**List of user stories that are fully-implemented:**
User story 1: hire physicians Harrish
User story 2: hire nurses Harrish
User story 3: resign physicians Harrish
User story 4: resign nurses Harrish
User story 5: store and retrieve vital signs of the patient Amira
User story 6: find a patient and display their information Amira, Mehregan
User story 7: Family doctor to get the patient's information and be searched Parmoun
User story 8: Setting the patient's consent form status Parmoun
User story 13: email family doctor with patient information after discharge Mehregan
User story 15: registration, login and validation module: Gaurav

**Main developers of:**
User story 1: Harrish Elango
User story 2: Harrish Elango
User story 3: Harrish Elango
User story 4: Harrish Elango
User story 5: Amira Mohamed
User story 6: Amira Mohamed, Mehregan Mesgari
User story 7: Parmoun Khalkhali Sharifi
User story 8: Parmoun Khalkhali Sharifi
User story 13:Mehregan Mesgari
User story 15: Gaurav Charan Moturi

Harrish - Testing user stories 5,6
Amira - Testing user stories 1, 3
Parmoun - Testing user stories 2,4
Mehregan - Testing user stories 7,15
Gaurav - Testing user stories 8, 13

User story 9: Mehregan Mesgari
User story 10: Gaurav Charan Moturi
User story 11: Mehregan Mesgari
User story 12: Amira Mohamed
User story 14:Mehregan Mesgari
Involved
Testing

# Part 1

**User Story 8: Setting the Patient's Consent Form Status**

*Manual End-to-End Test Cases*

1. Positive Test Case: Setting Consent to Yes
   - Preconditions: Nurse account exists with username "Nur" and password "123".
   - Test Steps:
     1. Launch the application.
     2. Login as Nurse using Username: "Nur" and Password: "123".
     3. Navigate to "Submit Form" section.
     4. Enter a valid patient ID known to be in the system.
     5. Select "Yes" for the consent option.
     6. Confirm the submission.
   - Expected Result:
     - Consent status for the patient in the database updates to "Yes".
     - A confirmation message appears on the GUI indicating successful submission.

2. Negative Test Case: Submitting with Invalid Patient ID
   - Preconditions: Nurse account exists.
   - Test Steps:
     1. Follow steps 1 to 4 from the positive test case, but enter an invalid patient ID.
     2. Attempt to submit the form.
   - Expected Result:
     - An error message appears indicating "Invalid Patient ID".
     - The consent status in the database remains unchanged for any patient.

3. Edge Case: Submitting without Selecting Consent Option
   - Preconditions: Nurse account exists, valid patient ID is used.
   - Test Steps:
     1. Follow steps 1 to 4 from the positive test case.
     2. Do not select any option for consent.
     3. Attempt to submit the form.
   - Expected Result:
     - An error message appears indicating "Select a consent option".
     - No changes are made to the database.

**User Story 13: Email Family Doctor with Patient Information after Discharge**

*Manual End-to-End Test Cases*

1. Positive Test Case: Sending Email upon Discharge
   - Preconditions: Physician account exists, a patient is admitted and ready to be discharged.
   - Test Steps:
     1. Launch the application.
     2. Login as Physician.
     3. Navigate to the discharge section.
     4. Select a patient for discharge.
     5. Confirm discharge and choose to send email to family doctor.
   - Expected Result:

- The system triggers an email to the family doctor's email address on file, containing patient information.
- A confirmation message appears on the GUI indicating successful discharge and email sent.

2. Negative Test Case: Discharging a Patient without an Assigned Family Doctor
- Preconditions: Physician account exists, a patient without an assigned family doctor is admitted.
- Test Steps:
  1. Follow the positive test case steps up to discharge.
- Expected Result:
  - An error or warning message appears indicating that the patient does not have an assigned family doctor, and no email can be sent.
  - The discharge process continues without sending an email.

3. Edge Case: Network Failure During Email Sending
- Preconditions: Physician account exists, patient is admitted and ready to be discharged, simulate a network failure.
- Test Steps:
  1. Follow the positive test case steps up to confirming discharge.
  2. Simulate a network failure before confirming the email sending.
- Expected Result:
  - The system should catch the failure and notify the user with an appropriate message.
  - The discharge process should allow for a retry or manual notification to the family doctor.

# Part 2

**User Story 8: Setting the Patient's Consent Form Status**

*'DatabaseHelper.java'*
- Hardcoded Database Credentials: Database connection details are hardcoded, posing a security risk and reducing flexibility.
- Improper Exception Handling: The use of 'e.printStackTrace()' for exception handling can leak sensitive information.
- Single Responsibility Principle Violation: The class handles multiple database operations, making it complex and difficult to maintain.
- Primitive Obsession: Uses primitive data types ('String', 'int') extensively instead of more descriptive types or objects.

*'NurseGUI.java'*
- Large Class / God Object: Manages UI initialization, user interactions, and database operations, violating the Single Responsibility Principle.

- Long Method: The 'initializeUI' method handles multiple UI components' setup, suggesting the need for decomposition.
- Magic Strings & Numbers: Uses hardcoded strings for titles and messages, and magic numbers for sizes, which should be constants.
- Feature Envy: Excessive interactions with 'DatabaseHelper', indicating that some responsibilities might be better placed elsewhere.

*'Patient.java'*
- Static State for ID Generation: Uses a static variable for patient ID generation, which can lead to thread safety issues.
- Lack of Encapsulation: Directly exposes 'physician' and 'nurse' fields, potentially leading to undesirable state changes.
- Complex 'toString' Method: Manually constructs a string representation, which is error-prone and hard to maintain.

