Final Report

CSCI 5448

12/15/15

Project Time-Management System

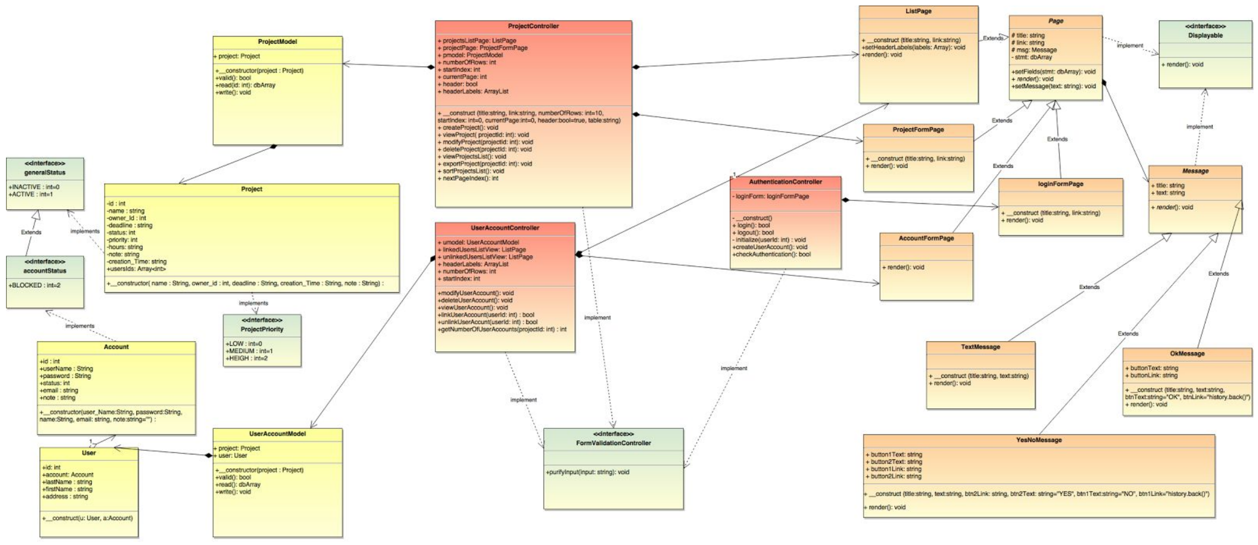
Team: Mohammad Alasmary, Kelly Goodman, Kevin Holligan, Elizabeth Lor

Implemented Features:

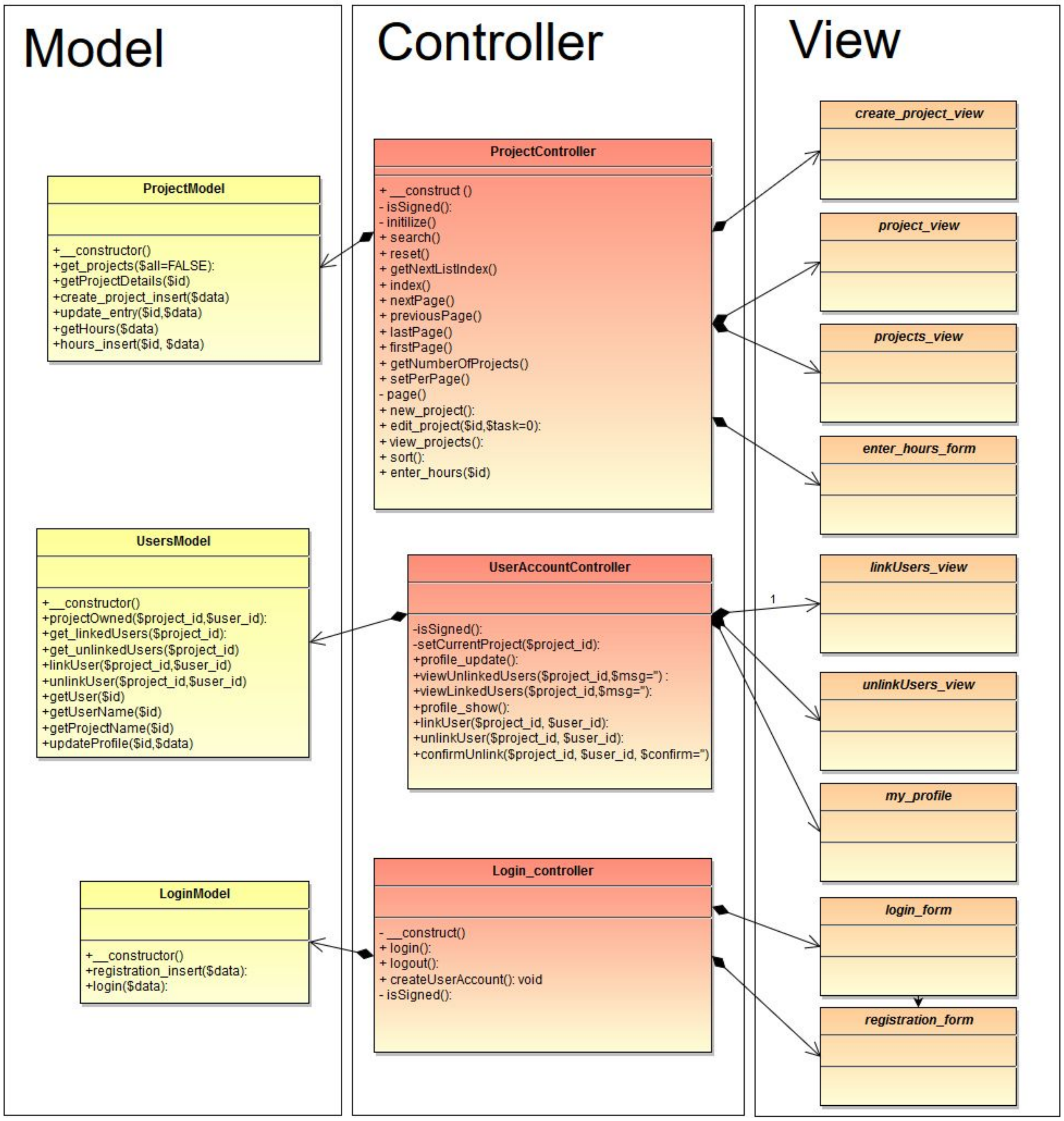
1. Create a user account. The property of checking whether the user has already signed in from another place is not implemented.
2. The user is able to login and logout of their account.
3. The user can change their email address, name.
4. The user can create new projects or modify the details of existing projects.
5. The user can list the projects they created as well as the projects in which they are a participant. This is the default view of the application.
6. Users can prioritize their projects. This feature was implemented as the ability to sort projects, as well as edit the textual description of the priority in the project.
7. We enabled users the ability to enter hours on a project and have the project reflect the changes.
8. The user can add and remove other users from your projects (i.e. the ability for other users to enter hours on the project).

A class diagram showing the final set of classes and relationships of the system.

The primary functions of the systems have not changed, but our internal design did. The following is the old and new class diagrams, respectively. Class Diagram from Project Part 2 (For a closer look, refer to the PDF file title [“class diagram.pdf” in the assignment-submission folder]).



New class diagram



Did you make use of any design patterns in the implementation of your final prototype? how?

We did not make use of any design patterns in the implementation of our final prototype due to time constraints, but we did apply the architectural pattern, Model-View-Controller (MVC). The MVC architectural pattern, which combines other design patterns (such as strategy, observer, or composite), was a good fit for our overall system as we wanted to create a dynamic web application. MVC fits this need by providing quick access from the database to the view through the use of controllers. Additionally, this helped us design a system with high cohesion, as each model, view, and the controllers are closely related in their purpose.

