Initiatives Platform

Analysis and Design Document

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Description** | **Author** |
| 04/Apr/2018 | 1.0 | First Draft | Robert Varadi  Ieremias Viorel |
| 25/Apr/2018 | 2.0 | First iteration | Robert Varadi  Ieremias Viorel |
|  |  |  |  |
|  |  |  |  |

Table of Contents

I. Project Specification 4

II. Elaboration – Iteration 1.1 4

1. Domain Model 4

2. Architectural Design 5

2.1 Conceptual Architecture 5

2.2 Package Design 7

2.3 Component and Deployment Diagrams 8

III. Elaboration – Iteration 1.2 8

1. Design Model 8

1.1 Dynamic Behavior 8

1.2 Class Design 10

1.3. Design Patterns 11

2. Data Model 12

3. Unit Testing 12

IV. Elaboration – Iteration 2 12

1. Architectural Design Refinement 12

2. Design Model Refinement 13

V. Construction and Transition 13

1. System Testing 13

2. Future improvements 13

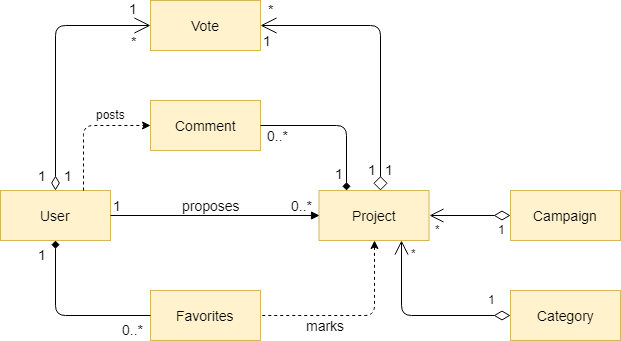
VI. Bibliography 13

# Project Specification

Implement a system where regular people can propose plans they would like to see implemented in their city. Other people, also users of the system, vote for the project they consider most useful. Projects are divided in some categories, on domain. This can however be extended on budget, for example, or implementation time (short-term, medium-term, long-term, etc.). Each person is able to vote for a limited number of projects and choosing the best project may take place in more than one round. The application must contain a comments section, too so people will be able to discuss about the different projects. Each person must have a “favorites” corner from where he/she can choose the projects he/she likes the most.

# Elaboration – Iteration 1.1

# Domain Model



*Domain Model Diagram*

The main entities involved in the system are the User, the Project and the Vote. A User entity abstracts personal and account information of a real user of the system. A Project entity contains relevant information related to the initiatives and ideas proposed by the citizens. In order to provide a useful classification for the projects, each one has attached a Category attribute. To provide an interaction between the users and to facilitate debating on a subject, a Comment entity is added in the system. Finally, in order to enhance the experience of the user, there is a “Favorites” corner, where a user of the system can mark the projects he/she wants to review later, so a Favorite entity, which maps a User and a Project is necessary. The central purpose of the application is to determine the needs of most of the people so a voting mechanism is built around the Vote entity. A Vote instance represents the decision of a User to support an idea. The multiplicity of the relationships is OneToMany both in case of User-Vote and in case of Project-Vote, because a User has a number of votes he/she can exert, while a Project can be supported, through the mechanism of votes, by multiple users. A more accurate result is obtained if the projects are filtered and eliminated in stages so campaigns are added in the system in order to achieve this purpose.

