School of Computing Sciences

Module: CMP-4008Y Programming 1
Assignment: Coursework Assignment 2

Set by: Jason Lines (j.lines@uea.ac.uk)
Date set: Friday 7th March 2025
Value: 55%

Date due: Friday 9th May 2025 3pm
Returned by: Monday 9th June 2025 *(20 working days)*
Submission: Blackboard

Learning outcomes

The aim of this assignment is to facilitate the development of **Java** and **object-orientated programming** skills by designing and implementing a program to simulate the operation of a fictional arcade company. In addition to using the fundamental concepts that were developed throughout the first assignment (such as **classes**, **objects**, **instance variables** and **instance methods**), this assignment will also develop skills in **UML class diagrams**, **file I/O**, **inheritance**, **exceptions** and basic **enumerative types** in Java. Further general learning outcomes include: describing abstract systems using technical diagrams; following specifications when developing software; documenting code to enhance readability and reuse; increased experience of programming in Java; increased awareness of the importance of algorithm complexity; and using inheritance to model relationships between classes.

Specification

A company has approached you to create a prototype system for simulating the use of their newly proposed contactless arcades. Driven by changes in many industries due to the adoption of contactless payments elsewhere, *GamesCo* wants to open a cashless arcade on a floating island in the middle of the UEA Broad (subject to planning permission).

Your task is to model *GamesCo*'s new system using an object-orientated approach in the Java programming language. You will need to model key objects in the system by creating classes to represent objects such as cabinet games (e.g. pinball, joystick games, etc.), active games (e.g. pool, air hockey, etc.), virtual reality games (e.g. using VR headsets, full-body tracking suits, etc.), customers, and arcades, in addition to a number of other classes to support the functionality of the system. A full description of the required implementation is given in the Description section, but in summary, you will:

1. create a UML class diagram to give an overview of the system;
2. implement the required classes and functionality in Java using an object-orientated approach, **including sufficient documentation and evidence of testing**;
3. simulate the use of the system through loading and processing a number of provided text files

Figure 1: An artist's impression of *GamesCo*'s proposed floating arcade in the UEA Broad.

Overview

To create a system that models customer interactions and transactions for arcades you will firstly need to be able to store information about **arcade games**. This will start with an abstract base class that stores basic information about generic arcade games, such as an ID number for the game, the game's name, and the basic price to play the game (in pence). Subsequently, you will need to *specialise* this class to represent *cabinet games*, *active games* and *virtual reality games*.

All **customers** in the system will have a unique customer ID, a name, a balance and an age. Functionality will also be required to both top-up and charge accounts when appropriate transactions are carried out. Anyone from the public can sign up for a standard customer account, but since *GamesCo* will be operating on UEA campus, they have agreed to special treatment of both *CMP* staff members and any UEA student. As a thank you for forcing their students to prototype this system, *GamesCo* will give all CMP staff a flat 10% discount when using any arcade game. Students, regardless of school of study, will get a flat discount of 5% off all arcade games, but will also be able to have a negative balance of up to £5, similar to an overdraft. Further, the arcade will operate peak and off-peak hours where specific discounts are applicable for different types of game (off-peak discounts are applied BEFORE customer discounts and specific details of the individual off-peak discounts for different game types are given in the Description section).

Once arcade games and customers can be modelled, each **arcade** should maintain its own collection of games and and its own collection of customers. It will need functionality to process transactions when passed a customer ID and a game ID, charging a customer the correct amount to play a game if the customer has a sufficient balance (or rejecting the sale if they do not). An age limit may also apply for certain types of arcade game and this should be checked too. An arcade will also need to keep track of the total value of all sales and it will also require some specialised methods that are of corporate interest, such as reporting the median price of all games within an arcade.

Finally, this will all be tied together by populating an arcade with customers and games by reading in a number of text files that have been provided. You will also be given a list of transactions in another file to **simulate** the use of an arcade.

Please note that you are expected to provide `toString` methods for all object classes, *appropriate* comments to aid someone reading your code, and evidence of testing in all classes (i.e. simple test harnesses in all of your non-abstract object classes).

(hint: please read the full assignment rather than diving in straight away - it will make your life a lot easier if you implement it in a logical order and plan ahead).

