IT UNIVERSITY OF COPENHAGEN

BDSA 2014

Calendar System
System Design Document

ASSIGNMENT 40

Anders Wind Steffensen - awis@itu.dk
Christopher Blundell - ppma@itu.dk
Pierre Mandas - cnbl@itu.dk

October 21, 2014

Contents

Contents i

1 Introduction 1
- 1.1 Purpose of the system 2
- 1.2 Design goals 3
- 1.3 Overview 5

2 Proposed Software Architecture 6
- 2.1 Overview 7
  - Overview of the program 7
- 2.2 Subsystem decomposition 9
- 2.3 Hardware/software mapping 15
- 2.4 Persistent data management 16
- 2.5 Access control and security 17
- 2.6 Global software control 18
- 2.7 Boundary conditions 19

3 Subsystem Services 20

CHAPTER 1

Introduction

1.1 Purpose of the system

The purpose of the Calendar system is to have a standard client-server calendar system like MS Exchange or iCal. The system must provide the user with the ability to save, edit events and be able to retrieve their calendar on any computer with the system installed. The system should furthermore provide the user with multiple kinds of overviews of the events he has planned, and prompt the user with a notification if an event has a notification time.

1.2 Design goals

- Strong and easy usability
  The goal with the systems usability, is to end up with a product which is user-friendly for all types of users. The gestalt laws are excellent examples on how to make a clean and simple user interface. The gestalt laws focuses on the placement of different GUI objects, such as buttons and text fields in a user interface, to make the interaction between user and user interface easy and painless for the user. By using the gestalt laws, we are able to make a clear distinction of which parts of the user interface that belongs to each other and thereby make it easier for the user to interact with the user interface. This will also mean that even the unexperienced user will have no trouble with interacting with our user interface.

- High Reliabiltiy
  It is desired to create a reliable program that will prevent the user from losing progress made to the system, due to unexpected events, whereas an event could be a computer turn-off by accident or any other such similar things. How our system will be handling this, is by doing frequent auto-saving of the data, to a local data file, and then synchronize the auto-saved data with our data storage, to have an external backup of the data. Furthermore, every time the user commits anything, whereas this for example could be a new calendar entry, it will also be saved immediately to the local data file, and thereafter synchronize the local data file with the database. Force restarting should not be acceptable and most exceptions must be caught and handled during runtime. By doing the abovementioned, data loss will be kept to a minimum by limiting it to the user's current activity, should a failure occur.

- Solid Performance
  The goal performance-wise is to be able to handle a large number of events daily, without setting a noticeable strain on the program. Heavy operations must run in the background, and must therefore not disturb the user while operating the program. The trade-off, however, will be the loading time of the program. This allows us to load all the initially required data, and prevent long waiting times while operating the system. The Calendar is a lightweight system, and stores data on the cloud, so space wise, a maximum of around

1gb should be achievable. Additionally, in its current form the CalendarSystem is only runnable on Windows OS.

- Strong achritecture with focus on extendability
  When rolling out future updates for the system, a full re-installation of the program will be necessary. This is due to resources allocated to other more desired design goals.

- Good documentation and Testing of the most important subsystems
  The system will be tested before release, but in a limited way. As a full system test can't be accomplished due to the time frame set for this systems development, a fully thorough testing will not be attainable, and we will therefore only focus on testing key components that are central parts of our system. Documentation will be a central part of our system, as it is important to keep documentation of how the system works as a whole, and how the modules or subsystem works separately, and by making this documentation, it will be easier to extend our system by developers that have no knowledge about the system

1.3 Overview

The architecture of the Calendar system will be centered around a model view controller pattern, and since data persistance is neccecary four ideal subsystems come inot mind.
Controller, View, Model and datastorage subsystems. Each of these systems have a specific role and by using these subsystems we loosen the coupling and increase the coherence.

