Project title	Project Management Application
Author(s)	Beáta Keresztes, Borbála Fazakas
Group	30422

Project Management Application

Overview

Our application is intended to help teams organize their shared projects, assign responsibilities and track the progress. The functionalities provided by our app are expected to increase the efficiency of teamwork, especially in a virtual environment.

Goals and Functionalities

Based on the main goals presented above, we plan to implement the following functionalities for the app:

1. User Registration and Authentication

A new user can register with a username and a password. These will be required later when the user wants to sign in and access the further functionalities of the application.

2. Team Creation and Management

Any user can create a new team with a specific name, and thus the creator will become the manager of it.

As a manager, the user has access to the following functionalities:

- Delete the team
- Add new team members based on the username
- Remove members from the team
- Change the name of the team
- Regenerate the unique code of the team which is used for joining the team
- Pass the manager status to another member

Furthermore, users other than the manager may

- Join the team using the unique code generated for the team
- Leave the team (the manager can leave the team only after passing the manager status to someone else)

3. Projects

a. Creation and Management

Any member of a team may create a new project which has the following attributes:

- Name
- Deadline

- Assignee: a team member who is responsible for this specific project
- Supervisor: a team member who monitors the progress, evaluates the proposed final version and decided whether it fulfills the requirements
- Description(optional)
- Status (to do/in progress/turned-in/finished)
- Importance (low/medium/high)

The creator of the project will automatically become the supervisor, and thus will gain access to the following options:

- Change the name, deadline, assignee, importance and description of the project
- The supervisor is the only one who can mark a project as “finished” after the assignee turned it in. If the supervisor decides that the requirements are not fulfilled, they may reject the project and set its status back to “to do” or “in progress”
- The supervisor may also pass the supervisor status to any other user, who is the member of the team

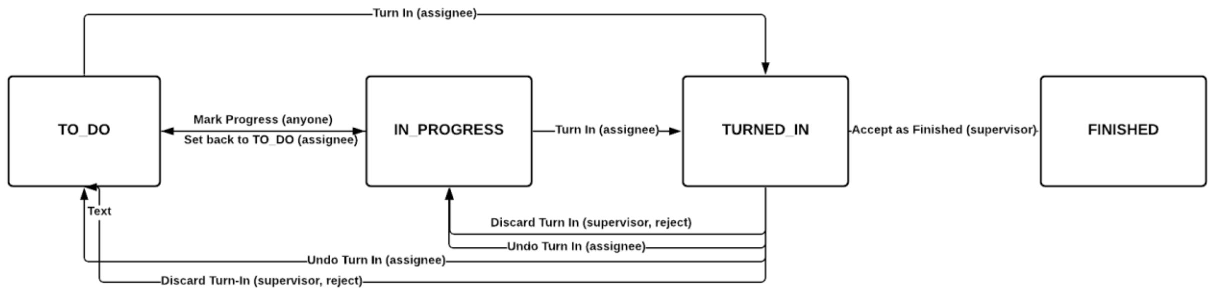
The Assignee of the project can

- Turn-in the project
- Undo the turn-in and set back the status to “to do” or “in progress”
- Set the status back from “in progress” to “to do”

Any member of the team in which the project was created can:

- Add comments to the project
- Mark the progress if the project’s status is “to do”

The following diagram illustrates the allowed status changes for a project, with the roles for which the operation is allowed in the brackets:



b. Progress Tracking

Any user can easily track the progress of

- Projects assigned to them
- Projects supervised by them

To make the monitoring process easier, they can view the projects categorized by

- Status
- Status with respect to the deadline
- Assignee
- Supervisor

And they can also sort the project based on

- The deadline
- The status
- The importance

Technologies used

The project will be implemented in form of a desktop application, using only Java language.

In order to create the user interface, we decided to use the Java Swing library. With the aid of the components provided by this framework, we can create a user-friendly interface for our application, with windows, forms, dialogs, etc.

Additionally, a database is needed in order to store the data across multiple executions of the program. Having the goal of easily transferring the app in form of a jar file, we needed to choose a lightweight embedded database, so we finally decided to implement our app with SQLite. Database access is achieved through the JDBC API and the SQLite driver.

Furthermore, to make building and importing dependencies easier, we use the Maven project management tool.

Design

Regarding the overall design of the implementation, we stick to the **MVC pattern** in order to achieve a high-level separation of concerns.

- **Mode**l: Implements the business layer and the data access layer for the application.
- **View**: represents the front-end part of the application. It is responsible for the creation of the user interface, displaying the model in a user-friendly way, and allowing the user to interact through it with the components of the model.
- **Controller**: The controller is the component which handles all the actions of the UI and sends the corresponding command to the model.

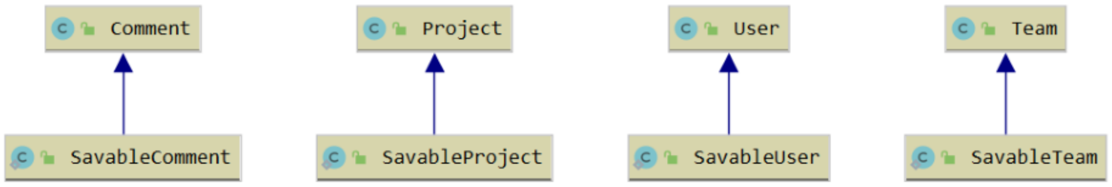
The communication between the components is done as follows: the user interacts with the view, these events are handled by the controller which sends command to the model to perform updates. When the model is updated, it fires events through the **Observer pattern,** and these events are again handles by the controller, which updated the UI corresponding to the data changes signaled by the model.

Implementation Details

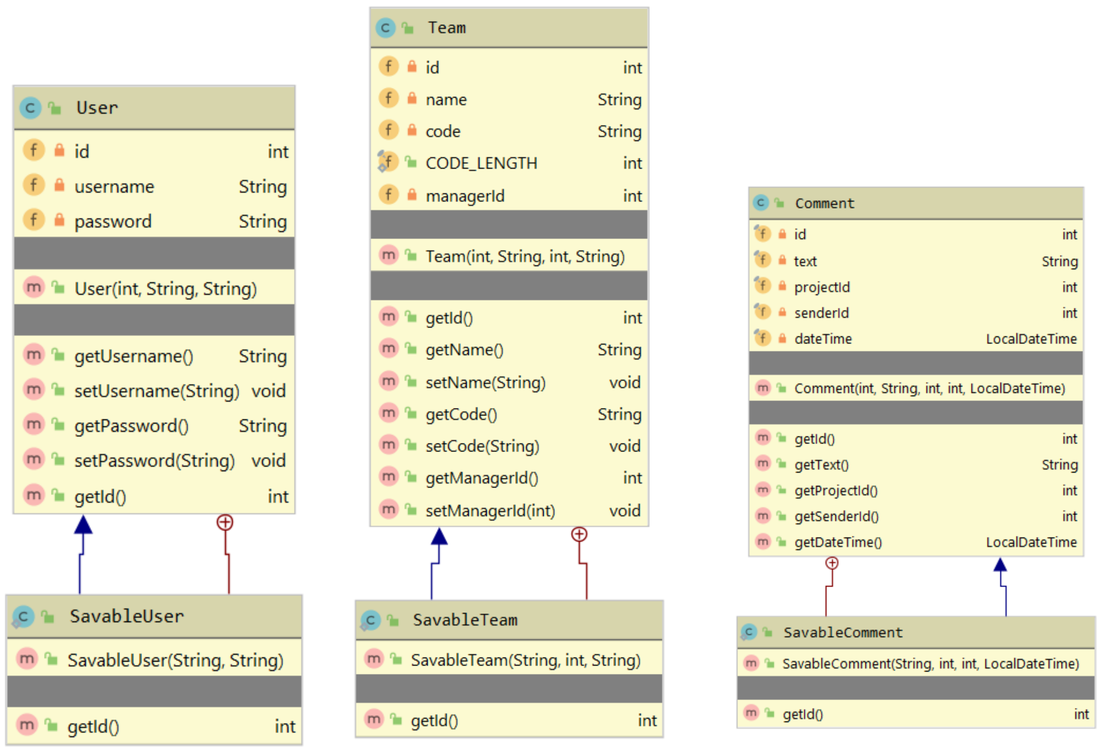
a. Model

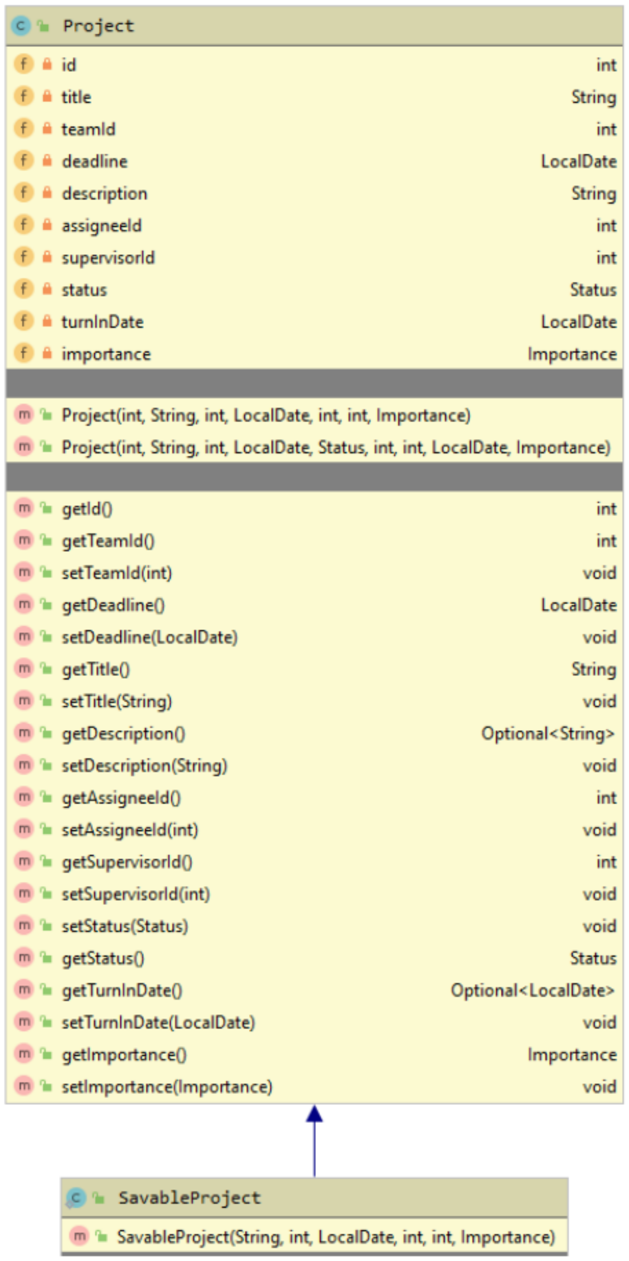
The model is responsible for the business logic and for updating and getting data from the database. Based on these two main purposes, we can categorize the classes used for implementing the model in 3 categories:

1. POJO classes for the main entities



- **User**
    - id (which identifies them in the database)
    - userName (unique)
    - password (required for the login)
- **Team**
    - id (which identifies it in the database)
    - name (unique)
    - code (a numeric code generated by the app. Other users may join the team if they know the code).
    - managerId (the id of the user who is the manager of the team)
- **Project**
    - id (which identifies it in the database)
    - title
    - teamId (the id of the team to which the project belongs)
    - deadline (the date until which the project should be finished)
    - Description
    - assigneeId (the Id of the user to whom the project is assigned)
    - supervisorId (the id of the user who supervises the progress of the project)
    - status (see Status Enum later)
    - turnInDate (the date on which the project has been turn in the last time. If the project is not turned in now, it is null)
    - Importance (see Importance Enum later)
- **Comment**
    - id (which identifies it in the database)
    - text
    - projectId (the id of the project to which it belongs)
    - senderId (the id of the user who sent the comment)
    - dateTime (the exact time when it was sent)


User
id int
username String
password String
User(int, String, String)
getUsername() String
setUsername(String) void
getPassword() String
setPassword(String) void
getId() int
SavableUser
SavableUser(String, String)
getId() int
Team
id int
name String
code String
CODE_LENGTH int
managerId int
Team(int, String, int, String)
getId() int
getName() String
setName(String) void
getCode() String
setCode(String) void
getManagerId() int
setManagerId(int) void
SavableTeam
SavableTeam(String, int, String)
getId() int
Comment
id int
text String
projectId int
senderId int
dateTime LocalDateTime
Comment(int, String, int, int, LocalDateTime)
getId() int
getText() String
getProjectId() int
getSenderId() int
getDateTime() LocalDateTime
SavableComment
SavableComment(String, int, int, LocalDateTime)
getId() int



