Technical University of Moldova
Inginerical department S.A.

Course Work

AMOO

Subject: Analysis and modelling of Facebook

Author: Terman Emil FAF161
Prof: R. Melnic, M. Gavrilita

Chisinau 2018

Contents

Intro to Theory

Project description, User stories

Facebook is a web platform for communication and networking. It offers a huge range of options: from simply chatting with friends, to event organization and page management. Nowdays, it is a very popular platform in the entire world.

Basically, the user is able to create an account and use the platform. I analyzed the following user actions:

- registration;
- login;
- commenting;
- adding a photo;
- sending a friend request;
- creating an event;

The admins on this platform have some special rights. For example, they are able to delete some posts if they find them inappropriate. In this project I tried to analyze this app with different UML diagrams.

User stories

- the user signs up, unless he is already registred, then he logs in. He can view the News Feed, search for different people, post a photo, comment and many other actions.
- the admin can receive email notifications about content that requires atentions - content that the platform decided that it is inappropriate, or it received user reports. He is able to remove the content or set it as a False Alert.

Modeling Lnaguages

Use case diagrams are usually referred to as behavior diagrams used to describe a set of actions (use cases) that some system or systems (subject) should or can perform in collaboration with one or more external users of the system (actors). Each use case should provide some observable and valuable result to the actors or other stakeholders of the system.

Modeling language: can be graphical or textual:

- Graphical modeling languages use a diagram technique with named symbols that represent concepts and lines that connect the symbols and represent relationships and various other graphical notation to represent constraints;
- Textual modeling languages may use standardized keywords accompanied by parameters or natural language terms and phrases to make computer-interpretable expressions;

Conceptual OO Analysis / Technical OO Design

Object-oriented analysis and design is a popular technical approach for analyzing and designing an application, system, or business by applying object-oriented programming, as well as using visual modeling throughout the development life cycles to foster better stakeholder communication and product quality.

The primary tasks in object-oriented analysis (OOA) are:

- find the objects;
- organize the objects;
- describe how the objects interact;
- define the behavior of the objects;
- define the internals of the objects;

Abstraction, Encapsulation, Decomposition, Generalization

All the following concepts represent basic analysis and design techniques. They are interrelated and normally used together during software development. We use them even though we are not always aware of it. Deeper understanding of these concepts helps us to be more accurate and effective.

1. **Abstraction** - in general, it's the process of consciously ignoring some aspects of a subject under analysis in order to better understand other aspects of it. In other words, it is some kind of a simplification of a subject. In software in particular, analysis and design are all about abstraction.
   - when the architecture is modeled, you concentrate on high-level modules and their relationships and ignore their internal structure;

  - each UML diagram gives a special, limited view on the system. Therefore, it focuses on a single aspect and ignores all other things (sequences abstract objects and messages, deployment abstracts network and servers, use cases abstract system users and their interactions with a system, etc);
  - so, abstraction is used to generalize objects into one category in the design phase. For example in a travel management system you can use Vehicle as an abstract object or entity that generalizes how you travel from one place to another;

2. **Encapsulation** - refers to the process of an object controlling outside access to its internal data. This also means to hide functions and methods of a class;

3. **Generalization** - is a relationship in which one model element (the child) is based on another model element (the parent). Generalization relationships are used in classes, components, deployments and use-case diagrams to indicate that the child receives all of the attributes, operations and relationships that are defined in the parent. This also doesn't have names.

4. **Decomposition** - is an application of the old good principle "divide and conquer" in software development. It is a technique of classifying, structuring and grouping complex elements in order to form more atomic ones, organized in a certain fashion and easier to manage. In all phases there are lots of examples:
   - functional decomposition of a complex process to hierarchical structure of smaller sub-processes and activities;
   - high-level structure of an complex application to 3 tiers - UI, logic and data;
   - UML packages are a direct use of decomposition on the model level - use packages to organize your model;
   - to have a good level of understanding of decomposition, you should first understand the concepts of association, composition, and aggregation;

Coupling and Cohesion, Separation of Concerns, Information Hiding, Conceptual integrity

Cohesion and Coupling deal with the quality of an OO design. Generally, good OO design should be loosely coupled and highly cohesive. Lot of the design principles, design patterns which have been created are based on the idea of "Loose coupling and high cohesion". The aim of the design should be to make the application:
- easier to add new features;
- easier to develop;
- easier to maintain;
- less fragile;

Cohesion g is the degree to which one class knows about another class. Let us consider two classes class A and class B. If class A knows class B through its interface only i.e it interacts with class B through its API then class A and class B are said to be loosely coupled. If class A apart from interacting class B by means of its interface also interacts through the non-interface stuff of class B then they are said to be tightly coupled. Suppose the developer changes the class B's non-interface part i.e non API stuff then in case of loose coupling class A does not breakdown but tight coupling causes the class A to break.

Coupling is used to indicate the degree to which a class has a single, well-focused purpose. Coupling is all about how classes interact with each other, on the other hand cohesion focuses on how single class is designed. Higher the cohesiveness of the class, better is the OO design.

Concerns are the different aspects of software functionality. For instance, the "business logic" of software is a concern, and the interface through which this logic being used is another. The separation of concerns is keeping the code for each of these concerns separate. Changing the interface should not require changing the business logic code, and vice versa. Model-View-Controller (MVC) design pattern is an excellent example of separating these concerns for better software maintainability.

Information hiding is the process of hiding details of an object or function. Information hiding is a powerful programming technique because it reduces complexity. One of the chief mechanisms for hiding information is encapsulation - combining elements to create a larger entity. The programmer can then focus on the new object without worrying about the hidden details. In a sense, the entire hierarchy of programming languages, from machine languages to high-level languages, can be seen as a form of information hiding. Information hiding is also used to prevent programmers from intentionally or unintentionally changing parts of a program.

Conceptual integrity it is the principle that anywhere you look in your system, you can tell that the design is part of the same overall design. This includes low-level issues such as formatting and identifier naming, but also issues such as how modules and classes are designed, etc. This is vitally important because inevitably unanticipated issues come up that must be resolved quickly.

SOLID

The term SOLID is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable.

- **S**ingle responsibility principle: a class should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class);
- **O**pen for extensions and closed for modification;
- **L**iskov substitution principle: "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program";
- **I**nterface segregation principle: "many client-specific interfaces are better than one generalpurpose interface";
- **D**ependency inversion principle: one should "depend upon abstractions, **not** concretions";

Grasp Principles

The different patterns and principles used in GRASP are controller, creator, indirection, information expert, high cohesion, low coupling, polymorphism, protected variations and pure fabrication. All these patterns answer some software problems, and these problems are common to almost every software development project.

These techniques have not been invented to create new ways of working, but to better document and standardize old, tried-and-tested programming principles in object-oriented design.

Software Development Process

It is the process of dividing software development work into distinct phases to improve design, product management and project management. It is also known as a software development life cycle. The methodology may include the pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application.

Most modern development processes can be vaguely described as agile. Other methodologies include waterfall, prototyping, iterative and incremental development, spiral development, rapid application development and extreme programming.

Top-down and Bottom-up Design

Component diagrams are different in terms of nature and behavior. Component diagrams are used to model the physical aspects of a system. Now the question is, what are these physical aspects? Physical aspects are the elements such as executables, libraries, files, documents, etc. which reside in a node.

Component diagrams are used to visualize the organization and relationships among components in a system. These diagrams are also used to make executable systems.

Purpose? Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Benefits

- imagine the system's physical structure;
- emphasize the service behavior as it relates to the interface;
- pay attention to the system's components and how they relate;

OntoUML

OntoUML is a ontologically well-founded language for Ontology-driven Conceptual Modeling. OntoUML is built as a UML extension based on the Unified Foundational Ontology (UFO). OntoUML has been adopted by many academic, corporate and governmental institutions worldwide for the development of conceptual models in a variety of domains. It has also been considered as a candidate for addressing the OMG SIMF (Semantic Information Model Federation) Request for Proposal, as is explicitly recognized as the foundations for the Data Modeling Guide (DMG) For An Enterprise Logical Data Model (ELDM) initiative. Finally, some of the foundational theories underlying OntoUML have also influenced other popular conceptual modeling languages such as ORM 2.0.

Analysis and Modeling

Main and Alternative Flows

Main Flow

Uploading a new photo

1. log in;
2. go to user profile;
3. press *photos*;
4. press *Add photos/video*;
5. choose file;
6. set as public;
7. add comment;
8. add tags;
9. upload photo;

Inviting a user to a chat grup

1. log in;
2. create a chat grup;
3. search for the desired user;
4. go to his profile;
5. press the invite button on his profile;
6. choose the *Invite to a chat* option;

Post on facebook

1. log in;
2. choose the *Create post* option;
3. write post content;
4. add a local image;
5. tag some friends;
6. write how you feel;
7. set publicity as *friends-only*;
8. post;

Alternate Flow

Log in with the wrong credentials

1. user goes to log in page;
2. enters wrong credentials;
3. the platform displays an error;
4. the user tries again;

5. another error is displayed;
6. a robot detection test is required this time;
7. user chooses to recover the password trough email;

Post rejected by group admin

1. log in;
2. enter the desired group;
3. submit a post;
4. wait for approval;
5. post gets rejected for being irrelevant to the group;

Registering with an existing email

1. go to register page;
2. enter credentials;
3. submit the request;
4. get an error that the email is already taken by another user;

Functional and Non functional requirements

Functional requirements

What will happen?