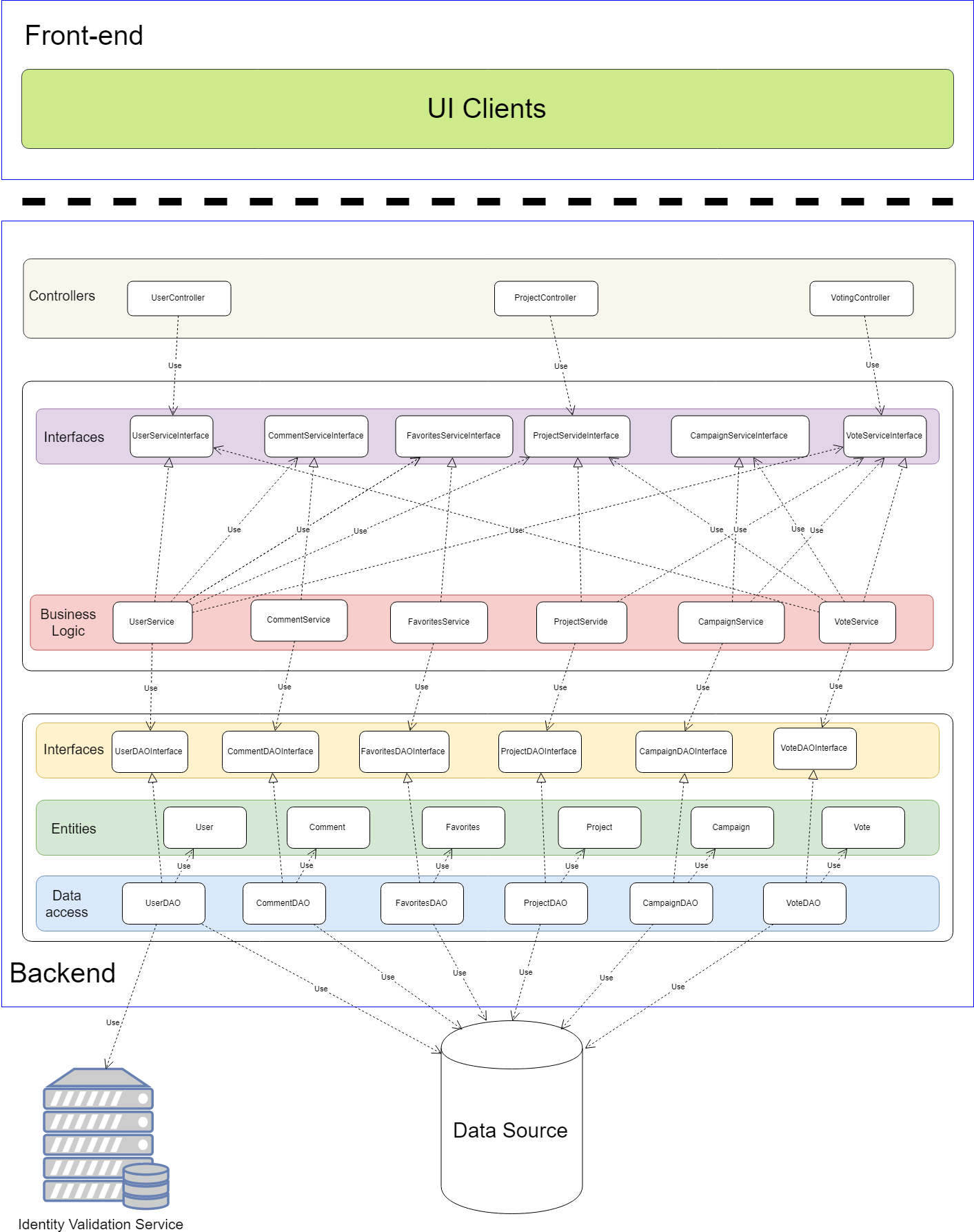
# Architectural Design

## Conceptual Architecture

The application is built as a decoupled system, composed of a REST API and a ReactJS – based Web Application client. The core API receives HTTP requests coming from the client, processes them and returns the HTTP response which will be displayed, in some format, by the client. From this perspective, the application is a distributed client – server system.

The back-end is structured using the layers pattern, which has the advantage of isolating specific processing steps into independent components which can be tested, replaced, maintained and extended separately. Also, the layers will communicate with each other only through services which are exposed from one to another through interfaces which hide the specific implementation.

