correct positions. In case the letter does not appear, the computer will fill in a new letter in the word "hangman", starting from the empty string. The game ends when the user has guessed the sentence (user wins) or when the computer fills in the "hangman" word (user loses). Program functionality is broken down as follows:

1. Add a sentence. While not in a game, the user can add a sentence **[1p]**. Each sentence must consist of at least 1 word. Every word in the sentence must have at least 3 letters. There can be no duplicate sentences **[1p]**.

2. Start the game. When the user starts a game, the computer selects one of the available sentences and displays it on screen, hangman-style. This means that the computer reveals the first and last letter of every word, as well as all the apparitions of these letters within the words **[2p]**. The sentence selection is random **[1p]**.
   e.g. for the sentence "anna has apples", the computer reveals "a _ _ a has a _ _ _ _ s".

3. Play the game. The game consists of several rounds. In each round, the user proposes a letter. If the sentence contains the letter, the computer reveals where these letters appear within the sentence. If the sentence does not contain the letter, or the user previously proposed the letter the computer will add a new letter to the word "hangman", which is displayed to the user **[3p]**.

4. *Game over. The game is over when the sentence is correctly filled in (user wins), or when the computer fills in the word "hangman" (user loses).* **[1p]**

Non-functional requirements:

- *Implement a layered architecture solution with Repository, Controller and UI.*
- *Functionalities without tests or specification in the Repository or Controller layers will be graded at 50% value.*

Observations!

- *Sentences are loaded from/saved to a text-file that must initially hold at least 5 entries.*

sentences.txt example:

```
anna has apples
patricia has pears
cars are fast
planes are quick
the quick brown fox jumps over the lazy dog
```

Gameplay example:

```
Output: "a _ _ a has a _ _ _ _ _ s" - ""
User guess: "m", output changes to: "a _ _ a has a _ _ _ _ _ s" - "h"
User guess: "n", output changes to: "anna has a _ _ _ _ _ s" - "h"
User guess: "m", output changes to: "anna has a _ _ _ _ _ s" - "ha"
User guess: "a", output changes to: "anna has a _ _ _ _ es" - "ha"
User guess: "x", output changes to: "anna has a _ _ _ _ es" - "han"
User guess: "t", output changes to: "anna has a _ _ _ _ es" - "hang"
User guess: "p", output changes to: "anna has app _ es" - "hang"
User guess: "l", output changes to: "anna has apples" - "YOU WON!"
```

Fawlty Towers

The Fawlty's need your help! Create a console-based application that helps manage the reservation process in their hotel during 2018. The hotel has 10 rooms. Rooms can be *single rooms* (1 person), *double rooms* (2 persons), or *family rooms* (4 persons). Each **Reservation** is identified by a unique **number** and includes the **room number**, a non-empty **family name**, number of guests, **arrival** and **departure dates**. Your program should include the following functionalities:

1. **Room number** and **room type** information must be read from a text file at program start-up. The hotel must have at least 2 of every room types **[1p]**.
2. Create a room reservation:
   - The user enters the guest's family name, room type, number of guests, arrival and departure dates. In case of a validation error (no family name, invalid arrival/departure, less than 1 or more than 4 guests), the program provides an error message **[1p]**.
   - The program creates a *reservation number* (unique, random, exactly 4 digits **[1p]**) and select a room from the available rooms of that type. If there are no rooms of the desired type available during the reservation dates, the program displays an error message **[1p]**.
3. Delete reservation. The user enters the reservation number. If the reservation exists, it is deleted. If it does not, the program displays an error message **[1p]**.
4. Show available rooms. Given a time interval in the form ***dd.mm – dd.mm*** (e.g. 25.06-12.07) the program will display all rooms available for reservation (both room number and type) **[1p]**.
5. Monthly report. The system will display an ordered list of months, sorted descending by the number of reservation days (e.g. a week-long holiday is 1 reservation, but counts as 7 reservation nights) **[1p]**. Make sure to handle the case of reservations that start during one month but end during the next one (e.g. from July 30th to August 5th) **[1p]**.
6. Day of week report. The system will display a days of week list (Monday – Sunday), ordered descending by the total number of reserved rooms for that day, for the duration of the entire year **[1p]**.

Non-functional requirements:

- Implement a layered architecture solution with Repository, Controller and UI.
- Provide specification and unit tests for Repository/Controller functions related with functionalities 2 and 3. In case specification or tests are missing, the functionality is not graded.

