Design Decisions Document

Decision to Use Singleton Pattern

Singleton Pattern for Session Management:

- **Implementation**: To manage user sessions, we implemented the `UserSession` class using the Singleton design pattern. This ensures that only one instance of the `UserSession` class exists throughout the application lifecycle. The class is defined in the `edu.duke.ece651.team1.client.model` package and includes properties for the username, session token, host, and port.

```
package edu.duke.ece651.team1.client.model;

public class UserSession {
    private static UserSession instance = new UserSession();
    private String username;
    private String sessionToken;
    private String host;
    private String port;
    private UserSession() {
    }
    public static UserSession getInstance() {
        return instance;
    }
    // Getters and setters for username, sessionToken, host, and port
}
```

- **Rationale**: The Singleton pattern was selected for its ability to ensure that a single user session is maintained across the client application. It prevents the creation of multiple instances of the session object, which could lead to inconsistencies and errors in session management.

Decision to Utilize Interface and Factory Pattern for Export Functionality

Interface-Driven Design for Flexible Exporting:

- **Implementation**: To support the flexible exporting of attendance records into various formats, we employed an interface-driven design complemented by the Factory Pattern. The core of this design is the `AttendanceRecordExporter` interface, which defines the contract for exporting records. Concrete implementations of this interface— `XmlAttendanceRecordExporter`, `JsonAttendanceRecordExporter`, and `CsvAttendanceRecordExporter` —provide the specific logic required to export data into XML, JSON, and CSV formats, respectively. This design is encapsulated within the `edu.duke.ece651.team1.shared` package.

```
package edu.duke.ece651.team1.shared;

public interface AttendanceRecordExporter {
    void exportToFile(AttendanceRecord record, String filename, String filePath) throws
IOException;
}
```

```
public class XmlAttendanceRecordExporter implements AttendanceRecordExporter {...}
public class JsonAttendanceRecordExporter implements AttendanceRecordExporter {...}
public class CsvAttendanceRecordExporter implements AttendanceRecordExporter {...}

public class AttendanceRecordExporterFactory {
    private static final Map<String, AttendanceRecordExporter> exporters = new HashMap<>
();

    static {
        exporters.put("json", new JsonAttendanceRecordExporter());
        exporters.put("xml", new XmlAttendanceRecordExporter());
        exporters.put("csv", new CsvAttendanceRecordExporter());
    }

    public static AttendanceRecordExporter createExporter(String format) {
        AttendanceRecordExporter exporter = exporters.get(format);
        if (exporter == null) {
            throw new IllegalArgumentException("Unsupported export format: " + format);
        }
        return exporter;
    }
}
```

- **Rationale**: The decision to adopt this interface and factory design was driven by the need for extensibility and maintainability in exporting functionality. By defining a common interface, we established a contract that all exporters must adhere to, allowing for the easy addition of new formats without modifying the core logic. The Factory Pattern further encapsulates the instantiation logic, providing a straightforward mechanism for obtaining exporter instances based on format requirements.

Decision to Utilizing the Observer Pattern for Notification Services

Observer Pattern for Notification System:

- **Implementation**: To establish a robust notification system, we embraced the Observer pattern, encapsulating this design within the `NotificationService` class. This service manages a list of `Notification` observers, each conforming to the `Notification` interface that mandates a `notify` method. Concrete implementations of this interface, such as `EmailNotification`, enable the delivery of messages across different mediums.

```
public interface Notification {
    void notify(String message, String recipient);
}

public class NotificationService {
    private final List<Notification> observers = new ArrayList<>();

    public NotificationService() {}

    public void addNotification(Notification notification){
```

```
        observers.add(notification);
    }

    public void deleteNotification(Notification notification){
        observers.remove(notification);
    }

    public void notifyObserver(String message, String recipient){
        for(Notification notification : observers){
            notification.notify(message, recipient);
        }
    }
}

public class EmailNotification implements Notification {
    @Override
    public void notify(String message, String recipient) {
        // Implementation for sending email notifications
    }
}
```

- **Rationale**: The decision to implement the notification system using the Observer pattern was driven by the need for a flexible and extendable architecture that can support multiple notification mechanisms (e.g., email, SMS, push notifications) without tightly coupling the notification logic to the consumer services. This pattern allows `NotificationService` to broadcast messages to all registered `Notification` observers, which can then handle the message delivery according to their specific protocol.