Final CA - Specification

Final CA
Class Group: TU259
Release date: 18/04/2023

Worth: 60% of the overall mark for the module	**Submission:** • 01/05/25 design document. • 16/05/25 final submission. Late submission: 10% first day, 20% second day, not accepted after that.

Report and video (submissions without a report and video will not be marked)
Understanding of the code will be evaluated based on a report written to explain your system and a video to demo it. It should contain one page describing the project classes and its methods, one paragraph as a user manual, and one paragraph describing the difficulties and more challenging parts. A short video also needs to be recorded to demonstrate the application use. This is detailed in the marking scheme.

Submission
Please submit your design document, solution (Python files and any necessary external files), report and video via Brightspace. It is your responsibility to make sure you've uploaded the correct file as changes after the deadline won't be accepted.

Marking Scheme
Correct functionality (50%), Additional Features (20%), Layout and use of comments (10%), Design document (10%), Report and video demo (10%). More about marks at the end of the sheet.

Plagiarism

Plagiarism will result in a zero mark (0%). You should make yourself familiar with the plagiarism policy of Technological University Dublin. **You might be contacted by the lecturer to demo your code if the submission is believed to contain suspected plagiarism.**

Problems Description

For this assignment you will have some choices of problems. **Pick 1 to implement.**

Library Management System:

You are asked to develop an application to manage library services, such as borrowing and returns activities. You should be able to **borrow/return** a **book**, an **article** in a journal, or **digital media**. All the library information should be stored in **four external files**: `library.txt`, `items.txt`, `members.txt`, and `borrowing.txt`. It is up to you to define the structure of each file, but each member, item or transaction should have an unique ID.

Use Python classes to implement the library. Some of the functionality your system should provide includes:

- At least the following classes: Library, Items, Books, Articles, Digital Media, Members
- Books, Articles, Digital Media should be a **subclass** of Items.
- Each class should have an **__init__** and **__str__** methods. For each __str__ method, think what information each class should provide when you print their instances.
- Persistent memory: when you start your system it should read the files `library.txt`, `items.txt`, `members.txt`, and `borrowing.txt` in the same folder as the python code and create all the necessary instances.
- Provide a command line interface for the user to:
    - Add/edit/delete instances belonging to each class,
    - Members browse library items and select items to borrow.
    - Members returning borrowed items.
    - Make sure to update the external files after any information is modified.

Bank Management System:

You are asked to develop an application to manage bank services. An account is a general account class that contains **balance**, **deposit**, **transfer** (send money to another account in the bank) and **withdrawal** methods. Your bank should allow the creation of **two bank accounts types**. The two types of accounts should include:

- Savings accounts: these only allow one withdrawal or one transfer per month, and might also be opened by teenagers from 14 years old.
- Checking accounts: these are regular accounts that can be opened by customers who are 18 years or older. They can also have a negative balance to a specified credit limit.

All the bank information should be stored in three external files: `customers.txt`, `accounts.txt`, `accountsTransactions.txt`. It is up to you to define the structure of each file, but each customer, account or transaction should have an unique ID.

Use Python classes to implement the bank system. Some of the functionality your system should provide includes:

- At least the following classes: Account, Saving Account, Checking Account, Customer.

- Saving Account and Checking Account should be a subclass of Account.
- Each class should have an **__init__** and **__str__** methods. For each __str__ method, think what information each class should provide when you print their instances.
- Persistent memory: when you start your system it should read the files `customers.txt`, `accounts.txt`, and `accountsTransactions.txt` in the same folder as the python code and create all the necessary instances.
- Provide a command line interface for the user to:
    - Customer creating a new account.
    - Customer viewing the transactions performed in one of his accounts and the respective balance.
    - Customer performing the operations allowed by its account type.
    - Customer deletes his/her account.
    - Make sure to update the external files after any information is modified.

GYM MANAGEMENT SYSTEM:

You are asked to develop an application to manage a GYM hall. The GYM should have equipment, trainers, customers, exercise plan, and simple customers subscriptions.

- An exercise plan consists only of a trainer, equipment and duration.
- A subscription contains a customer, a trainer, an exercise plan, a starting date and an end date.

Use Python classes to implement the Gym system. Some of the functionality your system should provide includes:

- At least the following classes: Customers, Trainers, Equipment, Exercise Plan, Subscriptions.
- Exercises Plan and Subscription should use aggregation or composition to contain relevant information about Customers, Trainers and Equipment.
- Each class should have an **__init__** and **__str__** methods. For each __str__ method, think what information each class should provide when you print their instances.
- Persistent memory: when you start your system it should read the files `customers.txt`, `trainers.txt`, `equipments.txt`, `exercisePlans.txt`, and `subscriptions.txt` in the same folder as the python code and create all the necessary instances.
- Provide a command line interface for the user to:
    - Login as customer or trainer.
    - Customer/trainer creating a new account.
    - Trainer CRUD operations (create, read, update and delete) for his exercise plans.
    - Customer CRUD operations for his subscriptions.
    - Customer deletes his/her account.
    - Make sure to update the external files after any information is modified.