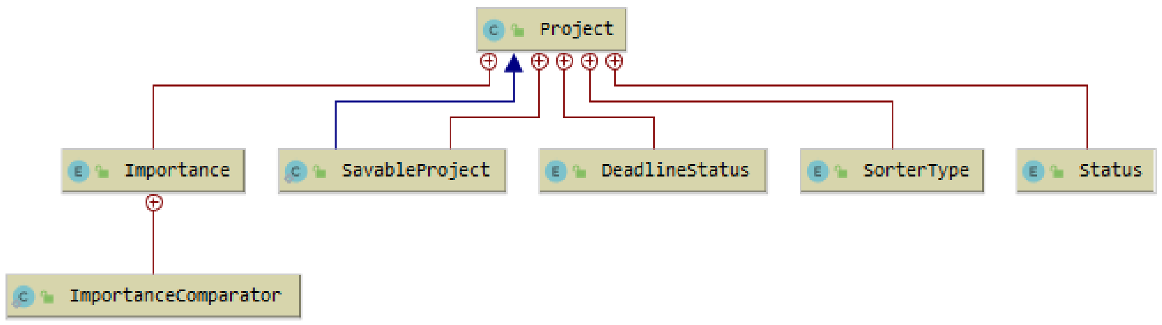
```
Project
id                                   int
title                                String
teamId                               int
deadline                             LocalDate
description                          String
assigneeId                           int
supervisorId                         int
status                               Status
turnInDate                           LocalDate
importance                           Importance

Project(int, String, int, LocalDate, int, int, Importance)
Project(int, String, int, LocalDate, Status, int, int, LocalDate, Importance)

getId()                              int
getTeamId()                          int
setTeamId(int)                       void
getDeadline()                        LocalDate
setDeadline(LocalDate)               void
getTitle()                           String
setTitle(String)                     void
getDescription()                     Optional<String>
setDescription(String)               void
getAssigneeId()                      int
setAssigneeId(int)                   void
getSupervisorId()                    int
setSupervisorId(int)                 void
setStatus(Status)                    void
getStatus()                          Status
getTurnInDate()                      Optional<LocalDate>
setTurnInDate(LocalDate)             void
getImportance()                      Importance
setImportance(Importance)            void

SavableProject
SavableProject(String, int, LocalDate, int, int, Importance)
```

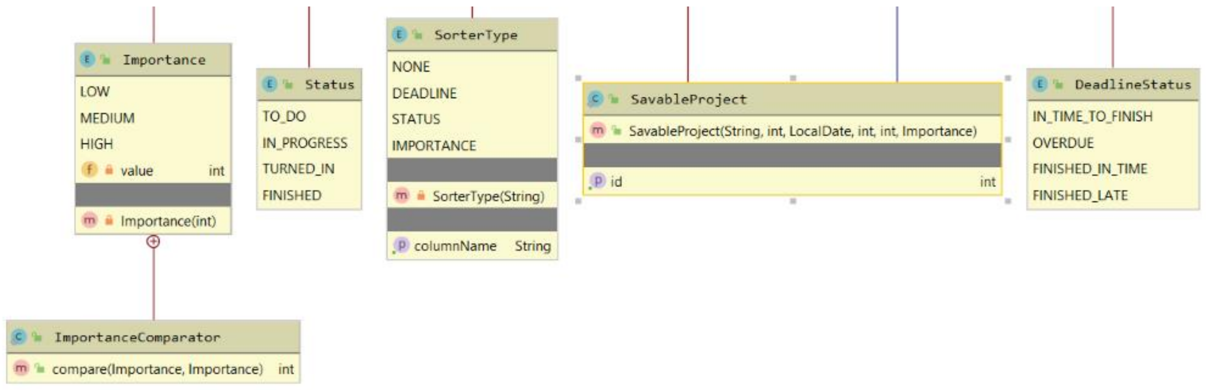
Remarks:

- Given that the data is stored in a database, the POJO classes don't have actual connections to each other, but they are only storing ids of the other entities (for example, in a Project object the assignee is not stored as a User, but only the id of the assignee is stored).
- The **SavableX classes** (SavableProject, SavableUser, …) are extensions of the basic POJO classes (Project, User, …), serving the purpose of holding data of one entity to be saved in the database: the SavableX objects do not have an id, and if a client tries to access their id, an exception is thrown. The reason for this is that the id is generated by the database, thus before saving the entity in the database it is unknown.

To the Project class there are multiple Enums associated, intended to increase code readability and safety. These correspond to the constants stored in the database.

- **Importance**: LOW/MEDIUM/HIGH, with a corresponding ImportanceComparator
- **Status**:
  - TO_DO: work not started yet (default status of new projects)
  - IN_PROGRESS: work has been started (progress can be marked by any user)
  - TURNED_IN/FINISHED
- **SorterType**: NONE/DEADLINE/STATUS/IMPORTANCE, specifies the attribute by which the projects returned from the database must be sorted
- **DeadlineStatus**:
  - IN_TIME_TO_FINISH: the project is not finished yet, but the deadline is today or later than today
  - OVERDUE: the project is not finished yet, and the deadline was earlier than today
  - FINISHED_IN_TIME: the project is finished, and the turn-in date is not later than the deadline
  - FINISHED_LATE: the project is finished but the turn-in date is later than the deadline

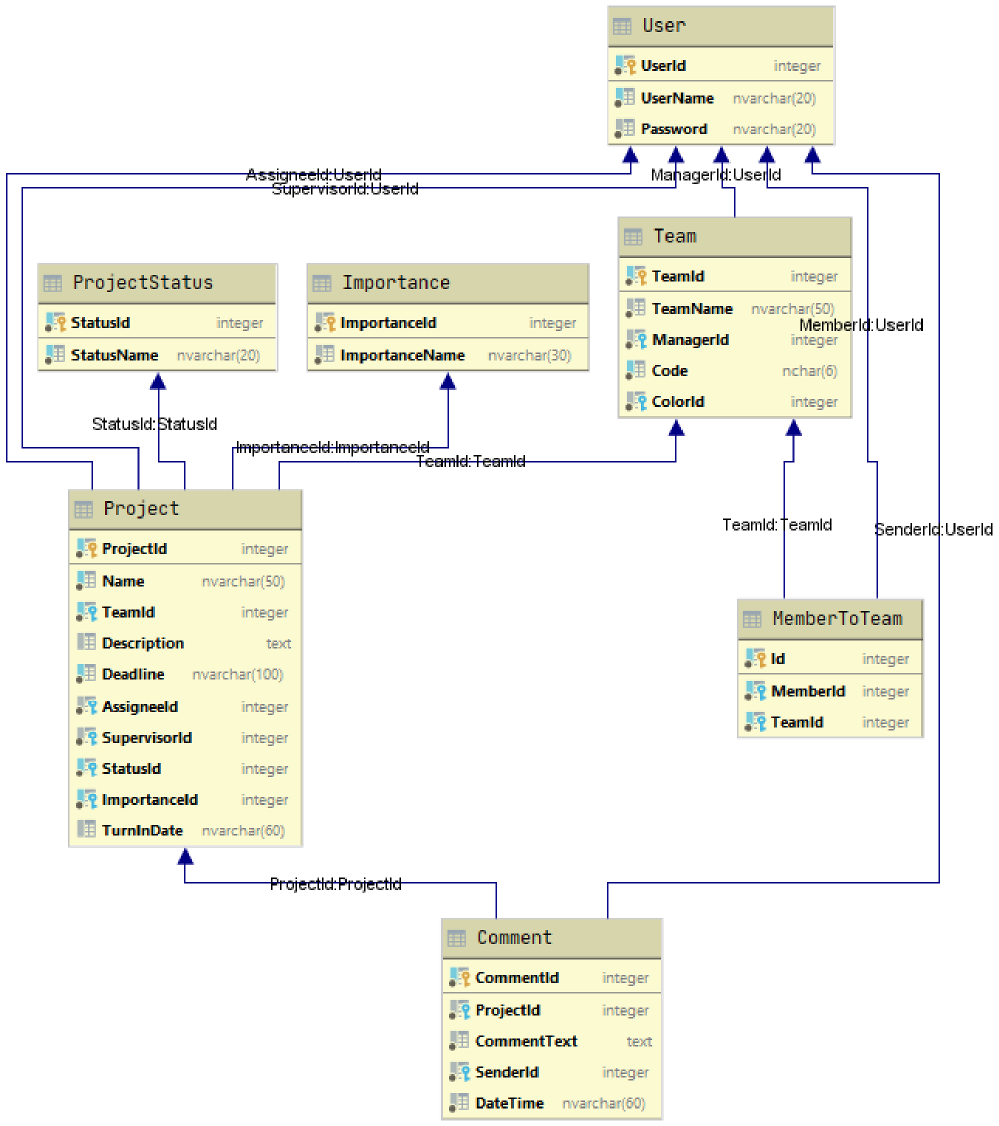


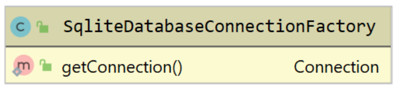


2. Data Access Layer

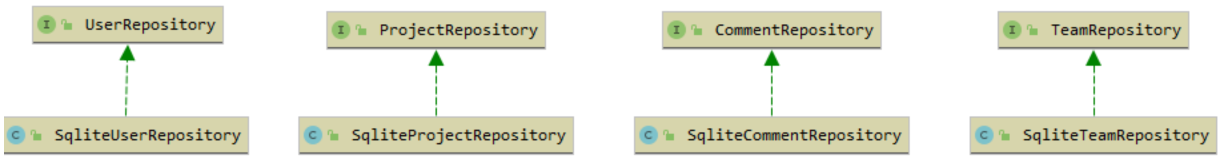
The database access in our application is guaranteed through the **JDBC API**, together with the **SQLite JDBC driver** required specifically to access an SQLite database.

The diagram of the underlying **SQLite database** looks as follows:



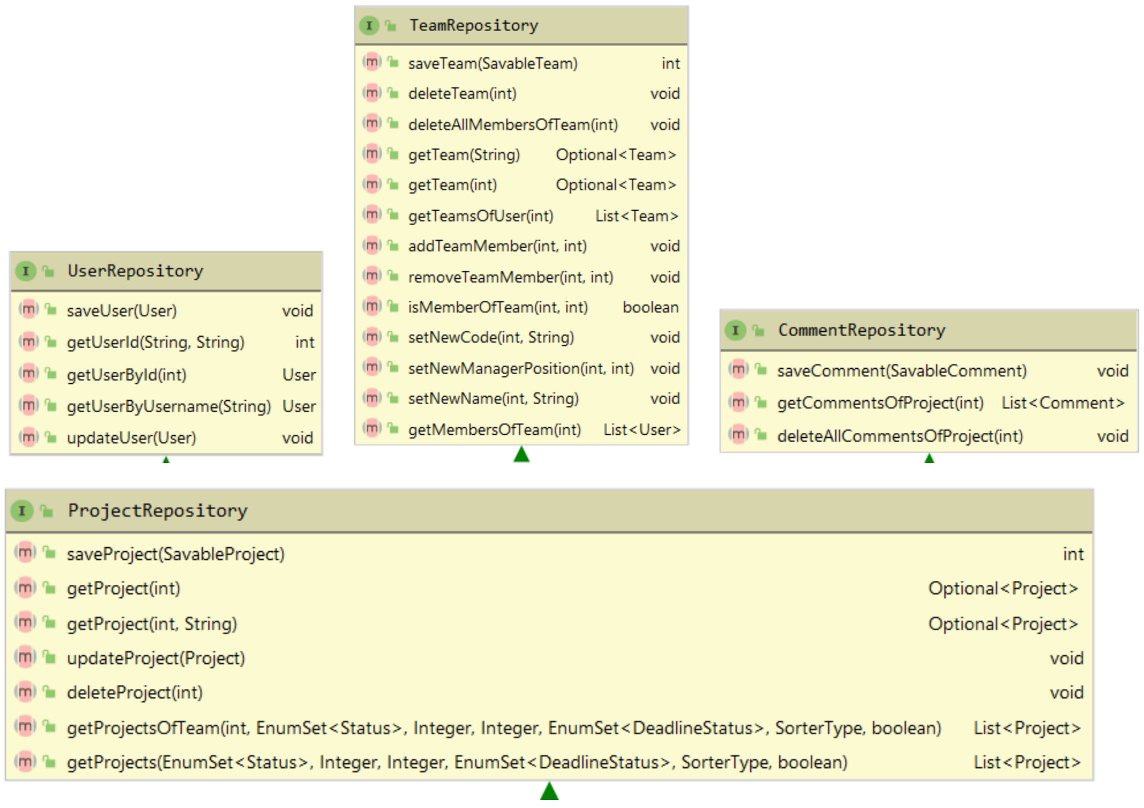


The database access is provided through a Connection object which is created by the **SqliteDatabaseConnectionFactory** class in a factory method, whenever it is needed. Given that the SQL server caches the connections, the connection created by SqliteDatabaseConnectionFactory is opened and closed each time it is used, but this does not cause any decrease in the execution speed. However, this method helps avoid issues such as locking the database due to reaching the maximum number of connections, and thus making it inaccessible for other clients, or running into problems due to temporary connectivity errors with the database.



