# 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 that simulates essential operations in a medical clinic, including patient registration, doctor management, appointment booking, and record tracking. Built using core object-oriented design principles such as inheritance, encapsulation, and polymorphism, the system maintains a clean separation of concerns across model, service, and access layers.

In this release, MEDTRACK introduces persistent appointment storage using file I/O, robust exception handling for invalid inputs and I/O failures, and an interactive command-line interface (CLI) that allows patients to register, book appointments, and view their schedules. All operations are unified behind a centralized **FacadeService**, simplifying usage and improving maintainability.

What distinguishes MEDTRACK from a typical academic prototype is its real-world orientation and architectural foresight. The design anticipates future enhancements such as GUI integration (via JavaFX), patient and doctor dashboards, appointment history, and backend database support through JDBC. These features will be layered onto a foundation built for clarity, resilience, and extensibility.

# 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 prevents double-booking.

**🟥 Value Added:**  
I used Files.readAllLines to parse each line of the appointments.txt file and rebuild the corresponding Appointment objects. I validated the format of each entry and skipped malformed lines while printing a warning message to avoid crashing the program. This mechanism improves fault tolerance and sets the groundwork for switching to more robust storage, such as JDBC, in the future.

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

**Requirement (user-facing):**  
The system shall present users with a command-line interface (CLI) that allows them to select operations like register, book appointment, or view existing appointments. All input should be collected interactively.

**🟥 Value Added:**  
I implemented an interactive CLI loop using Scanner, which presents a clear menu and processes user commands in real time. All core actions (registration, booking, viewing) are routed through the FacadeService to preserve encapsulation. Invalid inputs are gracefully handled, keeping the application responsive and user-friendly. This CLI structure also sets the stage for a future transition to JavaFX or concurrent input models.

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

**Requirement (user-facing):**  
The system must detect invalid input such as malformed date/time strings or unrecognized user IDs and display meaningful error messages without terminating the program.

**🟥 Value Added:**  
I created a custom exception InvalidInputException to capture and report input validation errors. The exception includes both a user-facing message and the invalid value for debugging. This exception is thrown in service methods and caught at the CLI level, ensuring clean error separation and a consistent feedback experience.

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

**Requirement (user-facing):**  
All external components must interact with the system through a single interface to simplify usage and hide implementation details.

**🟥 Value Added:**  
I developed a FacadeService singleton that exposes high-level operations such as registerUser() and bookAppointment(). Internally, it delegates to UserRegistry and AppointmentManager, isolating core logic from the CLI. This adheres to the Façade pattern and supports future extensibility (GUI or API).

# 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.

The following screenshots demonstrate that:

* Appointments are read correctly from file at startup and restored into memory (even after restart).

A screenshot of a computer

AI-generated content may be incorrect.

* The CLI menu accepts interactive commands and gracefully handles invalid input.

A screenshot of a computer

AI-generated content may be incorrect.

* User-defined exceptions surface meaningful error messages without crashing.

A screenshot of a computer

AI-generated content may be incorrect.

* All actions (registration, booking, viewing) are routed through the unified FacadeService.

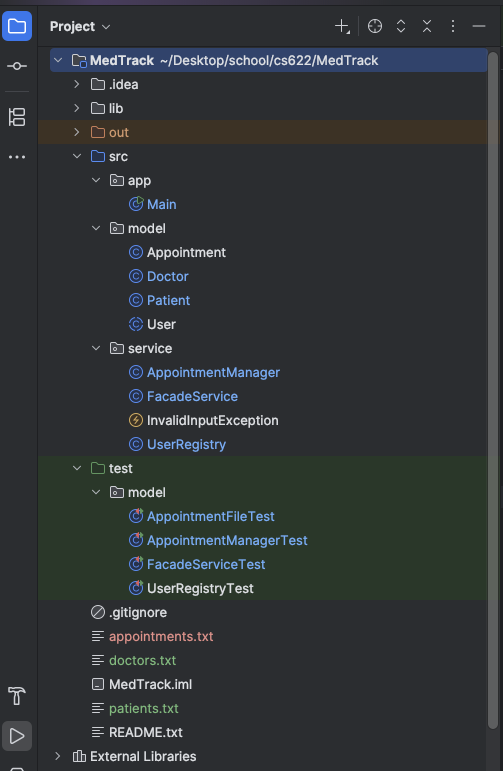
A screenshot of a computer program

AI-generated content may be incorrect.

