UNIVERSITY OF EXETER

Faculty of Environment, Science and Economy

ECM2414

Software Development CA 1

Continuous Assessment

Handed out date: 27October 2023.
Handed in date: 27November 2023.

This CA comprises 40% of the overall module assessment.

This is a paired assessment, and you are reminded of the University's Regulations on Collaboration and Plagiarism (https://vle.exeter.ac.uk/course/view.php?id=1957).

In this assignment you will implement an application using the various techniques that you are learning about during this module. You will tackle this task using the pair programming development paradigm that will reinforce the idea of programming as a team exercise.

1. Development Paradigm

Pair programming [1] is a popular development approach, primarily associated with agile development, but used across the software industry. In pair programming, two programmers work together to generate the solution to a given problem. One (the driver) physically writes the code while the other (the navigator) reviews each line of code as it is generated. During the development, the roles are switched between the two programmers regularly. The aim of the split role is for the two programmers to concern themselves with different aspects of the software being developed, with the navigator considering the strategic direction of the work (how it fits with the whole, and the deliverables), and the driver principally focussed on tactical aspects of the current task at hand (block, method, class, etc.), as well as allowing useful discussion between the developers regarding different possible solutions and design approaches.

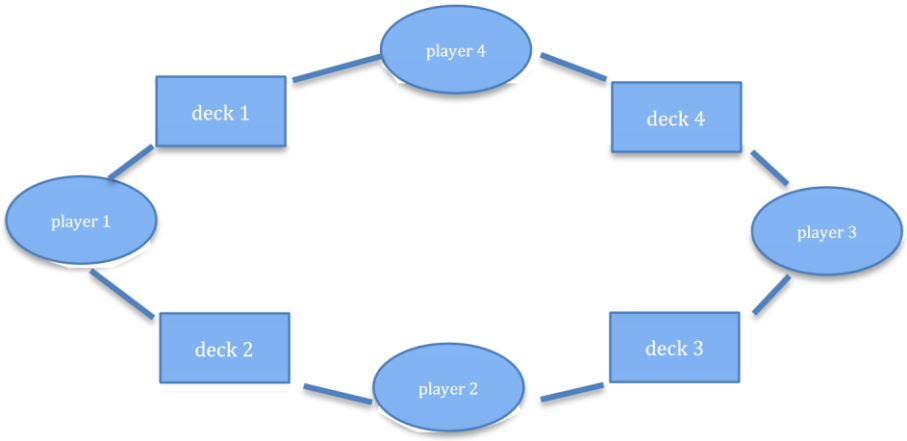
Research has indicated that pair programming leads to fewer bugs and more concise (i.e., shorter) programs. Additionally, it also facilitates knowledge sharing and flow between developers, which can be crucial for a software house, with pair programming often cycling through developers on a team so everyone is eventually paired with everyone else at some point. This assignment will introduce you to the paired programming approach in a practical fashion, through the development of a threaded Java game. Please find the pair information on ELE (in the Assessment tile). It is either paired by yourself or module leader will do it for you if you can't find a partner. **Note**, any changes of the members to the pairs need to be approved by the module leader, otherwise a mark of zero for the relevant teams may be applied.

2. Task Specification

You will develop, in Java, a *multi-threaded* card playing simulation. Within your design you will need to implement (at least) a thread-safe `Card` class and a thread-safe `Player` class (depending upon your design, you may also implement additional classes, for instance a `CardDeck` class). You will also develop an executable `CardGame` class.

The game has *n* players, each numbered 1 to *n* (which for clarity in the illustration below are named "player 1", "player 2", … , "player *n*"), with *n* being a positive integer, and *n* decks of cards, again, each numbered 1 to "*n*" (which for clarity in the illustration below are named "deck 1", "deck 2", …, "deck *n*"). Each player will hold a hand of 4 cards. Both these hands and the decks will be drawn from a **pack** which contains "8*n*" cards. Each card has a face value (denomination) of a nonnegative integer[1].

The decks and players will form a ring topology (see illustration in the figure below for the case where *n* = 4). At the start of the game, each player will be distributed four cards in a round-robin fashion, from the top of the pack, starting by giving one card to player1, then one card to player2, etc. After the hands have been distributed, the decks will then be filled from the remaining cards in the pack, again in a round-robin fashion.



Topological relationship of the game players and card decks, in the situation where *n* = 4.

To win the game, a player needs four cards of the same value in their hand. If a player is given four cards which are all the same value at the start of the game, they should immediately declare this (by their thread printing "Player *i* wins", where *i* should be replaced with the player index), that player thread should then notify the other threads, and exit[2].

