Assignment 3 – Model for a Medical Clinic System

You have just finished your work with the C language in this course, producing a system to handle a particular survey. Word has come through the community, and a doctor from a walk-in clinic in Victoria decided to hire you to work on the automation of the clinic's medical records. The doctor is friends to a system's analyst who did the basic system analysis, producing some user stories, a simple domain model and some integration test cases.

The analyst got in touch with you to discuss your work in this project. You will start from the user stories, domain model and integration tests, and you will first produce a model that handles the basic clinic user stories, initially in primary memory (RAM).

The analyst suggests you handle the user stories one at a time, creating the classes and methods in the model according to the user story needs. To check that you are advancing correctly, run the integration tests and check whether the test cases related to that user story you are working on are passing. You will also want to create unit tests for your model classes to increase your confidence you are doing a good job.

Programming environment

For this assignment, please ensure your work executes correctly on Python 3.9.19 (you can use later Python versions, but recall that you cannot use any new features that depend on those later versions). You are welcome to use your own laptops and desktops for much of your programming as well as your favorite IDE. For running tests, you will need the **`unittest`** framework that comes with Python 3. Some IDEs handle that testing framework smoothly or may have **`unittest`** plugins. Regardless of IDE, you may still use your Linux shell (or your command-line tool) to run your tests.

Start your Assignment 3 by copying the files provided on Brightspace (in Assignment 3) into your **`a3`** directory inside the working copy of your local repository (or, if you are working in pairs, inside the working copy of your group's local repository). Do not put your **`clinic`** and **`tests`** subdirectories in subdirectories of **`a3`**, put them in **`a3`** itself (this is important for the automated grading scripts).

Commit your code frequently (**`git add`** and **`git commit`** and **`git push`**), so you do not lose your work. We will look at the code commits you did in this assignment. **We require at least four different commits** (you should split your work into parts, and if you are working in pairs, we expect that both group members commit code, at least two commits for each member). The final grading will take that into account. Maybe the easiest way is committing your code after you pass each integration test case. We recommend you develop your changes in a separate new branch and only merge the changes to the master branch when you are confident your tests are passing. Do not forget to **`git push`** into your remote repo so that your new code is visible by the other stakeholders (e.g., collaborator member, grader).

Description of the task

By reading the email the systems analyst has sent you, you first find some user stories.

Table 1: The User Stories

User Story	Title	Brief Description
1	Log in	The user enters username and password. The system checks the user name and the password within the system and either logs in or not.
2	Log out	The user logs out of the system.
3	Search patient	The user searches a patient by Personal Health Number (PHN). If successful, it returns the patient.
4	Create new patient	The user creates a new patient in the system, with their personal health number, name and other personal data.
5	Retrieve existing patients	The user searches the patients by name, and retrieves a list of patients that have the searched name as part of the patient's name.
6	Update existing patient	The user searches the patient by PHN, retrieves the patient data, and updates any part of that data.
7	Delete existing patient	The user searches the patient by PHN and deletes the patient from the system.
8	List all patients	The user recovers a list of all the patients.
9	Choose current patient	The user sets the current patient by providing a valid PHN, the patient is searched and made the current patient. Then, it is available to work with no further need to provide a PHN. The current patient can be unset as well, if needed.
10	Create new note	The user creates a new note for the current patient's record, storing the note's code as the auto-incremented counter from the patient record, and also storing the note's details and current timestamp.
11	Retrieve existing notes	The user searches the current patient's record by text, and retrieves a list of notes that have the searched text inside the note.
12	Update existing note	The user selects a note by code in the current patient's record, retrieves the note's details and timestamp, updates the details, and changes the timestamp to the current timestamp.
13	Delete existing note	The user selects a note by code in the current patient's record, and deletes the note.
14	List the full patient record	The user lists the full patient record with all the notes, from the last created note to the first created note.

Then, you find a domain model (with a Controller added to exercise the model), expressed as an UML class diagram.

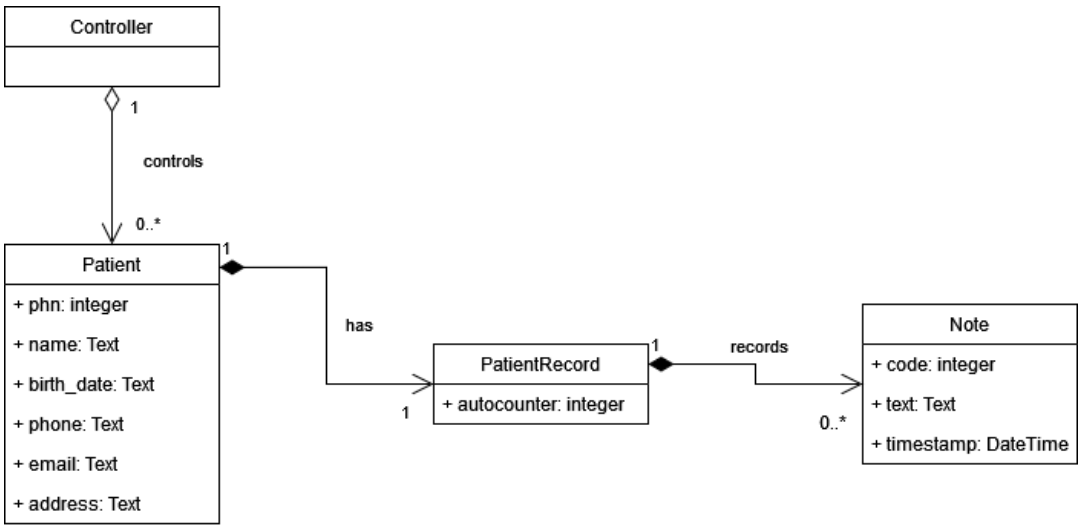

Figure 1: Domain Model and Controller

Implementing your program

The Python code you will develop in this assignment will be exercised from the integration tests file by using the **unittest** framework. You do not need to develop a user interface for your program now. That will be done as part of a follow-up assignment. You will notice that the integration test file, **integration_tests.py**, makes reference to a **Controller** class. You will have to create your **Controller** class in a file named **controller.py** inside the **clinic** directory to be consistent with the integration tests. Your **Controller** class may have simple objects as well as collections as fields from the model classes. But in order to keep the system cohesive and decoupled, do not put all the collections inside the **Controller** class. You will instead have both object and collection fields inside some model classes according to the relationships expressed in the domain model.

