Software Quality Assessment for Hospital Database Management System

Part 1: ISO/IEC 25010:2011 Standard (FURPS) Characteristics Analysis

1. Functional Suitability

Relevance to the Hospital Database Application:

The hospital database application is designed to manage critical healthcare data including patient information, provider details, visit records, and prescription information for a mid-size health insurance company. Functional suitability is paramount as the system must perform all required tasks accurately to ensure proper patient care, insurance claim processing, and regulatory compliance.

How it is Addressed in the Project:

1. **Comprehensive Data Management:** The application implements complete CRUD (Create, Read, Update, Delete) operations for all key entities (patients, doctors, drugs, prescriptions, visits, and insurance companies) through dedicated DAO classes:

```
public class PatientDAO implements BaseDAO<Patient> {
    @Override
    public void save(Patient patient) throws DatabaseException {...}
    @Override
    public void update(Patient patient) {...}
    @Override
    public void delete(String... ids) {...}
    @Override
    public Patient get(String... ids) {...}
    @Override
    public List<Patient> getAll() {...}
}
```

2. **Functional Completeness:** The system meets all requirements outlined in the project brief, covering all aspects of healthcare data management including tracking patient's primary care doctors, insurance details, prescriptions, and visit information.

3. **Functional Correctness:** Input validation ensures data integrity through a dedicated validation system:

```
public class ValidationConfig {
    private static final Map<String, FieldValidation[]> VALIDATION_RULES =
new HashMap<>();

    static {
        VALIDATION_RULES.put("patient", new FieldValidation[] {
            new FieldValidation("patientID", 10, true, "^[A-Z0-9]+$",
"Patient ID must be alphanumeric and uppercase"),
            // Other validation rules...
        });
    }
}
```

4. **Search Functionality:** Extensive search capabilities allow users to find records based on various criteria:

```
public List<Patient> findByFirstName(String firstName) {...}
public List<Patient> findByLastName(String lastName) {...}
public List<Patient> findByPostcode(String postcode) {...}
```

Alignment with Project Requirements:

The functional suitability directly addresses the requirement to "develop an application for a Hospital database for a mid-size health insurance company to keep track of health claims including patient information, provider(doctor) information, information about patient visits to their doctor as well as prescription drugs prescribed to patients." The application enables efficient management of all these data points with appropriate validation and relationship tracking.

2. Security

Relevance to the Hospital Database Application:

Healthcare applications handle sensitive personal health information (PHI) that is protected by regulations such as HIPAA (in the US) and GDPR (in Europe). Security is critical to protect patient privacy, prevent data breaches, and maintain compliance with healthcare regulations.

How it is Addressed in the Project:

1. **Input Validation:** Comprehensive validation prevents SQL injection and improper data entry:

```
public class FieldValidation {
    private final String fieldName;
    private final int maxLength;
    private final boolean required;
    private final String pattern;
    private final String errorMessage;
}
```

2. **Parameterized Queries:** All database interactions use prepared statements to prevent SQL injection:

```
String query = "SELECT * FROM patient WHERE patientID = ?";
try (Connection conn = DatabaseConnection.getConnection();
     PreparedStatement stmt = conn.prepareStatement(query)) {
    stmt.setString(1, ids[0]);
    ResultSet rs = stmt.executeQuery();
    // Process results...
}
```

3. **Exception Handling:** Proper exception management prevents information leakage:

```
try {
    // Database operations
} catch (SQLException e) {
    throw new DatabaseException("Error saving patient: " + e.getMessage());
}
```

4. **Database Connection Security:** Database credentials are stored as constants rather than in plain text configuration files:

```
private static final String URL =
"jdbc:mysql://localhost:3306/assesment_hospital";
private static final String USER = "root";
private static final String PASSWORD = "root";
```

Alignment with Project Requirements:

Security aligns with the requirement to manage sensitive healthcare data responsibly. While the current implementation provides basic security measures, a production environment would require additional security enhancements such as encryption, access control, and audit logging.

3. Maintainability

Relevance to the Hospital Database Application:

Healthcare systems typically have long lifespans and require regular updates to accommodate changing regulations, medical practices, and organizational needs. High maintainability ensures the system can be efficiently modified, enhanced, and debugged over time.