SCHOOL MANAGEMENT SYSTEM:

You are asked to develop an application to manage schools' services. These services include storing student information, reading and writing student information from files, computing final grades, printing a summary report to an output file, and so on. The school has students

separated in three categories: English, History, and Math students. All grades are based on a 100 points scale. Each module is broken down in the following way:

- English: Attendance 10%, Final exam 60%, Quiz 1 15%, Quiz 2 15%.
- History: Attendance 10%, Project 30%, Exam 1 30%, Exam 2 30%
- Math: Quiz average 15% (there are 5 quizzes worth 15% each that need to be averaged), Test 1 15%, Test 2 15%, Final Exam: 55%.

Use Python classes to implement the school management system. Some of the functionality your system should provide includes:

- At least the following classes: Student, MathStudent, HistoryStudent, EnglishStudent, School.
- The School class should use aggregation or composition to contain relevant information about students. MathStudent, HistoryStudent, and EnglishStudent should inherit from Student.
- Each class should have an **__init__** and **__str__** methods. For each __str__ method, think what information each class should provide when you print their instances.
- Persistent memory: when you start your system it should read the files `school.txt`, `mathstudent.txt`, `historystudent.txt`, `englishstudent.txt` in the same folder as the python code and create all the necessary instances.
- Provide a command line interface for a school manager to:
    - Login in the system
    - Add/remove students from the system.
    - CRUD operations (create, read, update and delete) for student data.
    - Print well formatted student reports for each academic term.
    - Make sure to update the external files after any information is modified.

SHOPPING CART SYSTEM:

For this project you are asked to develop an application to manage a shopping cart, such as the usual ones found online.

- Design a **shopping cart class** in Python. Think of what attributes and methods you have to include in your class and why. Use a **dictionary** to implement at least one attribute.
- Design a **customer class**, with **two sub classes**, Loyal Customers and Bargain Hunters. Loyal Customers can buy exclusive products, while Bargain Hunters can buy any other products listed from low to high price.
- Each class should have an **__init__** and **__str__** methods. For each __str__ method, think what information each class should provide when you print their instances.
- Include in one of your classes at least two methods available in Python to **overload operators**. Demonstrate the use of such operators with different instances of your class.
- Persistent memory: when you start your system it should read the file `products.txt`. This file will contain the list of products available in the shop. You need to define how each product is represented in the file and which ones are exclusive to Loyal Customers. The file should be in the same folder as the python code and create all the necessary instances.

Implement a command line menu with the following options:

```
1. Create a customer.
2. List products.
3. Add/remove a product to the shopping cart.
4. See current shopping cart.
5. Checkout.
```

Option 1 has to be performed before listing or adding any products to the shopping cart. Once a customer is created he/she should be able to see the products available to him/her, to loop over the options to add/remove products, and to see the current products in the shopping cart. The checkout is available once at least one product is added to the shopping cart. To perform the checkout, confirm that the customer is happy with the current items in the cart and the amount due. If it is confirmed, print a thank you message and exit the program.

Your Own System:

If none of these projects seem inspiring you can propose your own idea to me by email or in class. To have it approved it should contain at least 3 to 5 classes, use of inheritance/composition/aggregation, external files, complex data structures, and a command line menu with a meaningful number of operations.

Design Document

A **design document** helps you plan before coding, ensuring your project is well-structured and easier to implement. It clarifies the file structure, classes, and methods, helping you spot issues early and streamline development. Think of it as a blueprint that saves time and reduces errors. **Please submit it by 01/05 to get some feedback for your project**. This document is also required as part of your final submission.

Template

Problem title - Design Document

1. Project Overview
Describe the problem you have selected. What will your system be able to do? What are going to be the key functionalities? Which data structures are you planning to use?
2. Text Files Format
For each external file you are planning to use, give an example of data that will be stored in the file. Justify your decisions on the structure of the files.
3. Class Diagram (Optional)
4. Class Definitions
List the classes and their attributes/methods.
4. Planned Functionalities
Add the functionalities that will be available to the user.
5. Additional features
List the additional features you are planning to add.
6. Expected Challenges and Solutions

Hints and General Guidelines

- Start with the easy parts. Define only your constructors and add new functionalities one by one.
- Once a method has been coded test it before moving on to the next part
- Make sure you include any relevant **error checking** and handle unexpected input
- Make sure to use **function annotations**
- Once you have defined a class you can use its instances as any other type. Remember to use **composition, aggregation and inheritance** if applicable.
- The easiest way to update external files is likely to write content to a new file and replace the old file with the new file. To remove a file use the 'remove' function from the 'os' module.
- Think about which data structure is more appropriate to the pieces of information you need to store: list, string, dictionary, tuple, etc.
- Think if each attribute needs to be **private or public**. Remember to add **get/set** methods for private attributes that are accessed outside the class.
- Add **docstrings** to all your classes and methods
- After implementing a method and testing, see if it can be improved. You don't need to do an optimal solution in the first attempt. Most of the time you can reduce the number of lines and make an algorithm more elegant after an initial solution has been provided.