Furthermore, the database-queries and updates needed to be implemented by the repositories for the different entities are defined in the following interfaces, as shown on the below class diagrams:

- **UserRepository**
- **TeamRepository**
- **ProjectRepository**
- **CommentRepository**



TeamRepository	
saveTeam(SavableTeam)	int
deleteTeam(int)	void
deleteAllMembersOfTeam(int)	void
getTeam(String)	Optional<Team>
getTeam(int)	Optional<Team>
getTeamsOfUser(int)	List<Team>
addTeamMember(int, int)	void
removeTeamMember(int, int)	void
isMemberOfTeam(int, int)	boolean
setNewCode(int, String)	void
setNewManagerPosition(int, int)	void
setNewName(int, String)	void
getMembersOfTeam(int)	List<User>

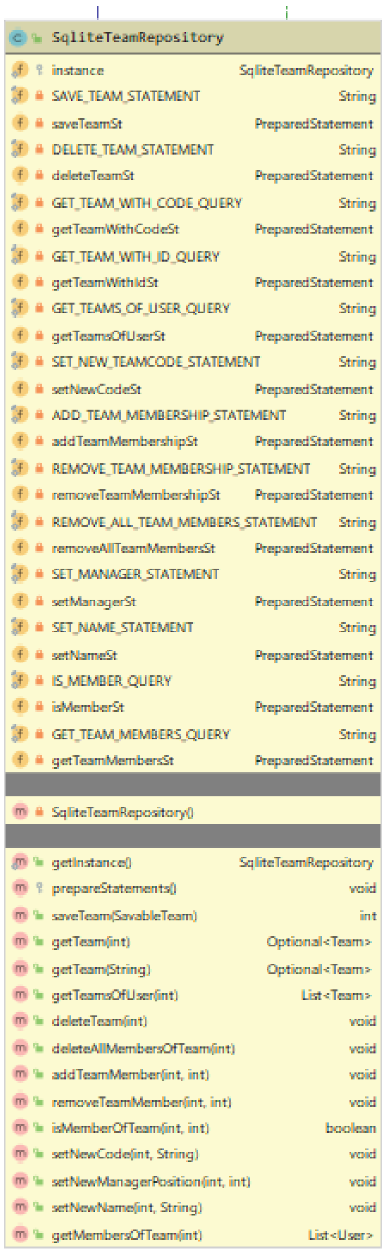
UserRepository	
saveUser(User)	void
getUserId(String, String)	int
getUserById(int)	User
getUserByUsername(String)	User
updateUser(User)	void

CommentRepository	
saveComment(SavableComment)	void
getCommentsOfProject(int)	List<Comment>
deleteAllCommentsOfProject(int)	void

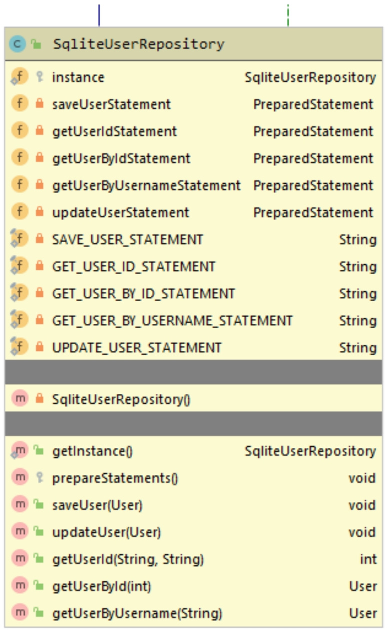
ProjectRepository	
saveProject(SavableProject)	int
getProject(int)	Optional<Project>
getProject(int, String)	Optional<Project>
updateProject(Project)	void
deleteProject(int)	void
getProjectsOfTeam(int, EnumSet<Status>, Integer, Integer, EnumSet<DeadlineStatus>, SorterType, boolean)	List<Project>
getProjects(EnumSet<Status>, Integer, Integer, EnumSet<DeadlineStatus>, SorterType, boolean)	List<Project>

Then, for each Repository, an implementation relying on an SQLite database is provided:

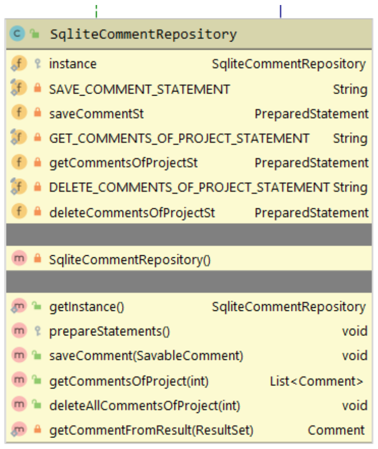
- **SqliteUserRepository** extends Repository implements UserRepository
- **SqliteTeamRepository** extends Repository implements TeamRepository
- **SqliteProjectRepository** extends Repository implements ProjectRepository
- **SqliteCommentRepository** extends Repository implements CommentRepository



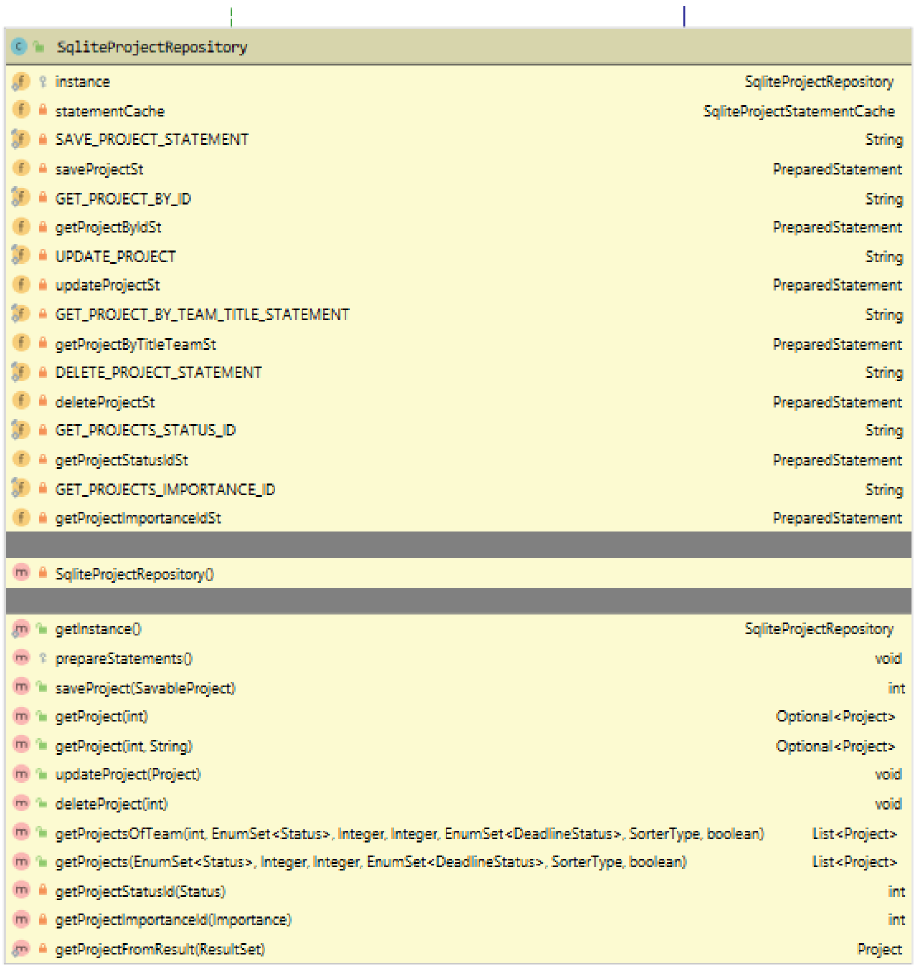
SqliteTeamRepository	
instance	SqliteTeamRepository
SAVE_TEAM_STATEMENT	String
saveTeamSt	PreparedStatement
DELETE_TEAM_STATEMENT	String
deleteTeamSt	PreparedStatement
GET_TEAM_WITH_CODE_QUERY	String
getTeamWithCodeSt	PreparedStatement
GET_TEAM_WITH_ID_QUERY	String
getTeamWithIdSt	PreparedStatement
GET_TEAMS_OF_USER_QUERY	String
getTeamsOfUserSt	PreparedStatement
SET_NEW_TEAMCODE_STATEMENT	String
setNewCodeSt	PreparedStatement
ADD_TEAM_MEMBERSHIP_STATEMENT	String
addTeamMembershipSt	PreparedStatement
REMOVE_TEAM_MEMBERSHIP_STATEMENT	String
removeTeamMembershipSt	PreparedStatement
REMOVE_ALL_TEAM_MEMBERS_STATEMENT	String
removeAllTeamMembersSt	PreparedStatement
SET_MANAGER_STATEMENT	String
setManagerSt	PreparedStatement
SET_NAME_STATEMENT	String
setNameSt	PreparedStatement
IS_MEMBER_QUERY	String
isMemberSt	PreparedStatement
GET_TEAM_MEMBERS_QUERY	String
getTeamMembersSt	PreparedStatement
SqliteTeamRepository()	
getInstance()	SqliteTeamRepository
prepareStatements()	void
saveTeam(SavableTeam)	int
getTeam(int)	Optional<Team>
getTeam(String)	Optional<Team>
getTeamsOfUser(int)	List<Team>
deleteTeam(int)	void
deleteAllMembersOfTeam(int)	void
addTeamMember(int, int)	void
removeTeamMember(int, int)	void
isMemberOfTeam(int, int)	boolean
setNewCode(int, String)	void
setNewManagerPosition(int, int)	void
setNewName(int, String)	void
getMembersOfTeam(int)	List<User>



SqliteUserRepository	
instance	SqliteUserRepository
saveUserStatement	PreparedStatement
getUserIdStatement	PreparedStatement
getUserByIdStatement	PreparedStatement
getUserByUsernameStatement	PreparedStatement
updateUserStatement	PreparedStatement
SAVE_USER_STATEMENT	String
GET_USER_ID_STATEMENT	String
GET_USER_BY_ID_STATEMENT	String
GET_USER_BY_USERNAME_STATEMENT	String
UPDATE_USER_STATEMENT	String
SqliteUserRepository()	
getInstance()	SqliteUserRepository
prepareStatements()	void
saveUser(User)	void
updateUser(User)	void
getUserId(String, String)	int
getUserById(int)	User
getUserByUsername(String)	User



SqliteCommentRepository	
instance	SqliteCommentRepository
SAVE_COMMENT_STATEMENT	String
saveCommentSt	PreparedStatement
GET_COMMENTS_OF_PROJECT_STATEMENT	String
getCommentsOfProjectSt	PreparedStatement
DELETE_COMMENTS_OF_PROJECT_STATEMENT	String
deleteCommentsOfProjectSt	PreparedStatement
SqliteCommentRepository()	
getInstance()	SqliteCommentRepository
prepareStatements()	void
saveComment(SavableComment)	void
getCommentsOfProject(int)	List<Comment>
deleteAllCommentsOfProject(int)	void
getCommentFromResult(ResultSet)	Comment



