System design document for ICE

Version: 0.1

Date: 2016-05-17

Author: Robert, Axel, Simon, Niclas, Jonathan

This version overrides all previous versions.

1 Introduction

1.1 Design goals

The application should consist of three distinct parts. A client, which in this apps case is a smartphone, a webserver that serves request from the client on the domain, and the domain. The domain should consist of one part that makes sure all constraints on the data are withheld, and one part that offers persistency. i.e a SQL server or in our case serializable repository that is saved and read from the local disk. This part that offers persistency should be very easy to switch out for a proper SQL database.

All dependencies will be handled through the build tool Gradle. See appendix for more information.

1.2 Definitions, acronyms and abbreviations

- client, a smartphone running the android operating system.
- user, a person linked to a client.
- java, platform independant programming language.
- GUI, graphical user interface.
- server, webserver that servers requests from clients.
- swipe, a client has made a decision and has clicked either yes or no on another user.
- spring, framework for writing easy to implement-webservers.
- view, a XML document defining the layout of an android activity.
- activity, something that loads a view on the android, enabling a user to click on buttons and use the app properly.
- Generated code, a gcode, is a code generated when creating a course, which uniquely identifies the course.
- matchrequest, a request sent from **one** user to another.
- match, a match is made when two users have both agreed to work with eachother.
- RAD, Requirement analysis document.

2 System design

2.1 Overview

The project will consist of three parts. A webserver, a domain and a client. In our MVC mindset the domain is our model, the webserver is our controller and the client is our view. The client by itself also has a MVC design when handling its activities and requests on the server and so on.

2.1.2 Model functionality

The functionality offered by our domain is exposed through the interfaces ICourse and IUser. Instead of offering one big interface we've split it up to make two interfaces. One regarding the courses and one regarding the users.
The server offers methods with the same names, and one or a few other ones. These other methods use the methods offered by the interfaces, but makes another operation on them. For instance getNotMatchedWithMe first queries the domain for matchedwithme and then for allUsers, and then returns the difference.

2.1.3 Users

Users are clients who have chosen to sign up as a user, and not an administrator. A user Object stores a users email, name and password. A user can join courses and match with users enrolled in the same courses.

2.1.4 Admins

Admins are clients who have chosen to sign up as a admin. An admin can not join courses, but it can create courses. An admin can't match with users either. An admin could i.e be a teacher or a TA (teachers assistant).

2.1.5 Courses

Courses are represented by the Course Object. A course object keeps track of the course name, the courses generated code, and the admin of the course. It also keeps track of who is matched with who, who is enrolled at the course and who has sent a matchrequest to who.

2.1.6 Match requests and matches

A match request is a simple data-holding object that keeps track of who has agreed to work with who. There can at no time exist a MatchRequest where the sender in the first one is the same as the receiver in the second. When such an object is encountered both requests should be removed and turned into a Matched instead.
A match is another simple data holding object that keeps track of who has mutually agreed to match with eachother.

2.1.7 ILocalXRepo

The local repos are simple classes to offer persistency. All they do is store data on disk. No particular logic to mention. These implement their own interface, and this interface is what the Domain classes make calls upon. These are made to be easilly interchangable with a SQL database, for instance.

2.1.8 CourseDomain / UserDomain

Hold logic for what is and what is not allowed. If a user requests to join a course, then these domains will make sure the user is not an admin, that it is a registered user and so on. Once all constraints are ok the data is passed on to the LocalUserRepo/LocalCourseRepo to store the data persistently.

To clarify; the domains implement the domain of our application. They hold state, logic and constraints. The repositories **only** offer persistency, nothing else. Whenever the domains alter state they will pass on the new state to the repositories that will save the state.

2.2 Software decomposition

2.2.1 General

Package diagram. For each package an UML class diagram in appendix

2.2.2 Decomposition into subsystems

2.2.2.1 Domain

The domain is represented by the core-classes Person, Admin, User, Course, MatchRequest, Matched, Partner and PartnerRequest. These are all data-holding classes.

There is a package called util which only holds one class, GCode. This class generates a unique identified for each course object. Courses may have the same name, but not the same identifier (courses run over many periods etc).

Next is a package called repos. This package offers persistency, and only that. There is one repository for each of our domains (2).

The package domains implements both our domains, CourseDomain and UserDomain. These classes implements all the logic our domain should offer. When all the operations are done on any given data it sends the information through to our repositores which saves it.

interfaces is a package containing all the interfaces our domain expose. One each for the domains and one each for the repositories.

2.2.2.2 WebServer

Our main method is in the class Application.java. This class starts a spring webserver where we have restcalls to our domain.
The webserver is divided into three controllers. AdminController,UserController and CourseCOntroller, each offering restcalls to their respective domains.

The API these controllers expose are shown by swagger, a tool for creating interactive documentation.

2.2.2.3 Client

The Client is implemented as a Android Application. It consists of a View provided by Activities [1] toghether with Layout resources [2]. The Activites in this application acts as a Controller and in a way a Model controlling some simple logic that felt not necissary to implement new classes for. The communication between the WebServer API and the Client is done by a utility class that uses the Jackson-library [3] to parse JSON-data to Java classes.

The Client implements the functionallity described in the Use Cases. One can also login/register users/admins in the Client.

