The Database:

Table name: user

userId	number
userName	varchar
email	varchar
paswd	varchar
title 1.Manager 2. Developer 3. Tester	varchar

Table name: project

projectId	number
projectName	varchar
projectDetail	varchar
startDate	date
status	varchar
manager	userId* (Foreign key)
developer	userId*
tester	userId*

Table name: bug

bugId	number
bugTitle	varchar
bugDetail	varchar
projectId	projectId*(foreign key)
createdBy	userId of Tester
openDate	date
assignedTo	userId of developer
closedBy	userId of manager
closedDate	date
bugStatus > open >closed	varchar

Explanation about the implementation for the USER:

```java
package dao;

import java.util.List;

public interface UserDao {
    void addUser(User user)throws UserAlreadyExsistException ;
    User getUserById(int userId);
    List<User> getUsersByTitle(String title);
    User getUserByEmail(String email);
    List<User> getAllUsers();
    void deleteUser(int userId);
    void updateUser(User user);
}
```

Figure : Interface for the User Storage

1. **`void addUser(User user) throws UserAlreadyExistsException;`**
    - This function is used to add a new user to the system.
    - It takes a `User` object `user` as a parameter, representing the user to be added.
    - It may throw a custom exception called `UserAlreadyExistsException` if a user with the same details (ID) already exists in the system.
2. **`User getUserById(int userId);`**
    - This function is designed to retrieve a user by their unique ID.
    - It takes an integer `userId` as a parameter and returns a `User` object representing the user with that ID.
3. **`List<User> getUserByTitle(String title);`**
    - This function is used to retrieve a list of users with a specific title or role.
    - It takes a `String` parameter `title` representing the title or role and returns a list of `User` objects matching that title.
4. **`User getUserByEmail(String email);`**
    - This function is meant to fetch a user by their email address.
    - It takes a `String` parameter `email` and returns a `User` object associated with that email.

5. **`List<User> getAllUser();`**
   - This function is used to retrieve a list of all users in the system.
   - It returns a list of `User` objects containing information about all users.

6. **void updateUser(User user);**
   - This function is for updating the information of an existing user.
   - It takes a `User` object `user` as a parameter, which contains the updated user details.

7. **void deleteUser(int userId);**
   - This function is used to delete a user from the system based on their unique ID.
   - It takes an integer `userId` as a parameter and removes the user with that ID from the system.

Code explanation for each method in the `UserServiceImpl` class implementing "UserService" interface:

```java
import java.util.List;


public interface UserService {

        void addUser(User user) throws UserAlreadyExsistException;
        User getUserById(int userId) throws UserNotFoundException;
        List<User> getUsersByTitle(String title)throws UserNotFoundException;
        User getUserByEmail(String email)throws UserNotFoundException;
        List<User> getAllUsers()throws UserNotFoundException;
        void deleteUser(int userId)throws UserNotFoundException;
        void updateUser(User user)throws UserNotFoundException;
}
```

Figure:The interface for service layer of user

1. **`addUser(User user)`:**
   This method adds a new user to the system by calling the `addUser` method of the `UserDao`. It handles the `UserAlreadyExistException` if the user already exists.

2. **`getUserById(int userId)`:**
   Retrieves a user by their unique ID by calling the `getUserById` method of the `UserDao`. If the user is not found, it throws a `UserNotFoundException`.

3. **`getUsersByTitle(String title)`:**
   This method fetches a list of users with a specific job title from the database using the `getUsersByTitle` method of the `UserDao`. If no users match the given title, it raises a `UserNotFoundException`. This functionality can be used in the frontend to streamline project assignments by allowing filtering of users based on their job titles, making it easier to identify and assign projects to users with the desired roles.

4. **`getUserByEmail(String email)`:**
   Retrieves a user by their email address by calling the `getUserByEmail` method of the `UserDao`. If the user is not found, it throws a `UserNotFoundException`.

