# 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.  
     
   Note for the professor: my writing is highlighted in red.

# 1. SUMMARY DESCRIPTION, UPDATED AS APPLICABLE

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

**MEDTRACK** is a modular Java application built to simulate key workflows in a medical clinic—such as registering patients and doctors, booking appointments, and maintaining reliable records. Designed with extensibility and real-world usage in mind, the system applies object-oriented principles like inheritance and polymorphism to organize its model and service layers effectively.

In this release, MEDTRACK improves usability and realism by storing appointments in a persistent file, responding gracefully to input errors and I/O failures, and offering a single interface for external interaction via a simplified command-line menu.

What sets MEDTRACK apart from a typical academic project is its **focus on reliability and future readiness**: the architecture anticipates future features like appointment lookup, role-specific menus, graphical interfaces, and database integration. These will be added in upcoming modules using techniques like JavaFX, generics, and JDBC. The system’s core value lies in its **realistic scenario coverage**, **error resilience**, and **scalable foundation** for continued development.

# 2. REQUIREMENTS IMPLEMENTED IN THIS RELEASE NOT IMPLEMENTED BEFORE

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

**Requirement (user-facing)**:  
When a patient books an appointment, the system shall store that appointment in a persistent text file (appointments.txt). The saved record shall include the patient ID, doctor ID, date, time, and a system-generated confirmation code. This allows the system to preserve appointment history across restarts.

**🟥 Value Added**:  
Buffered file writing with append mode was used to retain appointment logs across sessions. I separated file persistence logic from domain logic for better maintainability. Exception handling ensures that I/O failures (e.g., missing directory, permissions) don’t crash the app and instead notify the user. This prepares the system for future replacement of text files with a database using JDBC.  
  
2.2 ***Requirement Title:*** *Load Appointments from File on Application Startup*

**Requirement (user-facing)**:  
When the application launches, it shall automatically read the saved appointments from appointments.txt and restore them into memory. This ensures that previously booked appointments are available and helps prevent double-booking by checking existing reservations.

**🟥 Value Added**:  
I implemented a file-reading mechanism using *Files.readAllLines* and parsed each line into an Appointment object. To ensure robustness, I introduced a user-defined *AppointmentParseException* that skips malformed lines while logging the error, allowing the application to proceed. This builds fault tolerance into the startup logic and prepares the architecture for seamless future transition to binary database-based persistence (e.g., with Java serialization or JDBC).

#### 2.3 **Requirement Title**: Interactive CLI Menu for User Operations

**Requirement (user-facing)**:  
The system shall present users with a command-line interface upon startup that allows them to select an operation (e.g., register, book appointment, view appointments). Input must be interactive, accepting user values at runtime without recompiling the application.

**🟥 Value Added**:  
I implemented a CLI loop using Scanner that dynamically handles user selections. The menu supports flexible branching (registering, booking, and exiting), and all options route through the FacadeService for decoupling. I ensured invalid inputs are caught early with try/catch and clear user prompts, which prepares the system for GUI extension in future modules like JavaFX (Module 4) or asynchronous input using concurrency (Module 5).

#### 2.4 **Requirement Title**: Robust Handling of Invalid Input via User-Defined Exception

**Requirement (user-facing)**:  
The system shall detect invalid user input—such as malformed dates, times, or unknown user IDs—and display a meaningful error message without terminating the program. Errors must be handled gracefully at runtime.

**🟥 Value Added**:  
I created a custom *InvalidInputException* to represent format and lookup errors, such as an unregistered patient ID or an incorrectly formatted time string. This exception is thrown from service methods (e.g., input validation inside *AppointmentManager*) and caught at the CLI layer, which maintains a clean separation of concerns. The message includes the invalid value and a human-readable explanation to aid in debugging and user correction. This pattern aligns with best practices in error recovery (covered in Module 2) and will later support deeper diagnostics or GUI-level validation in JavaFX.

#### 2.5 **Requirement Title**: Centralized Booking and Registration via Façade Pattern

**Requirement (user-facing)**:  
The system shall offer a single unified interface for booking appointments and registering users. All external operations must be invoked through this entry point to simplify usage and hide internal implementation details.

**🟥 Value Added**:  
I introduced the *FacadeService* class as a unifying access point that exposes high-level methods like book() and register(). Internally, this class delegates to *UserRegistry* and *AppointmentManager*, shielding the CLI and future front ends from internal service orchestration. This design aligns with the Façade design pattern and improves testability, encapsulation, and modularity. It also prepares the system for future enhancements such as GUI interfaces (JavaFX in Module 4) or remote API endpoints, which can interface with the same facade without changes to core logic.

# 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.