Observations!

- Room numbers and types are loaded from a text file that holds 10 entries.
- Reservation information is loaded from/saved to a text-file that must initially hold at least 10 entries.

reservations.txt example:

```
1223, 1, Albu, 2, 15.02,19.02
4550, 5, Negru, 4, 13.02,19.02
0118, 2, Popescu, 1, 16.02,21.02
9948, 3, Ionescu, 3, 13.02,15.03
4445, 4, Berindean, 2, 12.02,19.02
4531, 1, Smith, 1, 20.02,18.03
7939, 1, Marian, 2, 19.03,01.04
8890, 3, Liliac,4, 20.03,22.03
9567, 3, Bodnar, 4, 23.03,25.04
8056, 4, Vasilescu, 2, 20.02,27.02
```

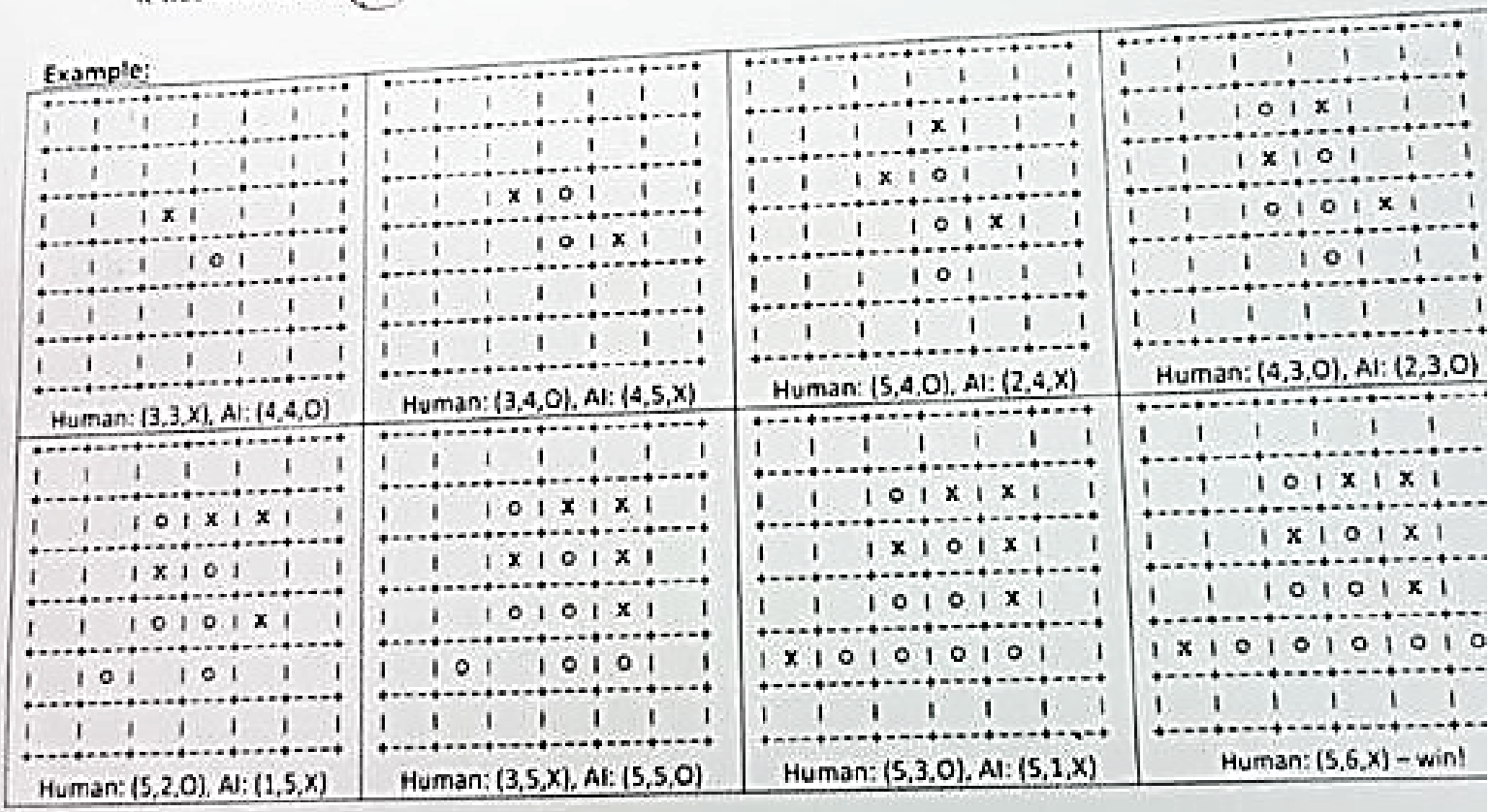
Default 1p

Order and Chaos (1st variant)