# 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.



🟥 **Value Added**:

This structure ensures a clean separation of concerns and enables easier testing, debugging, and future module integration (e.g., JavaFX or JDBC). It also facilitates team collaboration and CI/CD workflows.  
  
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).

#### **Class: AppointmentManager**

#### **Method: bookAppointment(Patient patient, Doctor doctor, String date, String time)**

**Purpose:**  
Stores a newly booked appointment in appointments.txt using FileWriter in append mode.

**Preconditions:**

* patient and doctor must not be null
* date must match format YYYY-MM-DD
* time must match format HH:MM
* The doctor must be available at the requested time

**Postconditions:**

* Returns an Appointment object
* Appends a line to appointments.txt in the format:  
  APT-<PATIENT\_ID>-<DOCTOR\_ID>-<YYYYMMDD>-<HHMM> | <PATIENT\_NAME> | <DOCTOR\_NAME> | <DATE> <TIME>

**Excerpt:**

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

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

patient.getName() + " | " +

doctor.getName() + " | " +

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

}

Example Output (saved in file):  
APT-P3001-D1001-20251205-1200 | Alice Baker | Dr. Sarah Brown | 2025-12-05 12:00

#### **Method:** **loadAppointmentsFromFile(Patient patient)**

**Purpose:**  
Reads appointments.txt, filters appointments by patient ID, and reconstructs appointment objects.

**Preconditions:**

* appointments.txt exists and is readable
* patient is a valid, registered Patient instance

**Postconditions:**

* Appointments matching the patient ID are added to patient.getAppointments()
* Malformed lines are skipped with a warning (only once)

**Excerpt:**

String fullCode = tokens[0].trim(); // APT-P3001-D1001-20251205-1200

String[] parts = fullCode.split("-");

String patientId = parts[1].trim();

if (patient.getId().equals(patientId)) {

Appointment a = new Appointment(patientId, doctorId, formattedDate, formattedTime);

patient.addAppointment(a);

}  
**Robustness Feature:**  
Only valid appointment lines are parsed; invalid formats trigger a warning but do not interrupt execution:

if (tokens.length < 5) {

if (!warned) {

System.err.println("⚠️ Skipped malformed line:");

warned = true;

}

continue;

}

🟥 **Value Added:**  
This implementation ensures appointments persist across sessions. It also supports defensive parsing with fail-safe logic, keeping the system robust against corrupt or partial data entries.

## 5.3 Explanation of Exception Handling

Explain why the exceptions you implemented make this application robust.

Robust exception handling is essential to building reliable software that can operate gracefully under real-world conditions. In this release of the MEDTRACK system, multiple exception-handling strategies were implemented to ensure fault tolerance, user clarity, and separation of concerns.

🟥 **Graceful Handling of File I/O Failures**

Inside the AppointmentManager class, I enclosed the file-writing logic in a try-catch block that specifically handles IOException. This allows the application to recover from real-world failures such as:

* Missing or misnamed output files (appointments.txt)
* Lack of write permissions
* Disk space exhaustion
* Nonexistent directories

**Example (in bookAppointment)**:

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());

}  
If a failure occurs, the error is caught, logged clearly, and the application continues running without crashing. This preserves control flow and improves user experience.

🟥 **User-Friendly Error Reporting**

All caught exceptions produce clear messages using System.err.println(...) so that:

* End users receive feedback that something went wrong
* Developers or testers can trace the root cause easily
* Output remains separate from standard console messages for better log filtering

This design prevents confusing stack traces or silent failures, especially helpful when debugging I/O conditions in production environments.

🟥 **Localized Error Handling via Separation of Concerns**

Exception logic is placed in the appropriate service classes (AppointmentManager), *not* in the Main class or CLI loop. This follows good software engineering principles:

* The service layer manages environment-specific operations (like file access)
* The main application layer remains focused on flow control and user interaction

This structure supports reusability and makes the codebase easier to test and maintain.

