Assignment 4 – Model Persistence in a Medical Clinic System

After your initial work on the Medical Clinic system model in Python, the clinic owner consulted the systems analyst whether they should proceed to a second phase. The analyst was happy with the initial results. With integration tests passing, the next logical phase would be assuring that the system keeps the patients' data (patient info and records) persistent. A simple solution would be using files for handling persistent data access.

The analyst got in touch with you again to discuss this next phase. A new set of integration tests was produced with improved expectations, but you may keep your previous unit tests. In the new integration tests, test cases were fixed to deal with exceptions when they are raised from incorrect system use. Besides, persistent data access is now being tested in the integration tests. The tests are similar, but now there is a boolean variable (*autosave*) that is used to choose whether to test persistence or not. With autosave off, you can test for the collections in memory only. With autosave on, the integration tests will check whether file saving and loading operations work well and keep data consistent.

The analyst also added that you should refactor your code to handle specific persistence technologies inside data access objects (DAO). In doing that, it will be easier to replace the persistence layer when the system use grows and files become inefficient. Anticipating changes that will lead to future replacement of files by either relational databases or other new tech is necessary.

The suggestion is that you handle this project phase in different steps. First, you should keep autosave off and handle the new assertions that handle exceptions. Then, with autosave still off, refactor your code and move all collection handling into the new data access objects, testing the system with the integration tests again. Finally, with your new data access objects, turn autosave on, add new code related to saving and loading files, and test the system with integration tests.

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 4 by copying the files provided on Brightspace (in Assignment 4) into your **`a4`** 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 **`a4`**, put them in **`a4`** itself (this is important for the automated grading scripts). Then add the files you developed in Assignment 3 into the correct places: **`controller.py`**, **`patient.py`**, **`patient_record.py`**, and **`note.py`** inside the **`clinic`** subdirectory, and **`patient_test.py`**, **`patient_record_test.py`**, and **`note_test.py`** inside the **`tests`** subdirectory.

Commit your code frequently (**`git add`**, **`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. We recommend that 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

As suggested by the systems analyst, handle the task in three different steps: 1) handle new assertions for exceptions; 2) refactor your code and move your collections into your new data access objects; and 3) turn *autosave* on, add new code related to saving and loading files.

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_test.py`**, makes reference to a **`Controller`** class in a file named **`controller.py`** inside the **`clinic`** directory. You will reuse your files from Assignment 3 as a starting point: **`controller.py`**, **`patient.py`**, **`patient_record.py`**, and **`note.py`** inside the **`clinic`** subdirectory, and **`patient_test.py`**, **`patient_record_test.py`**, and **`note_test.py`** inside the **`tests`** subdirectory.

Your controller's constructor will have a new parameter, **`autosave`**, with a default value as **`False`**. You need to add it to your controller's **`__init__()`** method so that you are able to run the integration tests. But keep **`autosave`** off in the **`IntegrationTest`** **`setUp()`** method during steps 1 and 2.

New Directories and Files

The analyst provided some new files to you together with the start files. Inside the **`clinic`** subdirectory, there is a **`users.txt`** file that will be used later for reading registered users (ignore it by now, you will need that only in step 3).

Also inside **`clinic`**, there is a new **`exception`** directory with all the new expected exceptions in the **`integration_test.py`** file. You will probably need to import those exceptions in your **`controller.py`** file so you handle them in step 1.

Still inside **`clinic`**, there is a new **`dao`** directory with the abstract classes that define the methods you will have to implement in your own concrete DAO classes. Your own concrete DAO classes as well as any auxiliary persistence-related classes will be inside the **`dao`** directory when you are in step 3.

Finally, for your reference, you will save the patient objects inside the **`patients.json`** inside **`clinic`**. Also inside **`clinic`**, there is a new **`records`** directory, which is initially empty. This is where the patients' records will be saved. This will all be done during step 3.

1. Handle new assertions for exceptions

For this step, keep **`autosave=False`** inside the **`setUp()`** method in the **`IntegrationTest`** class.

You will notice that the integration tests have new assertions called **`assertRaises()`** that replace some previous assertions that expected either **`None`** or **`False`** when the user exercises the system incorrectly. All those expected exceptions are given to you inside the exception directory, and you can import them inside **`controller.py`** the very same way **`integration_test.py`** imports them.

Fix your **`Controller`** class to raise the exceptions there when something incorrect happens as predicted by the integration tests.

2. Refactor your code and move your collections into your new data access objects

For this step, keep **`autosave=False`** inside the **`setUp()`** method in the **`IntegrationTest`** class.

Now that your exceptions are being well handled and your code passes all the integration tests once again, it is time to refactor your code to move the patients' and notes' collections into DAO classes.

For patients, your **`Controller`** class should instantiate a **`PatientDAOJSON`** class that inherits from the abstract **`PatientDAO`** class and assign it to a field. The patients' collection should be a field inside the concrete **`PatientDAOJSON`** class, and that class should manipulate patients with CRUD operations. Then, your **`Controller`** class should delegate its patients' CRUD operations to its **`PatientDAOJSON`** field object.

For notes, your **PatientRecord** class should instantiate a **NoteDAOPickle** class that inherits from the abstract **NoteDAO** class and assign it to a field. The notes' collection should be a field inside the concrete **NoteDAOPickle** class, and that class should manipulate notes with CRUD operations. By the way, the **counter** field in **PatientRecord** should also be transferred to this new class. Then, your **PatientRecord** class should delegate its notes' CRUD operations to its **NoteDAOPickle** field object.

Do not bother about the classes names mentioning *JSON* and *pickle*. Those technologies will be used later in step 3, but they are not needed in this step.