Order and chaos is a pen and pencil strategy game played on a 6x6 board. The two players are *order* (human player) and *chaos* (computer player), and they take turns placing **X**'s and **O**'s on the board. Order wins by having 5 consecutive pieces of the same type in a row, column or a diagonal. Chaos wins when the board is filled without this happening. Unlike other games, players can pick which symbol to place every move. Implement a console-based program that allows a human to play against the computer. The program must work as follows:

1. When the program is started, the empty board is displayed **[1p]**.
2. Players take turn in placing symbols on the board, with order having the first move **(1p)**.
3. User input is validated and an error message shown when trying to make an illegal move. **(1p)**
4. The computer makes random, but valid moves **(1p)**. The computer stops the human player's 1-move wins, whenever possible **[2p]**.
5. If there are 5 consecutive symbols of the same type in a row, column or diagonal order wins and the game is finished **(1p)**. **(NB!** This can also happen after chaos moves)
6. If the board is full, chaos wins and the game is finished **(0.5p)**.
7. The game can be saved to a file at any time during play **(0.5p)**. When starting the program, you must choose between loading an existing game and starting a new one. Loading a game causes it to continue from where it was interrupted **(1p)**.

Example:



Non-functional requirements:

- Make sure the program does not crash, regardless of user input.
- Implement a layered architecture solution.
- Provide specification and tests for functionality 4. In case specification or tests are missing, the functiona is not graded.

Default 1p.

QuizMaster 2000

You have to implement a program that allows creating and solving quizzes. Each quiz consists of several multiple-choice questions. Each question has an *id*, a *text*, 3 *possible answers*, one and only one *correct answer*, as well as a *difficulty level* (one of easy, medium or hard). The program will access a master question list and will allow the user to create quizzes as well as to solve previously created quizzes. Program functionalities are detailed as follows:

1. The user can add a question to the master question list **[2p]** using the following command:
   add ***<id>;<text>;<choice_a>;<choice_b>;<choice_c>;<correct_choice>;<difficulty>.***
   e.g. add *5;Which of the following numbers is prime?;5;56;75;5;easy.*

   In case the command is not provided in the correct format, the program will provide an error message **[1p]**.

2. The user can create a new quiz using the following command: **create** ***<difficulty>*** ***<number_of_questions>*** **<file>**
   The meaning of the parameters is as follows:
   - ***<difficulty>*** is one of easy, medium or hard. A quiz is easy if at least half its questions are easy. The same for medium and hard quizzes.
   - ***<number_of_questions>*** the number of questions in the quiz.
   - **<file>** the name of the file where the quiz is saved.

   e.g. create *easy 10 quiz1.txt*

   If the command is run successfully, the quiz is saved to the output file **[2p]**. If the command format is not correct, or there are not enough questions in the master list to create the quiz (e.g. not enough easy questions) the program will provide an error message. The quiz will not be created **[1p]**.

3. The user can take one of the saved quizzes using the following command: **start** ***<file>***
   e.g. start *myquiz.txt*

   Taking a quiz means providing answers to all its questions one at a time. The program will display the quiz's questions one by one, starting with the easiest questions first. For each question, the program will also display answer choices. Once the user provides an answer, the program will move to the next question **[2p]**.

4. The quiz is complete once every question is answered. At this point, the program calculates the user's score. The score is calculated by giving 1 point for each correct answer to an easy question, 2 points for a medium difficulty question and 3 points for correctly answering a hard question **[1p]**.