Sample Marking Schemes

Design document (10%) (To be submitted by 01/05 for feedback)
Your design document should be **clear, complete, and well-organized**. It will be evaluated based on whether the text file formats are well-defined and logical, the planned classes, attributes, and methods align with project requirements, the structure supports maintainability and readability, and the document clearly outlines core features and their implementation. A well-structured submission will help you get feedback before coding and complete the implementation more easily.

Correct functionality (50%)

- Code compiles and runs with no problem.
- All required classes have been coded.
- Required functionalities work properly.
- Inheritance and composition/aggregation applied.
- Ability to handle files and deal with exception handling.
- Functions/methods are used correctly to break down the problem. They are all used for a single purpose or for a single task.
- Nice use of data structures such as dictionaries and sets.
- Command line menu is well implemented, and all its options work as expected

Additional features (20%)

- Additional features that were not required and that fit well in the system have been included. For example:
    - Think about what else a library/bank/gym/school/cart system might have. Perhaps different types of items, customers, members, students etc.
    - Extra functionalities. For example, credit card options, mortgage calculator, fine system for late books, payment for customer subscription in the gym, etc.
    - Search options.
    - Other file types to store data. For example csv or json files (this can be easier to handle than txt files)
    - Additional classes have been developed.
    - Operator overloading has been used for some operations.
    - Use of other concepts you self-study outside the content delivered in the course **(please add references for this, i.e. by adding a URL as a comment in your code)**

Layout and use of comments (10%)
Indentation and white space have been used appropriately. Code is easy to read, and naming conventions have been adopted. Code is well documented, and it is easy to understand.

Report (10%, submissions without a report will not be marked)
Understanding of the code will be evaluated based on a report written by you and a video demos showcasing your system.

The report should contain:

- one page describing the project classes and its methods;
- one paragraph as a user manual;

- one paragraph describing the additional features;
- one paragraph explaining parts not done and why (if any)
- and one paragraph describing the difficulties and more challenging parts for completing the assignment.

The video should contain:

- 5 minutes max length.
- A quick scan over the code explaining the main concepts (which classes were implemented, which data structures were used, any important rationale for the design, etc.)
- Run all the operations in the command line menu.
- Run unexpected cases to demonstrate error handling
- Show the external files being updated.

You can use OBS to record your screen and voice.

Rubric

	Expectations Far Surpassed (70+%)	Expectations Exceeded (50-69%)	Expectations Met (40-49%)	Expectations Not Met (0-39%)	Not Done (0%)
Functionality	All requirements are met. All deliverables included in submission without deviation from requirements.	Requirements met. Deliverables included without major and few minor omissions or deviations from requirements.	Basic requirements met. Deliverables included without major omissions or deviations from requirements	Basic requirements not met or deliverables deviate substantially from requirements	Not convincingly attempted
Additional Features	Additional features are well thought and surpass expectations. They add complexity to the system, make it more interesting, useful, and close to a real system for the project area.	Additional features are well thought. They add complexity to the system and make it more useful.	Additional features are well implemented and work as expected. Not much complexity, but a basic implementation.	Additional features are attempted but not well implemented and/or not working as expected.	Not convincingly attempted
Design document	The document is exceptionally clear, complete, and well-organized. It includes a well-defined file structure, detailed class design, and a strong implementation plan. Readability and maintainability are carefully considered.	The document is well-structured and mostly complete, with good definitions of file structures, class design, and core functionality. Minor details may be missing.	The document covers the basic requirements but lacks depth or clarity in some areas. Some parts may be incomplete or underdeveloped.	The document is unclear, incomplete, or poorly structured, with significant gaps in planning and organization. It does not fully meet the project requirements.	Not convincingly attempted
Layout and use of comments	Layout and comments are flawless. Function annotations and docstrings have been used throughout the whole code. Naming conventions have been adopted. Comments are meaningful and make the reasoning more clear.	Layout and comments are well implemented. Function annotations and docstrings used throughout most parts of the code.	Layout and comments are ok. The code is well organised and not difficult to understand.	Code is not well organised. The main scope is not linear and it is hard to follow. Few or no comments. The code is hard to understand and requires a lot of thinking to follow.	Not convincingly attempted
Report	Report and video are complete. Report is well written and organised. It follows the appropriate length without being repetitive. Difficulties and challenges are well described and related to the solution delivered. User manual and class description are meaningful and help using and understanding the code. Video is well recorded, clear and complete.	Report is complete, well written and organised. Difficulties and challenges are well described. User manual and class description add relevant details. Video is complete.	Report contains all the sections. It describes all classes and adds values to the user. User manual is complete. Some challenges and difficulties are mentioned but could likely be expanded. Video is is clear but does not cover all the required points.	Report is incomplete. Significant parts are missing, such as classes documentation and/or user manual. Video is incomplete or missing. It does not demonstrate good knowledge of the code.	Not convincingly attempted