Description

1. UML Class Diagram (15%)

Your first task is to fully read this assignment specification and then create a UML class diagram for the proposed system. You should include all classes, and relationships between them, but you are not required to include accessor and mutator methods in this diagram, and you also should not include your main method class (`Simulation`). Marks will be awarded for the accuracy and correctness of your diagram, and presentation will also be taken into account (i.e. make sure that it is clear and easily readable, and make sure it follows conventions taught on this module for UML class diagrams and not conventions taken from anywhere else).

I recommend that you use diagrams.net but you are free to use any other simple tools (such as MS PowerPoint or Word) to *draw* your diagram if you wish.

To avoid issues when including your diagram in PASS, please make sure to save your UML class diagram as a .pdf and do not include spaces in your file name (you can use *export→pdf* in diagrams.net/PowerPoint/Word to do this, or ask for help from the lab assistants if you are struggling to format your work correctly - they cannot answer the coursework for you but they are free to help you with technical issues).

(hint: before starting, read the full coursework specification first and then come back to create your class diagram before writing code. It will help you understand all of the functionality and relationships between the classes, and give you something to refer to while working on the code - that is the whole point of class diagrams, after all).

2. Object Classes

This section describes the classes that you must implement. Please note that *all* non-abstract object classes should have a main method to demonstrate simple usage/testing and an *appropriate* implementation of `toString`. You do not need to include a full javadoc, but you should include a short comment at the start of each class to explain its purpose and use *appropriate* comments throughout to explain any complex operations or calculations. If it is not immediately obvious what a piece of code is doing then this is a good candidate for a comment that would aid a reader.

2.1 `ArcadeGame` (10%)

Arcade games are initially split into two types of game: cabinet games (e.g. pinball, pac-man, lightgun games, etc.) and active games (e.g. things that require active movement, such as pool, darts, air hockey, etc.). There is also a third type of game, virtual reality games, that further extend the active games class (more on that in Section 2.3). To start with however, you will need to begin by creating an abstract base class for `ArcadeGame` that will form the basis for the cabinet game and active game subclasses.

All arcade games have a name, serial number and price to play games.

- Game IDs should uniquely identify a specific game. IDs are 10 characters long and may contain a mixture of numbers and letters.
- Price per play is stored as a whole number in pence (e.g. if a game costs £2 to play, this would be stored as 200).

You are required to write an abstract class called `ArcadeGame`. This class should include fields for the ID, name and price per game of an arcade game. Your class should have a single constructor that takes values for each of those three fields and it should check that the provided ID is valid. If it is not, it should throw an `InvalidGameIdException` (you will need to implement your own Exception for this).

Further, your `ArcadeGame` class should have accessor methods for all fields and an abstract method called `calculatePrice(boolean peak)` that returns an int and takes a boolean as input (true for peak, false for off-peak). The purpose of this method is that subclasses can override it to return the correct price depending on whether it is used during peak/off-peak hours (more on that in Section 2.2).

2.2 `CabinetGame` and `ActiveGame` (10%)

You are required to extend `ArcadeGame` into two sub-classes:

- `CabinetGame` should have an extra field to determine whether or not the game can pay out a reward (for example, this would be true if a game can pay out tickets to a winner that they can trade for a prize). The class should have a single constructor that takes values for all fields **and the constructor should throw an** `InvalidGameIdException` **if the ID does not contain exactly 10 alphanumeric characters** or **if the ID does not start with the character 'C'**.

Figure 2: Real world examples of cabinet games.

- `ActiveGame` should include an additional field to determine the minimum age requirement for playing the game. The age limit should be stored as a whole number in years (e.g. beer pong would have an age limit of 18 in the UK). Similarly to *CabinetGame*, there should be a single constructor that accepts an argument for each field **and the constructor should throw an** `InvalidGameIdException` **if the ID does not contain exactly 10 alphanumeric characters** or **if the ID does not start with the character 'A'**.

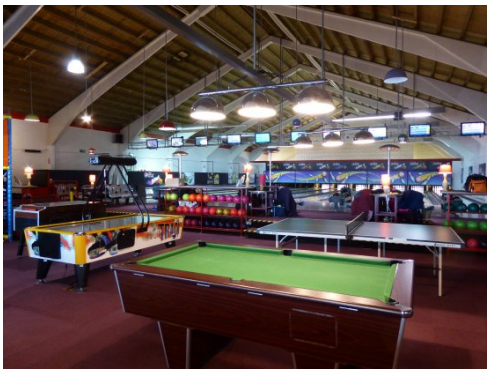

Figure 3: Real world examples of active games (e.g Foosball, air hockey, pool).