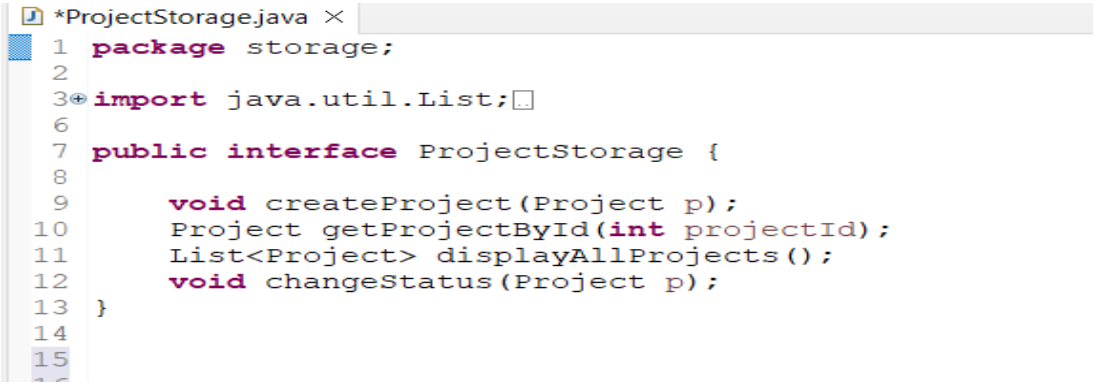
5. **`getAllUsers()`:**
   Retrieves a list of all users by calling the `getAllUsers` method of the `UserDao`. If no users are found, it throws a `UserNotFoundException`.

6. **`deleteUser(int userId)`:**
   Deletes a user by their unique ID by calling the `deleteUser` method of the `UserDao`. If the user is not found, it throws a `UserNotFoundException`.

7. **`updateUser(User user)`:**
   Updates an existing user's information by calling the `updateUser` method of the `UserDao`. If the user is not found, it throws a `UserNotFoundException`.

These methods encapsulate the core functionality of managing users, including adding, retrieving, updating, and deleting user records, while also handling exceptions when necessary.

Explanation about the implementation for the Project Interface in Storage/DAO:



*ProjectStorage.java

```
1  package storage;
2
3  import java.util.List;
6
7  public interface ProjectStorage {
8
9      void createProject(Project p);
10     Project getProjectById(int projectId);
11     List<Project> displayAllProjects();
12     void changeStatus(Project p);
13 }
14
15
```

1. **void createProject(Project p);**

    - This method in the storage layer is used to create a Project. This functionality is only restricted to the Manager.
    - This method does not have a return type and hence takes “void” but does have a formal argument “Project” for which we will take actual values for each of the fields in the Main Class and store all of them in another “Project” entity and send that to this method as an actual argument through the service layer, where we actually call this method.

2. **Project getProjectById(int projectId);**

    - This method in the storage layer is used to display a specific Project based on the “ID” of the same.Anybody in the team can view their team specific or to be more precise, specifically assigned projects but a Manager can view all the projects under him.
    - This method has a “Project” return type, which means that it gives back a “Project” to the related method in the service layer calling this method from the dao/storage layer.

3. **List<Project> displayAllProjects();**

    - This method in the storage layer is used to display all the projects and we achieve the same by storing it in a “List” which is type specific as it includes the generic “List<Project>”. This “List” is the return type of this

method which means that it returns a list to the related method in the service layer calling this method from the dao/storage layer.

- The Manager can view all the projects and can deal with multiple projects whereas a developer can view only the single project he/she has been assigned to at a point of time. A tester can view a maximum of two projects under the manager assigned to him/her.

4. **void changeStatus(Project p);**

- A project by default when created is set to “in-progress” status and later when the project is free of bugs and is completed, then comes the purpose of this functionality. This has a “void” return type and takes in a formal argument of “Project”. The fact that this method's code verifies and authenticates the project whose status has to be altered by getting its "ID" can be used to summarize the code's functionality.
- The status of the project can be changed by the Manager only if all the bugs have been closed after resolving them.

