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 9, 2014

Contents

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. By applying gestalt laws, the goal is to end up with a clean and simple user interface which is logical for all users, even the unexperienced.

- High Reliabiltiy
  Furthermore it is desired to create a reliable program, preventing users from losing all progress related to unexpected crashes. This will be handled by frequently auto-saving data and synchronizing with the data storage. Furthermore every time the user commits anything, for example a new calendar entry, it will also be saved immediately. Force restarting should not be acceptable and most exceptions must be caught and handled runtime. By doing the abovementioned, data loss will be kept to a minimum by limiting it to users current activity, should a failure occur.

- Solid perfomance
  The goal performance-wise is to be able to handle a large number of events daily, without setting a noticable strain on the program. Heavy operations must run in the background, and therefore not disturbing 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 there spacewise a maximum of around 1gb should be achievable. Additionally, in it's 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. Key

components will be tested, but due to the time frame set for this systems development thorough testing is unattainable. Documentation should be made for the majority of the classes - helping to the extendability

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: This subsystem contains a database storage. Users gain accesss to this database storage through an interface. Therefore, its assignment of functionality of this subsystem will be to store data, which will be used at a later time.

UML class Diagram

Along the project comes a UML Class Diagram file which can be viewed in Visual studio. Here we can see the different subsystems in action. Topmost is the Storage subsystem. The subsystem is in the form of a Abstract Factory pattern. The IAbsatract storage has a factory which creates Abstractstorages which has IStorage.
In this section the bridge pattern is also in use (the abstract storage works as a bridge) and the strategy pattern in the form of the Offline and Online connection classes.

Beneath the storage subsystem we can see the view subsystem. The classes all share the IViews interface which provides the classe with basic view methods such as show, hide and clear.

At the bottum of the UML class diagram we can see the Events and how they are implemented. The Event section of the model subsystem uses a couple of design patterns to increase coherence and loosen the coupling. The GoogleCalendarEvent is the result of the Adapter pattern and The LinkedEvents class is a result of the Composite pattern.

2.2 Subsystem decomposition

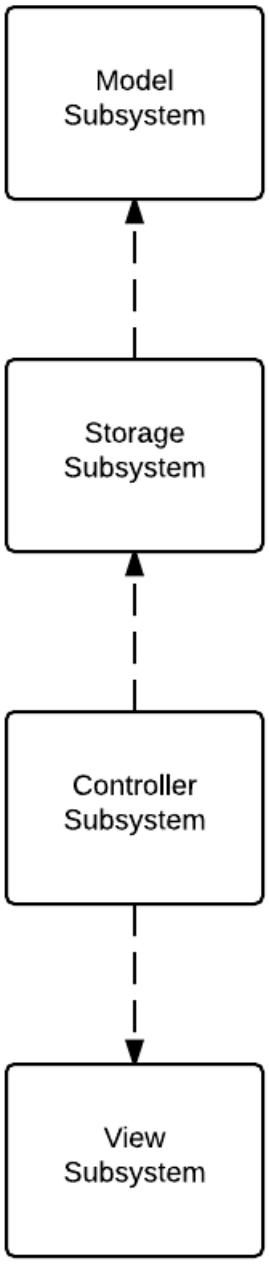
We have chosen to use the MVC design pattern, to make a clear distinction of what functionality should be assigned to which part of our system. The MVC pattern stands for Model-View-Control, and it's commonly used for implementing user interfaces.

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:

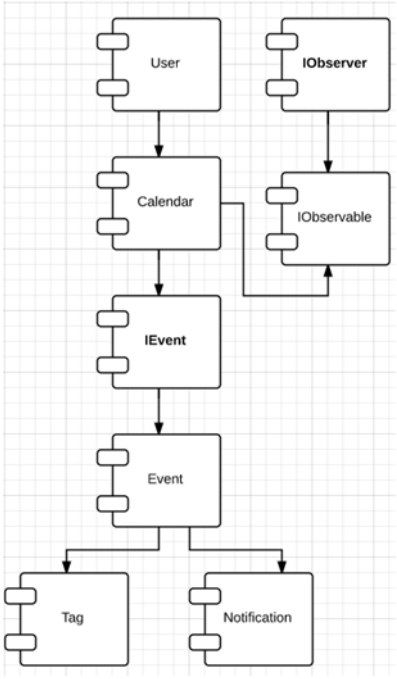
- Model
- View
- Control
- DataStorage

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.

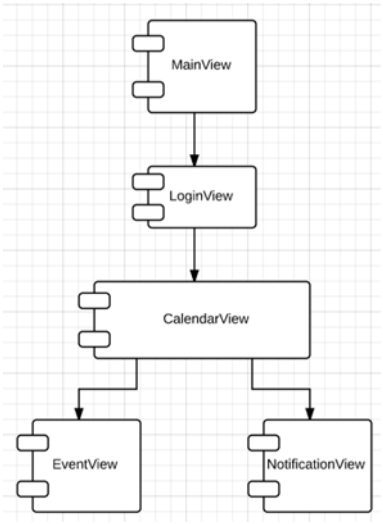


Our **Model subsystem** has the responsibility to keep notifying the

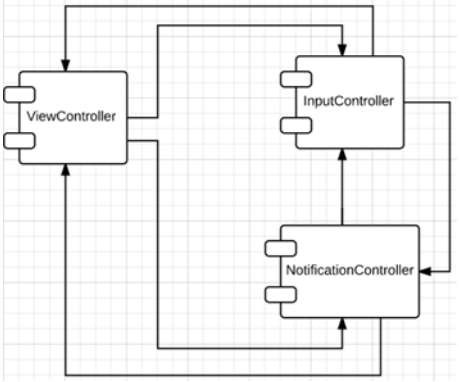
controllers to update the views, which the user is using. Here is a picture of our subsystem:



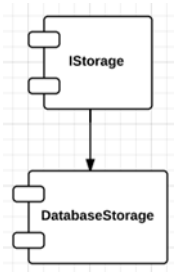
Our **View subsystem** has the responsibility to represent the model's state. Whenever the state of the model is changed, the view will also be changed to represent the new state of the model. Here is a picture of our subsystem:



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:



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

2.4 Access control and security

2.5 Global software control

2.6 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