System design document for Good Deeds

Lo, Charles, Richard, Saga, Anton

2019-10-25

1 Introduction

The Good Deeds application aims to target the need for a service or a need to be a good Samaritan without any monetary transaction. The basic functionality is to be able to put out an ad-like post if you want to get help with something or if you want to help your fellow human beings. This should be shown to users in a way that is easy to navigate so the users can filter and find exactly what service they need.

The stakeholders in this application are anyone who needs help with something or wants to help others and want to do it in a more old fashioned way without any monetary transaction.

This document tries to explain the application architecture, design, and overall quality. It is aimed to give a high-level understanding of the general structure and in some cases a detailed description of important parts.

1.1 Definitions, acronyms, and abbreviations

User - A person using the application. The user can be either a client or a volunteer, depending on the user's intentions. The user is not locked to be a client or a volunteer and will change the role if the intention changes.

Volunteer - A user that for this moment want to help others.

Client - A user that for this moment want to get help from others.

Receiving account - The client of the deed.

Giving account - The volunteer of the deed.

Deed - A listing made by users describing a service. As of right now, a Deed consists of a subject and a description. A deed can be either an Offer or a Request.

Offer - A Deed where a service is offered by a user. The user who created this offer will be registered as the volunteer by being the giving account. Other users can contact the volunteer and request to be provided the service.

Request - A Deed where a service is asked for by a user. The user who created this request will be registered as the client by being the receiving account. Other users can contact the client and offer to help with the service.

Claim deed - A user can claim a deed. If the deed is an offer, the user claiming the deed will be the client, by being registered as the receiving account. If the deed is a request, the user claiming the deed will be the volunteer, by being registered as the giving account. The deed will then be unavailable for others to claim.

Karma points - A user will use karma points as currency. When a person creates a new account, an initial amount will be assigned to the new user.

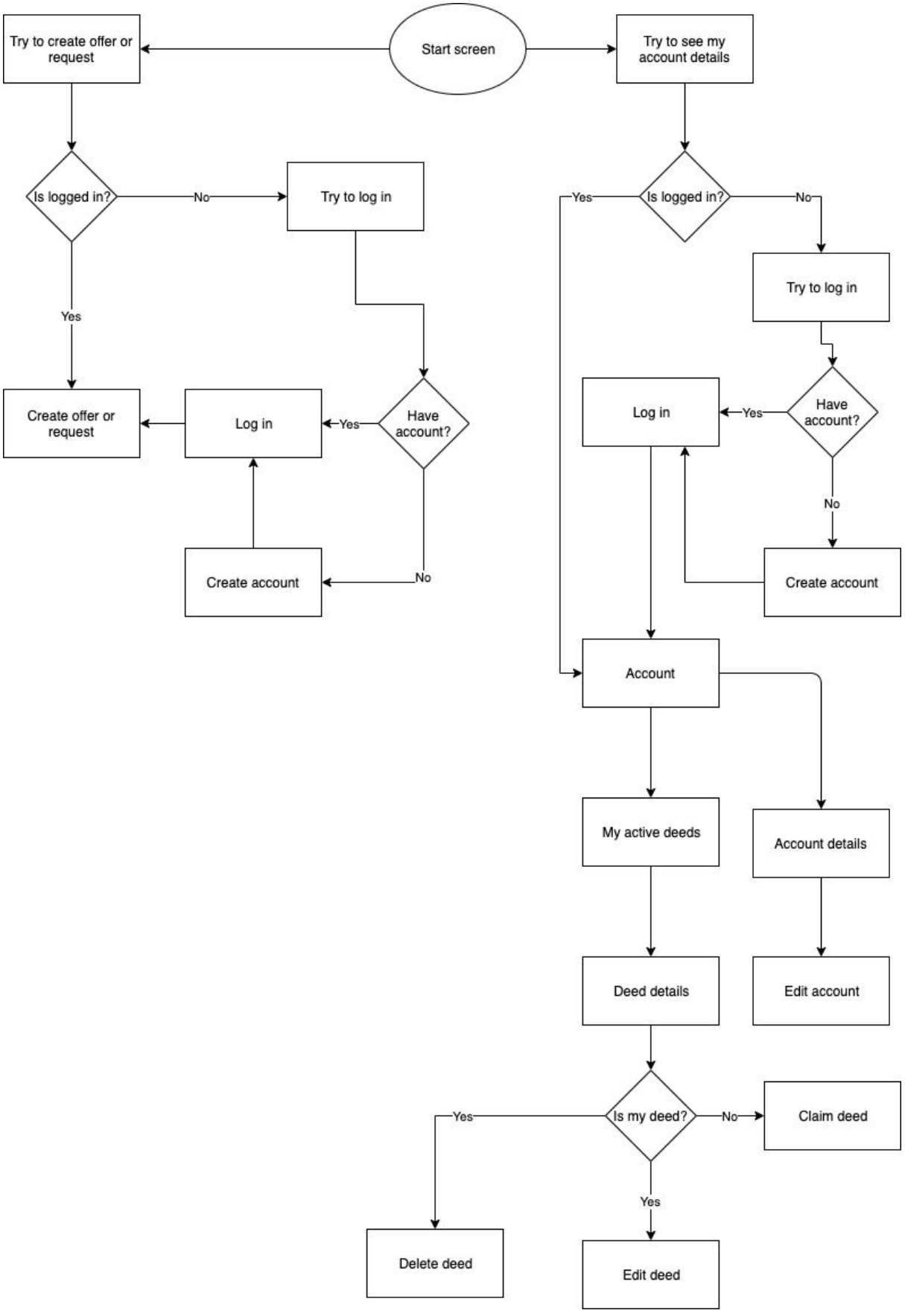
Done deed - A deed is done when a claimed deed is marked as done. Karma points will then be withdrawn from the client's account and deposited into the volunteer's account.