SqliteProjectRepository

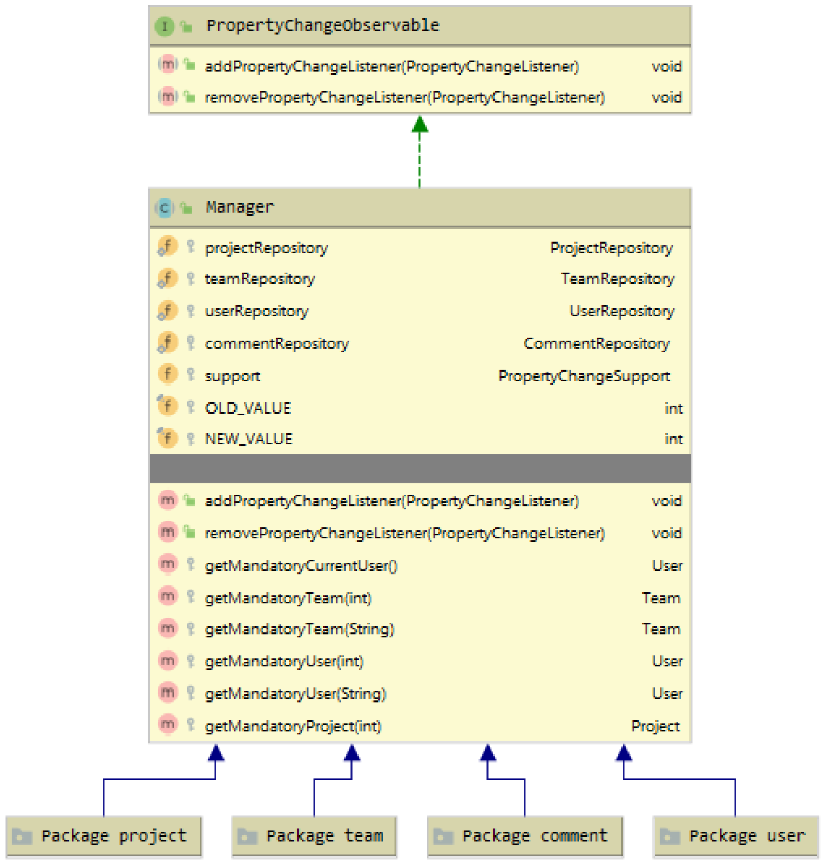
instance	SqliteProjectRepository
statementCache	SqliteProjectStatementCache
SAVE_PROJECT_STATEMENT	String
saveProjectSt	PreparedStatement
GET_PROJECT_BY_ID	String
getProjectByIdSt	PreparedStatement
UPDATE_PROJECT	String
updateProjectSt	PreparedStatement
GET_PROJECT_BY_TEAM_TITLE_STATEMENT	String
getProjectByTitleTeamSt	PreparedStatement
DELETE_PROJECT_STATEMENT	String
deleteProjectSt	PreparedStatement
GET_PROJECTS_STATUS_ID	String
getProjectStatusIdSt	PreparedStatement
GET_PROJECTS_IMPORTANCE_ID	String
getProjectImportanceIdSt	PreparedStatement
SqliteProjectRepository()	
getInstance()	SqliteProjectRepository
prepareStatements()	void
saveProject(SavableProject)	int
getProject(int)	Optional<Project>
getProject(int, String)	Optional<Project>
updateProject(Project)	void
deleteProject(int)	void
getProjectsOfTeam(int, EnumSet<Status>, Integer, Integer, EnumSet<DeadlineStatus>, SorterType, boolean)	List<Project>
getProjects(EnumSet<Status>, Integer, Integer, EnumSet<DeadlineStatus>, SorterType, boolean)	List<Project>
getProjectStatusId(Status)	int
getProjectImportanceId(Importance)	int
getProjectFromResult(ResultSet)	Project

Decoupling the interface of the repositories from the actual implementation makes the project database-independent: by simply changing the implementation of the repositories we could easily switch to another database type.

Each repository implementation has a set of SQL queries stored as strings with missing parameters. The PreparedStatements are prepared based on these strings only when needed, and they are closed immediately after the usage (using some try-with resources), in order to avoid storing too much unnecessary information on the SQL server, which anyway caches the statements which were already compiled.

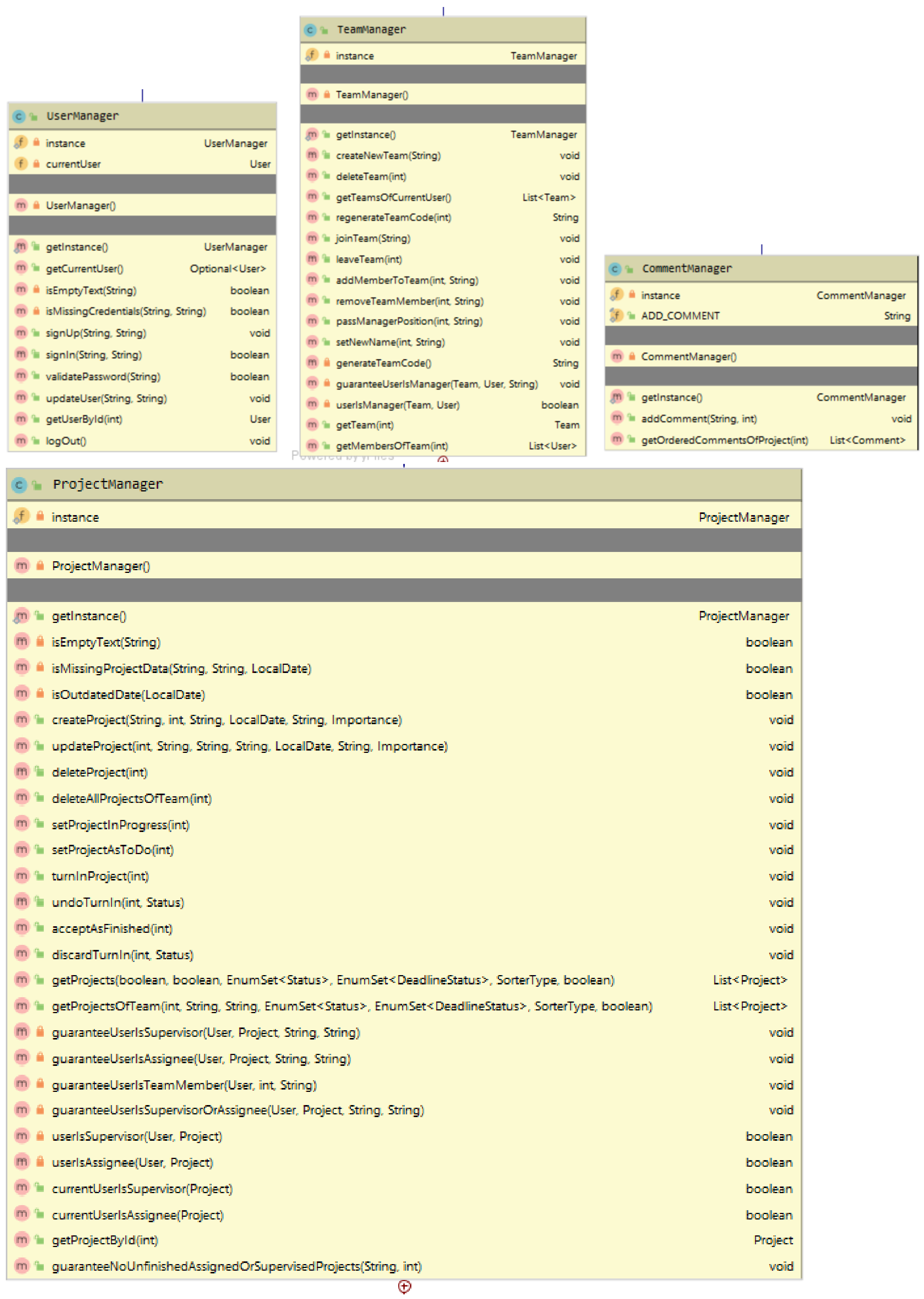
3. The Business Logic Layer

The actual business logic is implemented by the Managers. The Manager abstract class provides some utility functions for all managers (most of them are private), instantiates the repositories required for data access in the methods and implements the **PropertyChangeObservable** interface.


PropertyChangeObservable
addPropertyChangeListener(PropertyChangeListener) void
removePropertyChangeListener(PropertyChangeListener) void
Manager
projectRepository ProjectRepository
teamRepository TeamRepository
userRepository UserRepository
commentRepository CommentRepository
support PropertyChangeSupport
OLD_VALUE int
NEW_VALUE int
addPropertyChangeListener(PropertyChangeListener) void
removePropertyChangeListener(PropertyChangeListener) void
getMandatoryCurrentUser() User
getMandatoryTeam(int) Team
getMandatoryTeam(String) Team
getMandatoryUser(int) User
getMandatoryUser(String) User
getMandatoryProject(int) Project
Package project
Package team
Package comment
Package user

Then, each basic entity of the application has its own Manager class which provides the functionalities related to the given entity and extends the Manager class. The managers are singletons in order to guarantee that any data stored in them is shared across the application.

- **UserManager**: implements sign-up, login and logout and stores the data of the currently logged-in user.
- **TeamManager**: implements team creation and deletion, team joining and leaving, member addition and removal, and updates related to the manager position, the name and the code of the team.
- **ProjectManager**: implements project creation and deletion, updates for its attributes, status changes and finding the projects of a particular team with a given assignee and/or supervisor and status.
- **CommentManager:** implements comment addition functionality for projects and returns the project attached to a project.


TeamManager
instance TeamManager
TeamManager()
getInstance() TeamManager
createNewTeam(String) void
deleteTeam(int) void
getTeamsOfCurrentUser() List<Team>
regenerateTeamCode(int) String
joinTeam(String) void
leaveTeam(int) void
addMemberToTeam(int, String) void
removeTeamMember(int, String) void
passManagerPosition(int, String) void
setNewName(int, String) void
generateTeamCode() String
guaranteeUserIsManager(Team, User, String) void
userIsManager(Team, User) boolean
getTeam(int) Team
getMembersOfTeam(int) List<User>
UserManager
instance UserManager
currentUser User
UserManager()
getInstance() UserManager
getCurrentUser() Optional<User>
isEmptyText(String) boolean
isMissingCredentials(String, String) boolean
signUp(String, String) void
signIn(String, String) boolean
validatePassword(String) boolean
updateUser(String, String) void
getUserById(int) User
logOut() void
CommentManager
instance CommentManager
ADD_COMMENT String
CommentManager()
getInstance() CommentManager
addComment(String, int) void
getOrderedCommentsOfProject(int) List<Comment>
ProjectManager
instance ProjectManager
ProjectManager()
getInstance() ProjectManager
isEmptyText(String) boolean
isMissingProjectData(String, String, LocalDate) boolean
isOutdatedDate(LocalDate) boolean
createProject(String, int, String, LocalDate, String, Importance) void
updateProject(int, String, String, String, LocalDate, String, Importance) void
deleteProject(int) void
deleteAllProjectsOfTeam(int) void
setProjectInProgress(int) void
setProjectAsToDo(int) void
turnInProject(int) void
undoTurnIn(int, Status) void
acceptAsFinished(int) void
discardTurnIn(int, Status) void
getProjects(boolean, boolean, EnumSet<Status>, EnumSet<DeadlineStatus>, SorterType, boolean) List<Project>
getProjectsOfTeam(int, String, String, EnumSet<Status>, EnumSet<DeadlineStatus>, SorterType, boolean) List<Project>
guaranteeUserIsSupervisor(User, Project, String, String) void
guaranteeUserIsAssignee(User, Project, String, String) void
guaranteeUserIsTeamMember(User, int, String) void
guaranteeUserIsSupervisorOrAssignee(User, Project, String, String) void
userIsSupervisor(User, Project) boolean
userIsAssignee(User, Project) boolean
currentUserIsSupervisor(Project) boolean
currentUserIsAssignee(Project) boolean
getProjectById(int) Project
guaranteeNoUnfinishedAssignedOrSupervisedProjects(String, int) void

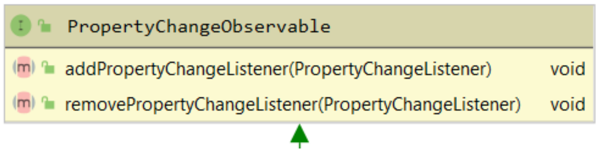
Exceptions

With the goal of keeping the application as safe and user-friendly as possible, we defined several custom exceptions to be thrown whenever the Managers get illegal data, commands, etc. Some of these may occur due to illegal user input, some other may occur only due to software bugs, but in both cases, we wanted to catch the bugs at an early stage, prevent corrupt data from being saved in the database or the application entering an invalid state, and to inform the user about the error that occurred.

The exceptions which are defined and handled separately in the application are the following:

- For User data:
  - **DuplicateUsernameException** is thrown if someone attempts to register with an already taken username
  - **EmptyFieldsException** is thrown if the sign-up data of a user contains empty fields
  - **InexistentUserException** is thrown if a command is related to a user which is not found in the database
  - **NoSignedInUSerException** is thrown if a functionality is accessed which is accessible to signed-in users only, but there is currently no signed-in user
- For Team data:
  - **AlreadyMemberException** is thrown if someone attempts to join a team but they are already a member of that team
  - **IllegalMemberRemovalException** is thrown if a member cannot be removed from a team because they still have unfinished projects
  - **InexistentTeamException** is thrown if a command is related to a team which is not found in the database
  - **ManagerRemovalException** is thrown if the manager of a team attempts to leave the team
  - **UnregisteredMemberRemovalException** is thrown if the Manager attempts to remove a user from the members who was not a member in fact
  - **UnregisteredMemberRoleException** if the manager position is passed to a user who is not the member of the team/a role in a project is assigned to someone who is not the member of the team
- For Project data:
  - **DuplicateProjectNameException** is thrown if someone attempts to create/rename a project, but a project with the same name already exists in the team
  - **IllegalProjectStatusChangeException** is thrown if someone attempts to change the status of the project but doesn't have permission to do that/that status change is illegal
  - **InexistentProjectException** is thrown if a command is related to a project which is not found in the database
  - **InvalidDeadlineException** is thrown if someone attempts to set a deadline for a project which is before the current date
- Others:
  - **UnauthorisedOperationException** is thrown if someone attempts to perform an action but doesn't have permission to do that (for example because they are not the manager/supervisor/assignee/...)

