System Design Document (SDD) for LuckyCalendar

Mads Frederik Madsen - mfrm, Holger Stadel Borum - hstb and Paw Høvsgaard Laursen - pawh

9. oktober 2014

Indhold

1 Introduction **3**
- 1.1 Purpose of the system . . . 3
- 1.2 Design goals . . . 3
- 1.3 Definitions, acronyms, and abbreviations . . . 3
- 1.4 References . . . 4
- 1.5 Overview . . . 4

2 Current System architecture **5**

3 Proposed system architecture **6**
- 3.1 Overview . . . 6
- 3.2 Subsystem decomposition . . . 7
- 3.3 Hardware/software mapping . . . 7
- 3.4 Persistent data management . . . 8
- 3.5 Access control and security . . . 8
- 3.6 Global software control . . . 8
- 3.7 Boundary conditions . . . 8

4 Subsystem services **9**

5 Glossary **10**

Version system:
First decimal increments whenever we have made complete changes.
Second decimal increments for every major change (Not addition).
Third decimal increments for every minor change or major addition.

Date	Version	Initials	Title	Description
02/10/14	0.1.0	pawh, mfrm, hstb	Assignment 39	Sections 1, 3.1 and 3.2

1 Introduction

1.1 Purpose of the system

The purpose of our calendar system is to replace the old fashioned paper calendar with a twist. The calendar stands out in its ability to set up appointments between strangers. When creating an appointment, it is possible to leave room for extra unknown participants to be added to appointment by our server. These "lucky"participants will, in their own calendar, create an "empty"appointment, a so called lucky appointment, where they want to be invited to a random event, at a certain time, in a specified city. That way people seeking, for instance participants for a street party, a bingo tournament or even a 4th player in a tennis double match, will be able to find their extra participants. And people lacking something to do while on a business trip to a foreign city, or simply someone who is bored on a Saturday, will get an invitation to try something new, with new and exiting people.

1.2 Design goals

Ease of Use. The system should be intuitive and easy to use for the average IT-user. It should be possible to understand and use the functions of the program without any introduction or documentation. The use of the LuckyAppointment part of the program, should also be easy to use, but would require reading of the documentation for some users to fully understand.

Scalability in terms of number of calendar users and appointments. The server system of the program must only linearily increase in response time with the addition of users and appointments. It should be possible to add to the capacity of the system hardware-wise.

Accessibility. Whenever the user has internet connection and the Client installed. It should be possible to access the calendar system on the server, no matter the state of the server. This should be accomplished by back-up servers.

No loss of data on the server side. Since it is incredibly important for a Calendar system to be reliable, in terms of no lost appointments, it is the stated goal of this system that no program and/or server crashes should result in the loss of saved appointment.

Quick loading time. There should be very little to no noticeable delay when retrieving appointments to the Client from the Server.

1.3 Definitions, acronyms, and abbreviations

Term	Definition
Appointment	An appointment between 1 or more people. It can be seen as a time frame wherein the users plans to be doing a certain and defined thing. Internally in the system the Appointment contains a time, place, title, desired number of Participants and description for the event. If the desired number of participants is bigger than the added users, LuckyParticipants can be added through matching Lucky Appointments.
LuckyAppointment	A representation of the time space, the user wishes to join the appointment of another (pseudo-random) user. It contains a desired time frame and city.
LuckyParticipant	A user participating in an Appointment through a LuckyAppointment.
Participant	A user participating in an Appointment by having been adding to the Appointment. The user does not need to, or even desire to, participate in the appointment, in order to be a participant. The user is simply added as a participant, once another user adds them to the Appointment.

1.4 References

1.5 Overview

We aim to build a system that is a simple and smart replacement of the old-fashioned paper calender. In order to achieve this, we wish the system to be easy to understand, easy to use and easily accessible. Because how can we replace something if our alternative is not better than what we want to replace?
Furthermore we aim to build a system that brings people closer. We want to enable our users to meet new strange and exciting people and try new things. And we want them to be able to do this all the time. All day, every day. So we must ensure that our system support a lot of data without performance loss, never losses track of saved appointments, and, of course, back up our users' data securely.

2 Current System architecture

There is no current system, so there is none to describe.