Marketplace - A page where users can see all the created deeds.

2 System architecture

The system is, at the moment, only an android application without any data storage or servers needed. In this iteration, we store all data in the model and it's not saved after the application closes. The aim is to create a server running a RESTful API with a relational database as storage. The means of communication with the API should be with JSON.

First when the application starts data is pulled from the data storage in order to show active deeds. See flow diagram below.



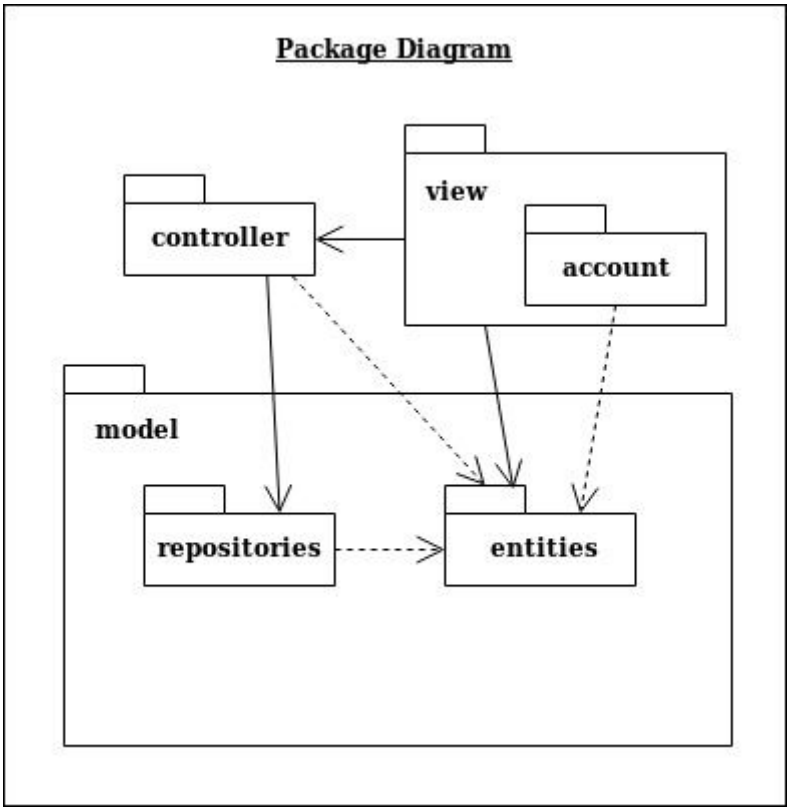
To further explain the above diagram of how a user can interact with the application, please see below for how the application handles the requests:

1.	Start screen	The application starts up and shows the default start view
2.	Press create offer/request button	The application checks whether the user is logged in already by asking the persistence layer if the user is logged in. If the user is logged in it gets sent to either the login view or create an offer/request.
3.	User is not logged in	The user is prompted to either login or create a new account. If the user tries to log in, the login details get validated with a service in the model. If the login is successful the account is stored as logged in.
4.	Tries to create account	The user fills in values in the forms which get validated with f.ex a postal code must consist of five numbers or an email address must have an “@” and a “.” in it. If the fields get validated a create account process is started which tells the controller to handle the account creation. The controller communicates with the repository and the repository communicates with our aggregate class that contains all our data.
5.	Create offer/request	When the user is logged in, they can create an offer or request by filling out the specific form. This form has minor validations with f.ex the fields is not allowed to be empty. If the fields get validated the view tells the controller to handle a create offer or request. The controller then tells our repository to create an offer/request which then contacts the aggregate class which handles the creation and data storage.

That is the flow when you login, create an account, create an offer/request. In a more general sense the flow within the code is:
A user interacts with a view that calls a controller method that handles the tasks at hand. It could be to get a deed to the view or more complex tasks as creating an Account. The controller usually uses the repository that then gets the data from GoodDeeds-aggregate class.