The Observer pattern


PropertyChangeObservable
addPropertyChangeListener(PropertyChangeListener) void
removePropertyChangeListener(PropertyChangeListener) void

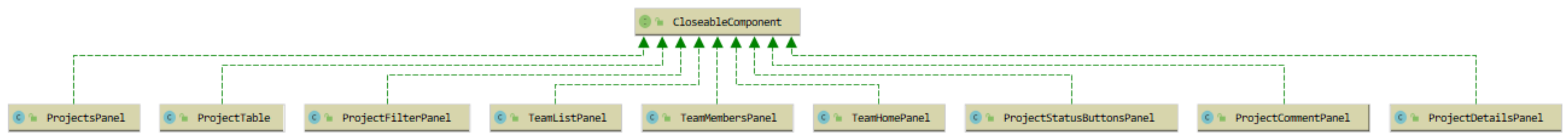
Given the MVC pattern, the Model can notify the View about the changes in the data which should be reflected in the UI. To do so, all the Managers described above are implementing the **PropertyChangeObservable** interface, and they have an instance of **PropertyChangeSupport** which can keep track of the registered listeners.

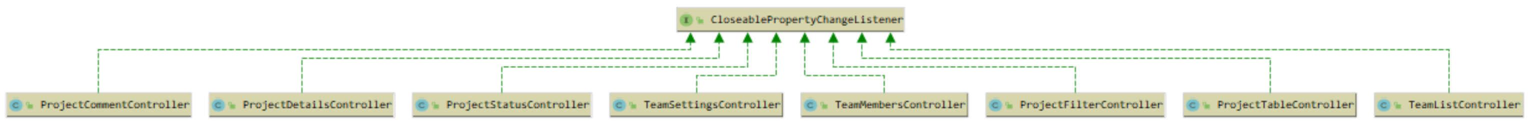
Thus, if a frame must be updated once certain data is changed, the controller of the frame, which implements **ProperyChangeListener**, is registered as a listener to the given Manager. When the data changes, the Manager fires a **PropertyChangeEvent** (with a given title), and the previously registered listeners are automatically notified about the event, and thus they can immediately update the UI.

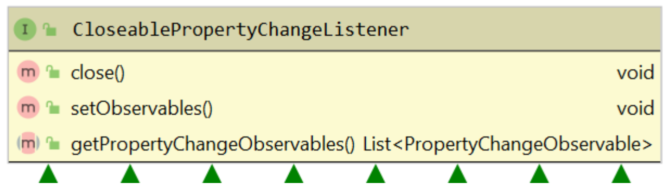
The observer pattern made our code more robust and simplified the implementation.

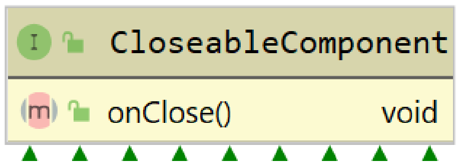
On the front-end side, we had to solve the issue of memory leaks caused by objects being registered as listeners for a certain manager would not be collected by the garbage collector even if the frame/UI component to which they belong would be closed/deleted. To deal with this issue, two additional interfaces were defined:

- **CloseableComponent** (interface in the View)**:**
    - implemented by all UI components other than frames which are closed (together with their parent frame), and thus their controllers should be collected by the garbage collector.
    - The interface provides the onClose() method, which calls the close() method of the controller (which implements CloseablePropertyChangeListener) object corresponding to the CloseableComponent.
    - CloseableComponents are built up as in the composite pattern: one CloseableComponent may be composed of multiple CloseableComponents. In this case, the onClose() method is called recursively for the subcomponents.
- **CloseablePropertyChangeListener** (interface in the Controller):
    - is an interface which extends PropertyChangeListener and it guarantees that a PropertyChangeListener (controller) previously registered at a PropertyChangeObservable (Manager) is collected by the garbage collector when the corresponding UI component is closed.
    - The UI component can call the close() method of CloseablePropertyChangeListener, which then unregisters the listener from all the Observable objects to which it was previously registered, and thus the unnecessary references to the PropertyChangeListener object, which prevented the garbage collector from collecting it, are removed, and the garbage collector can successfully collect the object.






CloseablePropertyChangeListener
close() void
setObservables() void
getPropertyChangeObservables() List<PropertyChangeObservable>



b. View and Controller

The **views** are responsible for displaying the relevant information of the model, in a user-friendly, readable way. In addition, the user can change the data of the model through the view, inserting, updating or deleting information about their account, teams or projects. Through the user-friendly interface, the user can interact indirectly with the underlying model.

Each element of the view extends either the **JFrame** or **JPanel** classes or other UI components.

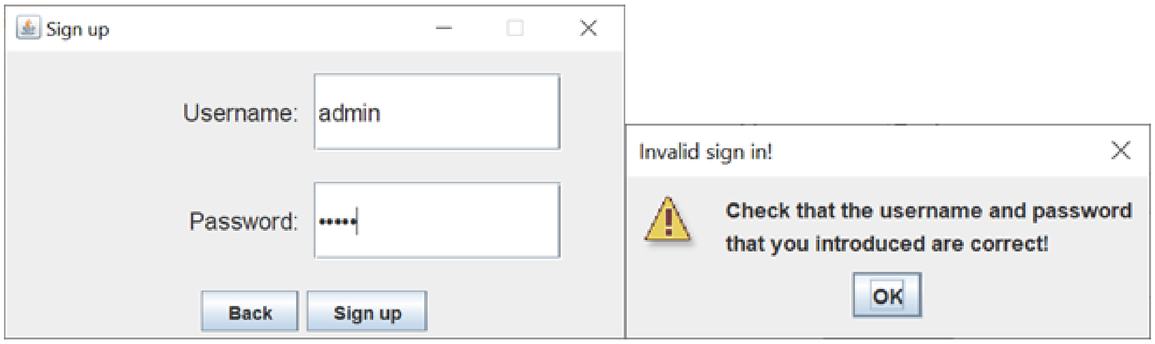
The **controllers** which manage these views, are mainly responsible for handling the requests of the user and invoking the corresponding methods of the model to perform certain operations. Moreover, they update the views whenever a change occurs in the model, thus ensuring that always the correct information is displayed to the user.

The controllers which are responsible for operations related to opening or closing frames, extend the **FrameController**, which handles the operation of closing a given frame.

1. Account-related actions

SignInFrame

- Allows the user to sign into the application with the corresponding username and password.
- If the user does not have an account yet, by clicking on the *"Create Account"* button, he/she will be redirected to the SignUpFrame.
- If the login credentials introduced by the user are incorrect, an error dialog pops up to inform the user about the unsuccessful sign in.

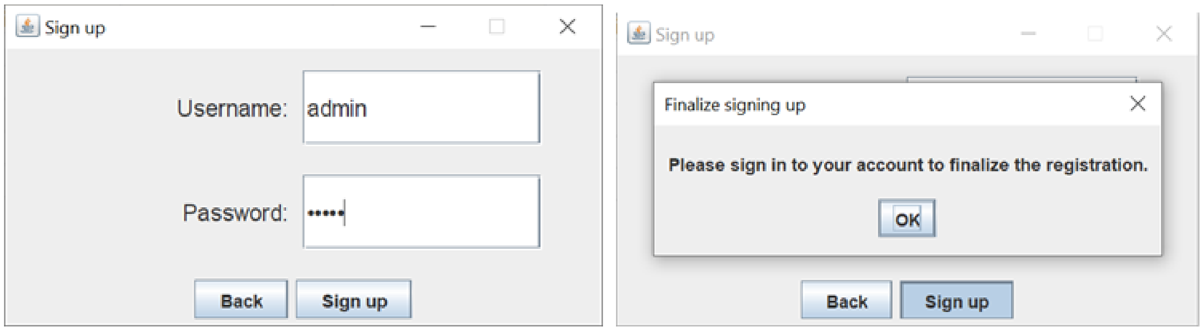


Check that the username and password that you introduced are correct!

It is managed by the SignInController, which sends a login validation request to the model when the user attempts to sign in, and opens the SignUpFrame, when the user chooses to create a new account.

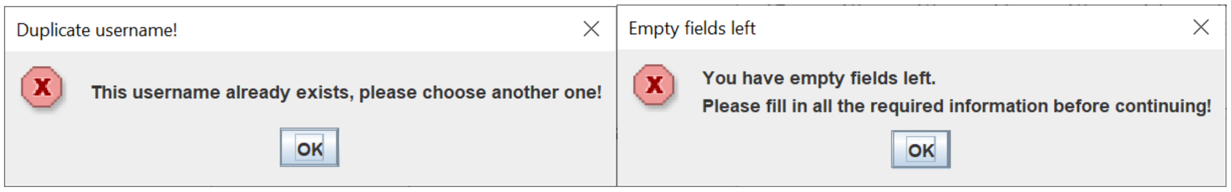
SignUpFrame

Allows the user to create a new account by setting their own password and a unique username, and save these by clicking on the "Sign up" button. After that, the user is redirected to the SignInFrame and asked to log in with the created credentials to finalize the signing up process.



Please sign in to your account to finalize the registration.

- The user can choose to return to the login page, upon clicking on the *"Back"* button or closing the frame.
- If the user attempts to create an account with an already existing username or tries to sign up, without completing all the required fields, a corresponding error message is shown.



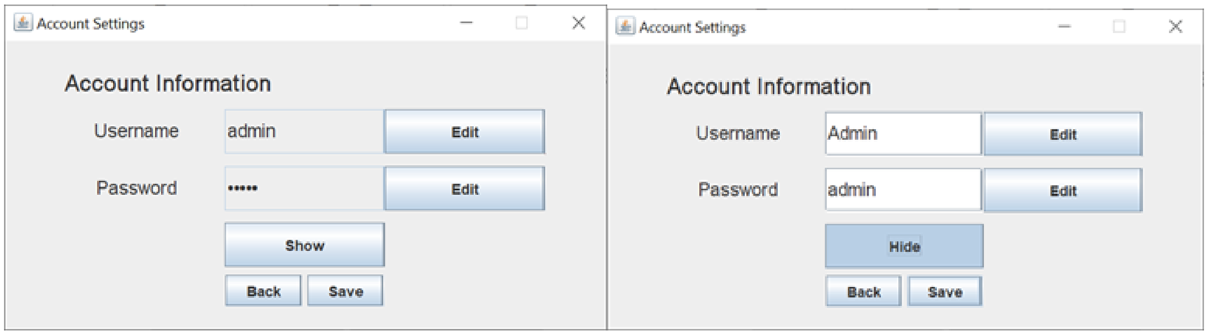
This username already exists, please choose another one!

You have empty fields left.
Please fill in all the required information before continuing!

- It is managed by the **SignUpController,** which is responsible for accessing the model and creating a new user account if all the introduced data is correct.

AccountSettingsFrame

- Allows the user to view the details of their account and to modify the account data, such as the username or password.
- The fields become editable when the user clicks on the *"Edit"* button.
- The user can choose to save the changes made to the account or to go back to the MainFrame.


Account Settings

Account Information

Username admin

Password •••••

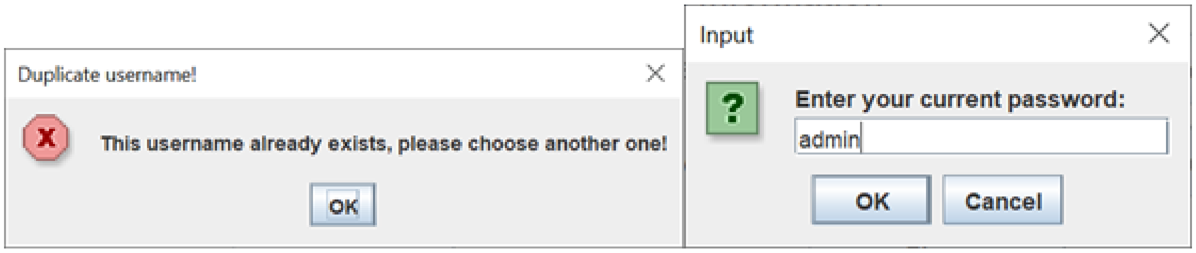
Account Settings

Account Information

Username Admin

Password admin

- The toggle button (*"Show"/"Hide"*), under the password's field, allows the user to show/hide the password, but only after the editing is enabled for that field.
- The username can be set directly, by typing in the new value. If the introduced username is already used in another account, an error message shows up to inform the user that the corresponding username is already taken.
- However, in order to change the password, the user is asked to introduce the old password as a minimal security measure.


Duplicate username!

This username already exists, please choose another one!

