Healthcare – Technical notes

The code is composed of three parts.

- `healthcare` contains the implementation of the specifications.
- `test` contains unit tests of `healthcare`.
- `console` contains an example application using healthcare. `Main.py` starts this part of the code.
- `initializer` contains an additional package to start `console` with example data.

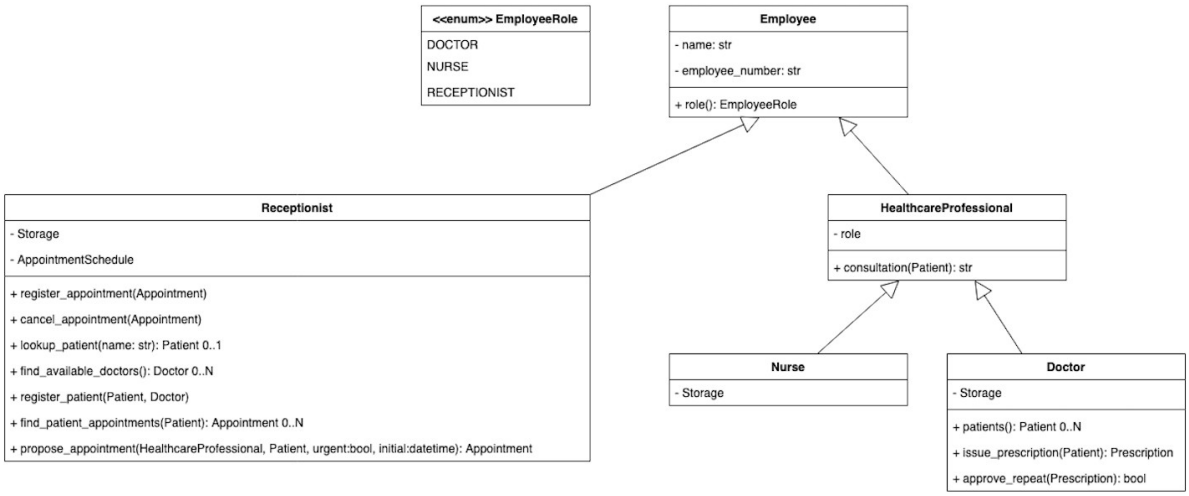
Employee domain

The similarities between `Receptionist` and `HealthcareProfessional` suggested creating a superclass `Employee`. This choice is also supported by the chosen representation on the database.

`Receptionist`'s method "make appointment" is split into "propose appointment" and "register appointment" to let `Patient` choose the appropriate moment.

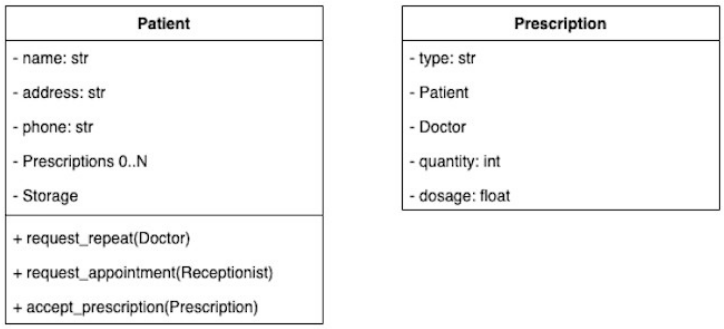
The original design did not suggest a solution to register patients. The implementation adds "lookup patient", "find available doctors" and "register patient" to verify the existence of a registration, propose a list of `Doctor` to a new `Patient`, and complete a registration.

"Approve repeat" in `Doctor` was added to match "request repeat" in `Patient`.



