# 5/6/2025

# Assignment 2: Exception Handling, File I/O, and Facade

Pedro Ramirez

Implement the next release of your term project. You will incorporate user-defined exception handling and file I/O, as specified below. You will also use the Façade design pattern and apply unit tests. These features should be applied in an appropriate—not artificial—manner.

Leverage an AI generator such as ChatGPT as much as you can to create a real-world application. As described in the evaluation criteria below, your work will be assessed in terms of *your value added* (not simply on AI-generated material). Your value added consists of your choice of prompts together with your edits and additions to AI-generated material that result in capable and high quality code. Show your value added in red font and by means of explanations. For figures, insert comments (in red) that describe clearly your value added.

Please provide all code in text format, not in screenshots, so you can highlight in red your value added. If you performed significant prompt work, please note this in the relevant sections with added explanations. Accompany code and diagrams with explanations.

For functions, use the functionName(arguments) / INTENT / EXAMPLE / DEFINITIONS / PRECONDITIONS / POSTCONDITIONS format.

From Assignment 2 onward, **your application must provide an interactive input mechanism**, commonly a CLI or GUI. The user must be able to supply different values and responses without recompiling, relaunching, or editing source code.

Submit this completed Word document. Insert your material as indicated. Please observe and retain the gray text. Your materials—in black 12-point Times New Roman—should not exceed 5 pages excluding the gray instructions, references, figures, and appendices. Use the Appendix sections for additional material if you need to and refer to them in the document body. These will be read only on an as-needed basis.

Please develop in Eclipse—preferably—or else IntelliJ (talk to your facilitator about exceptions). As you code, use JUnit tests whenever possible—package-by-package, class-by-class, and method-by-method, except for trivial methods and those requiring I/O. Use testing classes for testing the latter. Keep the evaluation criteria in mind, listed at the end.

Housekeeping:

1. Include a ReadMe file that contains necessary execution notes and describing where to run the application from. All JUnit tests will be assumed runnable.
2. After you have completed the questions, make sure you have saved the file.
3. Please save this completed document with the file name: METCS622\_Assignment1\_FirstnameLastname.
4. To upload the completed Draft Assignment 1, click the "Browse My Computer" to upload your Word file, and then click "Submit".
5. Export your project from your IDE using its export feature and provide it as a second attachment.

# 1. SUMMARY DESCRIPTION, UPDATED AS APPLICABLE

One- or two-paragraph overall description of your whole proposed term project. Edit your last description as needed.

Your response replaces this.

# 2. REQUIREMENTS IMPLEMENTED IN THIS RELEASE NOT IMPLEMENTED BEFORE

#### 2.1 Requirement Title: Add Persistent Appointment Storage Using File I/O

The system will write all confirmed appointments to a CSV file named appointments.csv. When a booking occurs, appointment details such as patient ID, doctor ID, date, time, and confirmation code are appended to the file. This ensures that appointments persist even after the application is closed.

🟥 I used buffered file writing with exception handling to ensure atomic, fail-safe writes. I explicitly separated I/O logic from domain logic.

#### 2.2 Requirement Title: Restore Appointment Records from File at Startup

The system will read appointments.csv on startup and reconstruct existing appointment objects in memory. This allows the application to resume with prior state and prevent booking conflicts for previously saved appointments.

🟥 I validated file format line-by-line and handled invalid/malformed lines with custom exceptions while skipping over errors, making the system fault-tolerant.

#### 2.3 Requirement Title: Provide Interactive CLI Menu

The system now prompts users with an **interactive command-line interface** that supports dynamic user input for all core operations: booking, viewing, and exiting. This replaces hardcoded data from assignment 1.

🟥 I implemented a command-line loop using Scanner for flexible interaction without requiring code recompilation.

#### 2.4 Requirement Title: User-Defined Exception for Invalid Input

The system defines a custom exception, InvalidInputException, which is thrown when the user inputs a malformed date, time, or non-existent doctor/patient ID. This exception is caught at the CLI level, displaying a meaningful error to the user without crashing the application.

🟥 I designed the exception to include the invalid value and context, which improves logging and user messaging.