Input

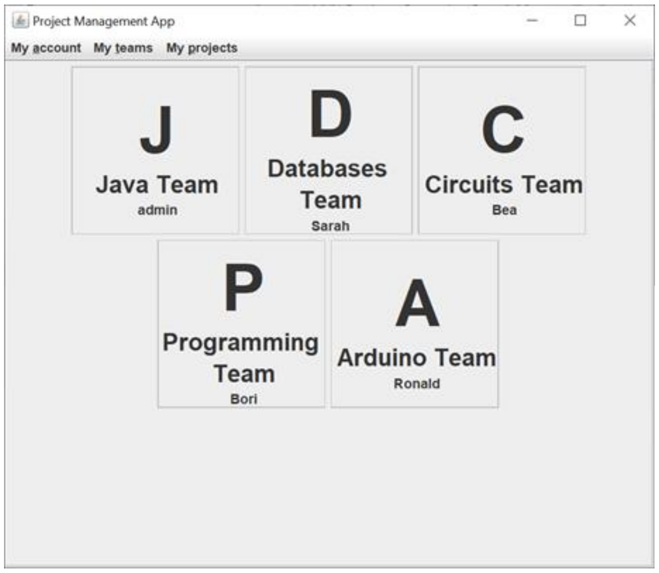
Enter your current password: admin

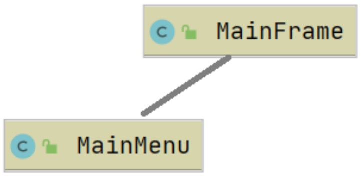
- The **AccountSettingsController** manages the AccountSettingsFrame, setting the model when the user modifies the account data, and updating the data displayed in the fields, when the model changes.

2. Navigation through the Main page

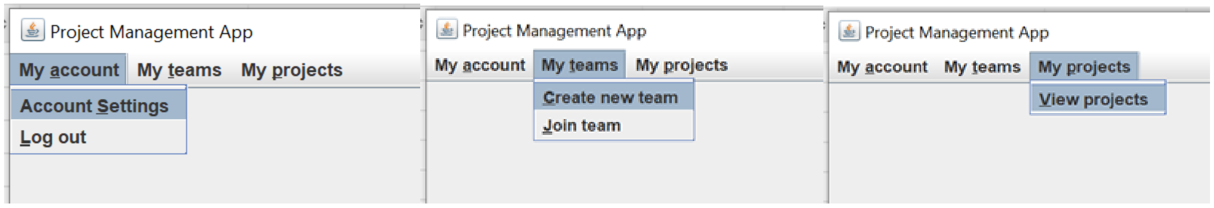
MainFrame

- Represents the main page to which the user is directed right after logging in, and it displays the list of teams in which the currently logged in user is a member.





- Contains a menu bar, with dropdown items which allow the user to select from the available functionalities.
- *"My account"* menu:
  - "Account settings": allows the user to view and edit his/her account details, by opening the AccountSettingsFrame.
  - "Log out": allows the user to log out from the application, in which case he/she will be redirected to the SignInFrame.
- *"My teams"* menu:
  - "Create new team": allows the user to create a new team by opening the CreateTeamFrame.
  - "Join team": allows the user to join an existing team by opening the JoinTeamFrame.
- *"My projects"*:
- *"View projects"*: allows the user to view the list of projects related to their account opening the UserProjectsFrame.



- It is managed by the **MainMenuController** which is responsible for opening the corresponding frame when a user selects an item from the main menu.

TeamListPanel

- Displays the list of labels of the user's teams in the MainFrame.
- It is managed by the **TeamListController**, which updates the TeamListPanel and the TeamViewModel, which contains information about the list of teams that are being displayed whenever the teams' data changes, for example a team is added or removed, its name or manager is changed.

TeamLabel

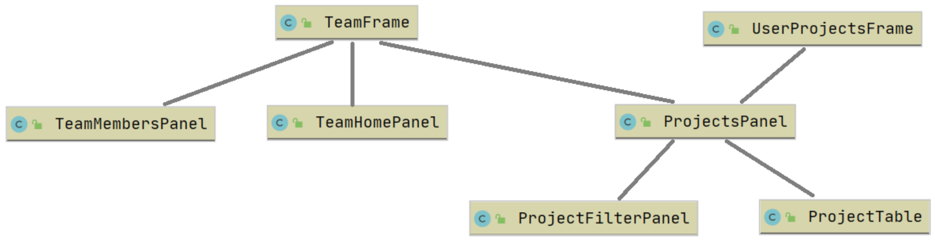
Represents the label through which the teams are displayed in the TeamListPanel.

- It generates an animation, when the user hovers over or clicks on the label, creating a more user-friendly experience.
- It is managed by the **TeamLabelController**, which opens a team, when a user double clicks on a label.

Team-related actions

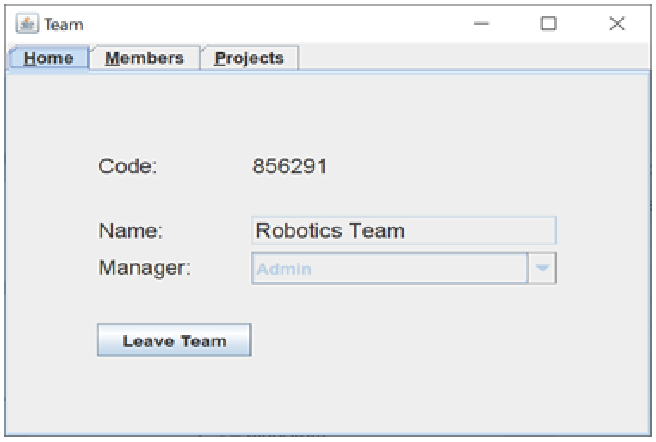
TeamFrame

- Allows the user to view the team selected from the MainFrame.
- It contains 3 tabs:



1. HomePanel

- Allows the user to view the details of the team or to leave the team.

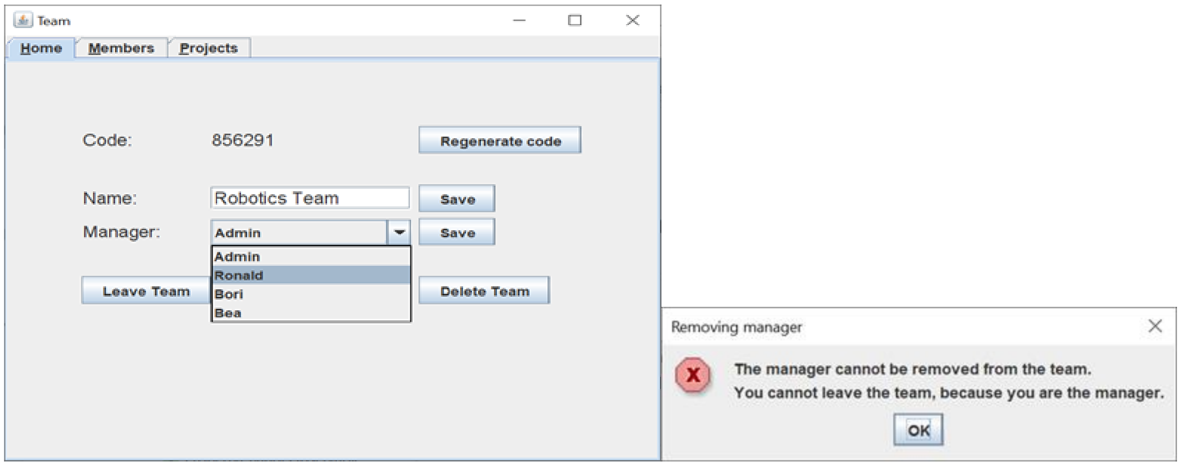


Code: 856291

Name: Robotics Team

Manager: Admin

- The manager can also edit the data related to the team, for example generate a new code for the team, change the name of the team or pass the manager position to another member.
- The manager is not allowed to leave the team unless he/she passes its position to someone else.

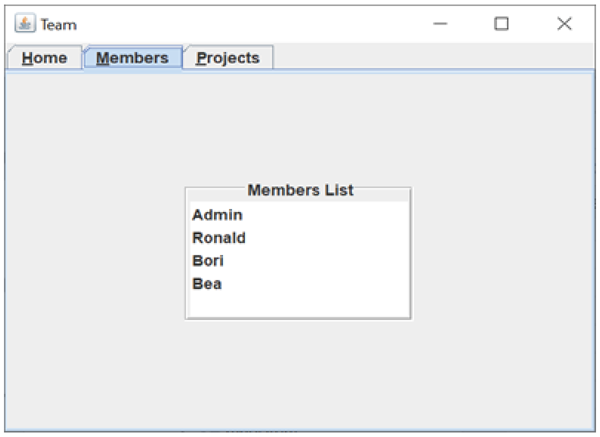


The manager cannot be removed from the team.
You cannot leave the team, because you are the manager.

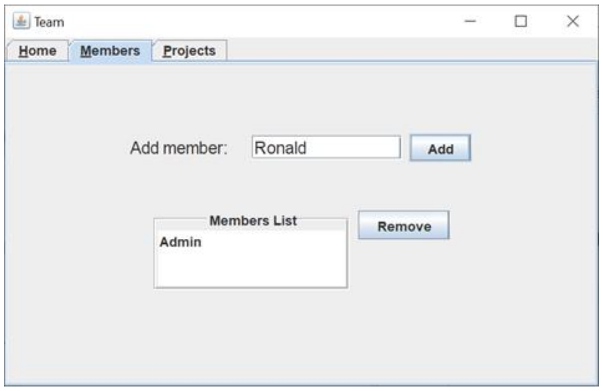
- It is managed by the **TeamSettingsController** which updates the data labels and the information displayed on the panel, if the data of the team changes in the model.

2. MembersPanel

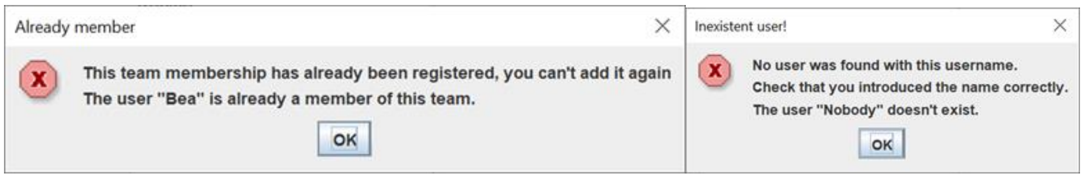
- Allows the user to view the list of members of the team.



- The supervisor can also add and remove members from the team:
- Add members by entering the name of the member in the corresponding textfield.



- The operation fails and the user is warned if the member requested is already a member of the team or if the member to be added doesn't exist in the database:

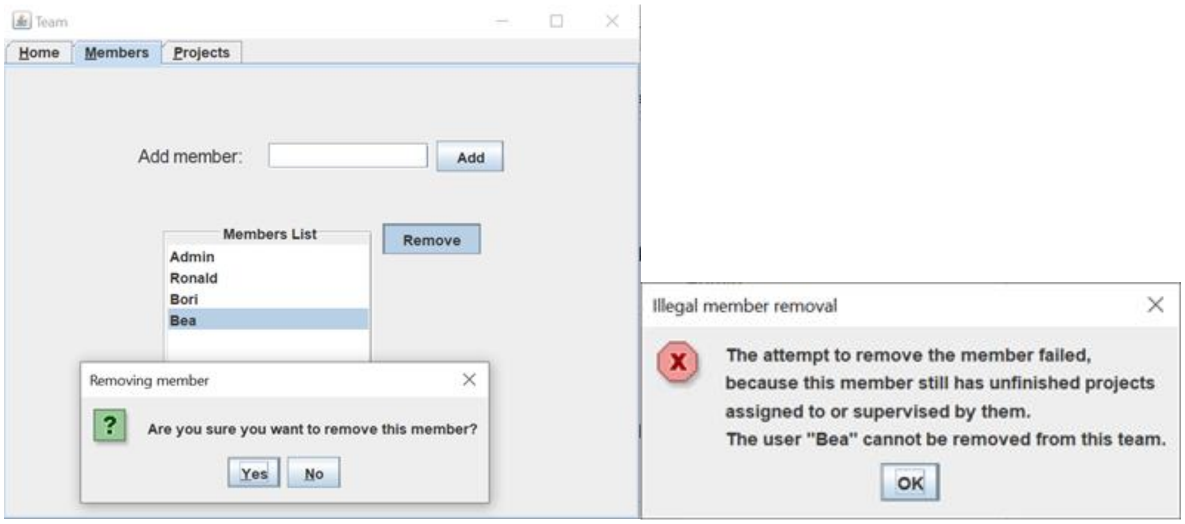

Already member

This team membership has already been registered, you can't add it again
The user "Bea" is already a member of this team.

Inexistent user!

No user was found with this username.
Check that you introduced the name correctly.
The user "Nobody" doesn't exist.

- Removing a member from the team can be done, by selecting it from the list and choosing the "Remove" option. Before removing the selected member, the user is asked to confirm the operation.
- When the manager attempts to remove itself or a member which still has unfinished projects an error message is shown, and the operation fails.



Removing member

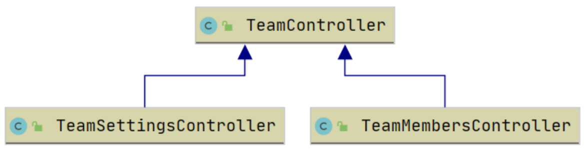
Are you sure you want to remove this member?