3 System design

3.1 Package Diagram



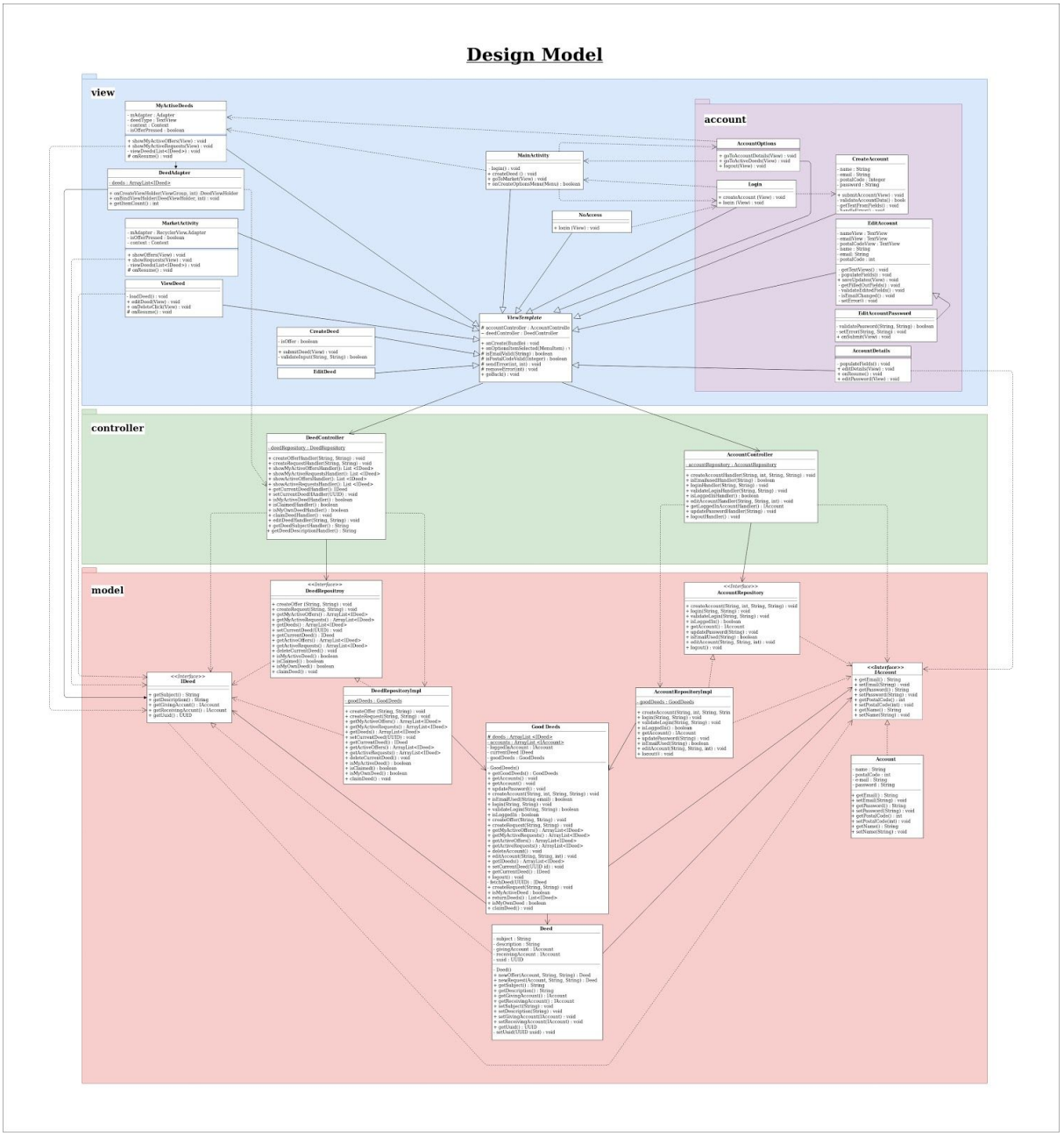
3.2 MVC

The Model is completely independent from the controller & view packages. The Model uses interfaces as abstractions in order to limit unnecessary exposure to the Controller and the View. This follows the dependency inversion principle with the interfaces as abstraction of the concrete implementation in the Model. It also follows the open-closed principle by ensuring that the Model exposes a distinct interface to depend on and it can still change while preserving the interface offered.

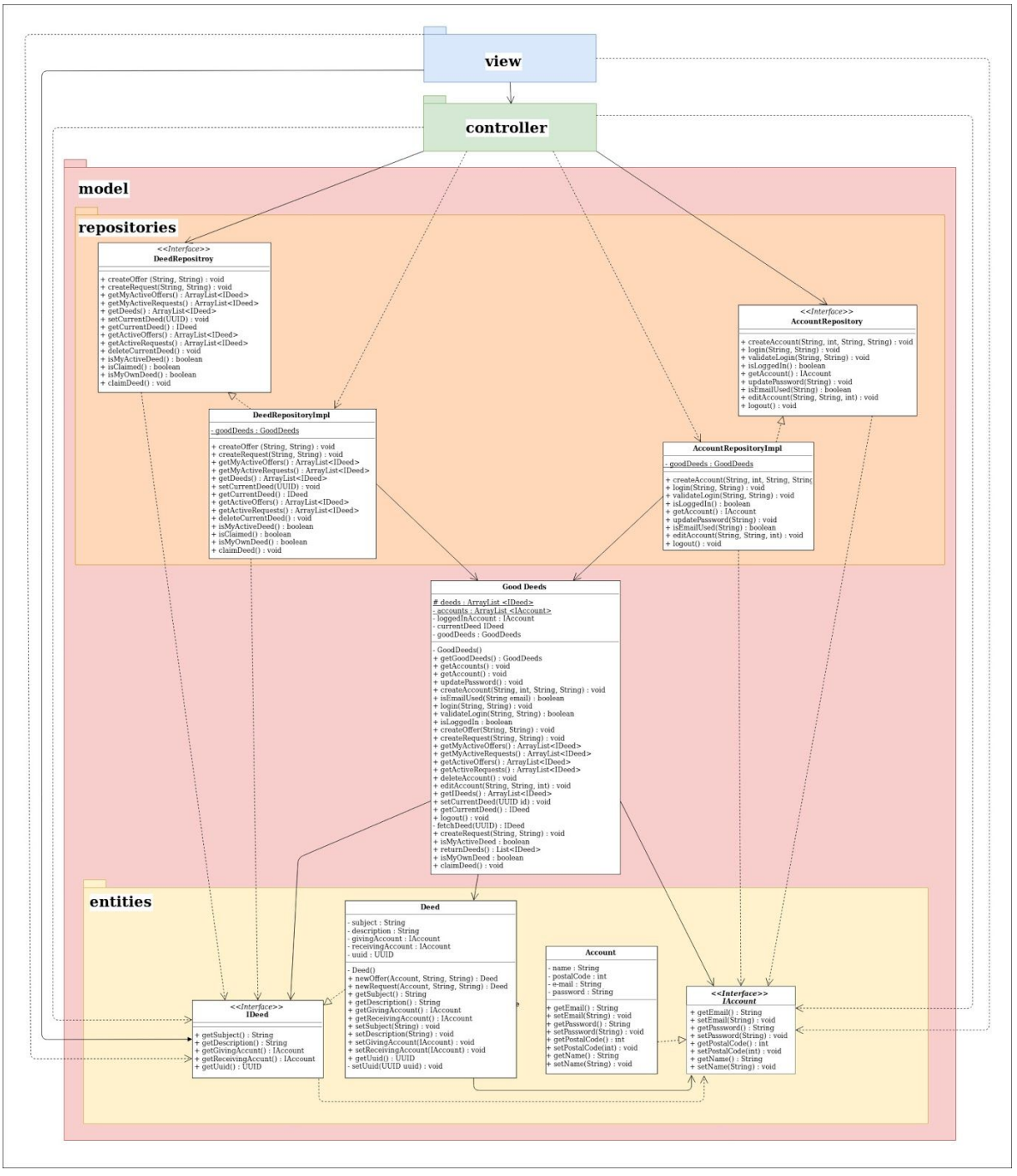
The View is responsible for managing the activities (used by the android application to interact with the user) of the application. It has a dependency on the controller which it uses to pass on requests to and get information from the Model. The View also has a dependency on the interfaces in the Entities package.