Explanation about the implementation for the Project Interface in Service:

```
import java.util.List;

public class ProjectServiceImpl implements ProjectService{
    ProjectStorage storage=new ProjectStorageImpl();

    @Override
    public void createProject(Project p) throws ProjectAlreadyExistsException {
        storage.createProject(p);

    }

    @Override
    public Project getProjectById(int projectId) throws NoProjectException {
        Project p=storage.getProjectById(projectId);
        if(p==null) {
            throw new NoProjectException();
            }
        return p;
    }

    @Override
    public List<Project> displayAllProjects() throws NoProjectException {
        List<Project> list=storage.displayAllProjects();
        if(list.isEmpty()) {
            throw new NoProjectException("No projects in the list");
        }
        return list;
    }

    @Override
    public void changeStatus(Project p) throws NoProjectException {
        Project p1=storage.getProjectById(p.getProjectId());
        if(p1==null) {
            throw new NoProjectException("No project exists with that id");
        }
        storage.changeStatus(p1);

    }
```

- The above img. shows the Service Layer Interface's Implementation and handles all the exceptions that the methods in the storage layer might throw. A few of them are explicitly propagated to the methods calling each of these in the view layer and handled there using a "Try-Catch" block. The methods in the service layer might have different names, but either have a similar functionality or are related to the methods in the storage layer which is absolutely necessary for communication between the layers, which sums up " Layered Architecture". This layer purely contains business logic and CANNOT have "print" statements or any other storage functionality.

- To connect to the storage layer, a memory of the implementation class of the storage interface is created and assigned to its parent, which is the interface itself. The reference used here can access the methods of the storage layer in each of the methods accordingly.

  Methods:

1. **void createProject(Project p) throws ProjectAlreadyExistsException;**

    - This method checks all the possibilities this method can ideally throw. Like for suppose, if a project that we're creating already exists, it will throw the "ProjectAlreadyExistsException".

2. **Project getProjectById(int projectId) throws NoProjectException;**

    - This method checks all the possibilities it can ideally throw. According to the method naming above, we deduce that it can throw a "NoProjectException" which can usually have two reasons:

        - ☐ A user might have not given the ID during the dynamic input process.
        - ☐ The project ID trying to be accessed by a team member doesn't exist at all.

3. **List<Project> displayAllProjects() throws NoProjectException;**

    - This method can ideally throw a "NoProjectException" too. If a user triggers this method and say for suppose, no projects are present in the database, then this error occurs.

4. **void changeStatus(Project p) throws NoProjectException;**

    - This method might throw a “NoProjectException” too. This occurs when a user is trying to change the status of the project that never existed in the database only. This method can potentially have other exceptions too like InvalidIdException, ProjectAlreadyUpdatedException. This applies to all the other methods too!

```
import java.util.List;

public interface ProjectService {
    void createProject(Project p) throws ProjectAlreadyExistsException;
    Project getProjectById(int projectId) throws NoProjectException;
    List<Project> displayAllProjects() throws NoProjectException;
    void changeStatus(Project p) throws NoProjectException;

}
```

The above img. displays the service interface named as “ProjectService”.

Project in Entity Class:

- The Project code also includes and stores various data types for its parameters in the entity class. The entity class has two important parameters out of all, namely “startDate” and “teamMembers[ ]”. The start date stores the input in a date format and it is vital for this to take a value minimum 2 days prior to the current date.

- The team members parameter is an array that consists of the team members in a specific project. It stores the user id’s of the team members in the array. The potential problem with declaring this as an array is the incompatibility with the data type of the variable related to it in the MySQL Database. A database cannot store an array, hence there can either be duplication of values or we can implement the same using a complex JOIN query solution.

```java
package entity;

import java.time.LocalDate;

public class Project {
    private int projectId;
    private String projectName;
    private String description;
    private LocalDate startDate;
    private String status;
    private int teamMembers[];
    public int getProjectId() {
        return projectId;
    }
    public void setProjectId(int projectId) {
        this.projectId = projectId;
    }
    public String getProjectName() {
        return projectName;
    }
    public void setProjectName(String projectName) {
        this.projectName = projectName;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public LocalDate getStartDate() {
        return startDate;
    }
    public void setStartDate(LocalDate startDate) {
        this.startDate = startDate;
    }
    public String getStatus() {
        return status;
    }
```

Explanation about the implementation of BUG :

Overview
The Bug class represents a bug entity in a software application. It encapsulates information about a bug, including a unique identifier, title, and description. This class is used to create and manage bug objects, making it an essential part of the bug tracking system.