1. Register user on the platform;
2. Register a friend request;
3. Register an event;
4. Send email confirmation;
5. Find requested users;

Non-functional requirements

How will it happen?

1. Only registred users can send friend requests;
2. Users can cancel the friend request, until it is accepted;
3. Users can receive notifications;
4. An event invite can be set to be public or private;
5. Event invites can be canceled;

Technologies

Facebook is a social web platform, so I suppose that a user privatness shoul come first. With that said, I will use *Triple DES* or *RSA* encryption algorithms, to ensure everything stays private.

Moving on to profit, like the current facebook, I will sell ads. For that, I would use Google Ads, because they are quite profitable and easy to use.

Since I need to work in a team and continually maintain the web app, I would need a platform where I can easily collaborate with other developers. And **Github** comes with plenty of tools, that fit just right for this problem. Using git, I can create different branches and descriptively commit my changes.

SWOT Analysis

It is a strategic planning technique used to help a person or an organization to identify the Strengths, Weaknesses, Opportunities and Threats related to business competition or project planning. Strengths and Weakness are frequently internally-related, while Opportunities and Threats commonly focus on environmental placement.

- ***S**trenghs*: characteristics of the business or project that give it an advantage over others;
- ***W**eaknesses*: characteristics of the business which can place the business or project at a disadvantage relative to others;
- ***O**pportunities*: elements in the environment that the business or project could exploit to its advantage;
- ***T**hreats*: elements in the environment that could cause trouble for the business or project;

Domain Analysis

- this platform is from the social networking **domain**. People are continously looking for a faster, intuitive and safier app, so I think, my application's main advantage is the full comunication privacy. Another strong point is that this project will be available as open source;
- *The importance of this theme*: a user's privacy is often secretly ignored, or at least, the user isn't 100% sure his chats are private. So, it would be a good idea to make the project open source so that other programmers can confirm my project's privatness;
- **Twiter** and **Instagram** are **similar applications**, but the user's private content is used for advertistment targeting and other nasty things. Instagram is more media based platform, whereas Twiter, it's more focused on comunication;
- *The purpose and objectives*: This app is supposed to be open source, meaning that there will be many comunity contributions, which will greatly affect the popularity.

Project Setup

To work on this project, I would require Visual Studio, because I would write it in ASP.NET Core 2. I would definetly need access to git to work with my team. As for the database, I would install SQLite. I will also need multiple browsers, like Chrome, Firefox, Opera, Sefari, Explorer. For testing on mobile platforms, I would need multiple types of phones: Android, IOS, etc.

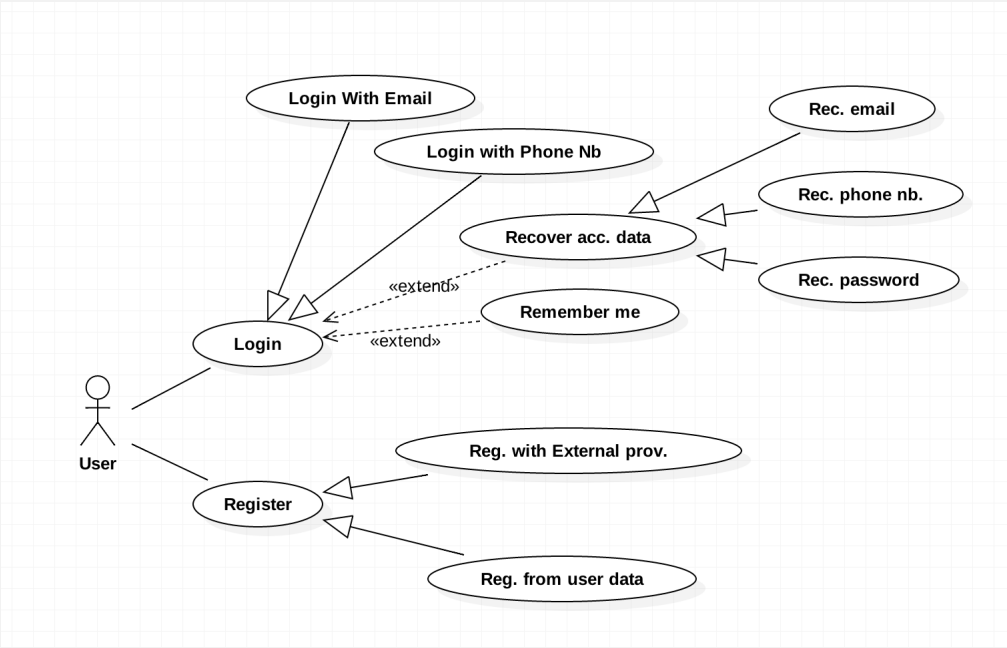
Analysis and Modeling in UML

Use Case Diagram

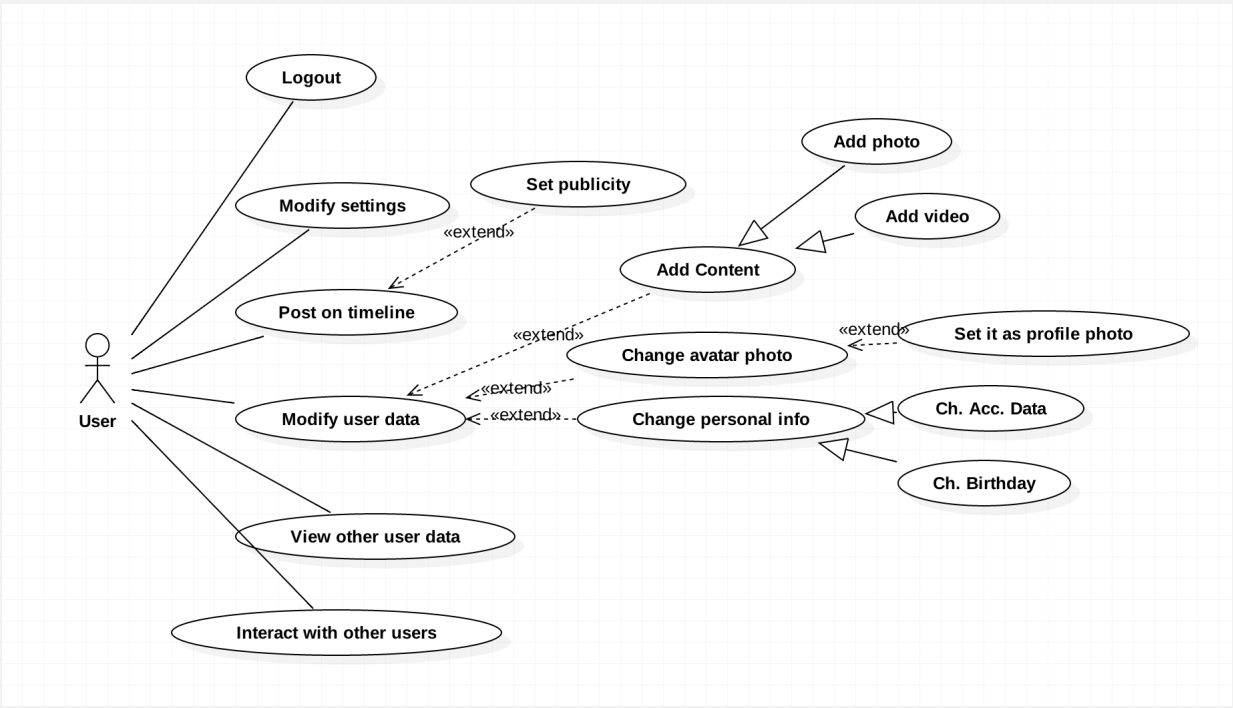
Figure 1 is the *Use Case Diagram* for a user, when he wants to enter FB. The most basic features are *Login* and *Register*. The login can be done through Email or Phone number. If the user forgets his password/email/phone number, he is able to recover it. Registration may be done either through an external provider, like Google account or like the usual.

In *Figure 2* is represented the most basic features a user can access. The first, most natural option, is to log out. He is also able to edit FB settings, like notifications. Moving on to *Modify user data*, this option offers to add photos or videos, change the avatar photo, change birthday and other such data. As many other social platforms, the user is able to post on the timeline, with configurable publicity. As well as **Interact with other users**.