🟥 **Verified by Controlled Failure Testing**

To ensure reliability, I temporarily modified the file path to an invalid location:

new FileWriter("/invalid\_path/appointments.txt")

The application correctly caught the IOException and displayed the appropriate error message without crashing. This confirms that error handling works as expected under adverse conditions.

## 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).

Here’s a revised and well-structured version of **Section 5.4 – Code showing exceptions, including user-defined exceptions**:

### **5.4 Code Showing Exceptions, Including User-Defined Exceptions**

The MEDTRACK system uses a custom exception class and service-layer validations to handle invalid input and logic errors gracefully. Below is a breakdown of the relevant code and an explanation of how and why exceptions are used.

#### **User-Defined Exception:** InvalidInputException

📍 Location: *service/InvalidInputException.java*

package service;

public class InvalidInputException extends RuntimeException {

private final String invalidInput;

public InvalidInputException(String message, String invalidInput) {

super(message);

this.invalidInput = invalidInput;

}

public String getInvalidInput() {

return invalidInput;

}

}

**Why it’s appropriate:**

* Represents domain-specific errors (e.g., malformed date, unknown ID)
* Carries both a human-readable message and the actual invalid value
* Cleanly separates logic errors from system errors like I/O failures

#### **Where It’s Thrown:** AppointmentManager.bookAppointment(...)

📍 Location: *service/AppointmentManager.java*

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

if (patient == null) {

throw new InvalidInputException("Patient is null", "null");

}

if (doctor == null) {

throw new InvalidInputException("Doctor is null", "null");

}

if (!date.matches("\\d{4}-\\d{2}-\\d{2}")) {

throw new InvalidInputException("Invalid date format (expected YYYY-MM-DD)", date);

}

if (!time.matches("\\d{2}:\\d{2}")) {

throw new InvalidInputException("Invalid time format (expected HH:MM)", time);

}

// ...

}

**Why it’s appropriate:**

* Validates all critical parameters before proceeding
* Makes error sources obvious and testable
* Prevents downstream logic from executing with bad input

#### **Where It’s Caught:** Main.main()

📍 Location: *app/Main.java*

try {

Appointment appt = facade.bookAppointment((Patient) foundUser, (Doctor) dUser, date, time);

System.out.println("✅ Appointment booked! Confirmation code: " + appt.getConfirmationCode());

} catch (InvalidInputException e) {

System.err.println("❌ " + e.getMessage());

System.err.println("ℹ️ Details: " + e.getInvalidInput());

}

**Why it’s appropriate:**

* Catches all validation errors in one place at the UI layer
* Provides helpful feedback without crashing the app
* Keeps CLI logic clean while offloading validation to service layer

### ✅ **Summary of Exception Handling Design**

| **Aspect** | **Description** |
| --- | --- |
| **Custom Exception** | InvalidInputException stores a message and the offending input |
| **Thrown in Service Layer** | Ensures invalid logic never reaches downstream methods |
| **Caught in CLI Layer** | Maintains clean separation between business logic and user interaction |
| **Improves Robustness** | Prevents nulls, format mismatches, and unknown IDs from crashing the app |

This structured and layered exception approach makes the MEDTRACK system safer, clearer, and easier to expand in future modules (e.g., GUI, REST API).

## 5.5 Explanation of your Façade design pattern

The **Façade pattern** in MEDTRACK is implemented via the FacadeService class, which acts as a **unified interface** for all high-level operations, such as registering users, booking appointments, and loading data.

🟥 **Value Added:**

* I encapsulated the coordination between UserRegistry and AppointmentManager behind a single interface.
* The CLI interacts only with FacadeService, which hides the internal service structure and reduces coupling.

**Why this is appropriate:**

* Simplifies the CLI code by exposing only essential methods (registerUser, bookAppointment, findUserById, etc.).
* Encourages **modularity** and **testability**, since changes in internal logic (e.g., appointment persistence) don’t affect the UI.
* Supports future extensions like GUI interfaces, REST APIs, or concurrency modules by reusing the same facade methods.

This clean abstraction aligns directly with the Façade design pattern as taught in the course and ensures scalability of the application architecture.