**User Story 13: Email Family Doctor with Patient Information After Discharge**

*'FamilyDoctor.java'* **&** *'Nurse.java'*
- Duplicated Code: Potential duplicated code in handling patient information, suggesting a need for abstraction.
- Feature Envy: Excessive access to data or methods of another class, indicating that some methods might be better placed in the class they interact with the most.

*'NurseGUI.java'* **&** *'NursePatientAddFrame.java'*
- Large Class / God Object: Manages multiple responsibilities from UI components to event handling, violating the Single Responsibility Principle.
- Long Method: Methods handling events could become overly complex, indicating a need for decomposition.

*'PatientInformationGUI.java'*
- Long Parameter List: Methods requiring many parameters suggest the need for grouping parameters into a single object.
- Speculative Generality: Presence of unused methods designed for future flexibility, unnecessarily complicating the code.

**General Across Files**
- Shotgun Surgery: Making changes requires modifications in many classes, indicating poor encapsulation.
- Data Clumps: Repeated groups of variables suggest the need for creating their own object for better organization.

In User Story 8, related to setting the patient's consent form status, several code smells across different components highlight areas needing improvement. In 'DatabaseHelper.java', the practice of hardcoding database credentials compromises security and limits flexibility, while the direct printing of exceptions and the handling of multiple database operations within a single class breach the Single Responsibility Principle

and demonstrate a primitive obsession with basic data types. The 'NurseGUI.java' file suffers from being a Large Class that encompasses UI initialization, user interaction, and database operations, all of which make the class overly complex and violate the Single Responsibility Principle. Additionally, the use of hardcoded strings and numbers, along with an excessive dependency on 'DatabaseHelper', points towards a need for better separation of concerns and code abstraction. Meanwhile, 'Patient.java' indicates a static approach to ID generation that could affect thread safety, an absence of proper encapsulation for critical fields, and a manually crafted 'toString' method, all of which cumulatively suggest a lack of modern Java practices that could enhance code readability, maintainability, and efficiency.

In User Story 13, which focuses on emailing the family doctor with patient information after discharge, the code exhibits various smells that suggest areas for refactoring. Both 'FamilyDoctor.java' and 'Nurse.java' show signs of duplicated code, especially in handling patient details, indicating a potential for creating more abstract methods or classes to encapsulate common functionality. This duplication is coupled with instances of feature envy, where methods excessively use data or functionalities of other classes, suggesting that some methods may be more appropriately located within the classes they primarily interact with. For the UI components, 'NurseGUI.java' and 'NursePatientAddFrame.java' are identified as Large Classes that take on too many responsibilities, from managing UI elements to processing user events, thus violating the Single Responsibility Principle. Additionally, these files contain long methods associated with event handling, signalling a need for breaking down complex methods into smaller, more manageable pieces. 'PatientInformationGUI.java' presents its challenges, with methods that require long lists of parameters, hinting at the possibility of encapsulating these parameters into a single object for cleaner code. Moreover, the presence of speculative generality, through methods that are currently unused and intended for potential future use, complicates the codebase unnecessarily. Addressing these code smells across the various components involved in User Story 13 could lead to a more maintainable, readable, and efficient codebase.

## Addressing the Code Smells

Addressing code smells in User Stories 8 and 13 involves prioritizing issues based on their impact on security, maintainability, and scalability. For User Story 8, the most critical issue is the hardcoding of database credentials in 'DatabaseHelper.java', as it poses a significant security risk and limits the code's flexibility across environments. This can be fixed by externalizing credentials into configuration files or environment variables. The next severe smell is improper exception handling, which can leak sensitive information; adopting a more robust logging framework would mitigate this. The Single Responsibility Principle Violation in both 'DatabaseHelper.java' and 'NurseGUI.java' significantly affects maintainability; refactoring these classes to ensure they have a single responsibility would greatly enhance code quality. Primitive Obsession and Magic Strings & Numbers indicate a need for using more descriptive types or constants, improving readability and making the codebase more resilient to changes.

In 'NurseGUI.java', the Large Class / God Object and Long Method smell suggests the class is doing too much, which complicates understanding and modifying the code. Decomposing the class and methods into smaller, focused units would alleviate this. The Feature Envy in 'NurseGUI.java' highlights misplaced responsibilities, which could be resolved by refactoring methods closer to the data they manipulate most.

For 'Patient.java', the use of a static variable for ID generation creates thread safety issues and makes testing challenging; adopting a database-managed approach or a dedicated ID generator could resolve this. The lack of Encapsulation and a Complex 'toString' Method suggests the need for better data hiding and leveraging IDE-generated 'toString' methods for maintainability.

In User Story 13, Duplicated Code between 'FamilyDoctor.java' and 'Nurse.java' is concerning as it bloats the codebase and complicates updates; abstracting common functionality into a shared class or method would be beneficial. Similarly, Feature Envy suggests a lack of proper class responsibility distribution, which could be fixed by relocating methods to the classes they interact with the most.

For UI classes in User Story 13, addressing the Large Class / God Object and Long Method smells involves breaking down responsibilities and simplifying complex methods. This would make the UI code easier to manage and extend. The long Parameter List in 'PatientInformationGUI.java' can be remedied by grouping parameters into objects, enhancing method signatures' clarity. Meanwhile, Speculative Generality introduces unnecessary complexity; pruning unused methods would streamline the code.

General Across Files, Shotgun Surgery indicates poor encapsulation and scattered responsibilities, necessitating a review of the design to better encapsulate functionality. Data Clumps across various classes point to a need for creating new types to group related data, improving code organization and readability.

Tackling these code smells by severity and their impact on the application's security, maintainability, and scalability ensures a strategic approach to refactoring, ultimately leading to a more robust, understandable, and flexible codebase.