Where could you make use of design patterns in your system?

Decorator

We could make use of the decorator design pattern for all of our views instead of using Codeigniter's views. For example, we use codeigniter's header and footer, but we could instead implement the headers and footers as concrete decorator classes and have the primary view as the abstract decorator class. Thus we can call the concrete decorators when we want the header and footer applied.

Iterator

The iterator design pattern uses an iterator to traverse a container and access the container's element. We could make use of this design pattern when filtering out users and projects that pertains to them. Every user has specific projects that they are working on; therefore an iterator can be used to go through the list of projects and display the projects that pertains to that particular user.

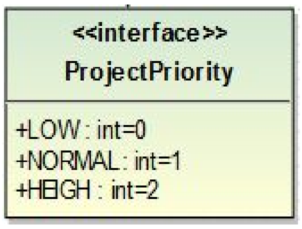
Memento

The memento design pattern captures and externalize an object's internal state which is useful for future restoration. This design pattern could be used in our project to allow users to revert back to previous values or information if they changed their minds or made a mistake. For example, if the user entered an incorrect number of hours and saved and closed it, they could undo their changes and restore it to the previous value.

What changed in your class diagram and why? Compare and contrast the old with the final class diagram that represents the final state of the system.

The class diagram was changed significantly to neatly fit the Codeigniter framework. We summarized the changes below:

Originally we had **Project**, **User**, and **Account** classes that were designed to contain variables regarding the instantiated objects of those types. However, as we implemented the MVC framework, we realized that all of these things would be stored in the database and would instead be accessed by the models.



The interfaces **generalStatus, accountStatus**, and **ProjectProirity** were not used since we migrated the project, user, and account classes to the database. However, we guaranteed their values by changing them in the database structure. For example, instead of priority in the project's table as a varchar data type, we made it an enum of the same values that were originally in the interfaces (low, normal, high).

FormValidationController interface was also removed. This was mainly due to our framework, Codeigniter, having a form validation class that performed the same functionality (thus not having to reinvent the wheel for this feature).

Models:

Originally, we had the login and logout processes go through the authentication controller. As we had moved the user fields to the database, we reorganized these processes so that they were performed through their own particular model (**Login_model**). This model performed database transactions for login, logout, and registering a new account.

We expanded the functions in our **Project_Model**. Originally, we had *read*, *write,* and *valid*, mostly as placeholders for performing the primary model functions. However, in the new diagram we updated to represent the entire range of functions we're performing, which include getting projects and project information, creating projects, updating projects, getting hours, and entering hours.

Similar to **project_model**, we expanded the function list for our **users_model**. Read() became get_linkedUsers, get_unlinkedUsers, and getUser. Write() became updateProfile, unlinkUser, and linkUser. We also needed additional functionality, so we added projectOwned(), which used to check ownership of a function, and some additional database retrieval functions for certain view output.