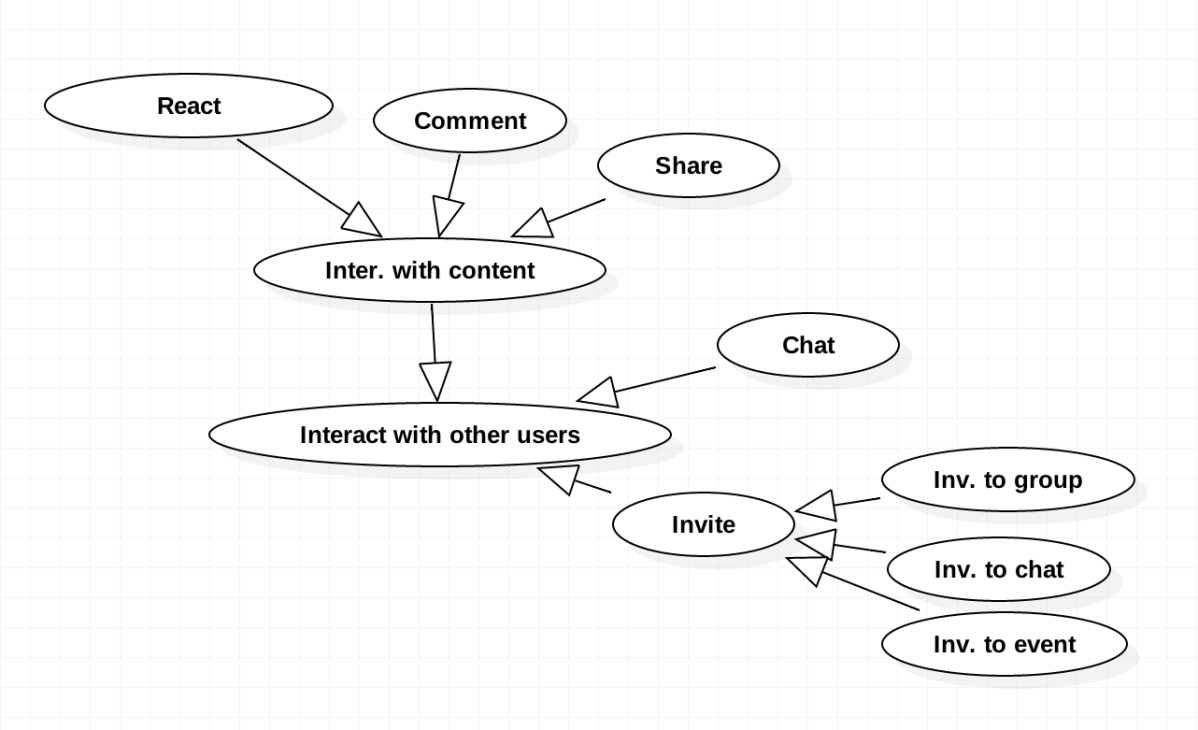
In *Figure 3* is shown the Interaction use case diagram. A logged user, can chat. He is able to interact with some Facebook Content, like reacting to a post, comment or share. He is also able to invite users to groups, events or chat groups.



Img 1: Enter Facebook Use Case Diagram



Img 2: General use of FB Use Case Diagram



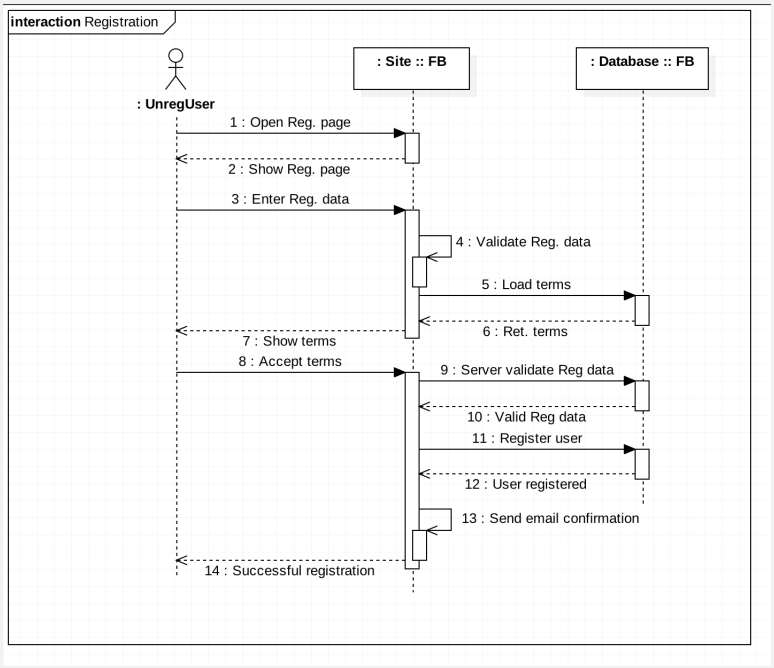
Img 3: Interactions with other Users Use Case Diagram

Sequence Diagram

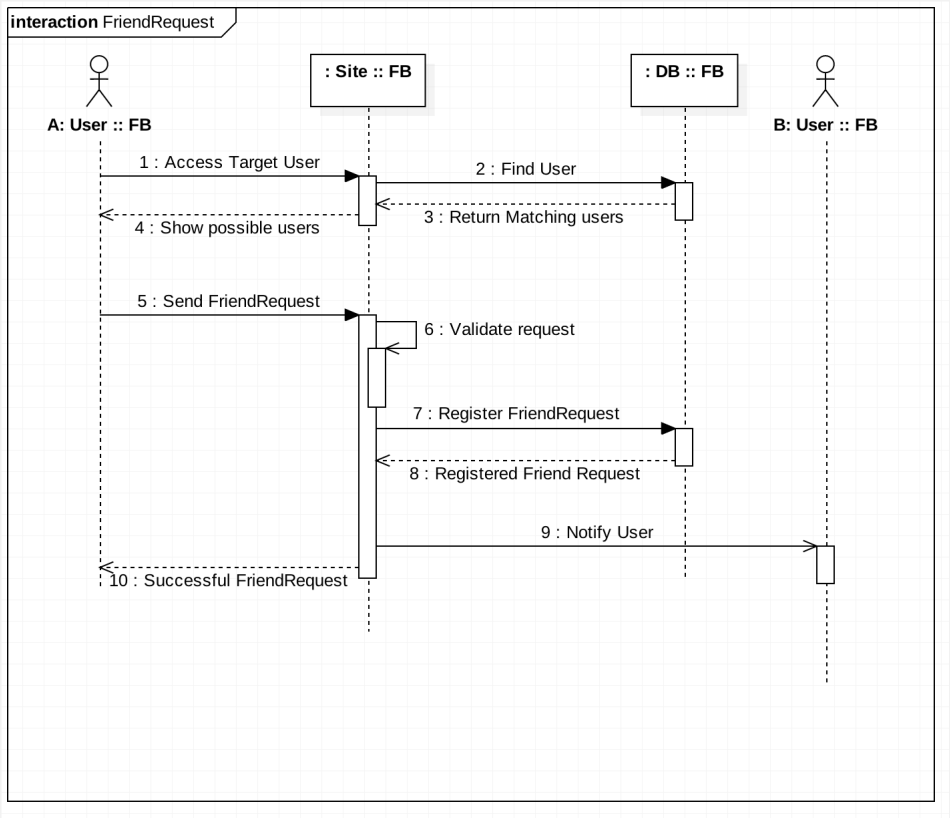
In *Figure 4* is explained how the process of registration looks like. The actor is an unregistred user, and he interacts only with the site. While the Facebook platform, has interactions with both user and the database. First, the user tries to submit some registration data. Then the platform validates the input and loads the *Terms of use* from the database. The user then accepts the terms and the platform validates the registration data on the server. After a successful validation, the user is then registred in the database as an unconfirmed user. After that, a message requesting email confirmation is sent back.

Moving on to user interaction, we have in *Figure 5* a friend request *sequence diagram*. In this case, two actors are involved: the sender (A) and the receiver (B). First, the sender tries to access the target. The platform then, returns a list of possible matches. After that, the first user submits a friend request. Then a validation is run and the request is registred on the database, followed by a notification to the *receiver*. Finally, the first user receives a *success* message.