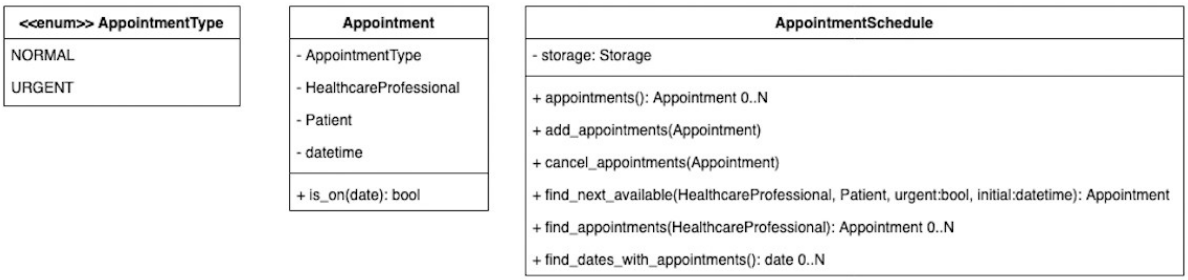
Patient domain

`Patient` has one addition to the original design: “accept prescription” to let the `Patient` become aware of a new `Prescription`. With “request appointment” `Patient` interacts with `Receptionist` who will register him (the first time) and propose a different `Appointment` until the `Patient` accepts.



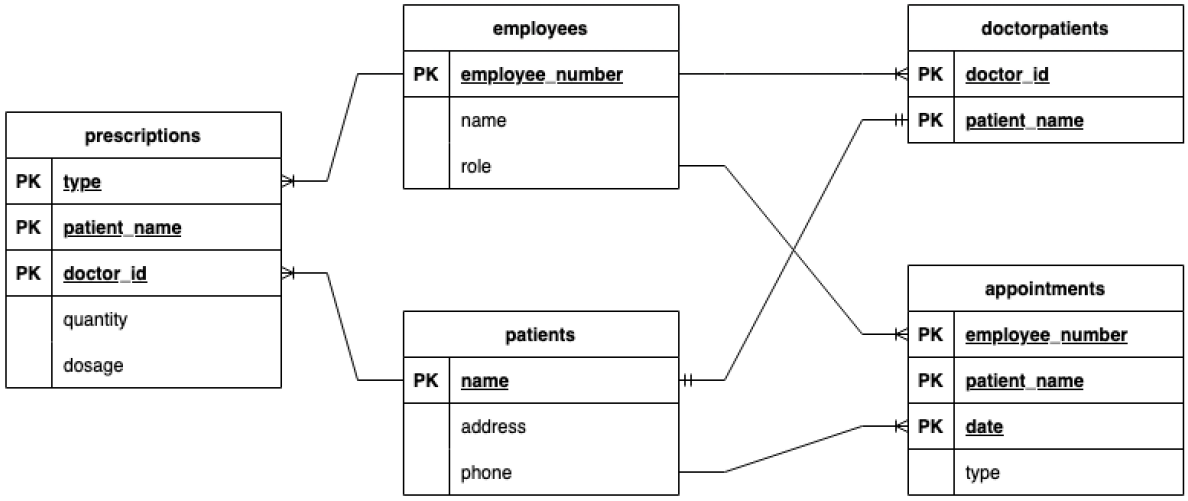
Appointment domain

This domain is faithful to the original design. The method “find dates with appointments” was added only for the convenience of `console`. `AppointmentSchedule` implements a simple logic to differentiate appointments: they can be scheduled from Monday to Friday, if urgent between 8 AM and 3 PM, otherwise between 9 AM and 2 PM.



Storage

The database schema is as follows:

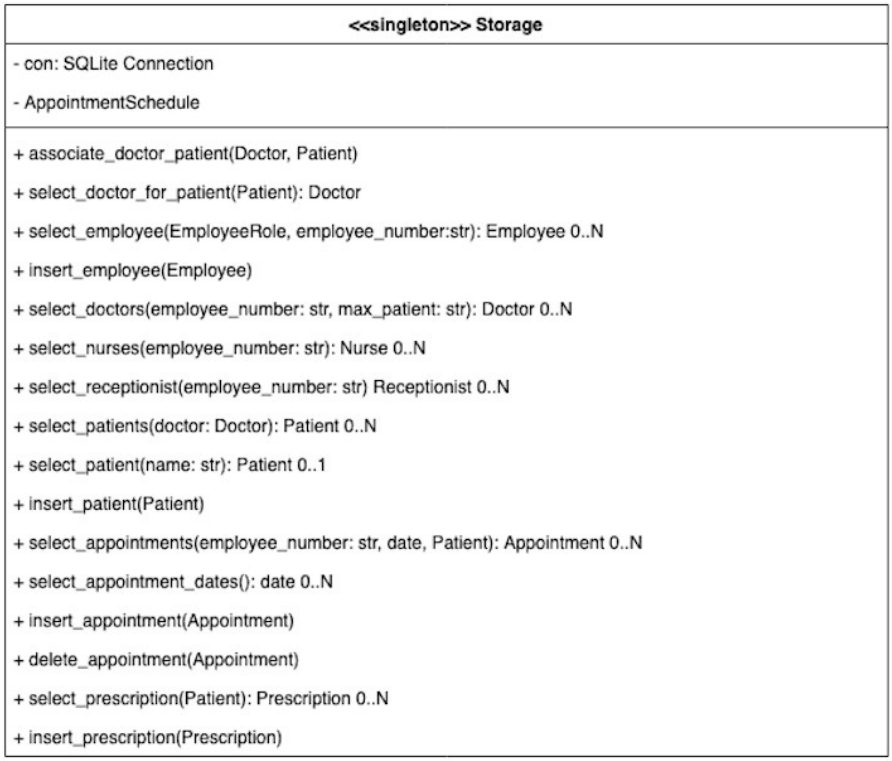


`Employees` corresponds to the classes `Doctor`, `Nurse`, and `Receptionist`.

`Patients` corresponds to `Patient`. The relationship between `Patient` and `Doctor` is modeled by `doctorpatients`. It is possible to collapse `doctorpatients` into `patients`, however the chosen design allows for reasonable extensions such as patients registered with multiple specialists.

`Appointments` and `prescriptions` are two many-to-many relationship tables between `employees` and `patients`.

This model does not make it clear that only doctors participates in the `prescription` table, nor that only doctors and nurses participates in `appointments`. Splitting employees into `healthcareproviders` and `receptionists` would have solved only one of these issues. Splitting employees into one table per role would have required also the creation of `doctorsappointments` and `nursesappointments`. The choice was to give precedence to simplicity and solve the issue programmatically.



`Storage` models the queries to an embedded SQLite database.

Singletons

`Storage` and `AppointmentSchedule` are implemented as singletons.

```
@classmethod
def instance(cls):
    if cls._instance is None:
```

```
            cls._instance = Storage()
        return cls._instance
```

Both classes mimic 3rd-party services such as a MySQL database and a shared calendar, therefore being singletons seems appropriate. The implementation as singletons also simplified the design because it made it unnecessary to pass references of `Storage` and `AppointmentSchedule` to users such as `Receptionist`.

Additional code

The additional code in `console` serves to manually test the application. It gives a view of the data let the tester interact with `Receptionist`, `Doctor`, or `Nurse`. The design of `console` is a simple state machine supported by utility methods for I/O (`ConsoleUtility`). A loop continuously checks the state to dispatch the execution to the appropriate `handler` mapped in the `handlers' dict`.

```
    def loop(self):
        self._state = State.CONNECTED
        context = {}
        while self._state != State.QUIT:
            self._state = self._handlers[self._state].handle(context)
```

The `initialization` package supports `console` inserting records in the database making it easier to perform manual testing.

How to run the code

Install the dependencies with:

```
pip3 install -e .
```

Run the tests with

```
python3 setup.py test
```

Start `console` (sample application) with

```
python3 main.py
```

Console accepts two optional parameters:

- `-h, --help` show the help message
- `-k, --keep` keeps existing db, with no initialization