Illegal member removal

The attempt to remove the member failed, because this member still has unfinished projects assigned to or supervised by them.
The user "Bea" cannot be removed from this team.

- It is controlled by the **TeamMembersController**, which updates the list each time a new member is added or removed from the team.

The **TeamSettingsController** and **TeamMembersController** extend the **TeamController**, which contains the id of the team that is viewed and grants access to the currently logged in user to edit the details and members of the team if he/she is the manager.



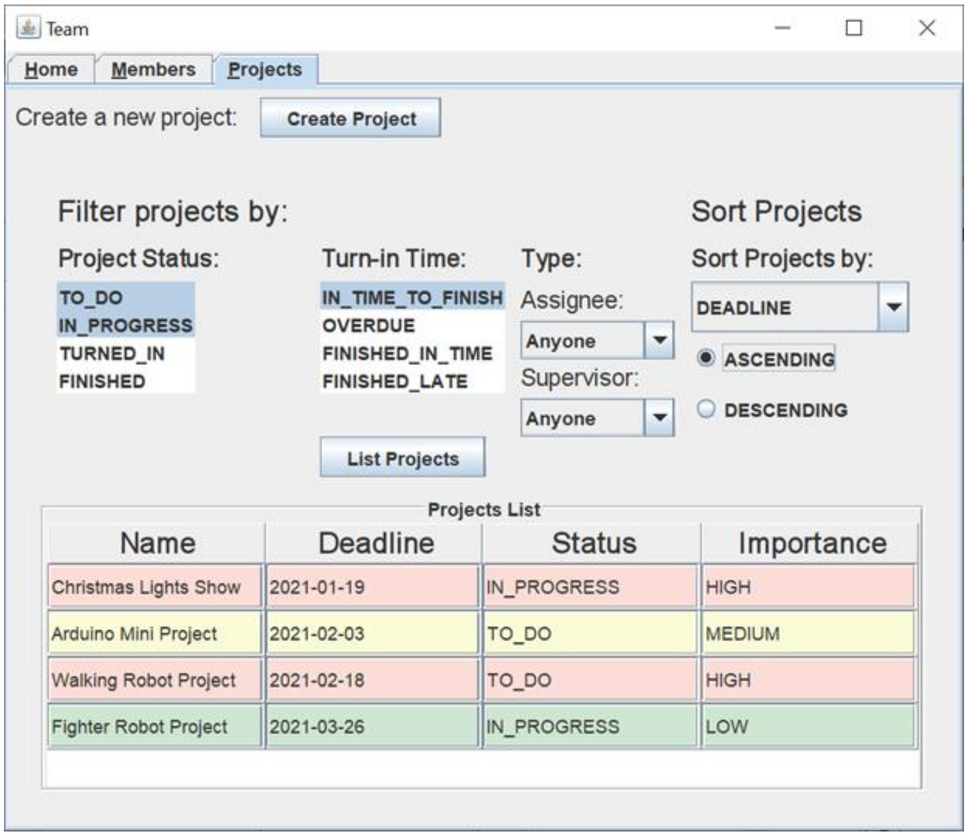
3. ProjectsPanel

- Displays the list of projects that belong to the given team.
- Allows the user to filter the projects by
  - **Status** (multiple selections is allowed)
  - **Turn-in time** (multiple-selections is allowed)

  - **Assignee** or **Supervisor** of the projects (single selection is allowed only), by default *"Anyone"* is selected.
- The user can also sort the list of projects by
  - **Deadline**
  - **Status**
  - **Importance**

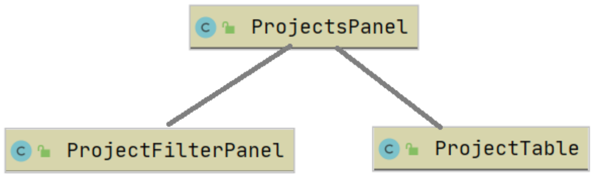
  in *ascending* or *descending* order.

- The selected filters and sorting options are applied only when the user clicks on the *"List projects"* button.
- Each project is colored based on its level of importance:
  - HIGH importance is marked with a red color.
  - MEDIUM importance is marked with green color.
  - LOW importance project has a yellow color.
- The user can view the details of a project by selecting the corresponding row from the table and double clicking on it.



Name	Deadline	Status	Importance
Christmas Lights Show	2021-01-19	IN_PROGRESS	HIGH
Arduino Mini Project	2021-02-03	TO_DO	MEDIUM
Walking Robot Project	2021-02-18	TO_DO	HIGH
Fighter Robot Project	2021-03-26	IN_PROGRESS	LOW

- The user can also create a new Project when selecting the "Create Project" Button, which opens the CreateProjectFrame.
- It contains 3 main components:



a. ProjectTable

- Extends **JTable** and is responsible for displaying the list of projects in form of a table, with four columns, listing the **title, deadline, status** and **importance** of each project.
- It is managed by the ProjectTableController, which listens to any changes of the ProjectListModel and updates the table, displaying the modified list.

b. ProjectListModel

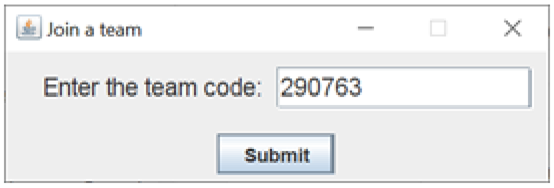
- It contains the list of projects that are displayed in the ProjectTable.
- It implements the PropertyChangeObservable Interface, such that it triggers a property change event whenever the project list is changed.

c. ProjectFilterPanel

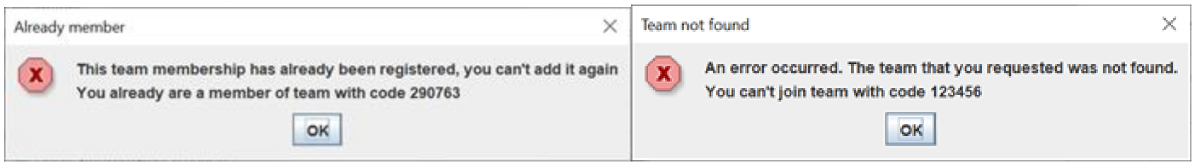
- Contains the filter components that are applied on the list of projects.
- It is managed by the ProjectFilterController, which listens to the changes in the ProjectListModel and updates the table containing the projects, in addition it sets the model whenever the user sends a request, pressing the "List Projects" button, or when a project is modified or deleted from the ProjectFrame.

JoinTeamFrame

- Allows the user to join an existing team by entering a unique, six digits code that identifies the corresponding team.



- If the user tries to join a team, that he/she is already a member of, or if the user introduced an incorrect or non-existing code a corresponding error message is shown.


Already member

This team membership has already been registered, you can't add it again
You already are a member of team with code 290763

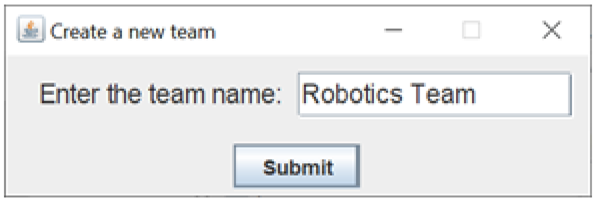
Team not found

An error occurred. The team that you requested was not found.
You can't join team with code 123456

- The **JoinTeamController** handles the user's request to join a team and displays the error dialogs above in case of a failed operation, and it closes the JoinTeamFrame upon successful join and redirects the user to the MainFrame.

CreateTeamFrame

- Allows the user to create a new team by entering the name of the team.



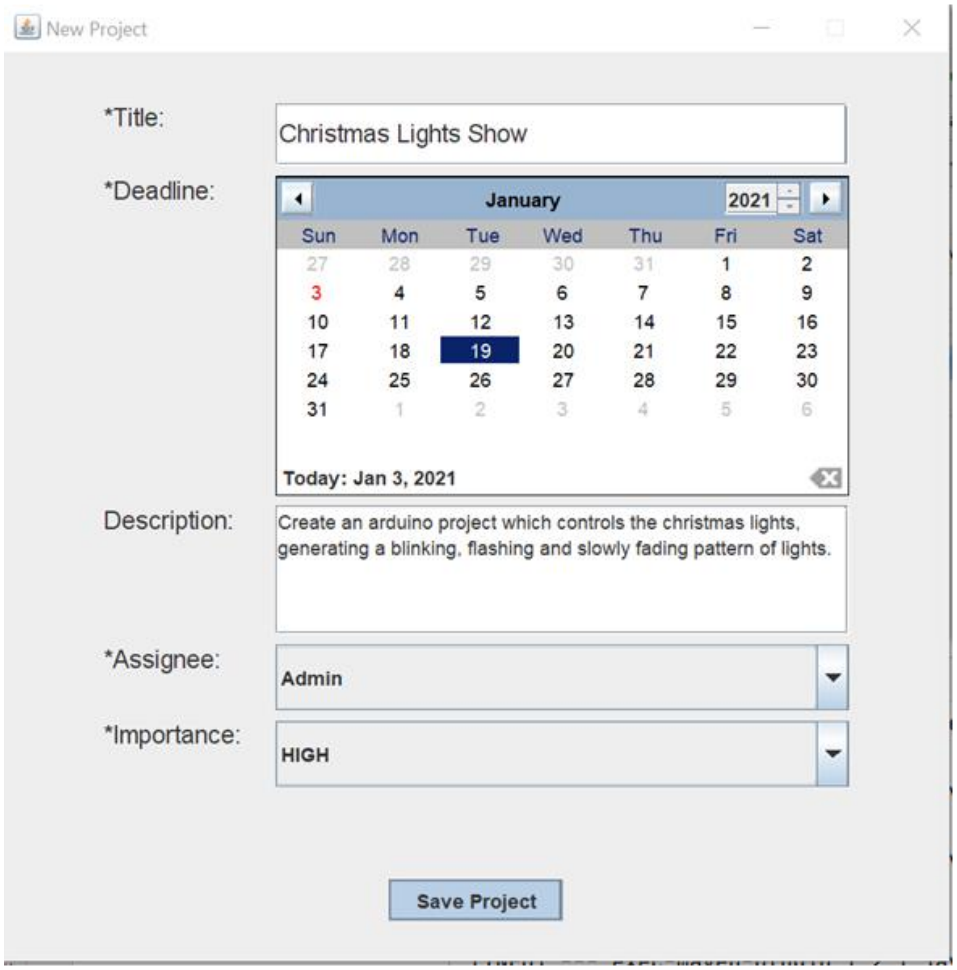
Enter the team name: Robotics Team

- It is managed by the **CreateTeamController**, which invokes the corresponding method from the model to create a new Team and closes the CreateTeamFrame upon successful team creation and redirects the user to the MainFrame.

Project-related actions

CreateProjectFrame

- Allows the user to create a new project by filling in the required fields.
- It is controlled by the **CreateProjectController,** which saves the new project, updating the model, if the introduced data is valid.



*Title: Christmas Lights Show

*Deadline: January 2021

Sun	Mon	Tue	Wed	Thu	Fri	Sat
27	28	29	30	31	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

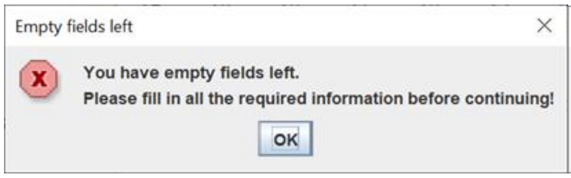
Today: Jan 3, 2021

Description: Create an arduino project which controls the christmas lights, generating a blinking, flashing and slowly fading pattern of lights.

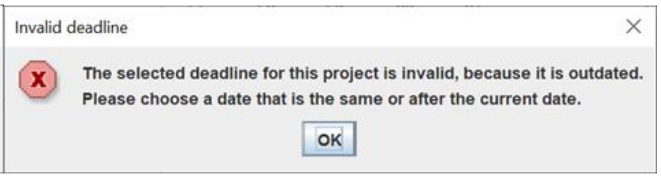
*Assignee: Admin

*Importance: HIGH

- If some of the mandatory fields marked with a "*" are left empty, or if the deadline chosen by the user is "outdated", meaning that it is before the current date, an error message is shown and the project is not saved.