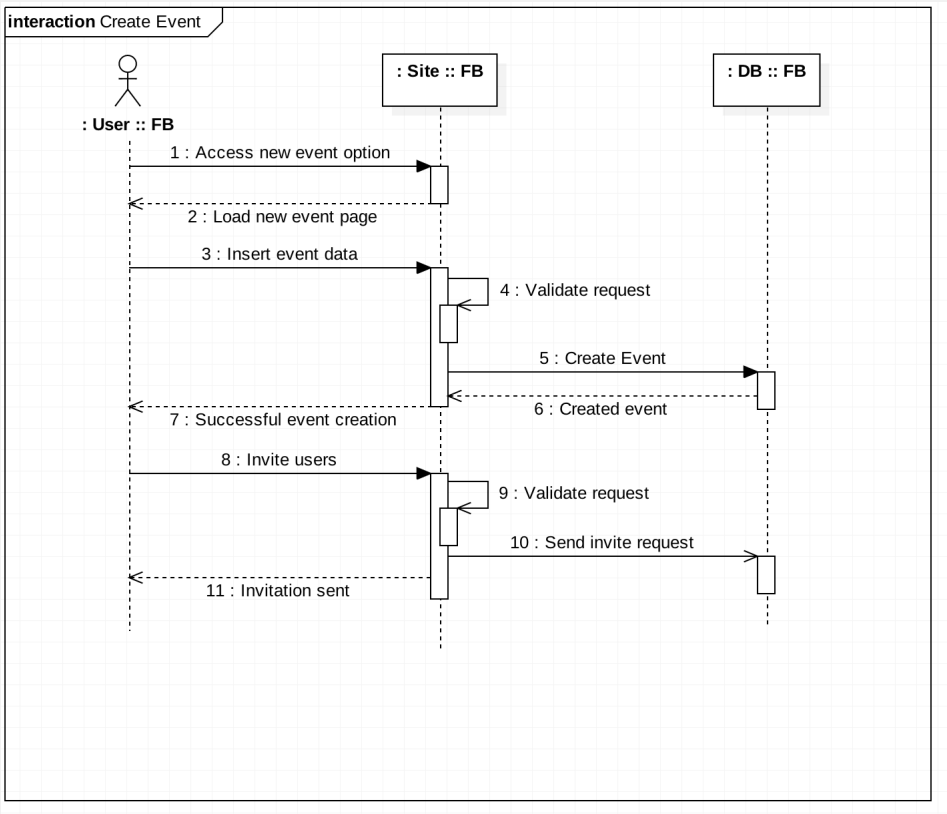
Finally, in *Figure 6*, the diagram of creating an event is shown. First, the user starts with accessing the *Create event* page. Then, he submits the event's data, followed by a validation on the platform. The event is then created in the database. After that, the user, ussually decides to invite people to his event: invitations are sent and the requests are registred in the database.



Img 4: Seq Diag: Registration



Img 5: Seq Diag: Friend request



Img 6: Seq Diag: Create event

Collaboration Diagram

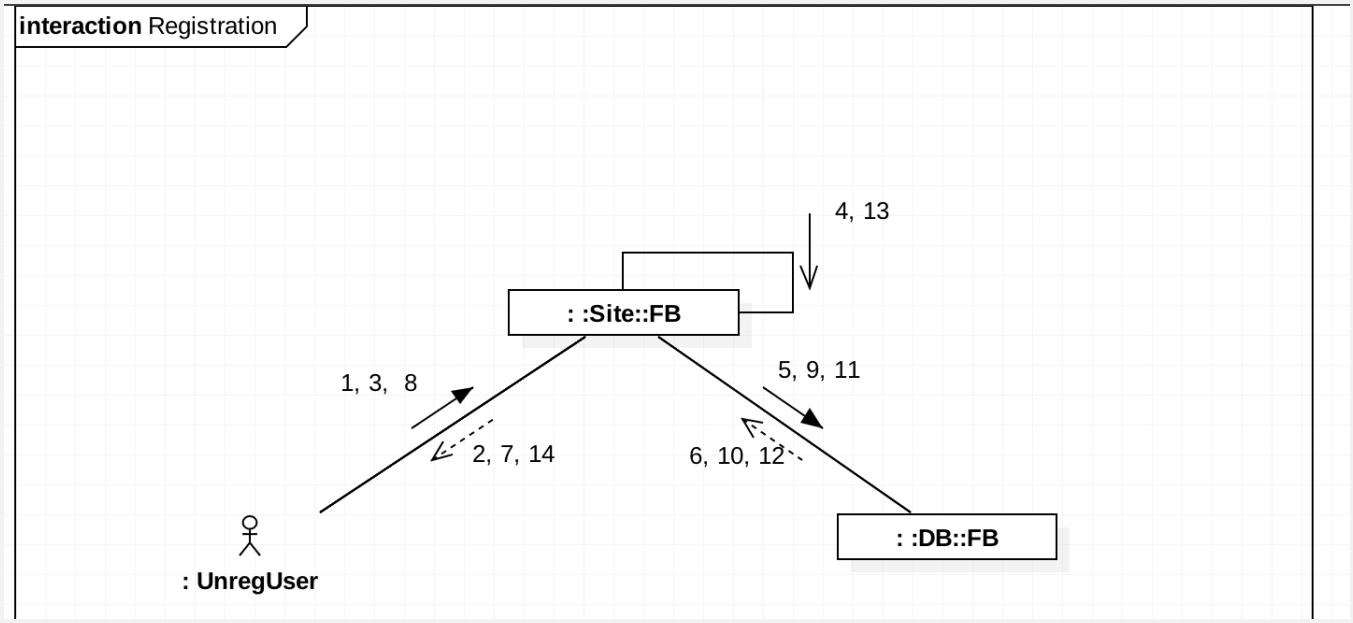
The following diagrams are the exact copy from the previous laboratory task, but transformed in collaboration diagrams. For a more detailed description, please refer to my previous laboratory task.

- ***Figure 7*** **- Registration**
  1. open reg. page;
  2. show reg. page;
  3. enter reg. data;
  4. validate reg. data;
  5. load terms;
  6. return terms;
  7. show terms;
  8. accept terms;
  9. server validate reg. data;
  10. valid reg. data;
  11. register user;
  12. user registered;
  13. send email confirmation;
  14. confimation sent;
- ***Figure 8*** **- Friend request**
  1. access target user;
  2. find user;

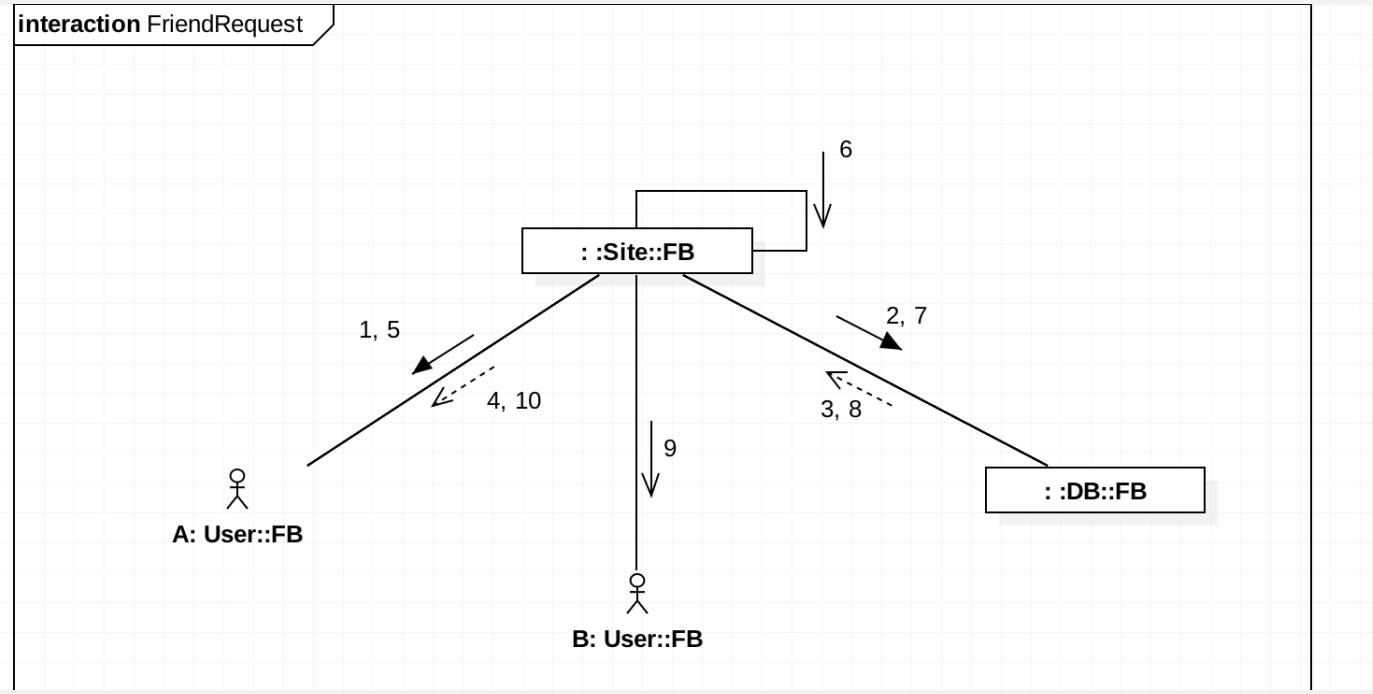
3. return matching users;
4. show possible users;
5. send friend request;
6. validate request;
7. register friend request;
8. registered friend request;
9. notify user;
10. successful friend request;