The Controller has a dependency on the Model, though composition, which uses interfaces from the Repositories package to be able to pass on request and information. It also uses instances of interfaces from the Entities package.

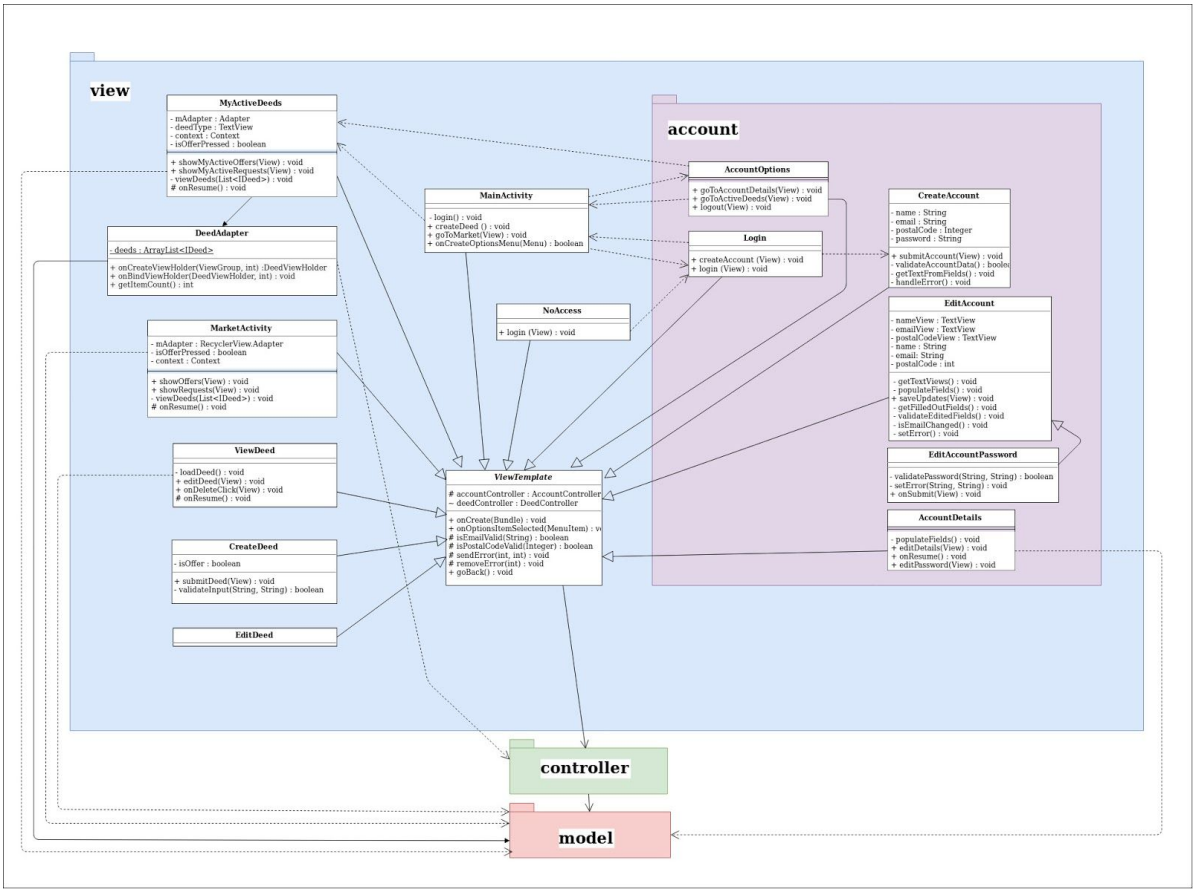
3.3 Design Model



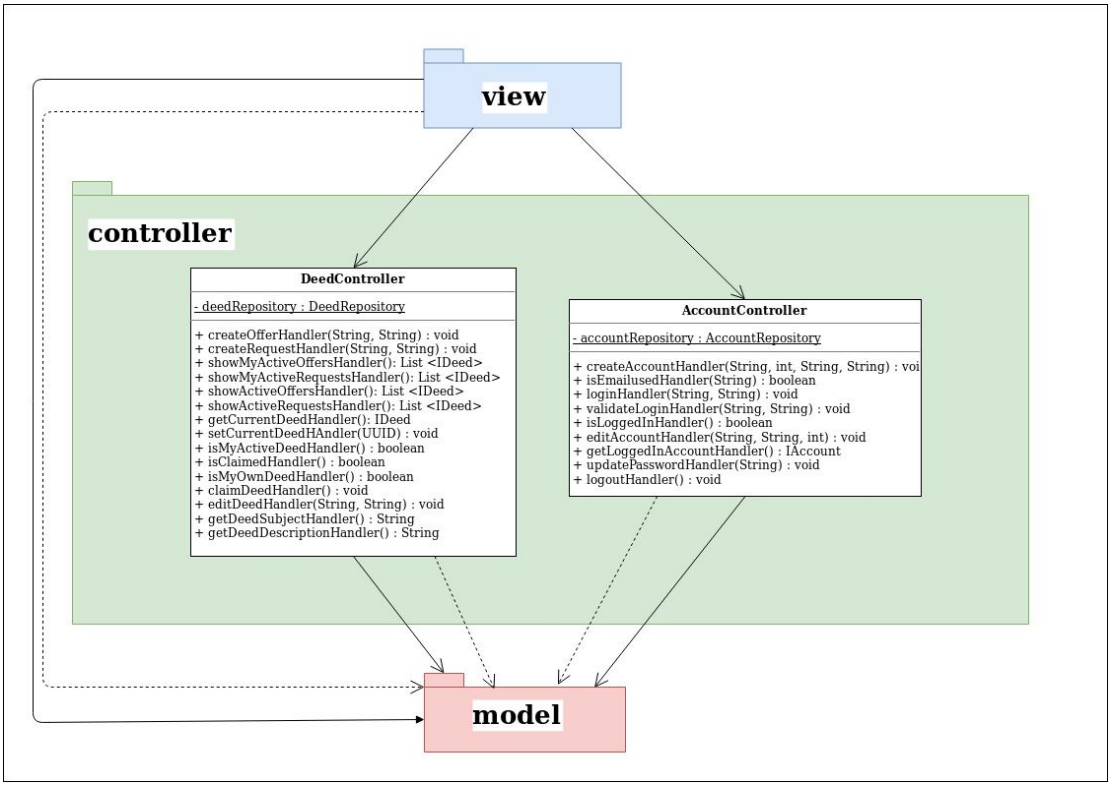
Model


view
controller
model
repositories
<<Interface>>
DeedRepositroy
+ createOffer (String, String) : void
+ createRequest(String, String) : void
+ getMyActiveOffers() : ArrayList<IDeed>
+ getMyActiveRequests() : ArrayList<IDeed>
+ getDeeds() : ArrayList<IDeed>
+ setCurrentDeed(UUID) : void
+ getCurrentDeed() : IDeed
+ getActiveOffers() : ArrayList<IDeed>
+ getActiveRequests() : ArrayList<IDeed>
+ deleteCurrentDeed() : void
+ isMyActiveDeed() : boolean
+ isClaimed() : boolean
+ isMyOwnDeed() : boolean
+ claimDeed() : void
DeedRepositoryImpl
- goodDeeds : GoodDeeds
+ createOffer (String, String) : void
+ createRequest(String, String) : void
+ getMyActiveOffers() : ArrayList<IDeed>
+ getMyActiveRequests() : ArrayList<IDeed>
+ getDeeds() : ArrayList<IDeed>
+ setCurrentDeed(UUID) : void
+ getCurrentDeed() : IDeed
+ getActiveOffers() : ArrayList<IDeed>
+ getActiveRequests() : ArrayList<IDeed>
+ deleteCurrentDeed() : void
+ isMyActiveDeed() : boolean
+ isClaimed() : boolean
+ isMyOwnDeed() : boolean
+ claimDeed() : void
<<Interface>>
AccountRepository
+ createAccount(String, int, String, String) : void
+ login(String, String) : void
+ validateLogin(String, String) : void
+ isLoggedIn() : boolean
+ getAccount() : IAccount
+ updatePassword(String) : void
+ isEmailUsed(String) : boolean
+ editAccount(String, String, int) : void
+ logout() : void
AccountRepositoryImpl
- goodDeeds : GoodDeeds
+ createAccount(String, int, String, String
+ login(String, String) : void
+ validateLogin(String, String) : void
+ isLoggedIn() : boolean
+ getAccount() : IAccount
+ updatePassword(String) : void
+ isEmailUsed(String) : boolean
+ editAccount(String, String, int) : void
+ logout() : void
Good Deeds
deeds : ArrayList <IDeed>
- accounts : ArrayList <IAccount>
- loggedInAccount : IAccount