Controllers

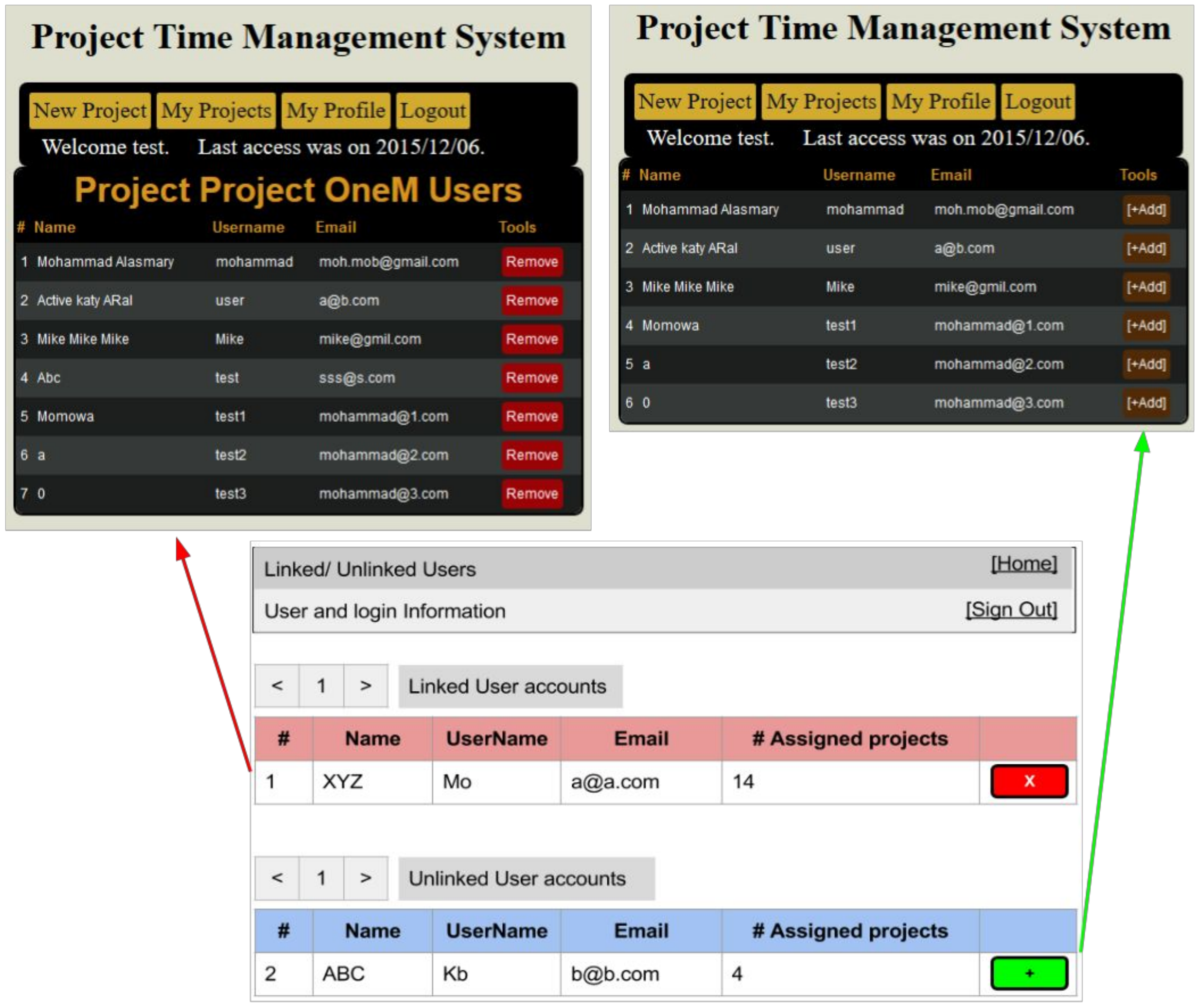
We kept the same number of controllers but reworked the contents of each. For example, as mentioned earlier, we removed all the variables from the controllers and retrieved them within the database instead (or through session variables).

We removed the initialize() function from the **login_controller**, as it was rendered irrelevant by the Codeigniter framework. We also renamed the checkAuthentication() function to isSigned().

Within the **projects_controller**, we never implemented the export() function, so its signature is no longer present in the class diagram. Additionally, we implemented a number of functions that assist the table and pages in the view (such as nextPage(), previousPage(), etc.). We used these as an alternative to creating javascript pagination functions or CodeIgniter pagination classes. We also renamed some functions and updated the class diagrams to reflect the actual code. On the other hand, including pagination functions, in the projects_controller is considered breaking single responsibility principle. However, we had implemented the pagination code in a separate class, but we could not neither use it or load to the framework. Therefore, we added them to the controllers.