– ***Figure 9*** **- Create an event**

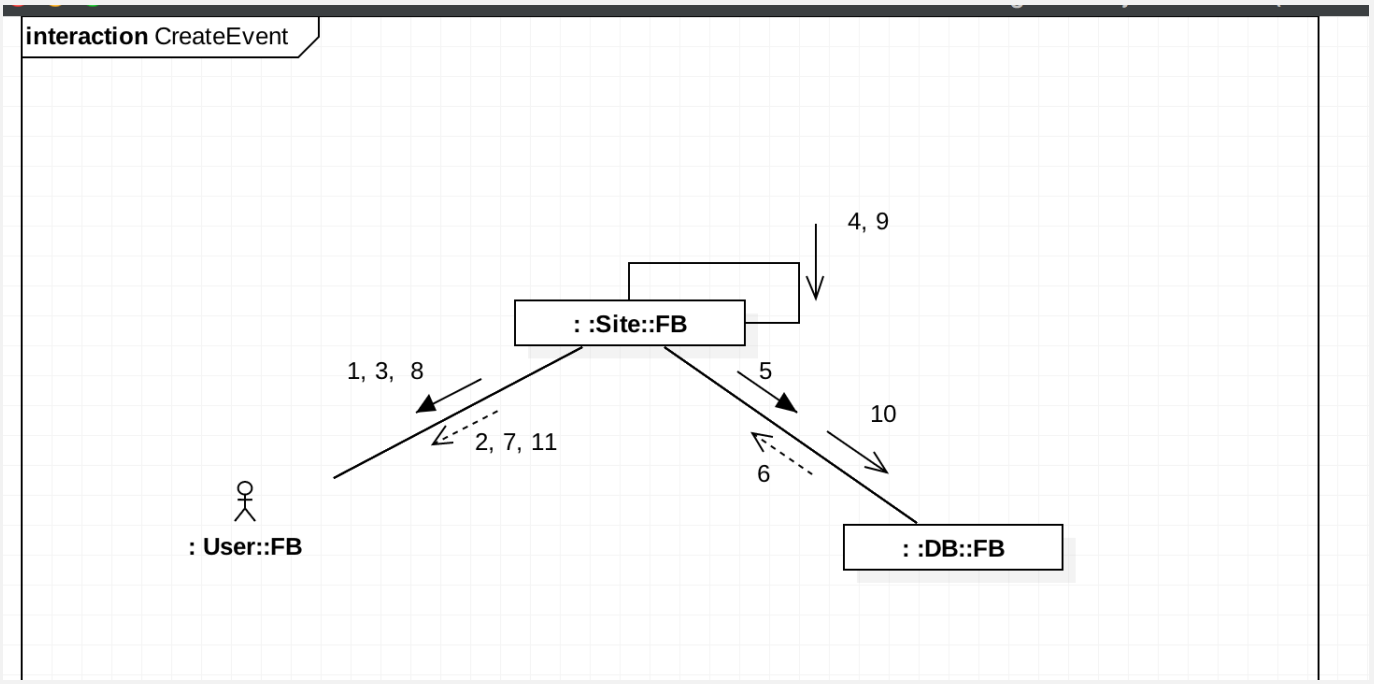
1. access *new event* option;
2. load *new event* page;
3. submit event data;
4. validate request;
5. create event;
6. created event;
7. successful event creation;
8. invite users;
9. validate requests;
10. send invitation requests;
11. invitations sent;



Img 7: Registration



Img 8: Friend request



Img 9: Create event

Class Diagram

In *Figure 10* it's represented the general structure of Facebook that is related to the user. It contains:
- Account;
- Profile;
- Converstaion;
- Message;
- Comment;
- RCSContent (React Comment Share content)
- Photo;
- Share;
- Reaction;
- Account State interface;
- Active account state;
- Banned account state;
- IssuedContent interface;

The most important class, is the **Profile**. It has an *Agregation* from **Account**. (Why not *Composition*? Because the account may get deleted, but the profile may still remain. The profile is also used to identify comments, reactions, shares, converstations and other things). All the user's friends are stored inside the Friends list. FriendRequests list is a container for pending friend requests. The user may *ProcessFriendRequest()* and set it as accepted or rejected. The profile may be also deleted. In this case, all its public *Issued content* will become anonymus.

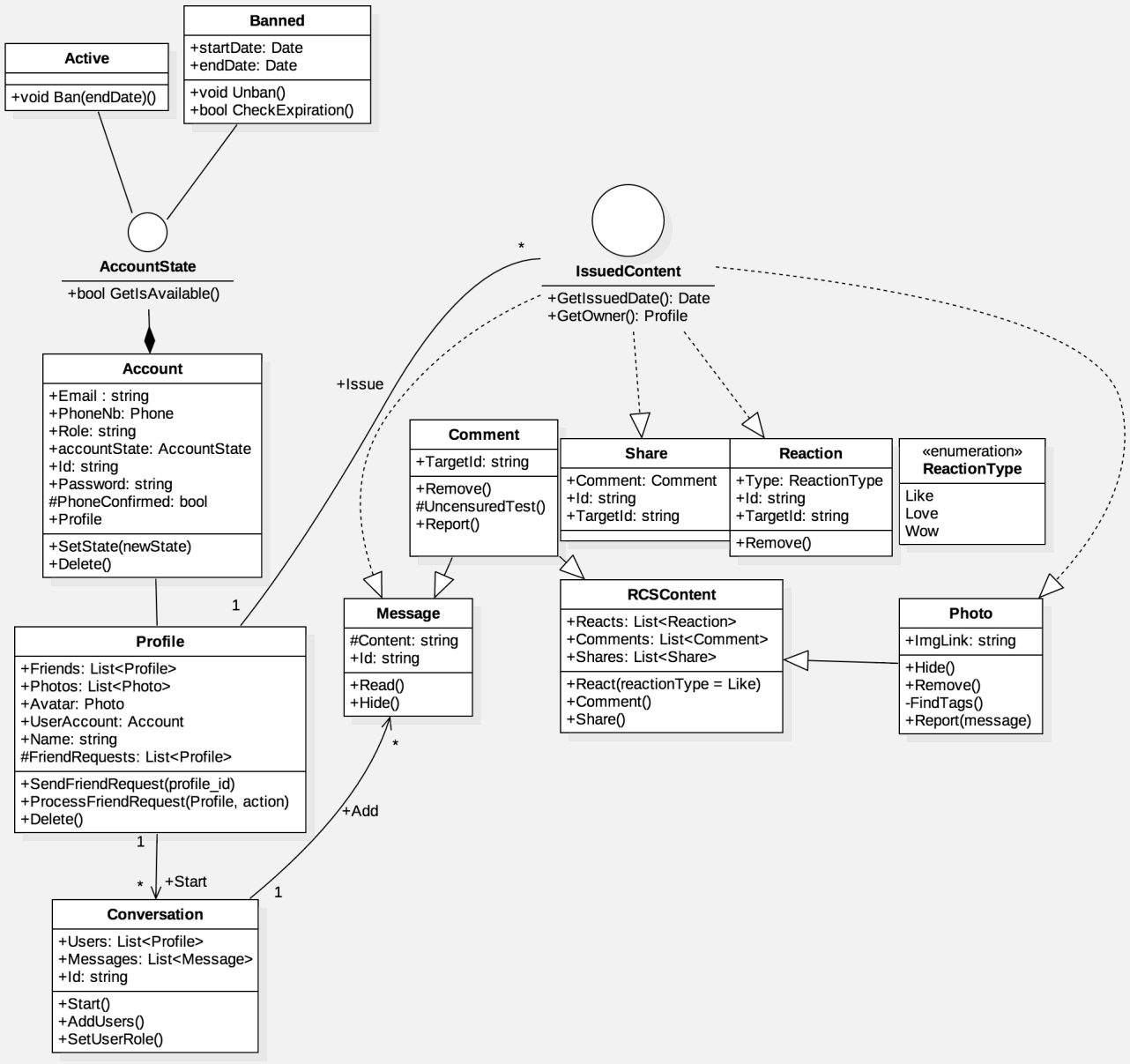
The **Account** class contains all the user's identity. Some of its fields may be set to public in the profile settings. The boolean which indicates if the phone is confirmed is used when the user tries to LogIn with his phone number. The Role is used to diffirenciate between a simple or a special account, like an Admin. Moving on to *Account state*: this represents an interface which shows if the account is available. For example, when some people try to find a user A, before they can see him, **accountState.isAvailable()** is checked first.

AccountState is an interface which is realized by **Active** or **Banned** classes. When a user is banned by an admin or multiple reports, an expiration time is set. If he is banned undefinetly, the *endDate* is set to its maximum value.

IssuedContent is an interface assigned to anything that has an Owner and a creation time.

RCSContent is a base class for anything that can be Commented, Reacted or Shared.

The **Comment** class inherits from both *Message* and *RCSContent* and has a protected opperation which checks if the content is Uncensured. If there are problems, the comment is hidden until a special user can approve. It also has the option to be reported by anyone who sees the comment, which may lead to banning the owner's account. The *TargetId* is the Id of the content that the comment was issued to. Any content on this platform has a unique ID. As said before, this class inherits from **Message**. This class has a protected string which represents the content. If the message contains a link to a video, or an image, or it's an audio/video message, then *Read()* opperation will properly display it.