- currentDeed IDeed
- goodDeeds : GoodDeeds
- GoodDeeds()
+ getGoodDeeds() : GoodDeeds
+ getAccounts() : void
+ getAccount() : void
+ updatePassword() : void
+ createAccount(String, int, String, String) : void
+ isEmailUsed(String email) : boolean
+ login(String, String) : void
+ validateLogin(String, String) : boolean
+ isLoggedIn : boolean
+ createOffer(String, String) : void
+ createRequest(String, String) : void
+ getMyActiveOffers() : ArrayList<IDeed>
+ getMyActiveRequests() : ArrayList<IDeed>
+ getActiveOffers() : ArrayList<IDeed>
+ getActiveRequests() : ArrayList<IDeed>
+ deleteAccount() : void
+ editAccount(String, String, int) : void
+ getIDeeds() : ArrayList<IDeed>
+ setCurrentDeed(UUID id) : void
+ getCurrentDeed() : IDeed
+ logout() : void
- fetchDeed(UUID) : IDeed
+ createRequest(String, String) : void
+ isMyActiveDeed : boolean
+ returnDeeds() : List<IDeed>
+ isMyOwnDeed : boolean
+ claimDeed() : void
entities
<<Interface>>
IDeed
+ getSubject() : String
+ getDescription() : String
+ getGivingAccunt() : IAccount
+ getReceivingAccunt() : IAccount
+ getUuid() : UUID
Deed
- subject : String
- description : String
- givingAccount : IAccount
- receivingAccount : IAccount
- uuid : UUID
- Deed()
+ newOffer(Account, String, String) : Deed
+ newRequest(Account, String, String) : Deed
+ getSubject() : String
+ getDescription() : String
+ getGivingAccount() : IAccount
+ getReceivingAccount() : IAccount
+ setSubject(String) : void
+ setDescription(String) : void
+ setGivingAccount(IAccount) : void
+ setReceivingAccount(IAccount) : void
+ getUuid() : UUID
- setUuid(UUID uuid) : void
Account
- name : String
- postalCode : int
- e-mail : String
- password : String
+ getEmail() : String
+ setEmail(String) : void
+ getPassword() : String
+ setPassword(String) : void
+ getPostalCode() : int
+ setPostalCode(int) : void
+ getName() : String
+ setName(String) : void
<<Interface>>
IAccount
+ getEmail() : String
+ setEmail(String) : void
+ getPassword() : String
+ setPassword(String) : void
+ getPostalCode() : int
+ setPostalCode(int) : void
+ getName() : String
+ setName(String) : void