Both `CabinetGame` and `ActiveGame` classes should have accessor methods for the new fields and each should have an appropriate implementation of `calculatePrice(boolean peak)` that incorporates the following rules:

- During **peak hours** no game is discounted;
- Cabinet games that **do not** give out rewards are discounted by 50% during **off-peak** hours;
- Cabinet games that **do** give out rewards are discounted by 20% during **off-peak** hours;
- All active games are discounted by 20% during off-peak hours, regardless of age limit.

The `calculatePrice(boolean peak)` in each subclass should follow the rules above and either return the full price to play a game during peak hours (i.e. `peak==true`) or the appropriately discounted price otherwise. These classes should also have a suitable `toString` implementation, helpful comments, accessors for all fields and evidence of testing. Please note that if you need to round prices after applying a discount then you should **round down** (e.g. if a discounted price was 208.8, it should be rounded down to 208 and not up to 209).

2.3 `VirtualRealityGame` (5%)

A third type of game should be represented with a class called `VirtualRealityGame`. This class is a special type of active game, and therefore should be a subclass of `ActiveGame`, and stores an additional field to state whether the virtual reality game requires either: a headset only, headset and controller, or a full-body tracking suit. .

A single constructor should be provided to accept arguments for all fields **and the constructor should throw an** `InvalidGameIdException` **if the ID does not contain exactly 10 alphanumeric characters** or **if the ID does not start with the characters 'AV'** (it must start with an 'A' because it is a subclass of `ActiveGame`, but also followed by 'V' to demonstrated that it is a `VirtualRealityGame` too).

Finally, `calculatePrice(boolean peak)` should again be overloaded so that there is no discount during peak times for any virtual reality game, but there is a 10% off-peak discount for

those that use a headset only, and a 5% discount for those that require a headset and a controller (no off-peak discount for full-body tracking virtual reality games). Similarly to the subclasses of `ArcadeGame`, this class should also have helpful comments, accessors for all fields, evidence of testing and a suitable implementation of `toString`.

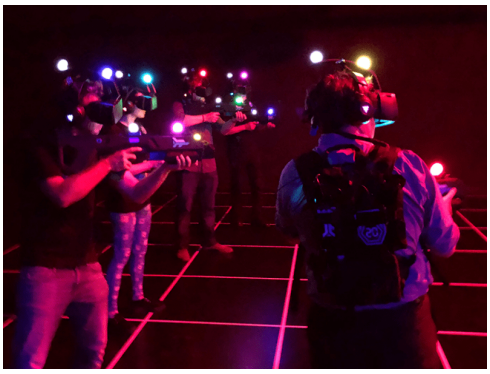

Figure 4: A real world example of people playing a virtual reality game.

2.4. `Customer` (20%)

The `Customer` class should store information about a customer's account. This includes basic information such as their account ID (a 6 digit alphanumeric identifier), their name, their age (a whole number in years), their level of personal discount (none, CMP staff, or student), and their account balance (as an integer in pence, e.g. 1001 for £10.01). You should have two constructors, one for ID, name, age and discount type only that sets a default balance of zero, and another constructor that takes arguments for all fields including a specified balance (but the initial balance must be at least zero). The account balance is the amount of money that is in a customers account and it will be reduced appropriately each time a customer uses an arcade game (see the `chargeAccount` method below). A customer's balance should never be allowed to become negative - with the only exception that those with a student discount are also allowed to have a balance that goes as low as -500 (similar to an overdraft), and it would not be sensible to allow a negative starting balance for an account. Therefore, your constructors should throw an `InvalidCustomerException` if the account ID is incorrect or a negative balance is provided (you will need to provide this Exception class).

Your class should include accessors for all fields and two methods for manipulating the balance of an account - one to add to the balance and one to simulate accounts being charged.

- First, the `addFunds(int amount)` method should allow a customer to top-up their account balance (only add a positive amount to a balance and do not alter the balance if a negative amount is provided).
- Second, there should be an instance method called `chargeAccount` that is used to simulate a customer using an arcade game. This method should take two arguments, where one is an arcade game object and the other is a `boolean` to determine whether the account is being charged during peak time. If a customer has a sufficient balance and is old enough to use an arcade game then the method should operate successfully and return an `int` equal to the amount that the customer was charged (after any applicable discounts - remember, there are off-peak discounts for some games and CMP Staff receive an additional 10% discount while students receive an additional 5% discount. Again, round *down* to the nearest integer if necessary when calculating prices). If the customer

does not have sufficient funds to use the arcade game then their balance should be left unaltered and an `InsufficientBalanceException` should be thrown. Similarly, if the customer is not old enough to use the arcade game, their balance should be left unaltered an `AgeLimitException` should be thrown (you will need to create both of these sub-classes of `Exception`).