3. Turn autosave on, add new code related to saving and loading files

For this step, first make **autosave=True** inside the **setUp()** method in the **IntegrationTest** class.

Here you will first handle the users' file to read usernames and passwords in 3.1. Then you will handle the patients' file in 3.2. Finally, you will handle a set of notes' files in 3.3.

For the next sub-steps, the integration tests will not be passing initially, except maybe for step 3.1. That is normal, since now persistence is being tested. You will fix your DAO classes to have the persistence integration tests pass.

3.1. Reading usernames and password hashes from the users.txt text mode file

Instead of having hardcoded usernames and passwords, we will load usernames and password hashes from a text file (**users.txt**). This file was produced elsewhere by the clinic IT support (recall that there are no user stories related CRUD operations with users). Your job is to load that file and store it in an appropriate users' collection in memory when your controller is constructed. Use a condition with **autosave** to isolate persistence code from the rest of the constructor.

Notice that the system will be using password hashes (safe practice) instead of storing passwords directly in a file (unsafe practice). Use what you learned in Lab 9 to handle password hashes appropriately.

Check whether your code is working with the **test_login_logout()** test cases.

3.2. Handling patients' load and save operations in the PatientDAOJSON class

First of all, for encoding and decoding patients with JSON, you should implement the **PatientEncoder** (inheriting from **json.JSONEncoder**) and the **PatientDecoder** (inheriting from **json.JSONDecoder**) classes. This is needed to respectively save and load patient objects as JSON strings.

After that, you will first have to pass autosave as a parameter to construct your **PatientDAOJSON** object. In doing that, you will be sure whether to use persistence or not for patients.

Then, change your **PatientDAOJSON** class to load patients from the **patients.json** file. The patients' load should usually be done when the controller is constructed (**Controller** delegates the load to the constructor of its **PatientDAOJSON** field). Use a condition with **autosave** to isolate persistence code from the rest of the constructor.

Then, change your **PatientDAOJSON** class to save patients into the **patients.json** file after any state change has happened to the patients' collection. State changes happen during create, update and delete operations. You may wish to perform those changes one at a time and test the related test cases afterwards. Use a condition with **autosave** to isolate persistence code from the rest of the CRUD method.

Check whether your code is working with the persistence integration test cases.

3.3. Handling notes' load and save operations in the PatientDAOPickle class

The systems analyst decided for a different strategy for patient records. Since record information is private, the decision was to use binary mode files to store records. For that reason, pickle was chosen as the tech for storing records. Another design decision was to use separate files for each patient record. Each patient record file will be named after the patient's personal health number (e.g., **9790012000.dat** for a patient with PHN 9790012000). In each file, you will store the patient record's collection of notes in binary mode by using the **pickle** library. All patient record files will be saved inside the records subdirectory inside clinic.

To implement those design decisions, you will first have to pass **autosave** as a parameter to construct both patients and patient records. In doing that, you will be sure whether to use persistence or not for a patient record.

After that, change your **NoteDAOPickle** class to load notes from the patient record file. The notes' load should usually be done when the **PatientRecord** is constructed (**PatientRecord** delegates the load to the constructor of its **NoteDAOPickle** object). Use a condition with **autosave** to isolate persistence code from the rest of the constructor. After you load the collection, do not forget to update the **counter** value with the code of the final note in that collection (one needs to know the counter value after file load, since it cannot start from zero after loading all notes from the record).

Then, change your **NoteDAOPickle** class to save all notes from the patient record after any state change has happened to the notes' collection. State changes happen during create, update and delete operations. You may wish to perform those changes one at a time and test the related test cases afterwards. Use a condition with **autosave** to isolate persistence code from the rest of the CRUD method.

Check whether your code is working with the persistence integration test cases.

Implementing your program

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).

You do not need to change your unit tests, they should still work regardless of autosave (i.e., they work for memory-only collections). Persistence testing is usually run as integration tests and should not affect unit tests. Run your unit tests as well to check whether your new file handling code does not affect your units.

To check that you are advancing correctly, run the integration tests and check whether the test cases related to a given 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 **a4** contains your model classes and your **Controller** class, and that your **tests**' subdirectory of **a4** contains the **integration_test.py** file, then the command to run the integration tests will be:

```
% python3 -m unittest -v ./tests/integration_test.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_test.IntegrationTest
```

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_test.IntegrationTest.test_login_logout
```

You should commit your code whenever you finish some functional part of your it. We recommend that 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 repo. You could do this various times during the development of your project.

What you must submit

- You must submit a solution to this assignment containing at all eight Python source files below within your individual **`git`** repository (and within its **`a4`** subdirectory) if you are working individually or within your group's **`git`** repository (and within its **`a4`** subdirectory) if you are working in pairs.
    - Inside **`clinic`**, you will submit **`controller.py`**, **`patient.py`**, **`patient_record.py`**, and **`note.py`**;
    - Inside **`clinic/dao`**, you will submit **`patient_encoder.py`**, **`patient_decoder.py`**, **`patient_dao_json.py`**, and **`note_dao_pickle.py`**.
- 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 encoder and DAO files.)
- Do not forget to submit on Brightspace your project's final commit hash and the 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 4 grading rubric is split into five parts.

1) Modularization - 10 points - the code should have appropriate modularization, adding the patients' and notes' file handling responsibilities inside the concrete DAO classes;
2) Documentation - 10 points – on documentation, we will only check the new classes inside **`clinic/dao`** since the other classes have already been documented. We will check for 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 - 8 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) In-memory integration tests (**`autosave=False`**) - 24 points – the 12 integration test cases will be run, and each test case passing is worth 2 marks when the tests are run against your code;
5) Persistence integration tests (**`autosave=True`**) - 48 points – the 12 integration test cases will be run, and each test case passing is worth 4 marks 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.