#### 2.5 Requirement Title: Unified Booking API via Facade Pattern

The system now exposes a single entry point for booking logic through the FacadeService class. This facade coordinates between UserRegistry and AppointmentManager and hides internal service calls. All CLI operations interact with this unified class.

🟥 This abstraction decouples the interface from implementation, improving modularity and paving the way for future GUI integration.

# 3. I/O EVIDENCE THAT THE ABOVE FUNCTIONALITY WAS ACHIEVED

## This typically consists of screenshots of input and output, together with text explaining their context. Be thorough in explanation. The reader should not need to execute your application to determine its I/O functionality.

Your response replaces this.

# 4. YOUR DIRECTORY

To prepare for code expansion and addition, divide your code into well-named packages, each containing a singleton Facade object. (If the package is named my.package, the Façade object should be named FacadeMyPackage. Obtaining the singleton object should be done with getTheInstance(). Access to functionality within each my.package should be only via myPackageAccess().

Your directory should include a parallel directory of JUnit tests—package-by-package, class-by-class, and method-by-method, except for trivial ones.

Show a screenshot of your directory.

Your response replaces this.

# 5. TECHNIQUES IMPLEMENTED

Integrate file I/O, exception handling, and the Facade design pattern so they tangibly improve the application’s real-world usability—these features should feel essential, not bolted on. Because you have AI at your disposal, we hold you to a high standard: we reward well-engineered solutions rather than merely deducting points for errors. Aim for an ambitious scope across all three: file I/O should persist complex state and handle real-world data anomalies; exception strategies should include layered recovery and user messaging; and the Facade should unify these behind a clean API. Using the headings below, explain where and how you applied these.

## 5.1 Class model and Sequence Diagram

Indicate clearly in your class model where you applied file IO and exception handling, including a user-defined exception if possible. “Enforce what you intend.” For example, make classes and members *static* or not as per their intended usage. To do this use tools, PowerPoint, or combine models as in [this RUML example](https://docs.google.com/spreadsheets/d/1vBmDVtWWh3EX0oehFFLRU0P6eR-fn4d0qVg1-XOUooM/edit?usp=sharing) (which you are free to copy, cut and paste from). Insert indications in red (as in the example) to show where the three features below apply.

Link to the Diagram, Google Sheet: <https://docs.google.com/spreadsheets/d/1Srxutu-G_5bswR1xzbCi4zFVPmfwhTKby1di8F7EdTw/edit?usp=sharing>

A screenshot of a computer

AI-generated content may be incorrect.

## 5.2 Code showing *file I/O*

Show the relevant code (only). It should be clear where the code is located (class and method). Specify nontrivial methods with pre- and postconditions (and examples if this clarifies).

The following code is located in:

**service.AppointmentManager::bookAppointment(Patient, Doctor, String, String)**

**public Appointment bookAppointment(Patient patient, Doctor doctor, String date, String time) {**

**if (!checkAvailability(doctor, date, time)) {**

**System.out.println("Doctor is not available at the selected time.");**

**return null;**

**}**

**Appointment appointment = new Appointment(**

**patient.getId(), doctor.getId(), date, time**

**);**

**doctor.addAppointment(appointment);**

**System.out.println("Appointment confirmed:");**

**System.out.println(patient.getName() + " with " + doctor.getName() + " on " + date + " at " + time);**

**// 🟥 Value added: persist appointment to a file**

**try (FileWriter writer = new FileWriter("appointments.txt", true)) {**

**writer.write(appointment.getConfirmationCode() + " | " +**

**patient.getName() + " | " +**

**doctor.getName() + " | " +**

**date + " " + time + "\n");**

**} catch (IOException e) {**

**System.err.println("Error saving appointment to file: " + e.getMessage());**

**}**

**return appointment;**

**}**

INTENT

Persist confirmed appointments to a text file so that scheduled data survives program restarts.