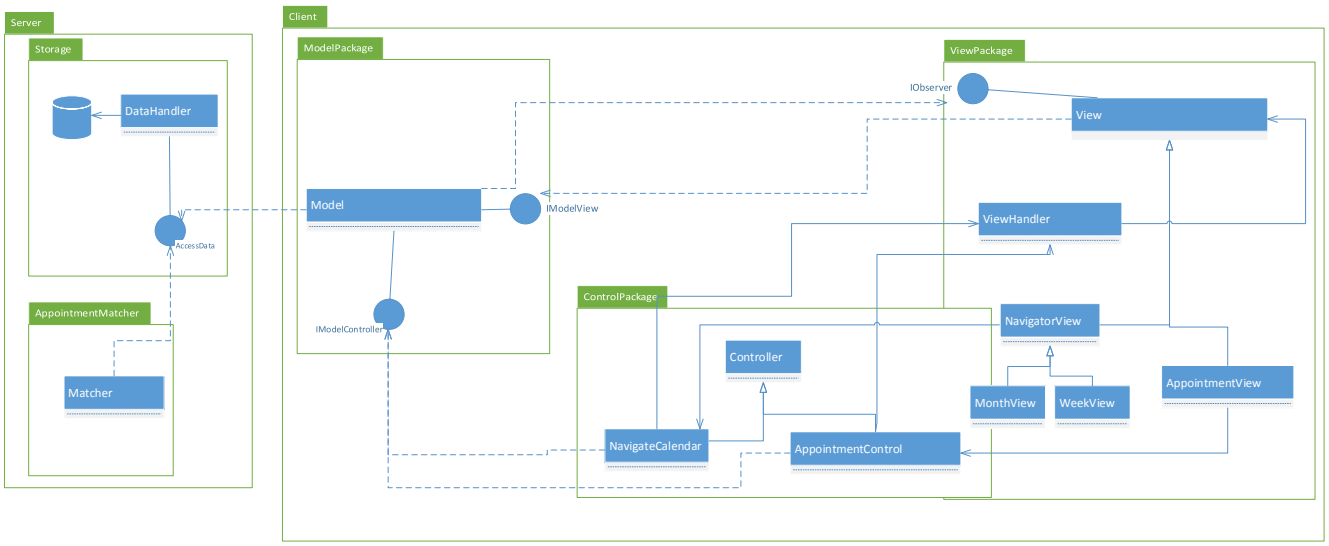
3 Proposed system architecture

3.1 Overview

The basic architecture for the system is a Client-Server setup. The server will handle persistent data and contain a small subsystem for manipulating data, we will for now not describe the server or the communication between the server and client greater detail and instead focus on the client.

The Client-system will be made with a Model-View-Controller (MVC) architecture. The system should have different views (Log in, MonthView... etc.) which would require different event-handling. The MVC allows this because it makes it easy to do runtime changes of views and controllers. It also results in high cohesion, in that the control objects and the view objects are separated in accordance to their respective responsibilities. Both view and control objects can be separated into small classes that only handles smaller tasks. However creating the MVC-architecture requires extra code. The main architecture can be seen below (it is possible to zoom in), we have not included all views and control objects to keep things a bit clean.

Architecture



3.2 Subsystem decomposition

The MVC architecture gives us three distinct types of subsystems, a model that handles date, a view that handles UI, and a Control object that handles event-flow and manipulation of data. The model subsystem should function independently from the rest of the system, therefore we used a facade design pattern to hide the internal implementation of the model from the rest of the system. In addition, the model communicates with the different views through a Observer/Observable-pattern, which mean the model isn't coupled to the Views.

The rest of the system is divided into View and Control objects. Each View requires a Control object, that handles the event-flow of the View. Each Control object has a reference to the ViewHandler, which allows them to create a new View when necessary.

At this point we have identified three types of views, a NavigatorView that shows calendar navigation, a AppointmentView that shows an appointment, and a login view that allows a guest to log-in and/or create a user. Each view have one of several control objects, which handles the events caused by the user. There might be several View objects for each Control object fx. the NavigateCalender control object could be responsible for MonthView, WeekView, and DayView under the common name NavigatorView. However, the opposite is not true. One view object cannot be controlled by more than one control object.

3.3 Hardware/software mapping

Since our system uses the Client-Server pattern, our software will have to run on two different pieces of hardware. First off, we have the clients - these are not required to be computation heavy machines. They are required to run windows, and in order to get the full functionality of our software, they should be able to connect to the Internet. The server(s) on the other hand, should be fairly quick machines, at least when it comes the internet connection. They are also required to run windows, and should preferably be able to communicate offline with each other in order to keep backup of our data. How this will be implemented in practice is yet to be determined.

The different responsibilities of the server and client are as follows: The Server:

- Storing all data
- Backing up data
- Running our LuckyMatchFinder algorithm
- Authorizing the clients requests

The Client:

- Keeping the local users credentials
- Synchronizing the local calendar with the server
- Saving changes when in a offline state, and later sending them to the server
- Providing the user with an easy-to-access UI

On the server side we will use the subsystem Datahandler to stay connected to the clients through a http-connection. The Client will use the model-subsystem to stay connected to the server. The clients will be logged on locally with after an authorization from the server. The servers DataHandler will function as a wrapper class around a SQL-database, through which it will load and save data after request from the client or the LuckyMatchFinder algorithm.

If a client for some reason should lose connection to the server, it will be able to save the changes locally instead, and will upon reconnection to the server, synchronize these changes with it. The clients model-subsystem uses a bridge pattern to apply the different data handling interfaces, which are assigned though a Strategy pattern which uses an Abstract Factory pattern to create the different data manager classes. The Client will also be able to synchronize with Google Calendar as a secondary server - that is, it will not replace our server, but rather work as a second back up for the user.

3.4 Persistent data management

We have identified the following persistent objects; APPOINTMENTS, LUCKYAPPOINTMENTS, USERS and NOFICATIONS. They are all saved by the server subsystem in a relational Database. When the user logs in, the Client synchronizes the users APPOINTMENTS, LUCKYAPPOINTMENTS and NOTIFICATIONS with the server, so that the Client subsystem is up to date with the changes, and the system can be used without connection. If local changes have not been saved to the server (probably due to loss of network connection), the system will synchronize next time a connection is established. This also means, that if a user has logged in to the system on a machine, it is possible to log in without connection later on, since the user-data is stored locally. There might be some problems with security or long periods without login, so we have yet to decide finer details of the synchronization between the Client and Server.

We considered saving persistent boundary objects locally, such as user preferences and last shown view, but we decided against it, since the client is supposed to be simple and intuitive, with ease of use and accessibility as declared design goals, making these boundary objects persistent, should not be necessary.

3.5 Access control and security

+ access matrix??

3.6 Global software control

3.7 Boundary conditions

+ inkluder i use-cases??

4 Subsystem services

5 Glossary

Glossary