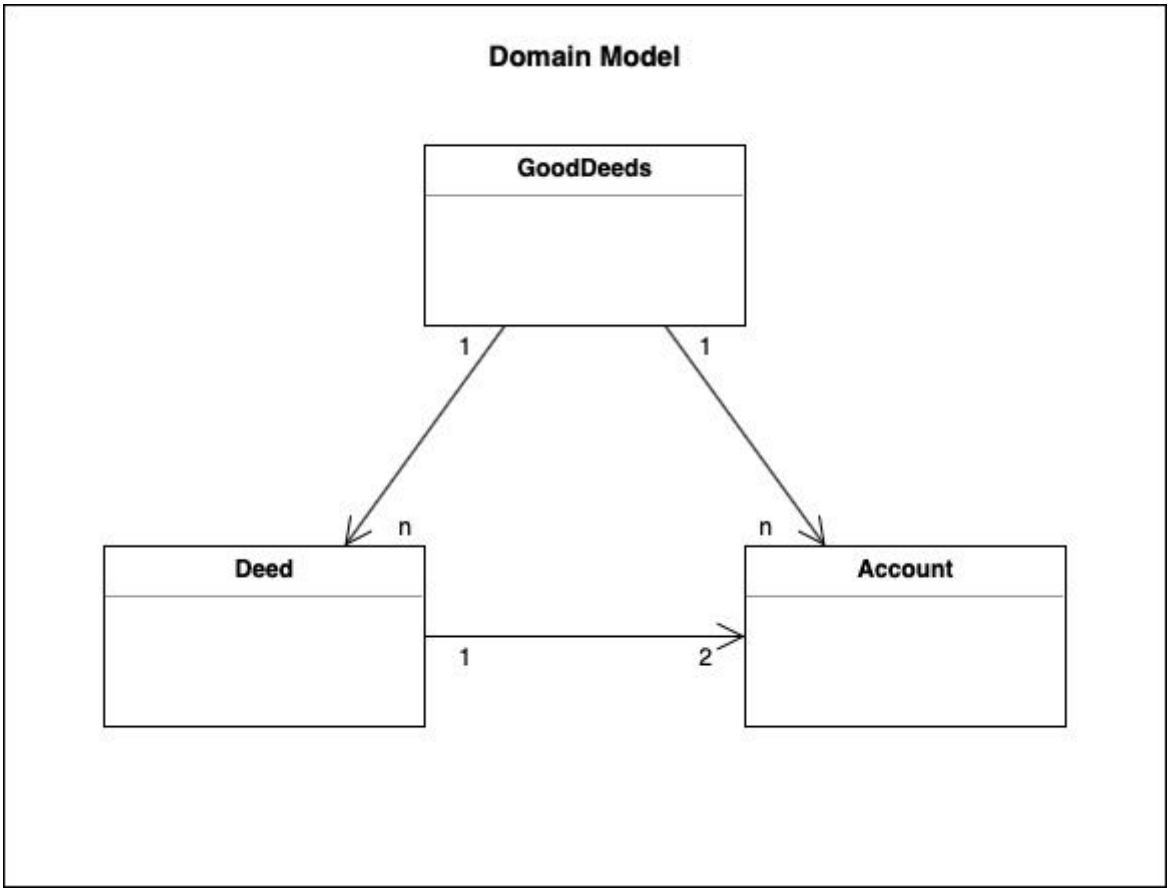
View



Controller



3.4 Domain model



These classes represent the core of our model.

3.5 Design Patterns

Singleton Pattern:
Data is currently stored in GoodDeeds and therefore this class is limited to only one object, ensuring that the data is handled correctly. With another storing system in place, this would not be an issue and the singleton pattern would not be used.

Module Pattern:
We grouped classes with the same responsibilities in separate modules, following the separation of concern principle and encapsulating the functionality. The high cohesion low coupling principle is followed by the coupling between modules being as few as possible and as weak as possible. The dependencies we have are regulated by using distinct interfaces. A strong connection is used by the components in the modules themselves to solve the tasks.