If the game is not won immediately, then the game progresses as follows: each player picks a card from the top of the deck to their left, and discards one to the bottom of the deck to their right[3]. This process continues until the first player declares that they have four cards of the same value, at which point the game ends[4].

1 It is legal for the face value of a card exceeding *n*.

2 There is a chance that, two or more players are given four cards with the same value at the start of the game. You don't need to handle this situation in your development.

3 By multi-threading, players should NOT play the game sequentially, i.e., NOT in a way that, when one player finishes actions another player starts.

4 By multi-threading, there is a chance that, two or more players have their four cards of the same value at the same time. You don't need to handle this situation in your development.

Game playing strategy

If a player does not start with a winning hand, they will implement a simple game strategy, as specified below (note, the strategy is *not* optimal).

Each player will prefer certain card denominations, which reflect their index value, e.g., player1 will prefer 1s, player2 will prefer 2s, etc. After drawing a card from their left, a player will discard one of their cards to the deck on their right, (e.g. player1 will draw from deck1 and discard to deck2). The card they discard must display a value which is **not** of their preferred denomination. Additionally, a player **must not** hold onto a non-preferred denomination card indefinitely, so you must implement your `Player` class to reflect this restriction (otherwise the game may stagnate).

Developing a solution

You will need to implement an executable class called `CardGame`, whose main method requests via the command line (terminal window) the number of players in the game (i.e. '*n*'), and on receiving this, the location of a valid input pack. The screenshot below shows how this works (note, I put the pack file in the same directory of `CardGame`, so there is no path needed in the example shown below.)

```
Yuleis-Air:example_answer_CA1 yw433$ java CardGame
Please enter the number of players:
4
Please enter location of pack to load:
four.txt
```

A valid input **pack** is a plain text file, where each row contains a single non-negative integer value, and has 8*n* rows. The screenshot below shows a partial pack

```
14
15
8
2
2
4
13
4
13
3
14
5
3
4
9
7
5
13
9
8
14
4
12
12
6
```

After reading in the input pack, the `CardGame` class should distribute the hands to the players, fill the decks and start the required threads for the players. If the pack is **invalid**, the program should inform the user of this, and request a valid pack file[5].

If a player does not start with a winning hand, as a player processes their hand, each of its actions should be printed to an output file which is named after that particular player (i.e. the output file for the first player should be named player1_output.txt)[6]. In game actions should be printed to the file in a similar form to the following example:

player 1 draws a 4 from deck 1
player 1 discards a 3 to deck 2
player 1 current hand is 1 1 2 4

5 The game should not start until there are valid inputs for the 'number of players' and the 'pack' file.

6 Each player should have its own output file, NOT a combined output file for all players.

Additionally, at the start of the game the first line of the file should give the hand dealt, e.g.

player 1 initial hand 1 1 2 3

and at the end of the game the last lines of the file should read either:

player 1 wins
player 1 exits
player 1 final hand: 1 1 1 1

if for instance player 1 wins, or

player 3 has informed player 1 that player 3 has won
player 1 exits
player 1 hand: 2 2 3 5

in the case where player 3 has won (and the values displayed match the hands and decks concerned).

There should also be a message printed to the terminal window (as is the case when a player wins immediately), i.e. if the 4th player wins, then

player 4 wins

should be printed to the screen.

There should only be one player declaring it has won for any single game[7]. If the game is won immediately (a winning hand is initially dealt), the output files should still be written for all players (although each file will only contain four lines). In addition to the player output files, there should also be *n* deck output file written at the end of the game (named, e.g. deck1_output.txt), which should contain a single line of text detailing the contents of the deck at the end of the game, e.g.

deck2 contents: 1 3 3 7

The combination of a card draw and a discard should be treated as a single atomic action. Therefore, at the end of the game every player should hold four cards. The program developed should follow the object-oriented paradigm.

You are encouraged to use a version control system, but this will not contribute to your final mark.

3. Submission

Your submission consists of a code part and a report part. You need to ZIP both parts into one file and submit it *electronically* using ELE2, by 12 noon on the hand in date. **Note**, only one member of each team needs to submit the paired work.

[7] By multi-threading, there is a chance that, two or more players declare they have won the game simultaneously. You don't need to handle this situation in your development. You can simply give it another run.

The Code Part. Your code part includes the following two files (i.e., `cards.jar` and `cardsTest.zip`).