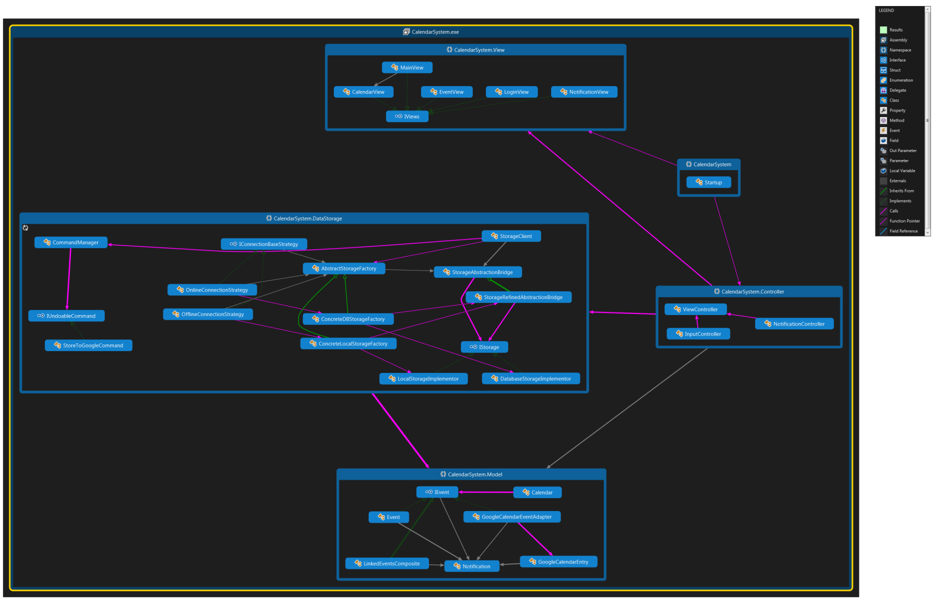
CHAPTER 2

Proposed Software Architecture

2.1 Overview

View subsystem: This subsystem contains the different types of views that will show, depending on how the user is using the system. Therefore, its assignment of functionality will be that this subsystem has to keep track of the different views.
Model subsystem: This subsystem contains the data that the system will be using. When the model changes state, it will notify the controllers, and the controllers will then change the view. Therefore, this subsystems assignment of functionality will be notifying the controllers, when its state is being changed.
Controller subsystem: This subsystem is being used to update the views, by receiving the models state. Its assignment of functionality will be that this subsystem will have to update the views with the received models state.
DataStorage subsystem: The DataStorage subsystem creates persistance of data either through a Database connection or locally in a file storage. The Client can gain access to the storage through a storage client which controls which kind of storage must be used at a given time and also controls the logic for the model to storage convertion.

Overview of the program



The figure shows the overall dependencies of the system, which object

refers to which. References between objects in the subsystems are not included. In the Model Subsystem the objects form the data we want represented. Events, notifications, and so on.
There are also a few extra classes the LinkedEventsComposite which is part of a Composite pattern used for linking together events, and the GoogleCalendarEventAdapter which is part of an Adapter pattern to make GoogleCalendarEvents able to be used along side our systems events.

The other interesting subsystem is the DataStorage. The classes of the subsystem form a number of objects which are used to store data. The objects form a number of design patterns together: The classes which is suffixed with Strategy forms a strategy pattern used to choose which factory to use.
The classes suffixed with Factory is part of the Abstract factory pattern. These factories creates the products LocalStorageImplementor and DatabaseStorageImplementor in a bridge pattern class (suffixed with Bridge).
The classes containing Command in their name are part of a command pattern, here the CommandManager can hold a number of commands, execute them and in case of a failure, call the commands undo method.

The classes in the subsystems Controller and View do not contain any design patterns.