Empty fields left

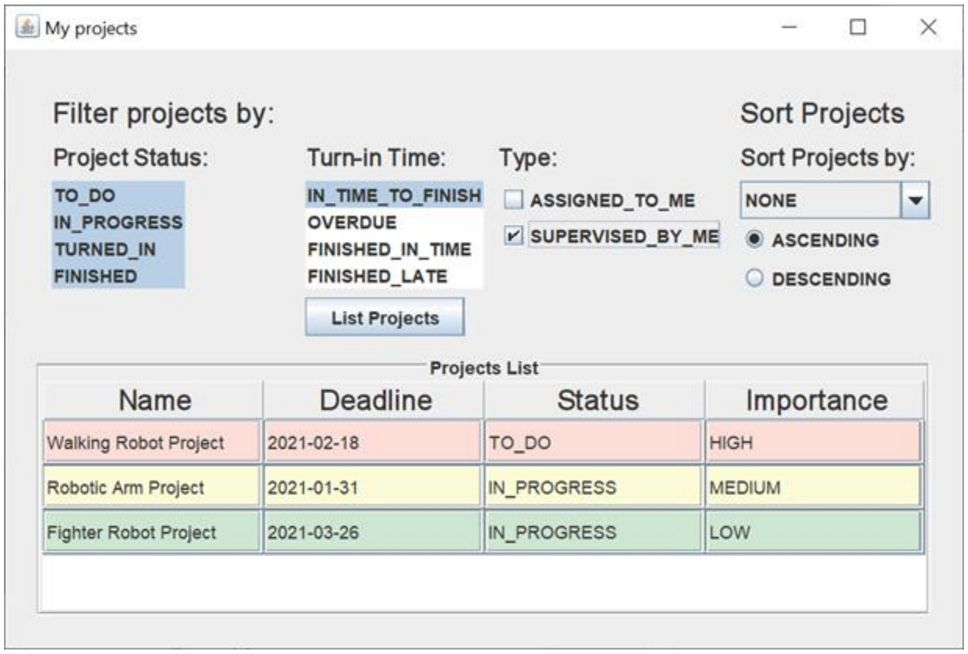
You have empty fields left.
Please fill in all the required information before continuing!


Invalid deadline

The selected deadline for this project is invalid, because it is outdated.
Please choose a date that is the same or after the current date.

UserProjectsFrame

- Displays the list of projects that are assigned to or supervised by the currently logged in user.


My projects

Filter projects by:

Project Status: TO_DO, IN_PROGRESS, TURNED_IN, FINISHED

Turn-in Time: IN_TIME_TO_FINISH, OVERDUE, FINISHED_IN_TIME, FINISHED_LATE

Type: ☐ ASSIGNED_TO_ME [✓ SUPERVISED_BY_ME]

Sort Projects

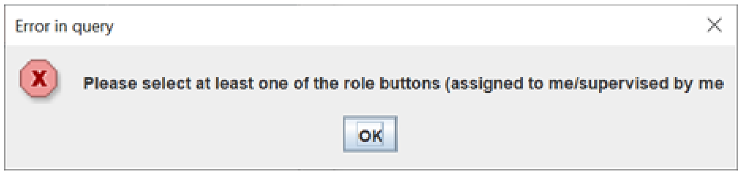
Sort Projects by: NONE

[● ASCENDING] ○ DESCENDING

Projects List

Name	Deadline	Status	Importance
Walking Robot Project	2021-02-18	TO_DO	HIGH
Robotic Arm Project	2021-01-31	IN_PROGRESS	MEDIUM
Fighter Robot Project	2021-03-26	IN_PROGRESS	LOW

- It is like the UserProjectsList, but in this case the selection of the assignee and supervisor is replaced by two checkboxes, which allow the user to filter the projects based on whether these were *assigned to* or *supervised by* the user. At least one option must be checked, either the "*Assigned to me*" or the "*Supervised by me*", otherwise an error message is shown.


Error in query

Please select at least one of the role buttons (assigned to me/supervised by me

ProjectFrame

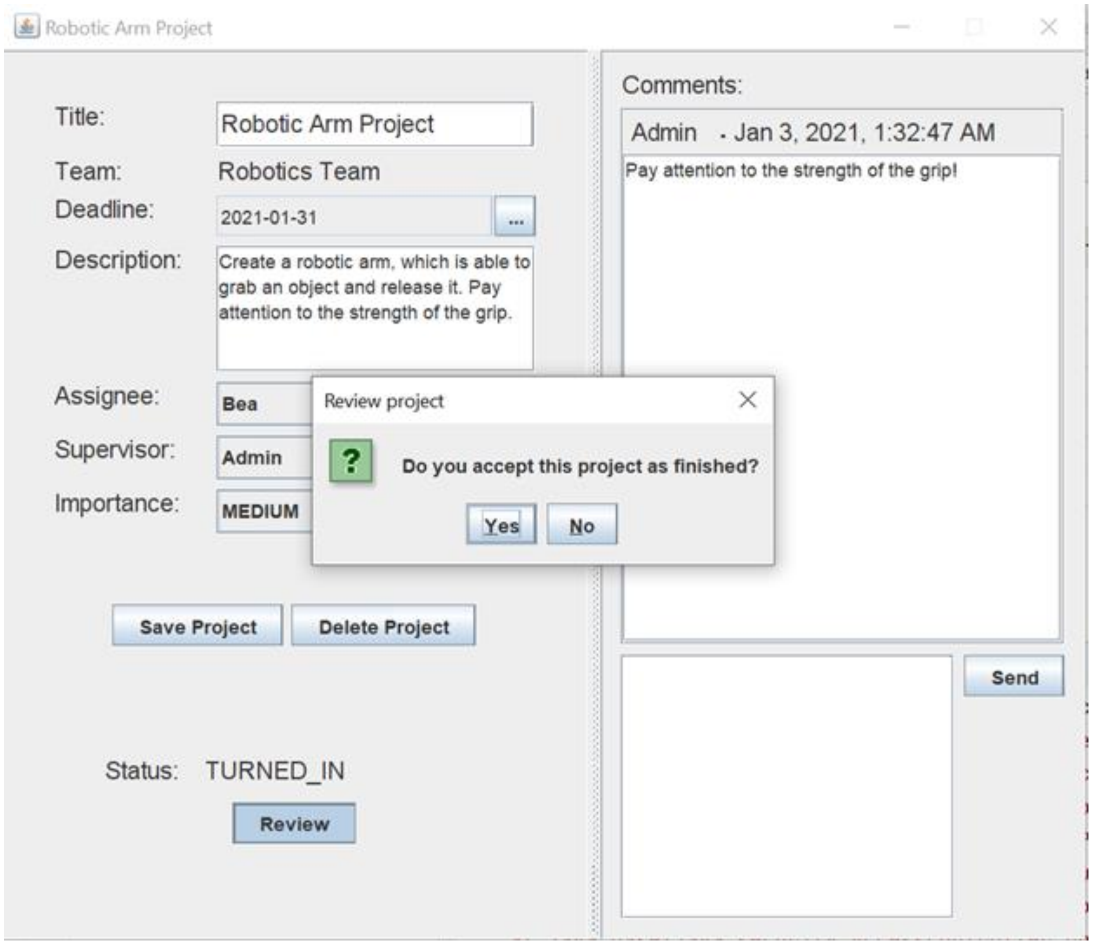
- Contains 3 panels for displaying the details of a single project and allow the user to mark the status of the project or leave a comment on the comment board.

1. ProjectDetailsPanel

- It lists the details of a single project and allows the supervisor of the project to save or delete the project.
- Only the supervisor has access to modify the data of the project, such as the title, deadline, assignee, etc. The “finished” project’s data cannot be modified anymore.
- It is controlled by the **ProjectDetailsController.**

2. StatusButtonsPanel

- It allows the user to change the status of the project, by displaying a set of available buttons.
- It is controlled by the **StatusButtonsController**, which sets the new status of the project and updates the list each time the status of the project in the model changes.
- Remark that everyone can mark the progress on the project, but only the assignee can turn it in. Similarly, only the supervisor can declare the turned in project as “Finished” after reviewing it, or even set the project’s status back to “To Do” or “In Progress”, if he/she finds it incomplete.

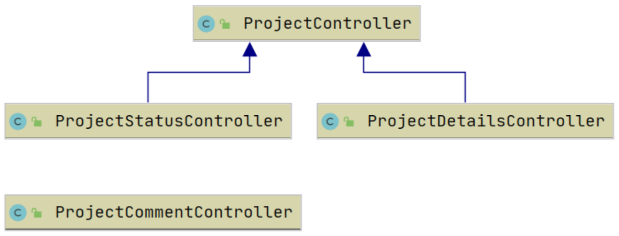


Title: Robotic Arm Project
Team: Robotics Team
Deadline: 2021-01-31
Description: Create a robotic arm, which is able to grab an object and release it. Pay attention to the strength of the grip.
Assignee: Bea
Supervisor: Admin
Importance: MEDIUM
Status: TURNED_IN

Admin - Jan 3, 2021, 1:32:47 AM
Pay attention to the strength of the grip!

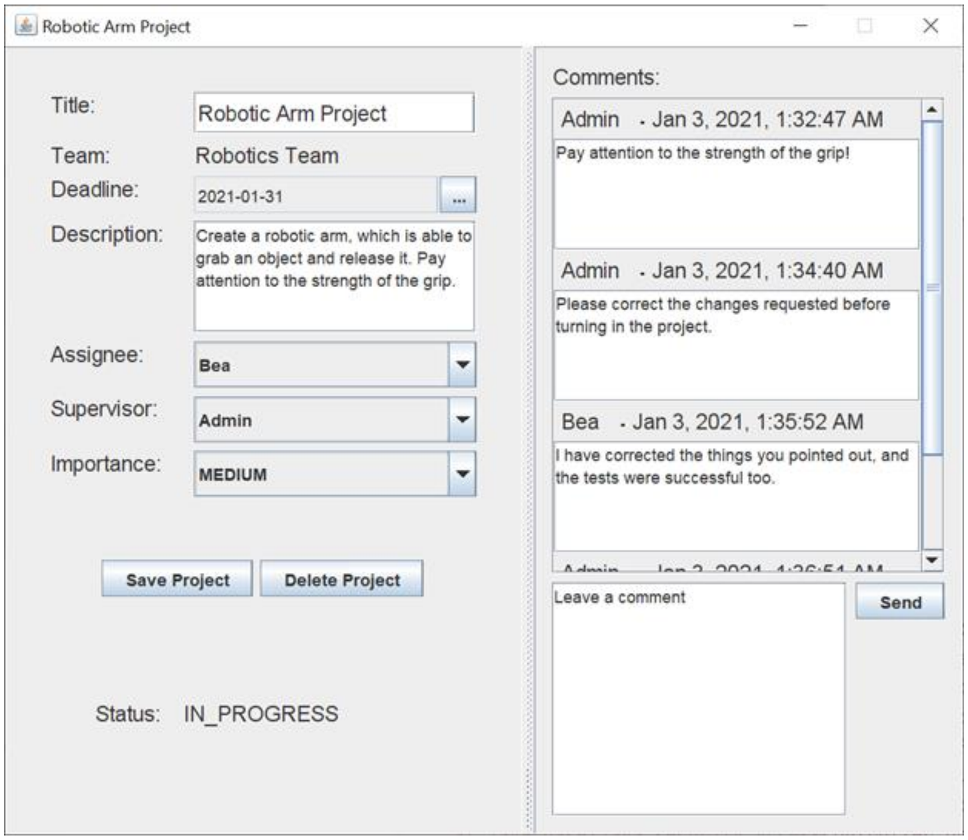
Do you accept this project as finished?

The **ProjectStatusController** and the **ProjectDetailsController** extend the **ProjectController,** which contains the data of the project that is currently viewed and is responsible for updating the project fields.



3. ProjectCommentPanel

- It displays the comments from the conversation related to the given project, and it allows the user to leave a comment.
- It also displays the date when the comment was posted and the name of the user who sent it.
- The user can send the comment by either clicking on the *"Send"* button or with the "*Alt+Enter*" key combination.
- The user cannot send an empty comment.
- It is controlled by the **ProjectCommentController**, which is responsible for updating the list of comments each time a new comment is added.



Title: Robotic Arm Project
Team: Robotics Team
Deadline: 2021-01-31
Description: Create a robotic arm, which is able to grab an object and release it. Pay attention to the strength of the grip.
Assignee: Bea
Supervisor: Admin
Importance: MEDIUM

Status: IN_PROGRESS

Comments:

Admin - Jan 3, 2021, 1:32:47 AM
Pay attention to the strength of the grip!

Admin - Jan 3, 2021, 1:34:40 AM
Please correct the changes requested before turning in the project.

Bea - Jan 3, 2021, 1:35:52 AM
I have corrected the things you pointed out, and the tests were successful too.

Leave a comment

UIFactory

It is responsible for the creating a uniform design across the interface, containing methods for creating predefined UI components, such as labels, buttons and text fields, and it also contains some constants, such as the dimension of a frame, paddings to align the component in the middle of the panel, and different font sizes. It guarantees mobility of changing the design of the project easily, by updating these components, instead of updating every element of the individual frames one by one.

ErrorDialogFactory

It is responsible for creating specific error message dialogs, depending on the type of exception that is given, and printing an appropriate message for each. It helps code reusability, by generifying repeated fragments of code.