- A copy of your finished classes in an executable *jar* file named `cards.jar`. The jar file should include **both** the bytecode (*.class*) and source files (*.java*) of your submission.
- A copy of your finished classes, and associated test classes and test suites, and any supporting files, in a *zip* file named `cardsTest.zip`. The zip file should include **both** the bytecode (*.class*) and source files (*.java*) of your submission (plus any testing files the tests may rely upon), and a README file, detailing how to run your test suite.

The Report Part. The report, with minimum 2 cm margins and 11-point text, should contain the items listed below.

- A **cover** page which details how you would like the final mark to be allocated to the developers, based upon your agreed input (i.e., 50:50 if both parties took equal roles, or perhaps 55:45 if you both agree that one party may have contributed a bit more than the other – do remember however that in pair programming both the *driver* and *observer* roles are vital, and should be switched frequently between developers). The maximum divergence allowed is 60:40, although non-contributors will receive a zero. **Note**, if a split is missing, a 50:50 will be used to calculate your marks.
- A (max 1 page) development **log**, which includes date, time and duration of pair programming sessions, and which role(s) developers took in these sessions, with each log entry signed by both members (using your candidate number as your signature). This part of the document should be no more than one side of A4.
- A (max 2 pages) document *detailing* your **design** choice and reasons with respect to both your **production code**, and any known **performance** issues. This part of the document should be no more than two sides of A4.
- A (max 3 page) document *detailing* the **design** choice and reasons with respect to your **tests**. You may use either of the JUnit 4.x or 5.x frameworks, but you should explicitly detail which framework you are using in your document. This part of the document should be no more than three sides of A4.

4. Some Questions You May Have

1. Which card a player should choose to discard?
   Answer: You can randomly choose one which is not the preferred denomination to discard.

2. Can a player who is given four cards which are all the same value at the start of the game, but the value is not their preferred denomination, still win?
   Answer: Yes, it still can win the game.

3. How many output files from the production code are required?
   Answer: 2*n*, where *n* is the number of players.

4. Do I need to use Javadoc?
   Answer: No, you don't need.

5. Do I need to test private methods?
   Answer: Yes, you can consider using Java Reflection.

6. My code reads in a valid pack file which doesn't contain the cards that allow a winning hand. What should I do?
   Answer: It's normal. You don't need to do anything. The code and pack are two separate things. You need to make sure your code is correct, which means if it takes in a valid pack that allows a winning hand, it can output a winning hand.

7. What should I do if my partner is a non-contributor?
   Answer: You need provide evidence to me, e.g., the evidence from the version control system. The module lead, maybe along with other colleagues, will evaluate and make a decision. You may be asked to attend a hearing meeting. The person who is doing ALL the work will be marked with more weights, which will be determined in the evaluation process.

If you have any further question about this CA, please ask Dr. M. Soufian (m.soufian@exeter.ac.uk).

5. Marking Criteria

This assessment will be marked using the following criteria.

Marking Scheme	**Description**	**Mark**
	The Report (40%)	
Structure and contents of the report	The report is well structured and presented. The design is well explained, matches the specification provided and the implemented code. The development log contains the specified information.	40%
	The Code (60%)	
Code comments	Code comments are useful and informative, and at the appropriate level (i.e., it should not contain spurious comments, or ones that do not serve an explanatory purpose).	10%
Production Code. Implementation & Testing.	The code is well structured and presented, with a coherent design and clear management of object states. The program is thread-safe, produces output and takes input in the formats specified. The JUnit tests are well formulated and cover the code well.	35%
README file.	The file details all that is required to run your test suite.	5%
Production Code. Handling exception states.	The code deals smoothly with exceptional inputs and is robust in use.	10%
	Penalties	
Penalty	Non-submission of cover page with weightings (a 50:50 will be assumed)	-5%
Penalty	Non-submission of development log	-10%
Penalty	Submission is greater than the stated number of pages (7) including cover sheet.	Only the first 7 pages will be marked.

Your CA will be marked based on the above criteria.

Reference

[1] J. Garber, Practical Pair Programming, O'Reilly, 2020.

Referencing, and academic conduct

You are required to cite the work of others used in your solution and include a list of references and must avoid plagiarism and collusion. https://vle.exeter.ac.uk/course/view.php?id=1957

Further Information and Guidance for students on submitting group work:

Only **one student per group** should submit the completed file. Nominate someone in your group to do this and make sure you arrange a time, before the deadline, to confirm that this has been done. If possible, do this together (or via a shared screen in Teams) so you can all check that the submission is correct, it has been uploaded to the correct link, and that the file hasn't been corrupted.

You MUST declare yours and your pair programming partner's details.

Please contact the Harrison Hub if you have any submission issue.