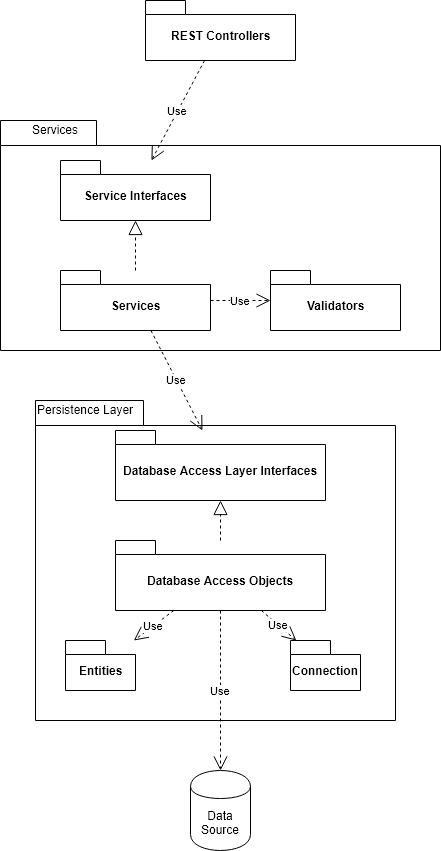
At the application level, we use MVVM (Model View ViewModel). The Model component is represented by the persistence layer and the business logic layer, where the state of the application is stored and the corresponding processing logic is implemented. The View component is the graphical user interface itself, which is only responsible for the “look” of the application, namely what the user sees and interacts with. The View – Model mechanism is specific to the ReactJS implementations, in which the view stores its own state from which it renders for the user. The advantage of this approach is that the view and the model are decoupled and any number of clients can be added or replaced without affecting the application.



*Architecture Diagram*

## Package Design

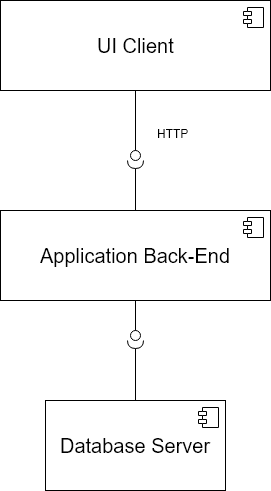
At package level design level, each layer is assigned a package, which is responsible for the whole functionality of the layer. The coupling between the layers is low, because the only way a higher – level layer communicates with a lower – level one is through the interface exposed by the latter one.



*Package Diagram*

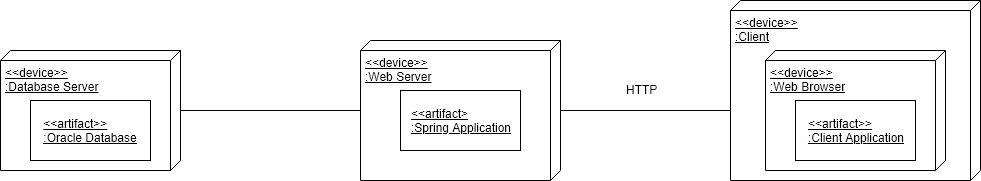
## Component and Deployment Diagrams

The component diagram contains the layout of the system at physical component level, that is, files, libraries, executables. From this perspective, we can draw the following component diagram:



*Component diagram*

The deployment diagram contains the physical layout of the system grouped by tiers. We might have the Database Server on a tier, the Application Server on a different one and the UI Client on a third one. Thus, the deployment diagram is the following one:



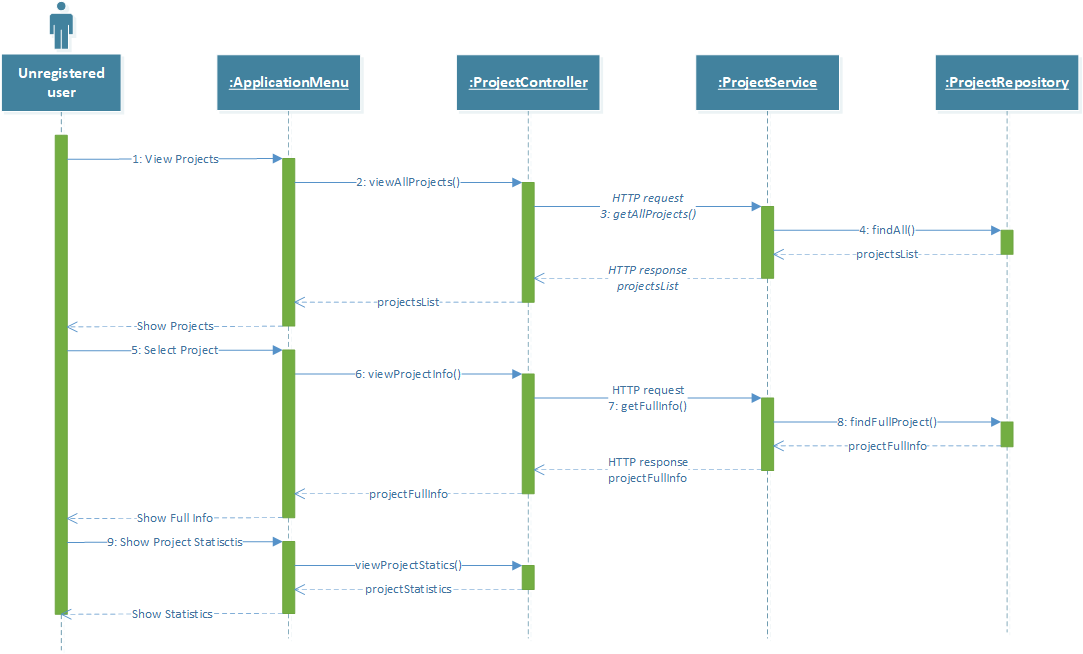
*Deployment Diaagram*

# Elaboration – Iteration 1.2

# Design Model

## Dynamic Behavior

In this subsection, the dynamic behavior of our system is being described on two use cases that we considered relevant. The first use case is the one in which the unregistered user wants to see the projects, the other one being the one in which the content supervisor wants to block a user who is the author of a flagged project or comment. The first use case is described using a sequence diagram, while the second one is described with the help of the communication diagram.

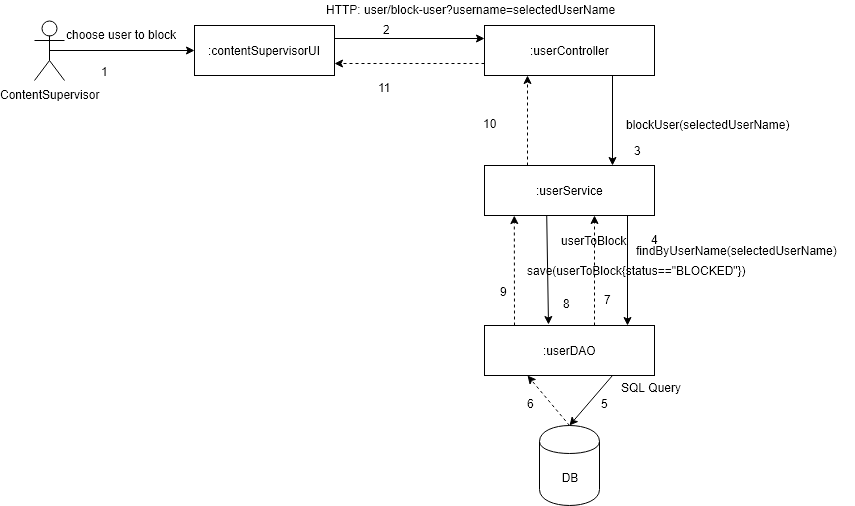


*Sequence Diagram*

The sequence di­­­­agram above presents the evolution of the system when the View Simple Project Statistics use case, of the actor Unregistered user is carried out. This use case coincides with the action of a casual user, that visits the platform without having an account or without logging in into the account and wants to see information about a specific project related to time of proposal, votes and views so far and category to which it belongs.