How it is Addressed in the Project:

1. **Modular Design:** The codebase is organized into clear packages with separation of concerns:

```
src/
├── main/
│   └── java/
│       └── com/
│           └── hospital/
│               ├── dao/          # Data Access Objects
│               ├── exceptions/   # Custom Exceptions
│               ├── gui/          # User Interface
│               ├── models/       # Data Models
│                ├── utils/       # Utilities
│                └── validation/  # Input Validation
```

2. **Design Patterns:** The application employs several design patterns:
    - Data Access Object (DAO) pattern for database operations
    - Factory pattern for creating forms and DAOs
    - Model-View-Controller (MVC) architecture for separation of concerns
3. **Code Reusability:** Abstract classes and interfaces promote reuse:

```java
public interface BaseDAO<T> {
    void save(T entity) throws DatabaseException;
    void update(T entity) throws DatabaseException;
    void delete(String... ids) throws DatabaseException;
    T get(String... ids) throws DatabaseException;
    List<T> getAll() throws DatabaseException;
}
```

4. **Inheritance Hierarchy:** Proper use of inheritance reduces code duplication:

```java
public abstract class Person {
    // Common person attributes and methods
}

public class Patient extends Person {
    // Patient-specific attributes and methods
}

public class Doctor extends Person {
    // Doctor-specific attributes and methods
}
```

5. **Consistent Naming Conventions:** Clear, descriptive naming throughout the codebase.

Alignment with Project Requirements:

Maintainability supports the long-term viability of the system, particularly the requirement that "eventually, the application will be used to track trends and for some extrapolative modelling based on the accumulated data." A maintainable codebase will allow for easier integration of these future analytical capabilities.

Part 2: Code Quality Review

1. Readability

Importance for the Hospital Database Application:

Readability is crucial for healthcare applications where code clarity directly impacts patient safety and data integrity. In a healthcare context, misunderstandings in code interpretation can lead to critical errors in patient data management or treatment information.

Analysis of Current Implementation:

1. **Consistent Formatting:** The codebase maintains consistent indentation and formatting:

```java
public void createTable(String tableType) {
    try {
        // Get the DAO from the factory
        BaseDAO<?> dao = DAOFactory.getDAO(tableType);

        // Create table with CustomTableModel
        JTable table = new JTable(new CustomTableModel(dao.getAll(),
tableType));

        // Add table formatting
        table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        table.getTableHeader().setReorderingAllowed(false);
        table.setRowHeight(25);
        table.setAutoCreateRowSorter(true);

        // Update panel
        JScrollPane scrollPane = new JScrollPane(table);
        mainPanel.removeAll();
        mainPanel.add(scrollPane, BorderLayout.CENTER);
        mainPanel.revalidate();
        mainPanel.repaint();

        // Add right-click functionality
        new TableRightClick(table, tableType);
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(
            this,
            "Error creating table: " + ex.getMessage(),
            "Error",
            JOptionPane.ERROR_MESSAGE
        );
    }
}
```

2. **Clear Method Names:** Methods have descriptive names that indicate their purpose:

```java
private void viewDoctorTable() {...}
private void showPrescriptionForm() {...}
private <T> void showForm(String type, T entity) {...}
```

3. **Organized Constants:** Constants are grouped logically and named clearly:

```
// Constants for table types
private static final String DOCTOR = "doctor";
private static final String PATIENT = "patient";
private static final String DRUG = "drug";
private static final String INSURANCE = "insurance";
private static final String PRESCRIPTION = "prescription";
private static final String VISIT = "visit";
```

4. **Comments:** The code includes descriptive comments, though they could be more consistent:

```
// Method to display search results depending on the different keys, search
types and search parameters in different tables
public void ResultSet(String tableType, String searchType, String...
searchParams) {...}
```

Strategies for Implementation and Assessment:

1. **JavaDoc Implementation:** Add comprehensive JavaDoc comments for all public methods, classes, and interfaces:

```
/**
 * Creates and displays a table based on the specified table type.
 *
 * @param tableType The type of table to create (patient, doctor, etc.)
 * @throws IllegalArgumentException if the table type is invalid
 */
public void createTable(String tableType) {...}
```

2. **Code Reviews:** Implement peer code reviews focused on readability using a checklist:
   - Are method and variable names self-explanatory?
   - Are complex algorithms explained with comments?
   - Is the code consistently formatted?
   - Are there any overly complex methods that should be refactored?