Pattern we wanted to use:
Repository Pattern:
Interfaces and classes exist but do not contain any additional logic. They were supposed to provide an abstraction of the data stored. Other modules would use these interfaces to access the data.

4 Persistent data management

We never implemented a database or any other way of saving the data. Instead, we just have our aggregate class of GoodDeeds to save information. But it will disappear when closing the application. Persistent data management is not available in our system as it looks right now.

In the future, our “Repository classes” would be responsible for handling data exchange, possibly between the application and a database.

5 Quality

All tests are in the com.goodpeople.goodDeeds (test) folder. We strive to have 100% test coverage and test all methods. Preferably at the lowest level and up. We test everything from entities up to the controller. Since the view is close to impossible to test, at least not with a satisfactory result, we test it manually. When a feature is done it’s first tested by the developer and then by two other developers in the code review.

Some known issues are :

One issue is that there are no tests for private methods. We didn’t find any reasonable solution for this.

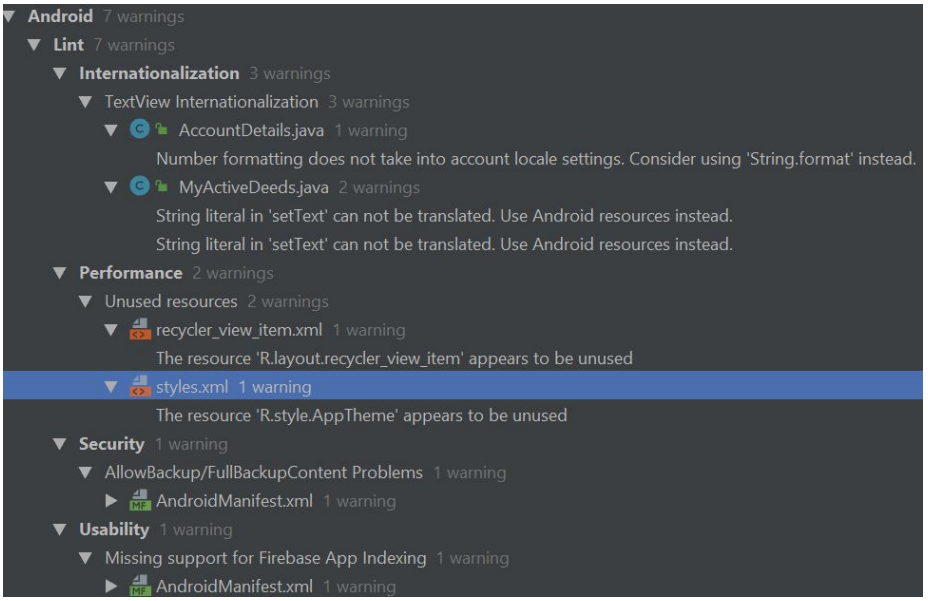
The implementation of repository is not fully in operation yet, it should have the persistence layer, in other words, it should be in contact with a database of some sort. At the moment all data is passed on to GoodDeeds. It is this way because currently all data is only held by GoodDeeds.

There exists no factory in our application. For example, we should/could implement a factory for Deeds and Accounts. This would add an abstraction level to it. It would also encapsulate how the objects are created.

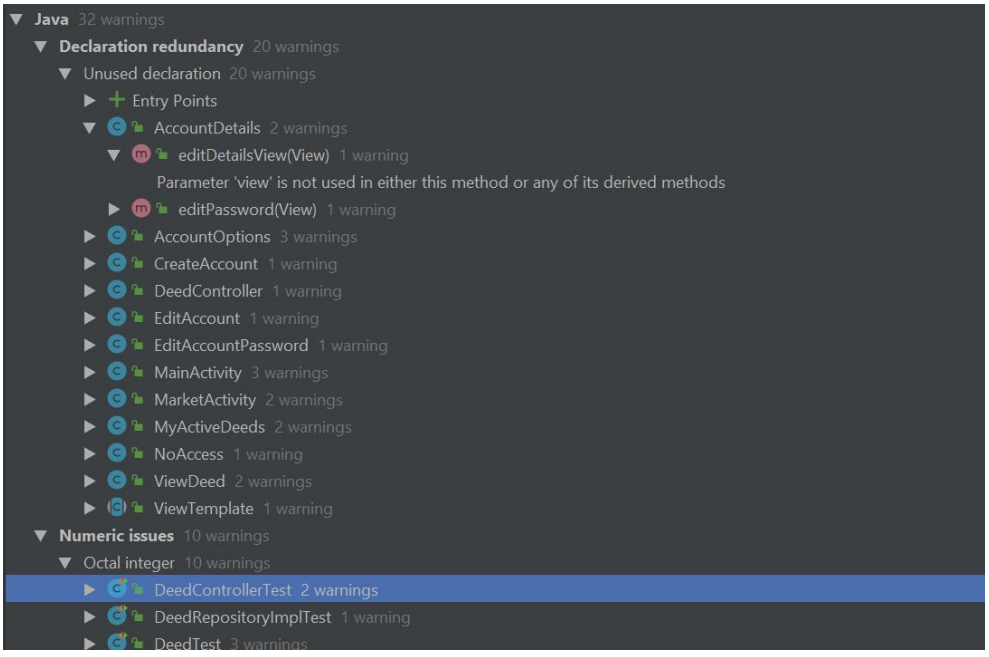
All sessions are only local. The changes someone does locally does not affect anybody else. The application needs to be online so people can interact with each other’s deeds. This is something that needs to be implemented further down the road.

The email addresses put into the email field aren’t fully validated, for example, you can put in the email “@.”.