As suggested by the analyst, handle the user stories one at a time, creating the classes in the model that are needed.

You will also want to create unit tests for your model classes to increase your confidence you are doing a good job with them. You do not usually test constructors, getters and setters (by the way, we tend to use less getters and setters in Python than in Java or C++). Some classes such as **Patient** and **Note** are simpler, so you may want to just test your objects for equality (**__eq__(self, other)**) as well their string representation (**__str__(self)**). On the other hand, you may want **PatientRecord** to have some more responsibilities and test them appropriately, since the **Controller** will usually delegate operations related to **Note**s to the **PatientRecord** class.

If you are working in pairs, you can either work together doing pair programming (even online pair programming is a good idea, as one of the developers may share their IDE screen and the pair may chat over an audio channel).

If you wish, you may also split the pair members' work. For instance, one developer handles the **Patient**'s user stories, and the other developer handles the **Note**'s user stories. If you do so, notice that the developer that handles **Note**'s user stories will have to work with a mock **Patient** (i.e., a simple **Patient** class that may not have the full code yet, but that has enough to allow the **Controller** to do the *Choose current patient* user story and the additional user stories related to **Note**s). Allow some extra time to do the integration of the real **Patient** class in place of the mock **Patient** class. This split may feel complicated for some people (especially because two developers may be changing the Controller class at the same time, and merge conflicts will have to be solved). If you feel that does not work for you, just do the pair programming work. It is a very helpful practice.

To check that you are advancing correctly, run the integration tests and check whether the test cases related to that user story you are working on are passing. Do not bother for the other integration test cases that should come later, just be sure you are still passing the test cases you were passing earlier as well as the new test case related to the current user story.

Assuming that the **clinic**'s subdirectory of **a3** contains your model classes and your **Controller** class, and that your **tests**' subdirectory of **a3** contains the **integration_tests.py** file, then the command to run the integration tests will be:

```
% python3 -m unittest -v ./tests/integration_tests.py
```

In some computer configurations, your system will use the **python** command instead of **python3**.

Instead of specifying a file path to your tests, you can also run your tests from the current directory by specifying the test module and the test class:

```
% python3 -m unittest -v tests.integration_tests.IntegrationTests
```

By the way, if you want to run just a single test case (one test method) from the current directory, add the test method to the previous command:

```
% python3 -m unittest -v tests.integration_tests.IntegrationTests.test_login
```

You should commit your code whenever you finish some functional part of your it. We recommend you develop your changes in a separate new branch and only merge the changes to the master branch when you are confident your tests are passing. That helps you keep track of your changes and return to previous snapshots in case you regret an unfortunate change. When you are confident that your code is working correctly, merge it to the master, and push your

local master commits to the remote rep. You could do this various times during the development of your project. For example, one full user story seems to be a good split for each new merge and push to the remote.

What you must submit

- You must submit a solution to this assignment containing at least four Python source files named **`controller.py`**, **`patient.py`**, **`patient_record.py`**, and **`note.py`** inside the **`clinic`** subdirectory within your individual **`git`** repository (and within its **`a3`** subdirectory) if you are working individually or within your group's **`git`** repository (and within its **`a3`** subdirectory) if you are working in pairs.
- You may also submit your unit test files named **`patient_test.py`**, **`patient_record_test.py`**, and **`note_test.py`** inside the **`tests`** subdirectory within your individual **`git`** repository (and within its **`a3`** subdirectory) if you want to get the marks related to unit tests.
- Do not use other names for those files because our automated scripts rely on that naming convention. Ensure your work is committed to your local repository and pushed to the remote before the due date/time. (You may keep extra files used during development within the repository, there is no problem in doing that. But notice that the graders will only analyze the controller file, the model files and your unit test files.)
- Do not forget to submit on Brightspace your project's final commit hash and your repository name (either your NetLinkID if you are working individually or your group ID **`groupXXX`**, where XXX is your group number, if you are working in pairs).

Evaluation

Our grading scheme is relatively simple and is out of 100 points. Assignment 3 grading rubric is split into five parts.

1) Modularization - 10 points - the code should have appropriate modularization, dividing each class responsibilities into simpler responsibilities (methods) as well as delegating some responsibilities to other classes (letting other classes do some work instead of being simple data containers);
2) Documentation - 10 points - code comments (enough comments explaining the hardest parts, such as loops, no need to comment each line), function comments (explain function purpose), and adequate indentation;
3) Version control - 5 points – Appropriate committing practices will be evaluated, i.e., your work cannot be pushed in just one single commit, its evolution will have to happen gradually; at least four commits are expected (if working in pairs, each collaborator must have authored at least two commits);
4) Unit tests - 15 points – each unit test file is worth 5 marks if it has adequate and passing unit testing test cases (see instructions in the *Implementing your program* section);
5) Integration tests - 60 points –the 14 user stories given have 12 associated integration test cases, and each test case is worth 5 marks if it is passing when the tests are run against your code.

We will only assess your final submission sent up to the due date (previous submissions will be ignored). On the other hand, late submissions after the due date will not be assessed.