Finally, ensure that your class also includes a suitable `toString`, helpful comments and evidence of testing.

(Note: in practice, a customer's age would not be stored as an int and would likely be represented by their date of birth. To keep it simple in this assignment though we are fixing it to be an int and you can assume a customer's age will not change during the execution of your program).

2.5. `Arcade` (25%)

You will also need to create a class to model an `Arcade`. This class should have fields for the arcade's name, a field for the revenue of the arcade, a collection of the arcade games that it offers, and and a collection of the customers that are registered with the arcade. The class should have a **single constructor that takes a single argument for the arcade's name**, and there should be methods to add **individual** customers and arcade games (e.g. `addCustomer(Customer c)`). Further, it should have accessor methods for the arcade's name and the revenue of the arcade, in addition to a suitable `toString` and evidence of testing.

You should also provide methods for:

- `getCustomer(String customerID) throws InvalidCustomerException`
- `getArcadeGame(String gameId) throws InvalidGameIdException`

Finally, you should also have `processTransaction(String customerID, String gameID, boolean peak)` method which will be used to process a transaction when given a customer ID, product ID, and `boolean` to represent whether the transaction was carried out during peak time. This method should tie together what you have already implemented - it should retrieve the correct game, the correct customer, and then try to reduce that customer's balance by the appropriate amount. If successful, this amount should be added to the arcade's `revenue` amount and you should return `true` to indicate that the transaction was a success. Otherwise, the method should throw an appropriate exception for why the transaction not be successfully processed.

Additionally, `GamesCo` has asked that you provide the following methods:

- `findRichestCustomer()` which should search the customers that are registered at a specific arcade to return customer with the highest balance;
- `getMedianGamePrice()` which will consider the price per game for all arcade games within this arcade and return the median (if there is an even number of games then this method should average the price of the two middle games);
- `countArcadeGames()` which should return an `int[]` of size 3, where the first element is the number of cabinet games in this arcade, the second is the number of active games in this arcade (not including virtual reality games), and the third is the number of virtual reality games in this arcade;

- `printCorporateJargon()` which should be a static method in the `Arcade` class that prints a message and does not return anything. It should simply print the corporate motto of "GamesCo does not take responsibility for any accidents or fits of rage that occur on the premises".

It is up to you to decide how you wish to store collections of products and customers. The simplest solution is to use arrays/`ArrayList`, but you can use any data structure that is implemented in Java (such as those that extend the Java `Collection` class or similar). A small number of additional marks will be awarded for using a more appropriate data structure than array-based collections, but **only** if the data structure used is indeed more appropriate **and** the choice of data structure is justified in the code with a short comment (i.e. why exactly is the data structure that you are using a better choice than an array/`ArrayList`). To be clear though, using an `ArrayList` or array will still lead to a good mark if implemented correctly.

3. Main method class: `Simulation` (15%)

Your main method class will simulate the creation and use of an arcade. You are provided with three files:

- `customers.txt`. This file contains information about customer accounts that should be created for your simulation. Each line includes information for an individual customer.
- `arcadeGames.txt`. This file contains information about the arcade games that should be created for your simulation. Again, each line includes information for an individual arcade game.
- `transactions.txt`. This file includes a chronological list of transaction that you should simulate using the appropriate methods that you have implemented.

Simulation will be the main method class of your project and, in addition to a main method (more on that later), you should implement two static methods:

- `initialiseArcade(String arcadeName, File gamesFile, File customerFile)`
  This method should return an `Arcade` with its name set to `arcadeName`. Two files should also be passed into the method: one with information about arcade games, and one with information about customers. This method should parse both of these files and add each customer and arcade game from the respective files to the `Arcade` (using the methods you implemented in the previous question) before it is returned.
- `simulateFun(Arcade arcade, File transactionFile)`
  This method should parse and proceed through a file that contains a list of transactions, simulating the running of an arcade (such as customers playing games, adding funds, or signing up for a new account). Each line should be processed individually and the action should be carried out on the `Arcade` that is also passed into the method (and catch any exceptions that occur). As you process each line of the transaction file **you should print out an informative summary line for each successful action**, and **also a summary line for each action that could not be performed** (such as a customer having an insufficient balance, or rejecting a transaction when an unrecognised game ID or customer ID is found, etc.). The actions that you can expect to see are adding new customers to the arcade (`NEW_CUSTOMER`), customers attempting to play games (`PLAY`), and customers adding funds to their account (`ADD_FUNDS`). Once all transactions have been processed, you should call your methods from `Arcade` to find the richest customer, the median game price and the number of each kind of arcade game. Print the results of each of these