Constructors
public Bug()
This is the default constructor for the Bug class. It creates an empty Bug object with default values for its attributes.

public Bug(int uniqueID, String title, String description)
This constructor creates a Bug object with the specified parameters:

uniqueID (type int): A unique identifier for the bug.
title (type String): The title or summary of the bug.
description (type String): A detailed description of the bug.
Methods
Accessor Methods
These methods allow you to retrieve the values of the Bug object's attributes:

public int getUniqueID(): Returns the unique identifier of the bug.
public String getTitle(): Returns the title of the bug.
public String getDescription(): Returns the description of the bug.
Mutator Methods
These methods allow you to set or update the values of the Bug object's attributes:

public void setUniqueID(int uniqueID): Sets the unique identifier of the bug.
public void setTitle(String title): Sets the title of the bug.
public void setDescription(String description): Sets the description of the bug.
@Override public String toString()
This method is overridden from the Object class to provide a string representation of the Bug object. It returns a formatted string containing the unique identifier, title, and description of the bug.

BugService Overview
The BugService interface is a contract for managing and interacting with Bug entities in a software application. It defines a set of methods that allow for registering new bugs, closing existing bugs, and viewing the status of a bug based on its unique identifier (ID). This interface serves as the foundation for any class or service that intends to handle bug-related operations.

Methods
void registerBug(Bug bug) throws BugAlreadyExistsException
This method is used to register a new bug in the system. It takes a Bug object as a parameter, representing the bug to be registered. If a bug with the same properties (e.g., title, description) already exists in the system, a BugAlreadyExistsException is thrown.

Parameters:

bug (type Bug): The bug to be registered in the system.
Throws:

BugAlreadyExistsException: Thrown if a bug with the same properties already exists in the system.
void closeBug(Bug bug) throws BugAlreadyClosedException
This method is used to close an existing bug. It takes a Bug object as a parameter, representing the bug to be closed. If the bug is already closed, a BugAlreadyClosedException is thrown.

Parameters:

bug (type Bug): The bug to be closed.
Throws:

BugAlreadyClosedException: Thrown if the bug is already closed when attempting to close it.
String viewBugStatus(int bugId)
This method is used to view the status of a bug based on its unique identifier (ID). It takes an int parameter bugId representing the unique ID of the bug, and returns a String representing the current status of the bug.

Parameters:

bugId (type int): The unique identifier of the bug for which the status is to be retrieved.
Returns:

String: A string representing the current status of the bug.

BugAlreadyClosedException Documentation
Overview
The BugAlreadyClosedException class is an exception type used in a software application to indicate that an attempt was made to close a bug that is already closed. This exception extends the standard Exception class and can include a custom error message to provide additional information about the exception.

Constructors
public BugAlreadyClosedException()
This is the default constructor for the BugAlreadyClosedException class. It creates an exception instance without a specific error message.

public BugAlreadyClosedException(String msg)
This constructor creates a BugAlreadyClosedException instance with a custom error message provided as a parameter.

Parameters:

msg (type String): A custom error message describing the reason for the exception.

BugAlreadyExistsException Documentation
Overview
The BugAlreadyExistsException class is an exception type used in a software application to indicate that an attempt was made to register a bug that already exists in the system. This exception extends the standard Exception class and can include a custom error message to provide additional information about the exception.

Constructors
public BugAlreadyExistsException()

This is the default constructor for the BugAlreadyExistsException class. It creates an exception instance without a specific error message.

public BugAlreadyExistsException(String msg)
This constructor creates a BugAlreadyExistsException instance with a custom error message provided as a parameter.

Parameters:

msg (type String): A custom error message describing the reason for the exception.