The user chooses from the Application Menu to see the projects listed so far, either form a specific category, or belonging to any category, based on a filter. A scrollable list is presented to him on the following page. Next the actor has the possibility to view more information about a project it selects. The page is filled with information available for that project. Finally, if besides the general information presented so far, the user wants to see detailed metrics, he uses a dedicated button on the current page that launches a screen with the required information.

The communication diagram models the interactions between objects or parts in terms of sequenced messages. For the “Block User” use case, the content supervisor chooses the user who will be blocked. At this time, the view will send an HTTP request to the userController object. This component parses the HTTP request from where it will take the information needed to identify the user, for example his/her username. This information will be sent to the UserService which will fetch the user to be blocked by asking the userDAO to query the database and return the user who has the given username. As soon as the userService has the User object associated to the user who has to be blocked, the service can modify the object’s status to “BLOCKED” and then ask the userDAO to save the modified user into the database.

**

*Communication Diagram*

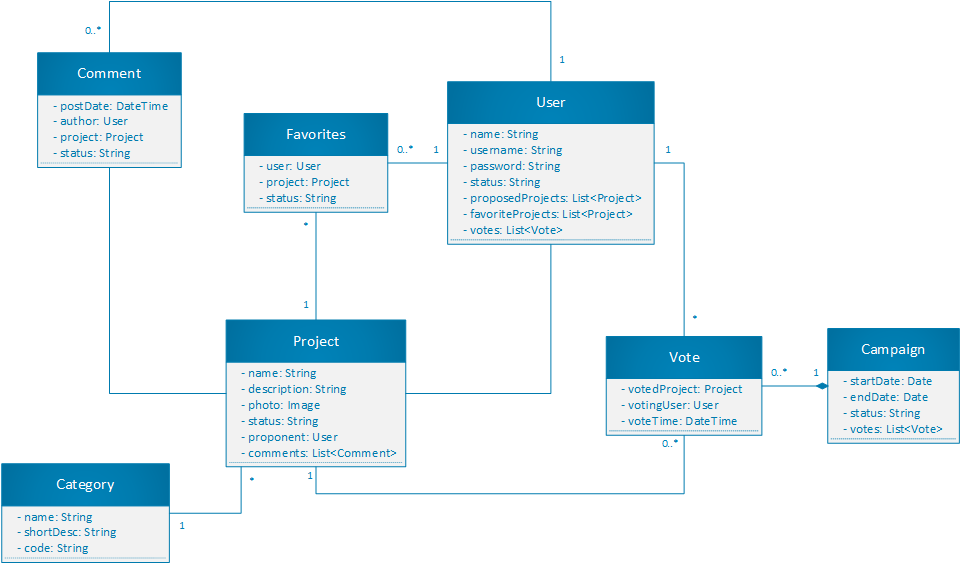
## Class Design

In this subsection we describe the design of the entities which take part in the system. These entities are the following ones:

* User
* Project
* Comment
* Vote
* Category
* Campaign
* Favorites

The entities have the following relations between them:

* A User object has more associated comments since a user can post more comments in different parts of the application;
* A User has more Favorites objects because he/she can mark more projects present in the application as favorites;
* A User has more votes associated since he/she is able to vote for more projects;
* A Project may have more comments related to it, so there is a one to many association relationship;
* More projects may have the same category, so the relationship between projects and categories will be many-to-one.
* A project may have more votes so the proper relationship between Project and Vote will be the one-to-many association relationship;
* During a campaign more users can vote and no vote should exist without a campaign so the relationship between Vote and Campaign is a many-to-one composition relationship.



*Class Diagram*

## Design Patterns

**Factory pattern** is used in the case of the User entity. The system has 3 types of users: regular users of the application, content advisors and system admins. The attributes related to personal information, are captured in a UserInfo field that is necessary and thus common to all the users. The differentiation between the different types of users is made based on a ProfileInfo field, that stores information about the allowed and restricted functions for each type. An implementation based on the factory pattern makes it easier to create and handle users.