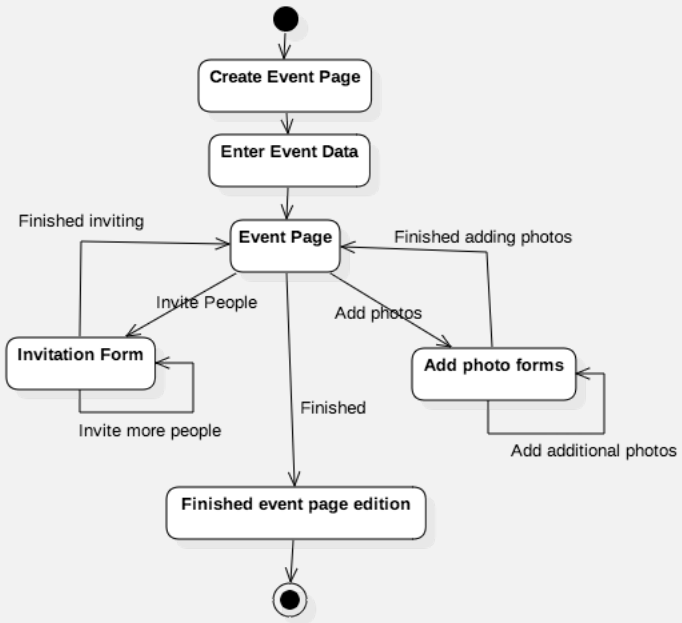
Img 10: **MiniFacebook** class diagram - user related

Statechart Diagram

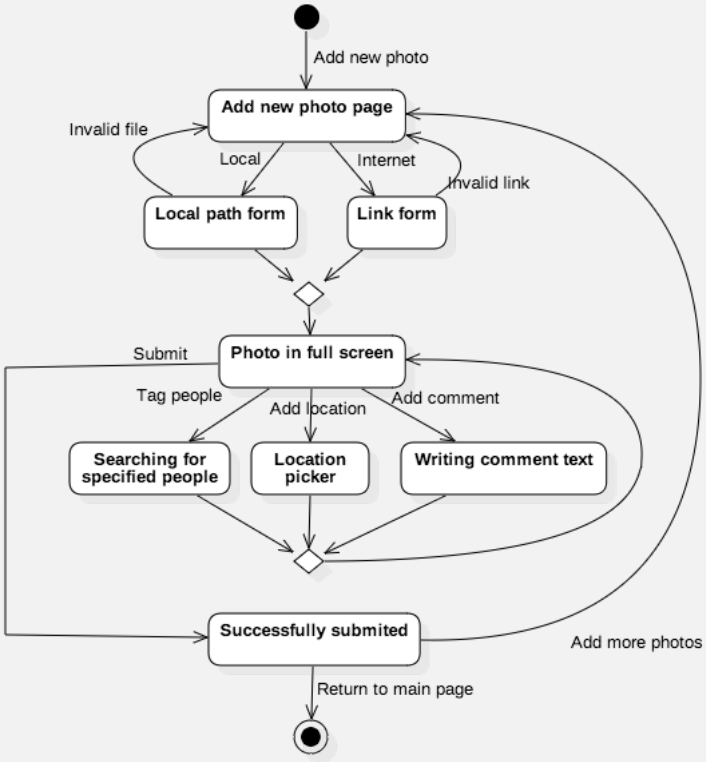
In *Figure 11* is represented the Statechart of creating a new event. First the user must enter the event details, then he is able invite people or add event photos. When he is done, the user is redirected to the event's View.

Figure 12 is about adding a new photo. The user is able to load a local image or indicate a link. If the file or the link is invalid, the user is returned to the initial state. When successfull, the loaded photo is shown in full screen. From here, it's possible to tag people, add a location or a comment. After submit is pressed, a corresponding message is displayed. In this state, it's possible to add more photos.

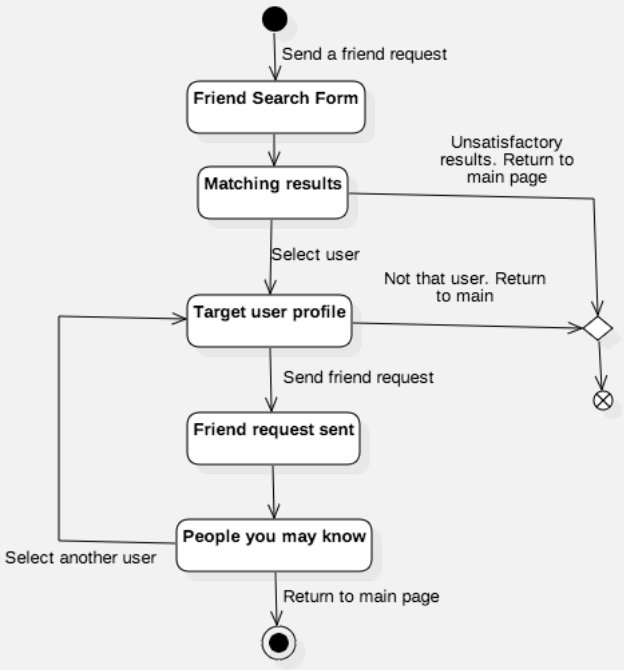
In *Figure 13* is shown the statechart of a friend request. First, the user tries to find some people in the search forms. If nothing satisfactory is found, he can cancel the action. Then profile is selected. It's also possible to cancel the action from this state. Here, a friend request button is available, if pressed, a message is displayed and the user is redirectioned to a page with people he may know. He can select other users or finish the action.



