Assignment 2

This describes the task for the second assignment. Be sure to document all assumptions and changes you are making. Also, be sure to specify who participated in the work and be prepared to answer any questions about your model.

Please use English in your models and documents.´

Goals

In this assignment you are to show that you:

- Understand basic Object Oriented Design and Implementation Principles (GRASP)
- Can work within an architectural pattern (MVC) and know how to assign responsibilities within this architecture.
- Can deliver a working piece of software with high quality and use version management.
- Understands how implementation and diagrams conform to eachother and the differences between the diagrams (class, sequence and object).
- Can design and implement more advanced features within the constraints of the architecture (Grade 3 and 4).

When working with the tasks it is important that you have the above in mind and do your best to clearly show that you have achieved the overall goals of the assignment.

The Stuff Lending System

A large problem for the environment is that we have too much stuff that we seldom use or need. It is wasteful to buy items that you use once or twice, instead it would be better if we could borrow such items from our friends, coworkers or neighbours etc.
One problem that arise in this situation is to know where the stuff actually is at a particular moment in time, what stuff is available for lending etc. In this assignment you will create design and implement the basics of such a system.

The general idea is that you have a number of members that offer a number of items for lending. For every item they get a number of credits in the system. Credits can be used to lend items. The items have a cost per day credits are transfered from the lender to the item owner when a lending contract is established. The lending contract defines the item and the time period (days) when the item should be picked up and returned to the owner. This establishes a basic economy in the system; mebers add value by adding items, and lending them to others, and can in return lend more items themselves.
A special case is when an owner of items needs to reserve an item they themselves own (i.e. the item is no longer available for others to lend) in this case the owner does not pay any credits.

Currently the system does not need to handle bad intentions such as registering non available items, not returning items etc.

Creating a working economy is a hard task and the goal here is merely to offer a basic proof of concept that could later be tested more in depth.

Requirements for Grade 2 (Passing Grade)

Task

Design and implement the stuff lending system. Implementation (Java source code), class-, object- and sequence-diagrams are to be created and presented. The sequence diagrams should show how a model-view-controller separation is achieved (i.e. start in the UI) and how the different requirements are met. The design and implementation should match *perfectly*.

The focus is not to create a very usable or fancy user interface but to have a robust and well-documented design that can handle change and follows the MVC, GRASP and possibly GoF patterns. The application should be a java console application and the ui should be menu based. You can think of the system as a sort of simulator that will let us check what happens in the system when we have members, items, contracts and time is advanced. That is, this is a single user system (no members that log in etc.).

Note: It is not permitted to use any type of framework, however, the standard class libraries, etc. are permitted. Basically, you should design and code your own application.

You may work in any way you like. A good process is to design a little, code a little, test a little, then iterate update and add more. Designing everthing up-front is often hard and requires a lot of experience. Coding everything without a clear (partial) design can lead to problems, esp. in a group setting.

Working process

- You work with this grade in your project A2 in the `main` branch.
- Commit and push to your work regularly.
- When done you create a merge request for merging your `main`-branch to the `release`-branch and select the `A2 Grade 2` milestone.
- Tick the corresponding check-boxes in the merge request. Do **not** close or merge, this is done by the course administration when the assignment is done.

Functional Requirements

1. Member
    1. Create with a name, email and mobile phone number. A *unique* member id should be generated and assigned to the new member and the day of creation should be recorded.
        1. The member id should be 6 alpha-numeric characters.

        2. The email adress and phone number needs to be unique (no other members can have the same email or phone number).
    2. Delete a member.
    3. Change a member's information.
    4. Look at a specific members full information.
    5. List all members in a simple way (Name, email, current credits, and number of owned items)
    6. List all members in a verbose way (Name, email, information of all owned items (including who they are currently lent to and the time period))
2. Item
    1. Create a new item for a member, the item should have a category (Tool, Vehicle, Game, Toy, Sport, Other), a name, a short description, the day of creation should be recorded, and a cost per day to lend the item.
        1. When created the owning member gets 100 credits.
    2. Delete an item
    3. Change an item’s information.
    4. View an items information including the contracts for an item (historical and future)
3. Contract
    1. Establish a new lending contract with a starting day, an ending day and an item.
        1. Credits should be transfered according to the number of days and the price per day of the item
        2. Can only be done if the lender has enough credits.
        3. Can obly be done if the item is available during the time period
4. Time
    1. Time is handled as a day counter, 0 is the first day and is set when the system starts. Time is not connected to the system in this proof of concept.
    2. Advance day. In order to properly test the system there needs to be a way to advance the current day without relying on the system time.
5. *Prepare* the design for persistence, i.e. add a persistence interface. Implement a hard coded "loading" of some members with items, i.e. create some hard-coded data. You should **not** implement any persistent loading or saving to file or database etc. for the passing grade.

Non-Functional Requirments

10. Strict Model-View-Controller architecture:
    1. The controller should be active and the view should be passive (controller depends on view, view does *not* depend on controller).
    2. The view should only be able to read data from the model, not change it.
    3. The model should not depend on the view/controller (user interface) in any way (direct or indirect).
    4. The model should not have view/controller (user interface) responsibilities.
    5. The view/controller (user interface) should not implement model/domain functionality.
    6. The model should encapsulate business requirements and make them easy to reuse and make it hard to get the system into an "invalid state".

11. Good quality of code e.g. variable names, code duplication etc.
12. An object oriented design and implementation. This includes but is not limited to:
    1. Objects are connected using associations and not with keys/ids.
    2. Classes have high cohesion and are not too large or have too much responsibility.
    3. Classes have low coupling and are not too connected to other entities.
    4. No use of static variables and operations as well as global variables (the ´main´ method is the only exception).
    5. Avoid hidden dependencies, i.e. magic constants in different parts of the code.
    6. Enforce encapsulation, do not expose internal class design choices to the outside, do not expose un-needed functionality to classes in other packages.
    7. Use a natural design - let the domain inspire the design
13. Simple error handling. The application should not crash but it does not need to be user friendly.
14. Proper use of versioning: there should be a number of commits (at least 20) that shows a natural progression of the application. "Big bang" delivery will result in a failed submission.
15. Use of gitlab build pipeline, gradle and quality-tests. Build using gradle regularley, fix any quality issues. Do not change the build-pipeline.
16. Should be built using `./gradlew build`
17. Should be run with `./gradlew run -q --console==plain`
18. You are free to add additional automatic tests of your code.
19. A class diagram that shows the final application - focus on classes/packages and relations (association, dependency, generalization, realization), add only some key attributes and operations. Use proper UML design class notation. The class diagram should correspond *exactly* to the final implementation. Automatically reverse engineered diagrams are **not** allowed. You should show that you have the skills to do such things manually.
20. A sequence diagram that corresponds to a scenario where a new third member is added to the system where there already exist two other members. It should involve objects with types from the model, view and controller (i.e. show the whole flow of messages). The sequence diagram should correspond *exactly* to the final implementation and use proper UML notation. Automatically reverse engineered diagrams are **not** allowed. You should show that you have the skills to do such things manually.
21. An object diagram that corresponds to the sequence diagram scenario.

Deliverables

- Everything should be neatly available in your branch as per the working process.
- Source code that can be immediately built and run by gradle. Do not add the compiled java files i.e. ´.class´ files, project files or other things to the project. There is a `.gitignore` that should cover most things but you may be working in a special environment, os etc. so take care. `git rm` is your friend.
- a `README.md` that explains usage of your application and if there are any parts missing.

- a `design.md` that shows your class diagram, sequence and object diagrams and any text needed to explain the design.
- a `testreport.md` that shows the results of your final tests of the application.