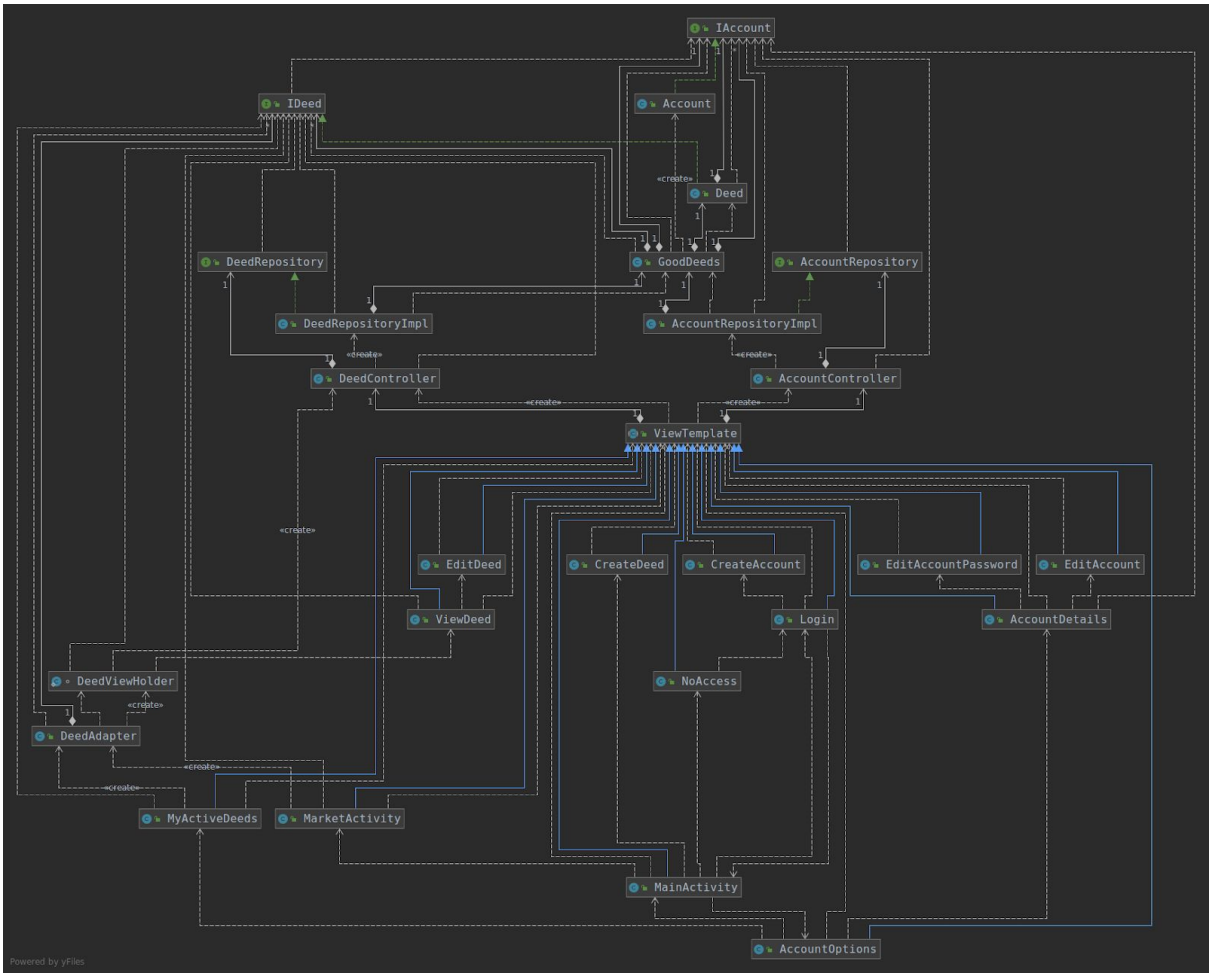
Analytic tools:
We have corrected the warnings that were sensible, but some warnings remain unfixed.
By running code inspection available by intellij, we got the following:



These issues were not rational. For example removing the R.Layout.recycler_view_item would result in errors.



The warnings about “unused declaration” are not valid. If you were to remove the parameter view, then the views wouldn’t work properly.
Also the warnings about “Numeric issues” are not reasonable because in the project we are saving “postal codes” as a five-digit number into an int.



The picture above is the UML generated by intellij Ultimate edition. There are some dependencies direct from view to IDeed for example. This is because we want to expose an abstraction of objects, but still encapsulate more sensitive methods.

We also use Travis CI which is a continuous integration tool, which runs the tests in the code that a developer commits to Github. This further enhances the quality of the code, because it gives a notice if something was broken, and where it needs to be fixed. When a pull-request is asked for, travis will give a “green checkmark” if it passed the tests, and a red warning if it failed the tests. This gives a good starting point for doing a review.
Here is a link to our travis https://travis-ci.com/antjanne/oopGoodPeople.

5.1 Access control and security

We only have one kind of login for the program, which is for a user. Further down the road, we had a plan to implement an admin type of user, that would be able to delete deeds that weren’t appropriate, and block users that used the app for purposes other than being a good citizen.

The users passwords are implemented as hashcode so that they aren’t saved as strings. As for now, it doesn’t matter much since we didn’t implement any persistent data management,

but it’s set up to be safer stored when we do.

6 References

- JSON - data interchange format http://json.org/
- RESTful API - architectural style for distributed hypermedia systems https://restfulapi.net/
- Travis - a continuous integration tool used to test software projects at Github https://travis-ci.org/
- Github - a hosting service for software development version control using git https://github.com/