Example master question list	Quiz generation command:
1;Which number is the largest;1;4;3;4;easy 2;Which number is the smallest;-3;3;0;-3;easy 3;Which number is prime;2;32;9;2;easy 4;Which country has the largest GPD;Brazil;China;UK;China;medium 5;Which is not a fish;carp;orca;eel;orca;medium 6;Name the first satellite;Apollo;Sputnik;Zaria;Sputnik;medium 7;Which Apollo mission did not make it to the moon;11;13;17;13;hard 8;A mole can be;animal;quantity;both;both;hard 9;Name El Cid's horse;Babieca;Abu;Santiago;Babieca;hard 10;The Western Roman Empire fell in;654;546;476;476;hard	**create** hard 6 hardquiz.txt hardquiz.txt 1;Which number is the largest;1;4;3;4;easy 4;Which country has the largest GPD;Brazil;China;UK;China;medium 5;Which is not a fish;carp;orca;eel;orca;medium 8;A mole can be;animal;quantity;both;both;hard 9;Name El Cid's horse;Babieca;Abu;Santiago;Babieca;hard 10;The Western Roman Empire fell in;654;546;476;476;hard

Non-functional requirements:

- Have at least 10 entries in the master question list file.
- Make sure the program does not crash, regardless of user input.
- Implement a layered architecture solution with Repository, Controller and UI.
- Provide specification and tests for the non-trivial methods in the Repository and Controller layers. Functionalities without specification or tests are graded at 50% value.

Scramble!

You have to implement the Scramble! console-based game. The objective of the game is to order the scrambled letters of a sentence into the correct order, by switching the places of pairs of letters, one pair at a time. In each word, the first and last letter will always be correct. There can be single-word sentences.

e.g. Given the word `'salembre'`, you can make the following switches: `salembre` → `eclembre` → `scramble`

Program functionality is as follows:

1. A new game starts each time the program is started. The word or sentence player is selected randomly by the program from one of the entries stored in a text file [1p]. The program scrambles the word or sentence in the following way: the first and last letter of each word are kept in their place, the rest are shuffled randomly (it is possible to move letters between words). The game starts with a score equal to the number of letters in the given word or sentence, not counting spaces. The scrambled sentence is printed at the console [2p].
   e.g. The sentence `'Dream without fear'` can be shuffled to `'Doahm woeiott fear'`. Its score is 16.

2. The user can swap two letters using the following command: `swap <word1> <letter1> - <word2> <letter2>`. The word/letter parameter pairs illustrate the indices of the word and letter to be swapped. After every swap, the updated sentence is printed to the console [2p]. In case the command is not complete, one of the indices supplied is incorrect or indices include the first or last letter of a word the program will provide an error message [1p].
   e.g. 'swap 0 1 – 0 3' in the previous example will swap the places of letters 'a' and 'c' (`salcmbre` → `sclembre`)

3. Each time the user swaps two letters, their score is decreased by 1. The updated score is printed to the console [1p].

4. The user can undo the last swap operation. This does not affect the score [1p].

5. The game ends when one of the following conditions are met: player score is 0 (defeat), or the letters comprising the sentence are put into correct order (victory!). In either case, the player receives a corresponding message [1p].

Example input.txt

```
scramble
dream without fear
The quick brown fox jumps over the lazy dog
Brevity is beautiful
Work hard dream big
```

Example playthrough:

```
Doahm woeiott fear [score is: 16]
swap 0 1 - 1 4
Drahm woeiott fear [score is: 15]
swap 0 3 - 1 1
Dream wheiott fear [score is: 14]
undo
Drahm woeiott fear [score is: 14]
swap 0 2 - 0 3
Drham woeiott fear [score is: 13]
swap 0 2 - 1 2
Dream wohiott fear [score is: 12]
swap 1 1 - 1 3
Dream wihoott fear [score is: 11]
swap 1 2 - 1 5
Dream witoht fear [score is: 10]
swap 1 3 - 1 5
Dream without fear [score is: 9]
You won! Your score is 9
```

Non-functional requirements:

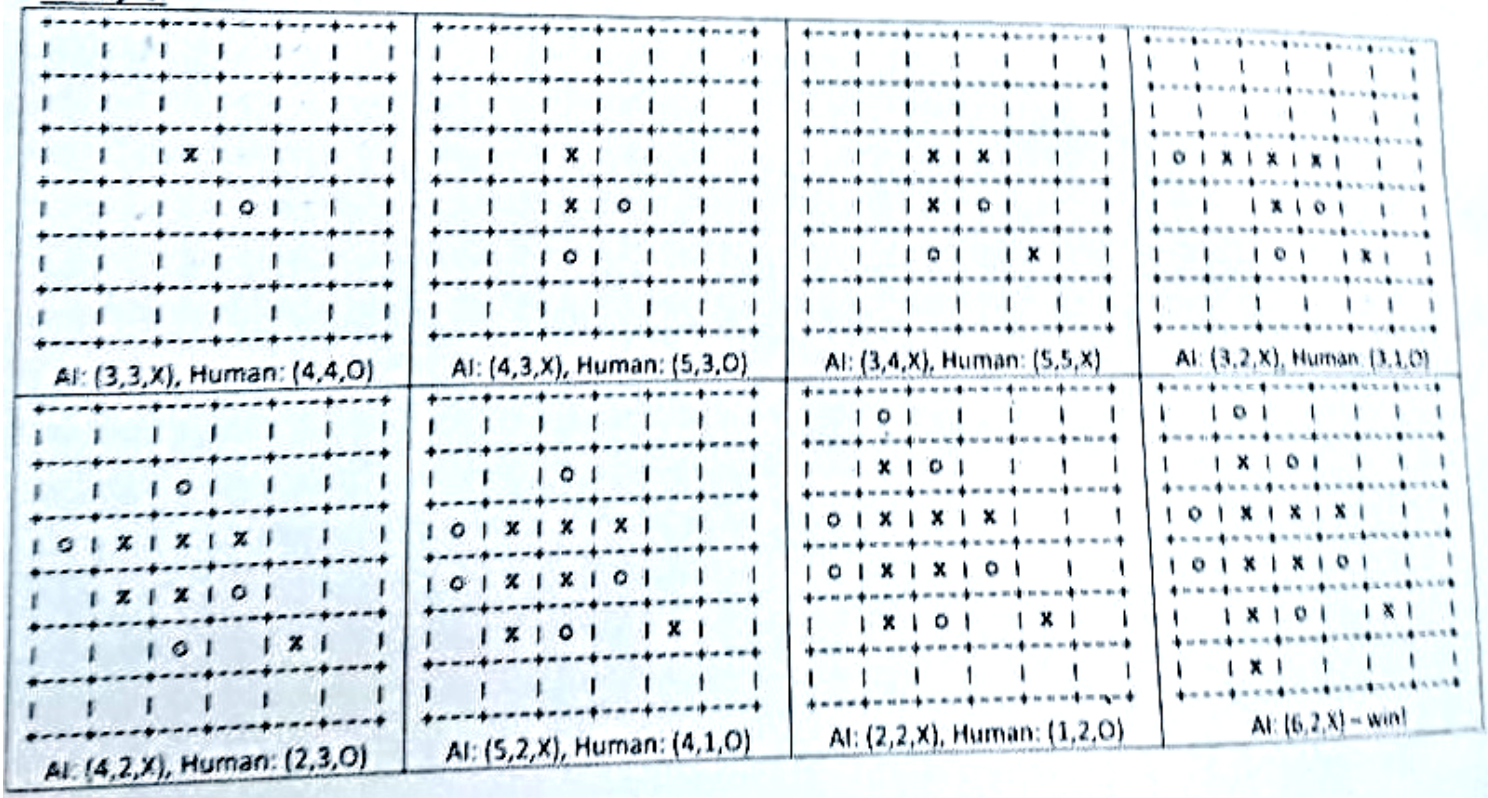
- Have at least 5 entries in the input text file.
- Make sure the program does not crash, regardless of user input.
- Implement a layered architecture solution with Repository, Controller and UI.
- Provide specification and tests for the non-trivial methods in the Repository and Controller layers. Functionalities without specification or tests are graded at 50% value.

Order and Chaos (2nd variant)

Order and chaos is a pen and pencil strategy game played on a 6x6 board. The two players are **order** (computer player) and ***chaos*** (human player), and they take turns placing X's and O's on the board. Order wins by having 5 consecutive pieces of the same type in a row, column or a diagonal. Chaos wins when the board is filled without this happening. Unlike other games, players can pick which symbol to place every move. Implement a console based program that allows a human to thwart the computer's efforts. The program must work as follows:

1. When the program is started, the empty board is displayed **[1p]**
2. Players take turn in placing symbols on the board, with order having the first move **[1p]**
3. User input is validated and an error message shown when trying to make an illegal move **[1p]**
4. In order to win, the computer uses the following strategy:
   a. Always makes valid moves **[1p]**.
   b. If it can make a winning move, it does **[1.5p]**.
   c. Chooses the symbol that appears most often on the board **[1p]**, and places it in the square that has the largest number of same-symbol neighbours **[1p]**.
5. If there are 5 consecutive symbols of the same type in a row, column or diagonal order wins and the game is finished **[1p]**. **(NB! This can also happen after chaos moves)**
6. If the board is full, chaos wins and the game is finished **[0.5p]**.