methods (formatted however you wish so that the user can read the results) and also print the company's corporate jargon using the `static` method that you implemented in `Arcade`. Finally, also print out the total revenue for the arcade after all transactions and other methods have been finished.

Your main method should call `initialiseArcade` to create an `Arcade`, and it should then pass that `Arcade` into your `simulateFun` method. It is up to you what you want to call the arcade, but make sure that you use the three files that have been provided (make sure that you place those files in the root directory of your project). Note: it is important that, when run, your code does not need any further information from the user (i.e. your main method should simply call `initialiseArcade` with the provided files, and then pass the returned `Arcade` into the `simulateFun` method without requiring any keyboard input to run when PASS calls your main method).

Relationship to formative work

Please see the following lectures and labs for background on each of the specified tasks:

- **Gavin's content from Semester 1**. In particular, the fundamentals from the first assignment are relevant here. Lecture 5 introduces how to use objects, lecture 11 introduces writing classes, lecture 14 includes using collections to store objects, and lecture 16 introduces inheritance.
- **Classes and objects**: the use case example that we went through at the start of semester 2 (`XboxGame`) demonstrates the structure of a typical class. The labs reinforced this.
- **UML class diagrams**: these were introduced in semester 2 week 2, and further explained in week 3 with relevance and examples including inheritance (hint: remember that the week 4 lab showed how to include exceptions in a UML class diagram...).
- **File I/O**: Gavin introduced this in semester 1, and I have shown an alternative method of doing this in slides at the start of semester 2 (i.e. using the Scanner class). Either approach is perfectly acceptable for file reading on this assignment.
- **Inheritance**: this was covered in more detail in week 3 of semester 2.
- **Exceptions**: this was covered in week 4 of semester 2 and examples were given in code during the lecture (recording available).
- **Data structures**: see the lecture in week 6 of semester 2 for a discussion of different types of data structure that you can use within your Arcade.

Deliverables

Your solution **must** be submitted via Blackboard. The submission **must** be a single `.pdf` file generated using PASS, using the PASS server at `http://pass.cmp.uea.ac.uk/` . The PASS program formats your source code, and compiles and runs your program, appending any compiler messages and the output of your program to the `.pdf` file. If there is a problem with the output of PASS, contact Gavin (gcc@uea.ac.uk) or Jason (j.lines@uea.ac.uk). Do not leave it until the last moment to generate the `.pdf` file, there is a limit to the amount of help I am able to give if there is little time left before the submission deadline.

PASS is the target environment for the assignment. If your program doesn't operate correctly using PASS, it doesn't work, even if gives the correct answers on your own computer:

- The PASS program is not able to provide input to your program via the keyboard, so programs with a menu system, or which expect user input of some kind are not compatible with PASS. Design your program to operate correctly without any user input from the keyboard.
- Do not use absolute path names for files as the PASS server is unable to access files on your machine. If your program is required to load data from a file, or save data to a file, the file is expected to be in the current working directory. If the program is required to load data from a file called `rhubarb.txt` then the appropriate path name would be just `"rhubarb.txt"` rather than `"C:some\long\chain\of\directories\rhubarb.txt"`.
- If you develop your solution on a computer other than the laboratory machines, make sure that you leave adequate time to test it properly with PASS, in case of any unforeseen portability issues.
- If your program works correctly under InteliJ on the laboratory machines, but does not operate correctly using PASS, then it is likely that the data file you are using has become corrupted in someway. PASS downloads a fresh version of the file to test your submission, so this is the most likely explanation

Resources

- **Previous exercises**: If you get stuck when completing the coursework please revisit the lab exercises that are listed in the *Relationship to formative work* section during your allocated weekly lab sessions. The teaching assistants in the labs will not be able to help you with your coursework directly, but they will be more than happy to help you understand how to answer the (very) related exercises in the lab sheets. You will then be able to apply this knowledge and understanding to the new problems in this coursework assignment.
- **Discussion board**: if you have clarification questions about what is required then you **must** ask these questions on the Blackboard discussion board to make sure that other students have the same information for fairness (there will be a specific discussion board topic for the second assignment like there was previously for the first). Also, please check that your question has not been asked previously before starting a new thread.
- **Course text:** *Java Software Solutions* by *Lewis and Loftus*. Any version of this textbook is helpful for Programming 1 and will have specific chapters on topics such as inheritance and Exceptions. You can buy your own copy, but I'd suggest doing a simple online search as many editions of the text are available online for free. The library also has a few copies of the latest version of this textbook too.