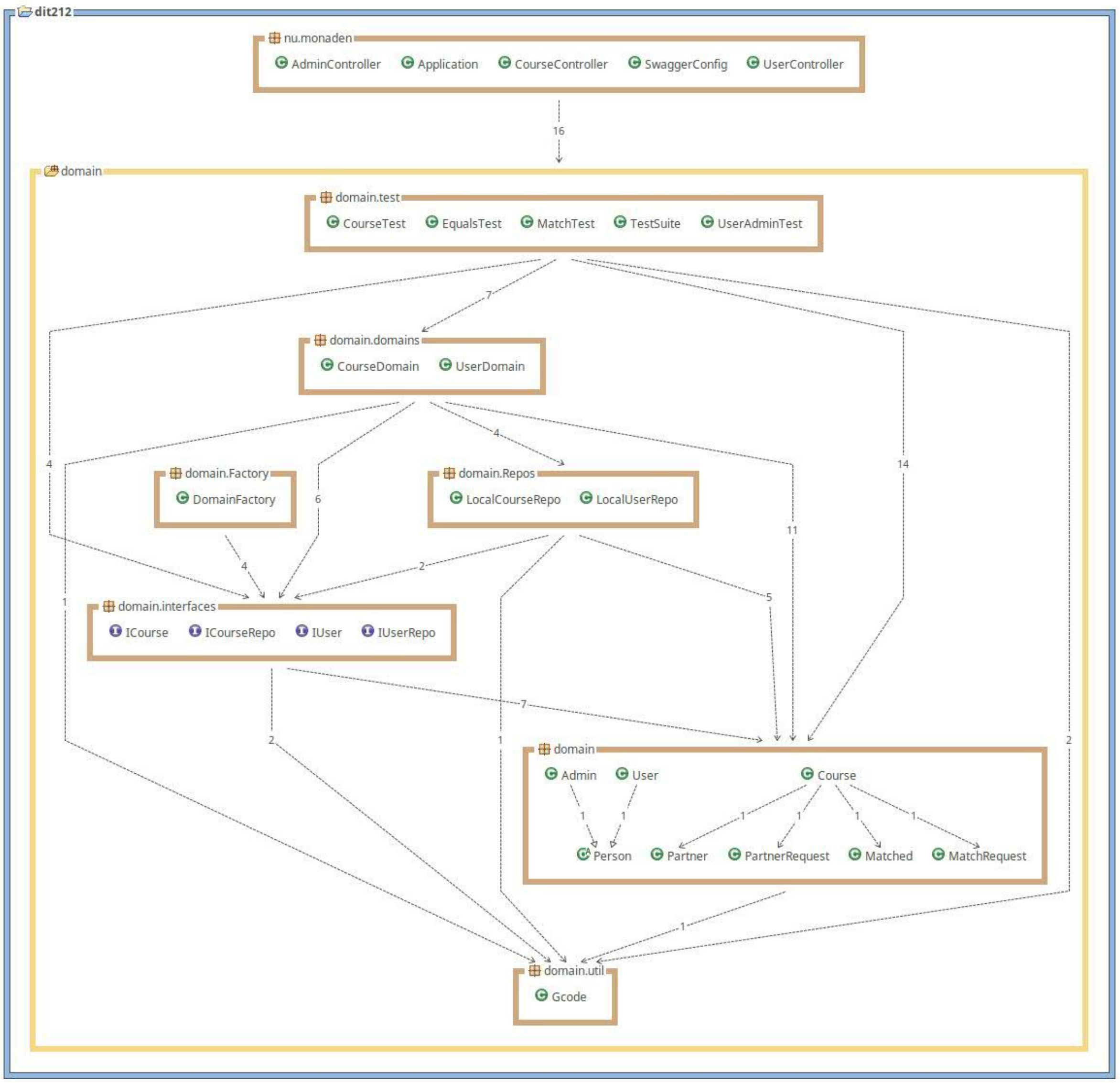
2.2.3 Layering

The user will see and interact with a client. The client is implemented as a android application. The client sends HTTP requests to a HTTP webserver. In our MVC (model view controller) design the client is our view. The client itself implements smaller MVC subsystems (activities), but in the big picture the client is the view.

The webserver is a HTTP server implemented with the spring framework. It maps requests to methods and passes on results back to the client. When the server gets a request it forwards the request to a method in the domain. The web server is thereby our controller.

The Domain is implemented as a standard java program, is uses both binary and text files for persistant data management however as metiond previously the method used for persistant data can easily be switched. The domain is the model in our MVC design.

2.2.4 Dependency analysis



As shown by this dependency analysis generated by Stan (see appendix) it is clear that there are no circular dependencies.
nu.monaden is the webserver, and it is dependant on the domain. The dependencies within the domain are all okay. We are also happy with the layering if our dependencies. The tests rely on the domains, the domains rely on the repositories who depend on the interfaces who lastly depend on the core classes Admin,User,Course and so on.

2.3 Concurrency issues

The domains are implemented using the singleton pattern, so there is plenty of room for race conditions. This will be left unresolved for this version as that is not something that’s specified in the RAD (Requirement Analysis Document).

2.4 Persistent data management

Persistency is offered by the repositories. They serialize the data and write them to binary files, and also read and populate our objects from the same files.
The domains handle all logic and whenever they are done they pass on their state to the repositories who save them.

2.5 Access control and security

All services the domain offer are exposed through the interfaces, and all other methods are private.

2.6 Boundary conditions

NA

3 References

1. Website, https://developer.android.com/reference/android/app/Activity.html
2. Website, https://developer.android.com/guide/topics/resources/layout-resource.html
3. Website, http://wiki.fasterxml.com/JacksonHome/

https://sv.wikipedia.org/wiki/Model-View-Controller - MVC
http://stan4j.com/ - Tool used to generate dependency analysis document
http://gradle.org/ - Build tool used to handle dependencies

APPENDIX