Img 11: Create an event



Img 12: Add new photo



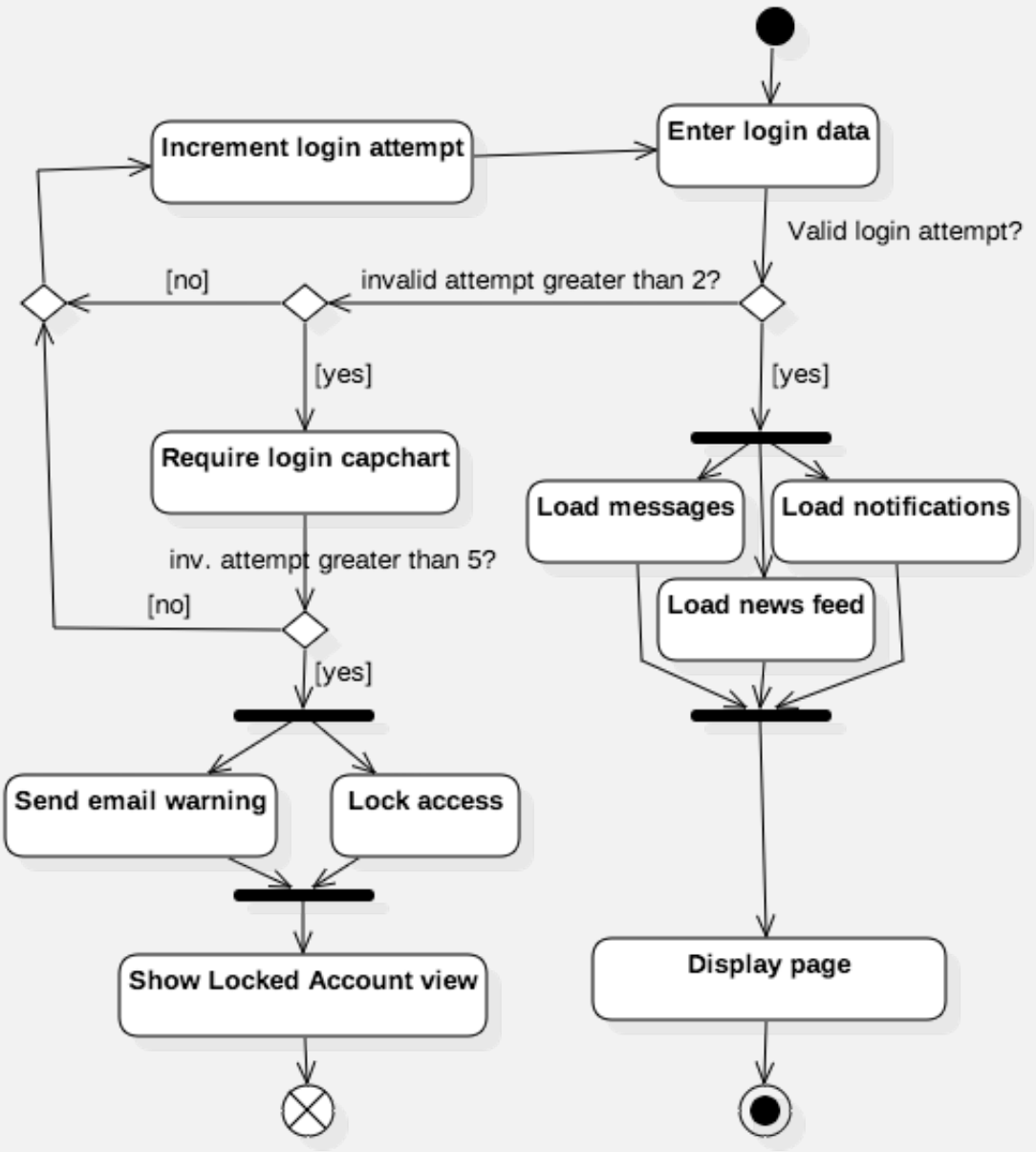
Img 13: Friend request

Activity Diagram

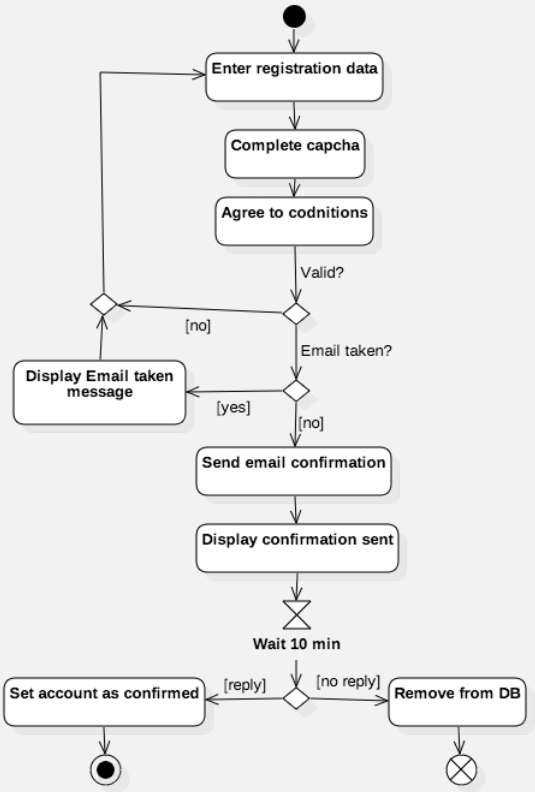
Figure 14 represents the Activity diagram of a user trying to login into the platform. If he fails 2 times, a capcha must be completed for additional attempts. If he fails 5 times, the access is locked for futher tries and an Email warning is sent to the account's user and the action is terminated. Otherwise, messages, notifications and other necessary data is loaded to display the user's Home page.

In *Figure 15* is shown the activity of registration. After successfully submiting the registration details, an email confirmation is sent. If in the following 10 minutes the user doesn't validate his account, the registration data is removed from the Database. Otherwise, the account is set as active.

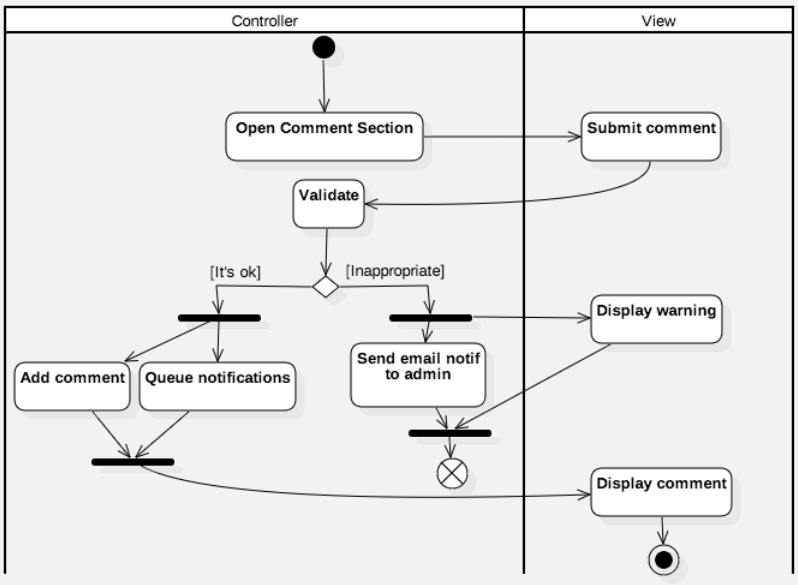
In *Figure 16* I tried to separate what activities are done in the controller and view. When the comment is submited, it is validated in the controller. If it is found to be inappropriate, then an email notification is sent to an admin and a warning is displayed in View. Otherwise, the comment is registred in the DB and notifications are sent to users interested in this comment. After that, the comment is displayed.


Increment login attempt
Enter login data
Valid login attempt?
[no]
invalid attempt greater than 2?
[yes]
[yes]
Require login capchart
Load messages
Load notifications
inv. attempt greater than 5?
[no]
Load news feed
[yes]
Send email warning
Lock access
Show Locked Account view
Display page

Img 14: Login



Img 15: Registration



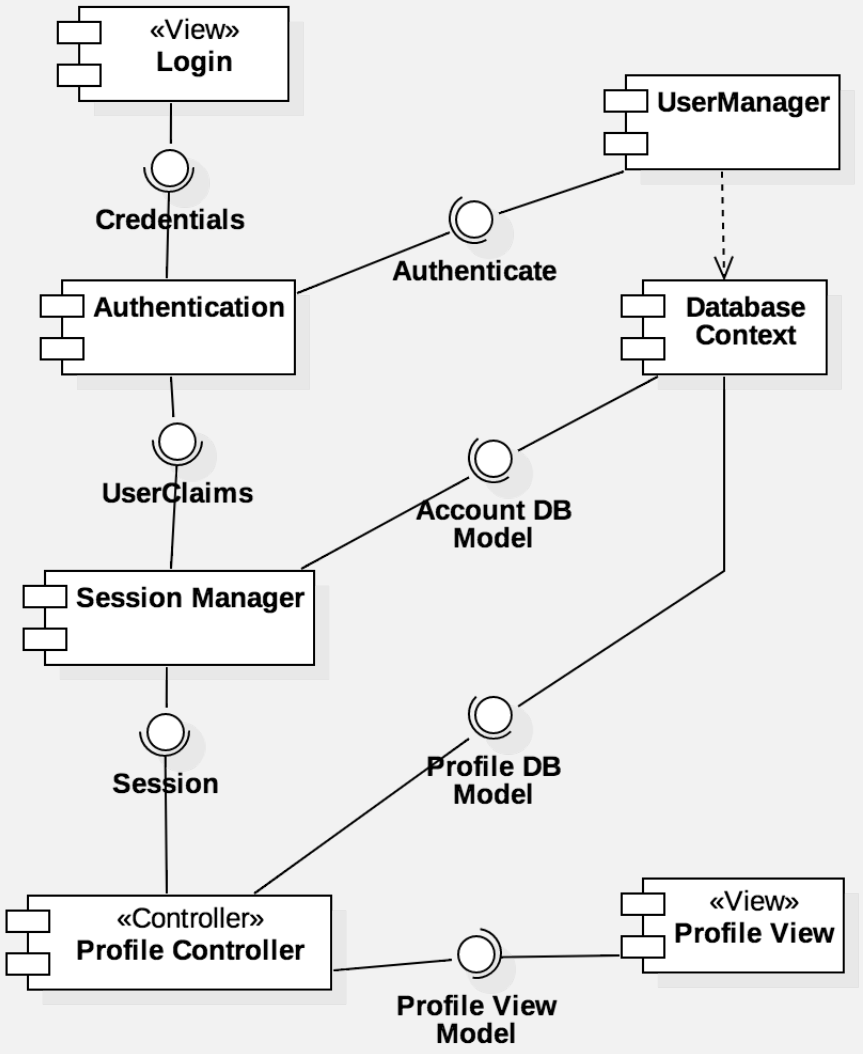
Img 16: Write a comment

Component Diagram

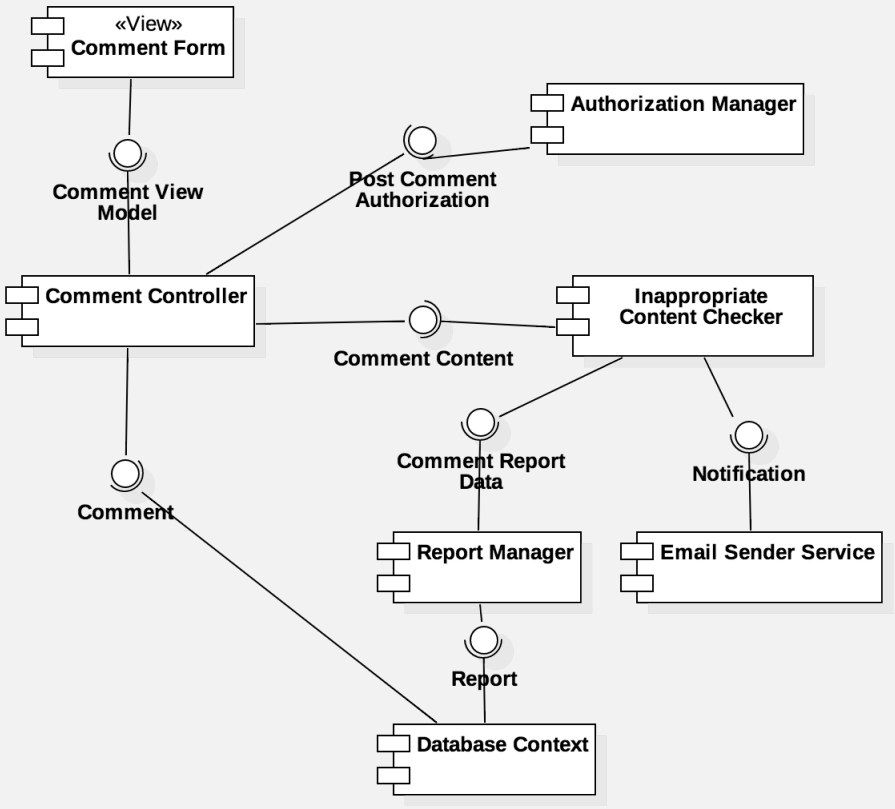
In *Figure 17* is represented the component diagram of Login. I tried to keep to the MVC patern. With that being said, the user interacts with the **Login** *View*, which sends the *Credentials* to the *Authentication*. It then communicates with the *UserManager* to authenticate and then form the user's *Claims*. A *Session* is formed and it is used for further user identification. Finally, the *Profile View* is fed with a *Profile View Model*.