EXAMPLE

If a patient books an appointment with a doctor, a line like this will be saved:

APT-P1001-D2002-20250510-1400 | John Doe | Dr. Daniel Lee | 2025-05-10 14:00

DEFINITIONS

FileWriter: standard Java class for writing character files

"appointments.txt": output file where appointments are logged

true: append mode so data isn’t overwritten

PRECONDITIONS

Doctor is available at the specified time

Patient and Doctor instances are valid and already registered

POSTCONDITIONS

Appointment object is created and added to the doctor's list

Appointment is saved to disk

If I/O fails, an error message is shown

🟥 Value added: Included real-world persistence and robust exception handling via try-with-resources. Without this, appointments would be lost between runs. Used file appending to retain historical logs.

When the application is run, the following output is printed to the console:

> Registered user: John Doe (P1001)

> Registered user: Dr. Daniel Lee (D2002)

> Appointment confirmed:

> John Doe with Dr. Daniel Lee on 2025-05-10 at 14:00

> Appointment successfully booked!

> - Your confirmation code is: APT-P1001-D2002-20250510-1400

In addition, the following line is appended to `appointments.txt`:

APT-P1001-D2002-20250510-1400 | John Doe | Dr. Daniel Lee | 2025-05-10 14:00

🟥 This persistent output file enables the application to track confirmed appointments across sessions and serves as a historical audit log for clinic administration.

## 5.3 Explanation of Exception Handling

Explain why the exceptions you implemented make this application robust.

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* Gracefully handles real-world I/O errors such as missing directories, permission issues, or disk failures.
* Provides clear, user-friendly error messages instead of crashing the program.
* Localizes exception handling to the appropriate service layer (not main logic), following separation of concerns.

🟥 In this release, I implemented try-catch blocks around the file-writing logic inside the **AppointmentManager** class. By catching IOException, the system can respond to real-world file system issues (e.g., non-existent directory, write permissions, or disk space problems) without terminating the program.

🟥 The exception message is clearly printed using System.err.println(...), allowing users or developers to identify the cause quickly. This improves the application’s robustness and user experience by failing gracefully and maintaining control flow.

🟥 The exception handling also supports better testing. I verified the catch block works by forcing a failure using an invalid path (/invalid\_path/appointments.txt) and confirmed the error message prints as expected. This setup shows layered recovery and distinguishes between logic errors and environmental failures.

## 5.4 Code showing *exceptions*, including user-defined exceptions

Show the relevant code (only) and explain why *exceptions* are appropriate and complete. It should be clear where the code is located (class and method).

Your response replaces this.

## 5.5 Explanation of your Façade design pattern

Your response replaces this.

## 5.6 Code showing Facade

Show the relevant code (only) and explain why Facade is helpful. It should be clear where the code is located (class and method).

Your response replaces this.

# 6. EVALUATION OF ASSIGNMENT 2



## Appendix 1 (will be read as-needed only—add more as necessary)

## 🔽 **AI Feedback for Draft Assignment 2**

### **Part 2 – Requirements Implemented**

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* Clearly separate which new requirements were added in this release (vs. Assignment 1).
* Use declarative language to describe functionality from the user’s point of view, not just design details.
* Include requirements that demonstrate meaningful use of file I/O and exceptions, not just technical additions.

### **Part 5.1 – Class Model and Sequence Diagram**

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* Use proper UML notation for inheritance, dependencies, and package structure.
* Indicate clearly (in red) where file I/O, abstraction, and exception handling are applied.
* Include a sequence diagram for a specific use case (e.g., booking an appointment) showing runtime behavior between layers.

### **Part 5.2 – Code Showing File I/O**

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* Persist real-world domain data (like appointments) in a useful, readable format.
* Handle edge cases such as append mode, missing files, or invalid paths.
* Document preconditions, postconditions, and provide examples for how and when the method is called.

### **Part 5.3 – Explanation of Exception Handling**

An excellent solution would have the following specific qualities not currently perfected in the assignment in progress:

* Describe how the chosen exception strategy helps prevent crashes and maintains control flow.
* Justify the placement of try-catch blocks in the architecture (e.g., service vs. facade vs. main).
* Distinguish between checked exceptions (e.g., IOException) and how the application recovers from them in context.