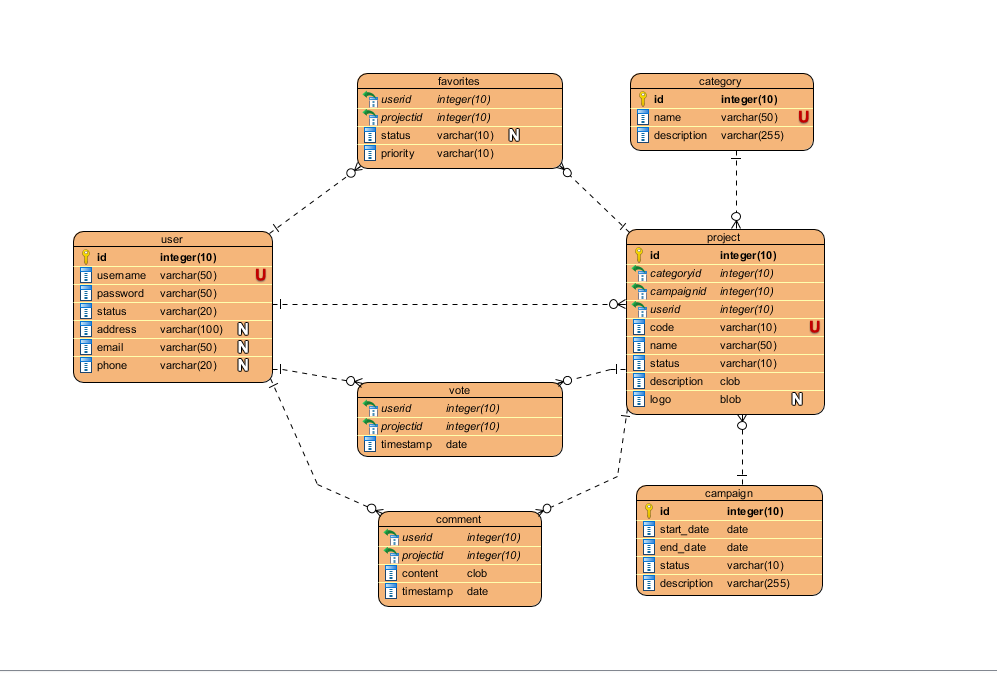
**Prototype pattern** is considered for handling large and costly objects. The entities that fit into this category are instances of the User class, and hold many fields, some of which are collections of references to other objects. Prototype objects that are cached and persisted only when changes occur are used with the purpose to decrease the effort to access the database and create new instances.

**Filter pattern** is necessary because the system is populated with many entities of the same class, mainly users, projects and votes. Complex filters have to be implemented on all these entities, so an approach based on the filter pattern allows splitting the functionality into small sized filters that can be composed into complex structures.

**Builder pattern**in order to allow users to generate some custom reports and statistics related to the projects and votes. They will choose the type of statistics they want to see and how to be displayed.

# Data Model

The data model or database diagram is almost the same as the class diagram presented above, the difference is that each object will have a primary key or a composite key and these keys will be used to reference the objects, instead of pointers. The multiplicities of the relationships are the same. The red “U” letter on the right of the attribute means that the attribute should be unique, while the “N” letter means that the attribute might be null.



*Data Model Diagram*

# Unit Testing

*[Present the used testing methods and the associated test case scenarios.]*

# Elaboration – Iteration 2

# Architectural Design Refinement

*[Refine the architectural design: conceptual architecture, package design (consider package design principles), component and deployment diagrams. Motivate the changes that have been made.]*

# Design Model Refinement

[Refine the UML class diagram by applying class design principles and GRASP; motivate your choices. Deliver the updated class diagrams.]

# Construction and Transition

# System Testing

*[Describe how you applied integration testing and present the associated test case scenarios.]*

# Future improvements

*[Present future improvements for the system]*

# Bibliography