2.2 Subsystem decomposition

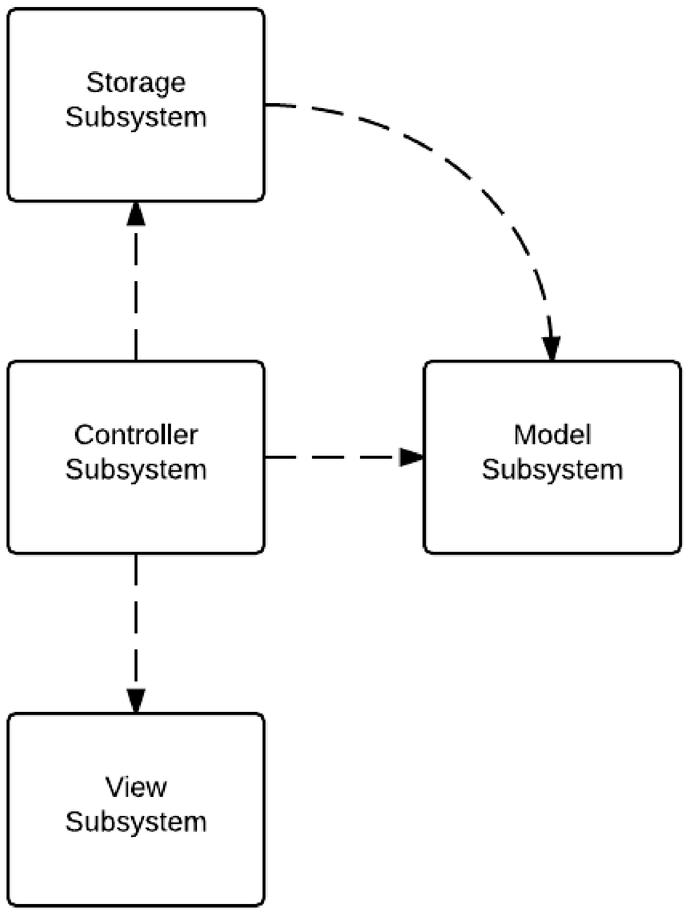
We have chosen to use the MVC design pattern, to make a clear distinction of what functionality should be assigned to which part of our system. The MVC pattern stands for Model-View-Control, and it's commonly used for implementing user interfaces.

We have also chosen to create a database, whereas this database will be used to store persistent data, to when the user will logout or close the program. This will make it possible to receive earlier used data.

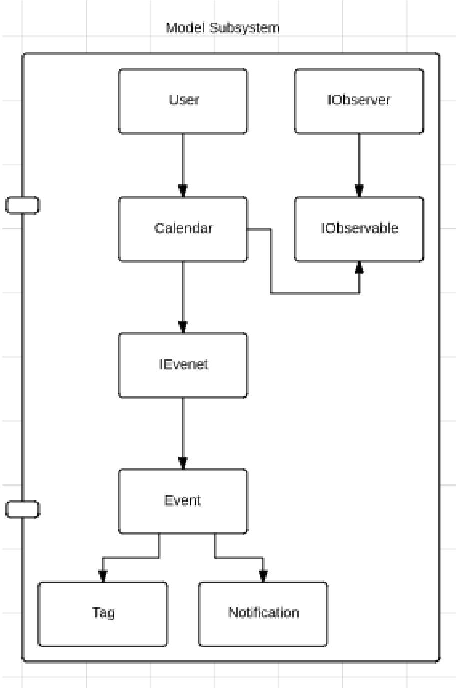
As we have chosen to use the MVC pattern, we will have 3 different subsystems, plus the subsystem of our database storage:

- Model
- View
- Control
- DataStorage

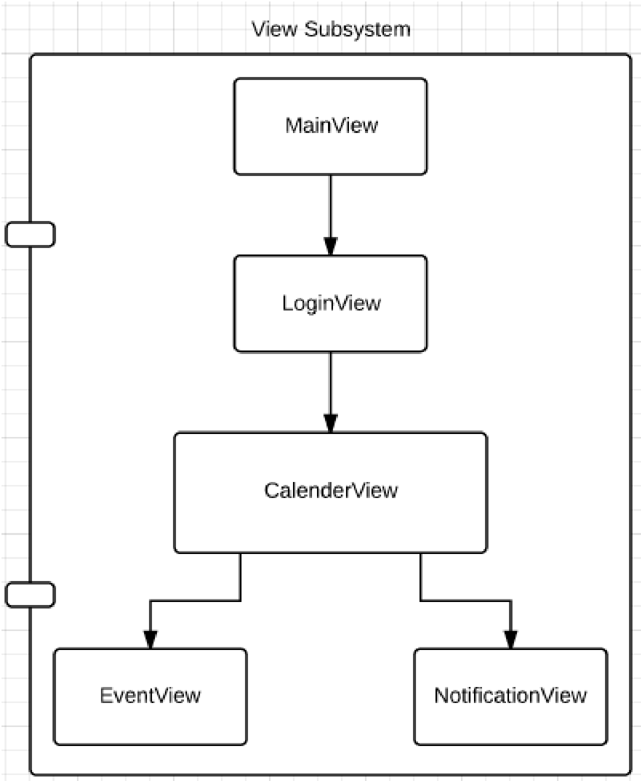
The overall relations between the subsystems can be seen here. The controller controls the flow of the software and takes requests and sends data on from the storage. Objects from the Model will be send to the controller and in some cases the view, from the storage.