3. **Readability Metrics:** Use tools like SonarQube to measure code complexity (cyclomatic complexity, cognitive complexity) and set acceptable thresholds.

4. **Style Guide:** Adopt and enforce a formal Java style guide such as Google's Java Style Guide.

2. Maintainability

Importance for the Hospital Database Application:

Healthcare systems require frequent updates to accommodate changing medical codes, regulations, insurance requirements, and organizational processes. Maintainable code is essential for cost-effective, timely updates while minimizing the risk of introducing errors.

Analysis of Current Implementation:

1. **Separation of Concerns:** The application properly separates data access, business logic, and presentation layers:

```
// Data Access Layer (DAO)
public class PatientDAO implements BaseDAO<Patient> {...}

// Business Model
public class Patient extends Person {...}

// Presentation Layer
public class PatientForm extends BaseForm<Patient> {...}
```

2. **Abstraction and Inheritance:** The application uses inheritance effectively to reduce code duplication:

```
public abstract class BaseForm<T> {...}
public class PatientForm extends BaseForm<Patient> {...}
public class DoctorForm extends BaseForm<Doctor> {...}
```

3. **Code Duplication:** Some areas show duplication, particularly in the UI event handlers:

```
searchByKeys.addActionListener(event -> {
    patientIDField.setEnabled(true);
    firstNameField.setEnabled(false);
    lastNameField.setEnabled(false);
    // Similar code repeated for other fields
});
searchByFirstName.addActionListener(event -> {
    patientIDField.setEnabled(false);
    firstNameField.setEnabled(true);
    lastNameField.setEnabled(false);
    // Similar code repeated for other fields
});
```

4. **Exception Handling:** Exception handling is present but could be more consistent:

```
catch (SQLException e) {
    e.printStackTrace(); // Some methods only print stack traces
}

catch (SQLException e) {
    throw new DatabaseException("Error retrieving visit: " +
e.getMessage()); // Others throw custom exceptions
}
```

Strategies for Implementation and Assessment:

1. **Code Refactoring Plan:**
   - Create helper methods for repetitive UI field enabling/disabling
   - Standardize exception handling across all DAO classes
   - Extract complex switch statements into separate strategy classes

2. **Unit Testing:** Implement comprehensive unit tests to ensure refactoring doesn't break functionality:

```
@Test
public void testPatientGetAll() {
    PatientDAO dao = new PatientDAO();
    List<Patient> patients = dao.getAll();
    assertNotNull(patients);
    assertFalse(patients.isEmpty());
}
```

3. **Metrics and Analysis:**
   - Track cyclomatic complexity and enforce maximum thresholds
   - Monitor code duplication percentage and aim for reduction
   - Measure test coverage and set minimum requirements

4. **Documentation:**
   - Create and maintain technical documentation including data flow diagrams
   - Document database schema and relationships
   - Maintain a change log to track system modifications

5. **Continuous Integration:**
   - Implement CI/CD pipeline with automated testing
   - Include static code analysis tools in the pipeline
   - Enforce code quality gates before allowing merges

Challenges and Considerations for a Hospital Database Application

1. **Regulatory Compliance:** Healthcare applications must comply with regulations like HIPAA, GDPR, or other regional healthcare data protection laws.

2. **Data Security:** Health information requires robust security measures including encryption, access controls, and audit trails.

3. **System Availability:** Healthcare systems often require high availability with minimal downtime.

4. **Integration Requirements:** Hospital systems typically need to integrate with other healthcare systems, insurance databases, and potentially electronic health record (EHR) systems.

5. **Data Volume Management:** Healthcare systems must efficiently handle large volumes of data while maintaining performance.

6. **User Experience:** Healthcare professionals require intuitive interfaces that don't impede their workflow.

7. **Data Quality and Validation:** Ensuring accurate data entry is critical for patient safety and proper insurance claim processing.

References

1. ISO/IEC 25010:2011. (2011). Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. International Organization for Standardization.

2. Martin, R. C. (2008). Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

3. Fowler, M. (2018). Refactoring: Improving the Design of Existing Code (2nd Edition). Addison-Wesley Professional.

4. Meyer, B. (1997). Object-Oriented Software Construction (2nd Edition). Prentice Hall.

5. Department of Health and Human Services. (2013). HIPAA Administrative Simplification: Regulation Text. Office for Civil Rights.

6. European Parliament and Council. (2016). General Data Protection Regulation (GDPR).