Example:



nctional requirements:

- Make sure the program does not crash, regardless of user input.
- Implement a layered architecture solution.
- Provide specification and tests for functionality 5. In case specification or tests are missing, the functionality is not graded.

Default 10.

Stellar Journey

Implement the Stellar Journey game, where the player controls the USS Endeavour spaceship and has the goal of eliminating all Klingon ships from a sector of the galaxy. The game is played in a number of turns, as described by the following rules:

1. When the game is started, display the sector of the galaxy according to the following rules:
   a. The game takes place on an 8x8 grid, having marked rows (1-8) and columns (A-H). [0.5p]
   b. Exactly ten stars are randomly placed in the sector, so that no 2 stars overlap, and no 2 stars are adjacent to each other on row, column or diagonal. [1p]
   c. The player's ship, the USS Endeavour starts in a random square of the grid that has no stars. The ship is represented as E. [0.5p]
   d. Three Klingon cruisers are placed randomly on empty squares of the grid. They must not overlap each other, the player's ship, or a star. [1p] The player can see only the ships directly adjacent to the Endeavour [1p]
2. The game is played in a number of turns. Each turn, the player can give one of the following commands:
   a. **warp <coordinate>** (e.g. warp G5). Moves the ship to the new coordinate. The new coordinate must be on the same rank, file, or diagonal as the starting position (e.g. from A1 you can warp to C3, but not to C4) [0.5p]. In case a star is in the way, the program displays an error message and the ship is not moved [1p] In case Endeavour would land on an enemy ship, the Endeavour is destroyed and the game is over [0.5p]. In case the command format or destination are invalid, an error message is displayed. [0.5p]
   b. **fire <coordinate>.** Destroy the Klingon ship at the given coordinates (e.g. fire B4). The fire command only works for ships adjacent to the player's ship. [0.5p] If the wrong coordinates are provided, an error message is displayed. [0.5p]
   c. **cheat.** This displays the playing grid, with remaining Klingon ships revealed. [0.5p]
3. Every time a Klingon ship is destroyed, the remaining ones reposition randomly, making sure that constraints given at 1.d are observed. [0.5p]
4. The player wins by destroying all enemy ships. [0.5p]

Non-functional requirements:

- Implement a layered architecture solution.
- The program must not crash, regardless of user input.
- Provide specification and tests for the warp command.
- The exam cannot be passed without working functionalities!
- Default 1p.

Example starting position:

```
+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
+===+===+===+===+===+===+===+===+===+
| A |   | E |   | * |   |   |   | * |
+---+---+---+---+---+---+---+---+---+
| B |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+
| C |   | * |   | * |   |   |   |   |
+---+---+---+---+---+---+---+---+---+
| D |   |   |   |   | * |   |   |   |
+---+---+---+---+---+---+---+---+---+
| E |   |   |   |   |   |   |   | * |
+---+---+---+---+---+---+---+---+---+
| F |   |   | * |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+
| G |   |   |   |   | * |   | * |   |
+---+---+---+---+---+---+---+---+---+
| H |   |   |   | * |   |   |   |   |
+---+---+---+---+---+---+---+---+---+
```

Initial configuration

```
+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
+===+===+===+===+===+===+===+===+===+
| A |   | E |   | * |   |   | B | * |
+---+---+---+---+---+---+---+---+---+
| B |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+
| C |   | * |   | * |   |   |   |   |
+---+---+---+---+---+---+---+---+---+
| D |   |   |   |   | * |   |   |   |
+---+---+---+---+---+---+---+---+---+
| E |   |   |   |   |   |   |   | * |
+---+---+---+---+---+---+---+---+---+
| F |   |   | * |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+
| G |   | B |   |   |   | * |   | * |
+---+---+---+---+---+---+---+---+---+
| H |   |   |   | * |   | B |   |   |
+---+---+---+---+---+---+---+---+---+
```

The **cheat** command reveals enemy sh

mple commands:

- ***warp B3***, followed by **warp E3**, followed by **warp H6** (as long as no enemy ships are on any of those sq
- ***cheat***