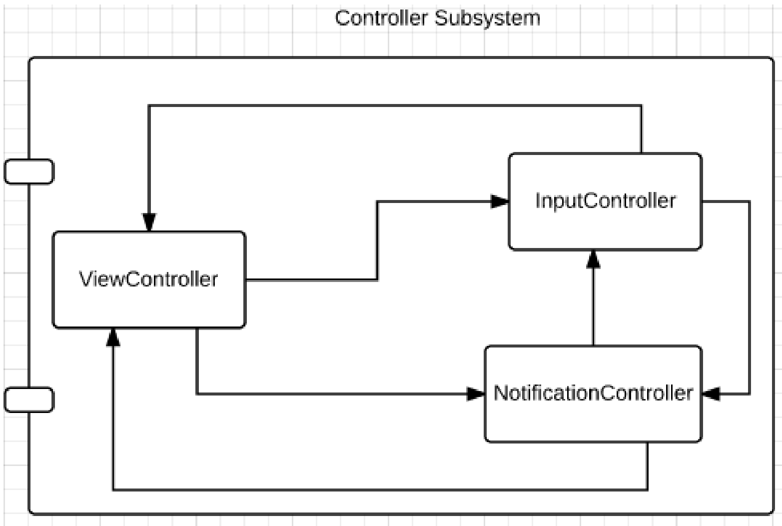
Our **Model subsystem** has the responsibility to keep notifying the controllers to update the views, which the user is using. Here is a picture of our subsystem:


Model Subsystem
User
IObserver
Calendar
IObservable
IEvenet
Event
Tag
Notification

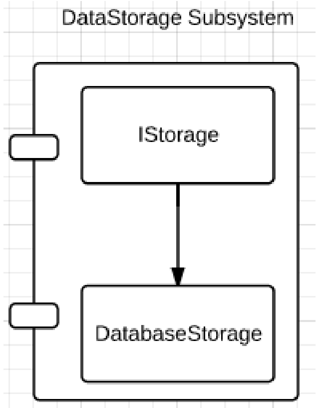
Our **View subsystem** has the responsibility to represent the model's state. Whenever the state of the model is changed, the view will also be changed to represent the new state of the model. Here is a picture of our subsystem:



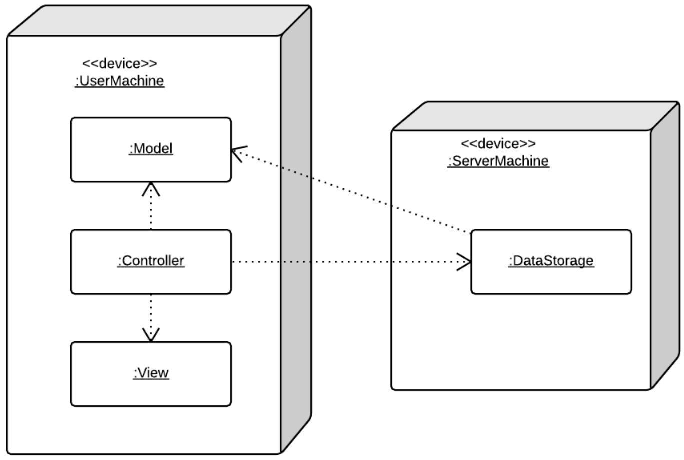
Our **Control subsystem** has the responsibility to update the view, whenever the model has changed. The controllers will be used to update the views, by being notified by the model about its state. Here is a picture of subsystem:


Controller Subsystem
InputController
ViewController
NotificationController

Our **DataStorage subsystem** has the responsibility to save persistent data, which the system will be using at another time. Persistent data could be a calendar for a specific user. When the user will be login into the system, the controllers will be using the DataStorage subsystem to receive the calendar. Here is a picture of our subsystem:



2.3 Hardware/software mapping



2.4 Persistent data management

To save the data in form of events, tags and so on, an online database connection will be used. The database gives access to data quickly and from multiple locations. The problem with this solution is that it requires an internet connection. To work around this a strategy design pattern can be implemented to provide the application the possibility to store data locally if no connection is available and vice. versa.
The Data which must be persistant is the following objects:

- Events
- Notifications - as long as they are not dismissed by the user
- Tags
- User info - passwords, usernames and their connection.

To peek, post, update, delete LINQ can be used since it supports these features in a strong cross storage-platform language.

2.5 Access control and security

Objects / Actors	Calendar	Notification	Event
User	createCalendar updateCalendar deleteCalendar	setNotification	createEvent updateEvent deleteEvent
System	createCalendarView	createNotifier	createEventView

Due to the fact that our Calendarsystem is available for many users, possibly with sensitive information, it is important to keep the individuals data private as seen necessary. This is done by providing each user with a password. When the user wishes to access his or her data, the password will be prompted, and after authentication through encryption, the respective data will be accessable to the user.

2.6 Global software control

Upon starting the program, the user is prompted with the loginView. Prior to the password being authenticated the CalendarSystem will proceed to load data belonging to the user. When the data has loaded, the CalendarView is initialized. The user can then interact with the system in multiple ways. For instance, the user can create/update or remove an event. This is then handled by the InputController, which will register the users activity and then load it in to the CalendarSystem. In the background this recent activity will be stored in to the database. When a change is committed, the CalendarView will then observe this, and the model will then proceed to update the view.

2.7 Boundary conditions

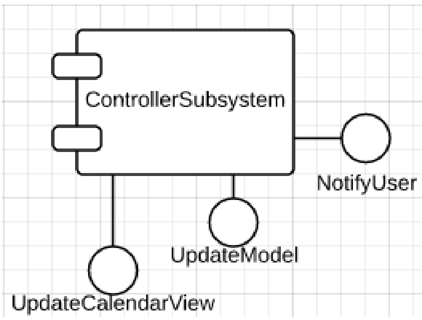
When looking at the data storage a boundary use case stands clear. If the storage must be able to save data locally and in the cloud(online database), the storage must have a method which can check for data integrity between the two, such that the online database at all possible time has the newest data.

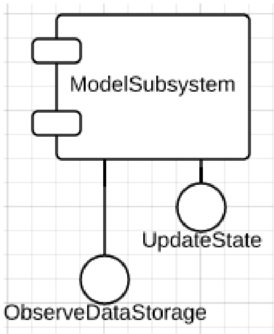
When looking at the start and shutdown of the system, making sure that data is saved and retrieved to/from the available storage(optimally the online database).

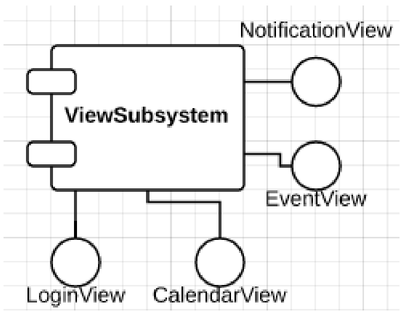
CHAPTER 3

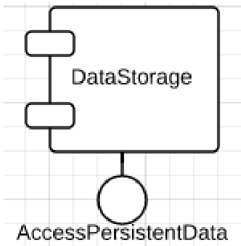
Subsystem Services

Subsystem Services








DataStorage
AccessPersistentData