For **users_controller**, the class diagram was relatively unchanged, except for the renaming of certain functions. However, we did split getNumberOfUserAccounts() into two functions (viewLinkedUsers and viewUnlinkedUsers). We did this to better follow the single responsibility principle, rather than trying to cram too many features into one function.

getNumberOfUserAccounts function divided into viewLinkedUsers, viewUnlinkedUsers. The reason is we though we can easily have many tables in one page and perform their operations; however, this turns out not to be not straightforward and produces a horrible coding. Therefore, we made separate views for every function, linking and unlinking users, which was slightly different from our original UI layout. We also added a few additional functions (such as confirmUnlink()) for features that we did not anticipate in the original diagram.



Project Time Management System

New Project | My Projects | My Profile | Logout

Welcome test. Last access was on 2015/12/06.

Project Project OneM Users

#	Name	Username	Email	Tools
1	Mohammad Alasmary	mohammad	moh.mob@gmail.com	Remove
2	Active katy ARal	user	a@b.com	Remove
3	Mike Mike Mike	Mike	mike@gmil.com	Remove
4	Abc	test	sss@s.com	Remove
5	Momowa	test1	mohammad@1.com	Remove
6	a	test2	mohammad@2.com	Remove
7	0	test3	mohammad@3.com	Remove

Project Time Management System

New Project | My Projects | My Profile | Logout

Welcome test. Last access was on 2015/12/06.

#	Name	Username	Email	Tools
1	Mohammad Alasmary	mohammad	moh.mob@gmail.com	[+Add]
2	Active katy ARal	user	a@b.com	[+Add]
3	Mike Mike Mike	Mike	mike@gmil.com	[+Add]
4	Momowa	test1	mohammad@1.com	[+Add]
5	a	test2	mohammad@2.com	[+Add]
6	0	test3	mohammad@3.com	[+Add]

Linked/ Unlinked Users [Home]

User and login Information [Sign Out]

< 1 > Linked User accounts

#	Name	UserName	Email	# Assigned projects	
1	XYZ	Mo	a@a.com	14	x

< 1 > Unlinked User accounts

#	Name	UserName	Email	# Assigned projects	
2	ABC	Kb	b@b.com	4	+

Views

All of the view classes in the old class diagram were replaced by the following views: create_project_view, enter_hours_form, linkUsers_view, login_form, my_profile, project_view, projects_view, registration_form, and unlinkUsers_view. Additionally, we removed all separate message classes and implemented their features within the main views.

How it helped doing the diagrams first before coding if you did not need to change much.

Although we did need to change much of the code on implementation, the diagrams helped. First, it reduces the inconsistency between team members' work. Everyone use them as references and stuck to them as much as possible when making decisions. We assigned different classes to different team members in order to give everyone an opportunity to learn and practice their coding. Because of the class diagram, it was easy to integrate the final code since each of the three controllers were listed separately in the class diagram and each participant knew the relationship to each other class in the system. Likewise, when writing only one segment of the code, even if another piece of functionality is not implemented, you know where it should be and what it should do, so it is easy to work around a stub or signature and have the code be integrated once it gets updated.

What have you learned about the process of analysis and design now that you have stepped through the process to create, design and implement a system?

We found that creating the activity diagram and use cases helped us solidify what we were actually trying to accomplish with the application. It helped alleviate any confusion about the overall structure of the program as we all knew the purpose of each feature.

As mentioned earlier, while we found it useful to have the class diagrams created before coding, it was quite challenging to create class diagrams in a language that most of us had never used before. As such, we think that the analysis and design process is best suited for languages you are familiar with, and gets better as you become more proficient in said language.

Frameworks often have an initial learning curve to understand the framework structure, and its pre-existing functions, but once you learn how to use it, it can be a valuable and time-saving resource. However, we did find it challenging to create class diagrams for a framework we had never used, hence the sweeping changes between the original diagrams and the completed diagrams.

Lastly, we learned that designs continue to change throughout the project. It is never set in stone until the final project has been completed, whether it’s from our UI designs to our class diagrams to our code, we have to learn to adapt and change things as we go along.