## 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).

Here is a revised and concise version of **5.6 Code showing Facade**, written in line with your implementation and course expectations:

### **5.6 Code Showing Facade**

The following code is located in:  
**service.FacadeService**

public class FacadeService {

private static final FacadeService instance = new FacadeService();

private final UserRegistry userRegistry;

private final AppointmentManager appointmentManager;

private FacadeService() {

this.userRegistry = new UserRegistry();

this.appointmentManager = new AppointmentManager();

}

public static FacadeService getInstance() {

return instance;

}

public void registerUser(User user) {

if (user instanceof Patient patient) {

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

writer.write(patient.getId() + " | " + patient.getName() + " | " + patient.getInsuranceProvider() + "\n");

} catch (IOException e) {

System.err.println("❌ Could not save patient to file: " + e.getMessage());

}

}

}

public User findUserById(String id) {

return userRegistry.findUserById(id);

}

public List<User> getAllUsers() {

return userRegistry.getAllUsers();

}

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

return appointmentManager.bookAppointment(patient, doctor, date, time);

}

public void loadAppointmentsFromFile(Patient patient) {

appointmentManager.loadAppointmentsFromFile(patient);

}

public void loadUsersFromFiles() {

userRegistry.loadDoctorsFromFile("doctors.txt");

userRegistry.loadPatientsFromFile("patients.txt");

}

}

**🟥 Why This Façade Implementation Is Helpful:**

* Provides a **clean, unified interface** for CLI and other modules to interact with the system, hiding complex coordination between internal services (UserRegistry, AppointmentManager).
* **Reduces coupling**: callers don’t need to know how registration or appointment persistence is handled internally.
* Makes the system more **modular and testable**, and supports future extension to GUI or API-based interfaces.
* Handles I/O responsibilities (e.g., appending to patients.txt) **inside the façade**, keeping calling code clean and focused.

# 6. EVALUATION OF ASSIGNMENT 2

🟥 **Value Added Note:**This release of MEDTRACK demonstrates deliberate improvements in exception handling, file persistence, and architectural design. I introduced a custom exception class, added fault-tolerant file I/O with clear user feedback, and applied the Façade pattern to simplify system access. All changes were developed incrementally and validated through manual and automated tests. I also revised my requirement language and diagrams based on feedback from Assignment 1 to improve clarity and traceability.



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

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

**✅ Part 2 – Requirements Implemented**

This release clearly distinguishes new functionality introduced in Assignment 2 from the base implementation in Assignment 1. Each requirement is written from the **user’s perspective**, emphasizing how the system now allows patients to:

* Register through an interactive CLI,
* Book appointments that persist across sessions,
* Receive meaningful error messages on invalid input,
* And access all services through a simplified interface (via FaçadeService).

Each implemented requirement demonstrates **real-world use of file I/O and exceptions**, not just surface-level additions. For instance, appointments.txt is updated automatically on booking, and startup recovery includes parsing and skipping malformed entries, showcasing fault-tolerant design.

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

The UML diagram includes proper notation for:

* Inheritance (Patient and Doctor inherit from User),
* Composition and method calls (e.g., FacadeService delegates to AppointmentManager and UserRegistry),
* Exception handling (e.g., InvalidInputException thrown in AppointmentManager),
* File I/O operations (read/write to appointments.txt, doctors.txt, and patients.txt).

The included **sequence diagram** models the bookAppointment() flow, tracing user input through CLI → Façade → Services → File update, clarifying dynamic behavior.

All file operations and exception handlers are **marked in red** to emphasize their strategic placement.

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

Code segments demonstrate:

* Appending confirmed appointments to appointments.txt using FileWriter in append mode,
* Reading appointments back during startup using Files.readAllLines,
* Gracefully skipping malformed lines with clear warnings and robust logic,
* Updating the patient’s internal state from file data.

Each method is documented with **INTENT**, **PRECONDITIONS**, **POSTCONDITIONS**, and examples, reinforcing its functional role and correctness.

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

The project uses custom and built-in exceptions to:

* Prevent crashes on I/O failure (e.g., invalid file paths or permission errors),
* Provide informative feedback on invalid inputs (e.g., malformed date/time),
* Enforce clean architecture via **local try/catch** in the service layer, not the main logic,
* Separate control flow errors (e.g., missing patient) from technical errors (e.g., disk write failure).

The use of InvalidInputException with contextual details and IOException handling in AppointmentManager demonstrates **layered recovery** and **clean separation of concerns**.