Tips

- Identify appropriate places where you could use enumerative types - these have not been specifically requested, but there are some obvious places where they would be more appropriate than using in-built Java types and it is expected that you can recognise this and use `enums` appropriately.
- File reading should **only** happen in the main method class (`Simulation`). You should not have *any* file reading logic within the object classes themselves.

- It is fine to implement additional helper methods that are not specified in the coursework description if you choose, but this is not required and you will not gain extra marks. Do not add additional constructors or functionality that would change the intended implementation of the classes that have been described, however.
- Make sure to follow the specification and do not make unnecessary changes to it. If specific class names and method names are requested then you must follow these requirements.
- You do not need to load the data from files to test any of your object classes - you should start by using hard-coded values in your test harnesses to check that you implementations are correct before even considering the data within the text files.
- Generally, exceptions are thrown within object classes and caught in the main method of your project - if you catch them too early (e.g. as soon as they are thrown in an object class rather than in the main method that called it) then there is no chance to handle the exceptional behaviour appropriately. Similarly, aside from when developing/debugging your code, printing to the console should normally only happen in test harnesses and the main method class (with the exception of any methods that specifically require this, such as the static method in `Arcade` that explicitly asks you to print a message). For example, if you print a message every time you construct an object, this may lead to annoying output if you were to reuse your class in another project (which is common in real-world development) because there would be no way to turn off the output.

Marking Scheme

Marks will be awarded according to the proportion of specifications successfully implemented, programming style (indentation, good choice of identifiers, commenting, testing, etc.), and appropriate use of object oriented programming constructs. Note that it is **not sufficient** to ignore the specification to simply produce the "correct" output - marks are not given for the output specifically and having the correct output only implies that the specification *may* have been implemented correctly. Professional programmers are required to produce maintainable code that is easy to understand, easy to debug when bug reports are received, and easy to extend.

Important Note on Code Simplicity: Students are expected to use only the programming techniques and language features covered in this module. **Solutions that are overly complex or make use of advanced Java features beyond the scope of the course may be subject to additional scrutiny**. While creativity and independent learning are encouraged, submissions should demonstrate a solid understanding of the core principles taught in this module as this is an assessment of your understanding of the content on CMP-4008Y.

Itemised marks are provided throughout the assignment description, but to summarise the marks available for each part:

1. UML Class Diagram (15 marks)
2. Object Classes
    2.1. ArcadeGame (10 marks)
    2.2. CabinetGame, ActiveGame (10 marks)
    2.3. VirtualRealityGame (5 marks)
    2.4. Customer (20 marks)

2.5. Arcade (25 marks)

2.6. Simulation (15 marks)

Total: 100 Marks

Plagiarism, collusion, and contract cheating

The University takes academic integrity very seriously. You must not commit plagiarism, collusion, or contract cheating in your submitted work. Our Policy on Plagiarism, Collusion, and Contract Cheating explains:

- what is meant by the terms 'plagiarism', 'collusion', and 'contract cheating'
- how to avoid plagiarism, collusion, and contract cheating
- using a proof reader
- what will happen if we suspect that you have breached the policy.

It is essential that you read this policy and you undertake (or refresh your memory of) our school's training on this. You can find the policy and related guidance at https://my.uea.ac.uk/departments/learning-and-teaching/students/academic-cycle/regulations-and-discipline/plagiarism-awareness (log-in required).
Note that, in this assessment, working with others is *not* permitted. All aspects of your submission, including but not limited to: research, design, development and writing, must be your own work according to your own understanding of topics. Please pay careful attention to the definitions of contract cheating, plagiarism and collusion in the policy and ask your module organiser if you are unsure about anything. For the avoidance of doubt, use of artificial intelligence to produce any aspect of the solution to this assignment is not authorised. Any suspicion of such tools being used to generate a solution will be referred to the Plagiarism Officer in accordance with the university's plagiarism regulations.