Figure 18 represents the components of commenting. It interacts with the *Authorization Manager* to check if the user is able for such an action. To check if the content does not contain anything inappropriate, the contents are sent to a checker. If it is found to have problems, an email notification can be sent to an admin trough the *Email Service* and a *Report* can be registered in the Database.

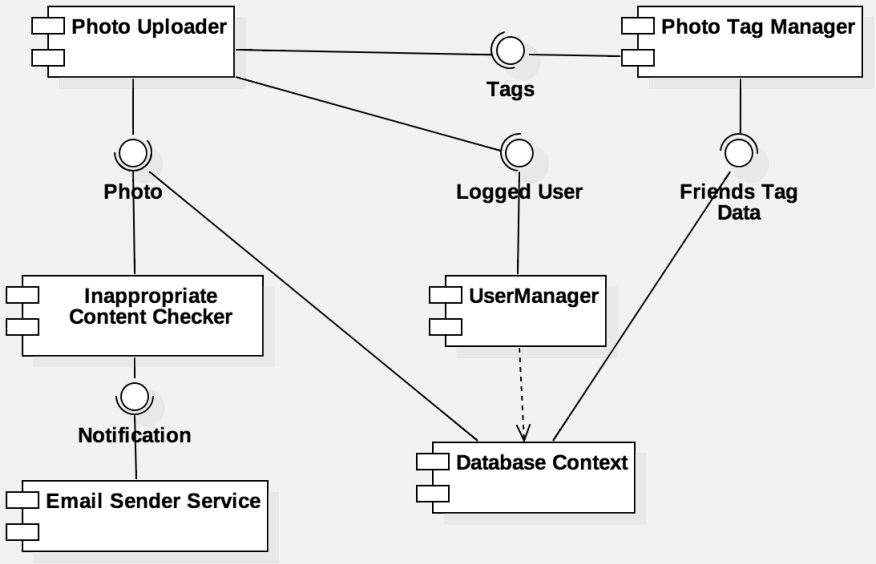
In *Figure 19* is shown the components needed for a Photo Uploader. First, it gets the Logged User to link the user ID to the photo. Then, it interacts with the *Tag Manager* to find possible tags on the uploaded image. It must also check for Inappropriate content which is done in the *Inappropriate Content Manage*. Again, if a problem is found, an email notification can be sent.



Img 17: Login Component Diagra



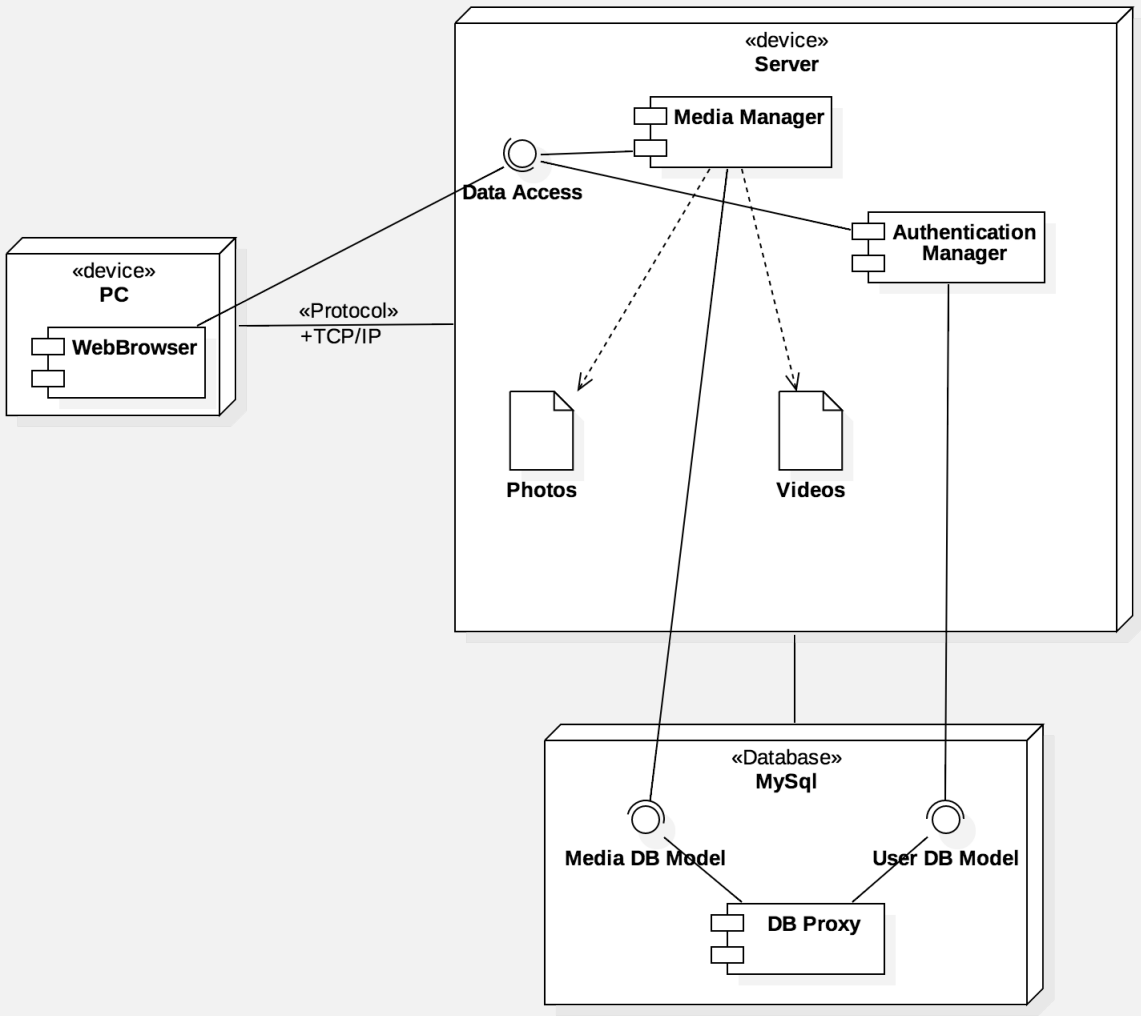
Img 18: Login Component Diagra



Img 19: Upload Photo Component Diagra

Deployment Diagram

Figure 20 represents a general Deployment Diagram conerning the *Media Manager*. The *Photos* and the *Videos* are saved on the server and only the paths are stored in the Database. The Database has Proxy which comunicates with our program. The client accesses the web app through a Web Browser. The Browser then communicates through a *Data Access* "interface", which represents the actual requests with its headers.



Img 20: Deployment Diagram

Conclusion

In this course I learned how to draw and read UML diagrams. UML language is a powerful tool wich helps a lot in designing a project. **Class diagrams** for example are extremly handy in developing a programmig library, because it offers a much clearer view on the structure. With that, it helps avoiding a lot of refactoring in the production stage.

Component diagrams, are very helpful to get a general idea about what parts of the system interact with each other, and what components can be reused. *Activity Diagrams* are very good to visualize the workflow of the application. *Use Case Diagram* is the best tool for brainstorming. In the early stage of the project, when the team gathers up to analyze the situation, Use Case Diagrams are usually used, because they are understood by non programmer people aswell. It also provides the main overview of what the app should be able to do. For programmers, it would be useful to have a Class Diagram on some complicated sections, to avoid missunderstanding.

Often, many non IT people are participating at the same project and often everyone comes with ideas of their own. This isn't really a bad thing in a small team, but when the project involves over 10+ people, it definetly requires to take the time and make some diagrams, so that everyone is working on **one** idea. Sometimes, that time can save a lot more time and money in futue. There are many examples of projects that got bankrupt, simply because everyone was working on different ideas, or because they lost a lot of time solving Merge Conflicts.

Speaking of merge conflicts: in this course I haven't learned only about UML diagrams, but I got a grasp on some Software Paterns, like SOLID. When working in a team, there is always this problem of dividing tasks. An example of a really bad practice would be to write the entire project in one single file and have a team of 5 developers work on it. *Divide and conquer* would suggest to divide the functionalities in different components. If the *Single Responsibility* principle were applied, the functionality would be divided even more, creating room for teamwork. With the *Interface Segregation* principle, there would be more client specific interfaces than one general-purpose interface. Applying these principles it helps avoiding a lot of bugs, "conflicts" and refactoring, which makes it really time worthy to make the project SOLID.

This course was very useful for me, because I have learned and applied many extremely usefull tools for working on scaled projects, which require long time maintenance. Even I started applying the *Class Diagram* on my personal project and now it's way much easier to see what needs improvement and more work.