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 24, 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. 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 commands, and then retrieving or generating the model. 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.

Figure 2.1 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).

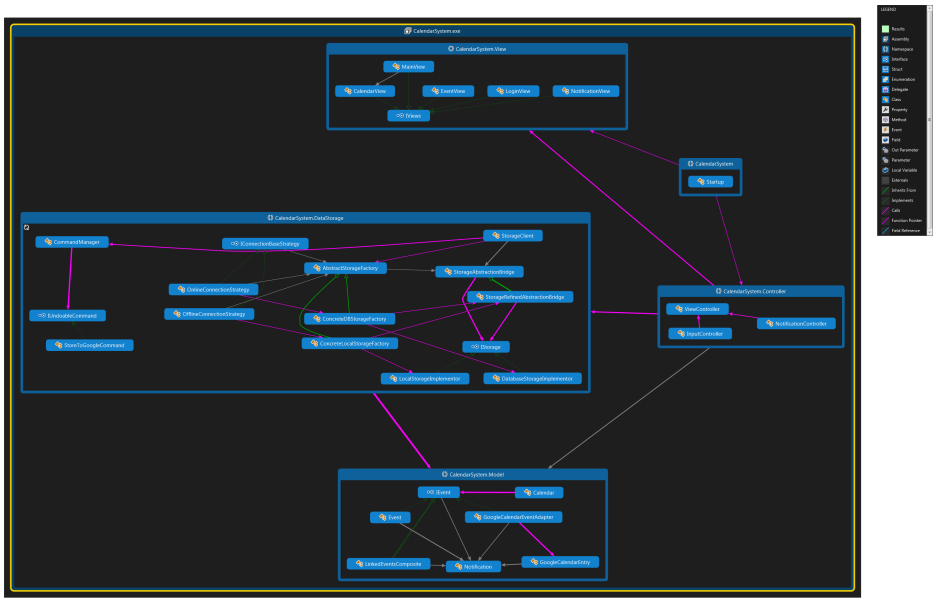

Figure 2.1 – Zoom in to see details - A dependenzy overview of the system

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.

Overall Subsystem Decomposition

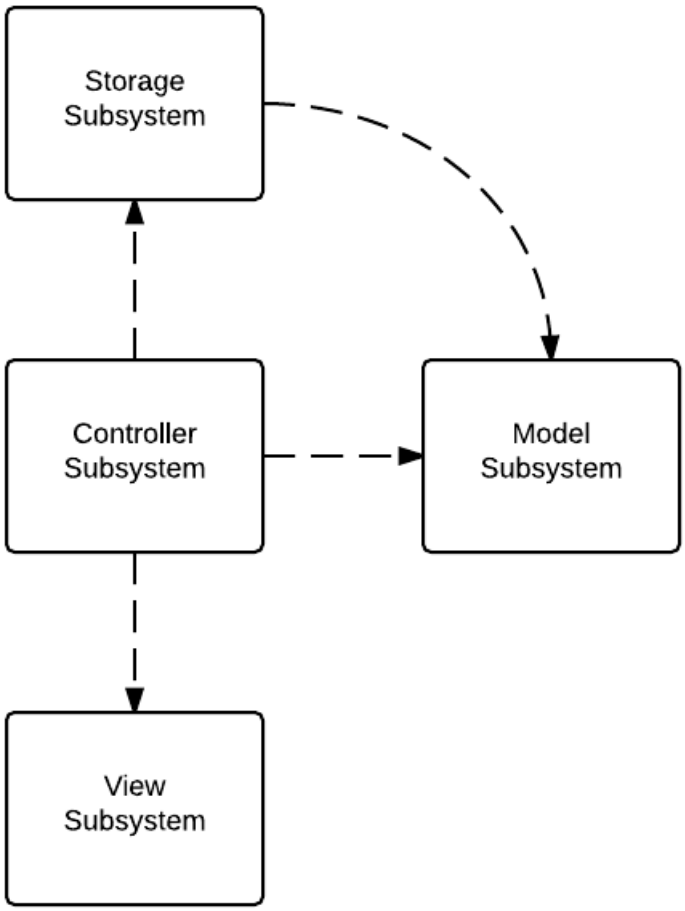

Figure 2.2 – The overall subsystem decomposition - not up-to-date

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:

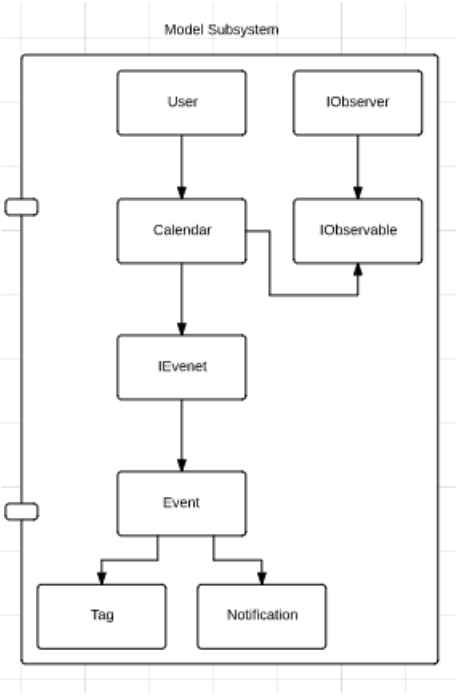

Figure 2.3 – The model subsystem decomposition - not up-to-date

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:

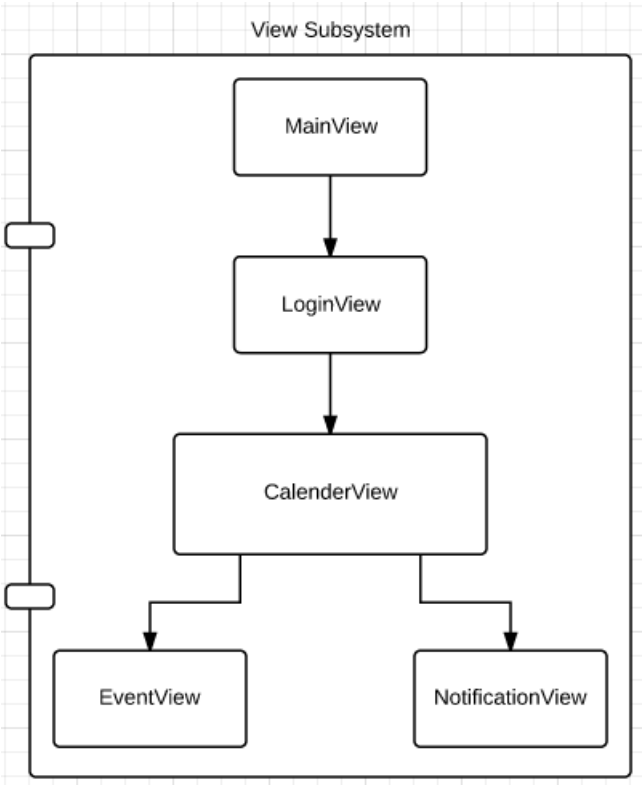

Figure 2.4 – The view subsystem decomposition

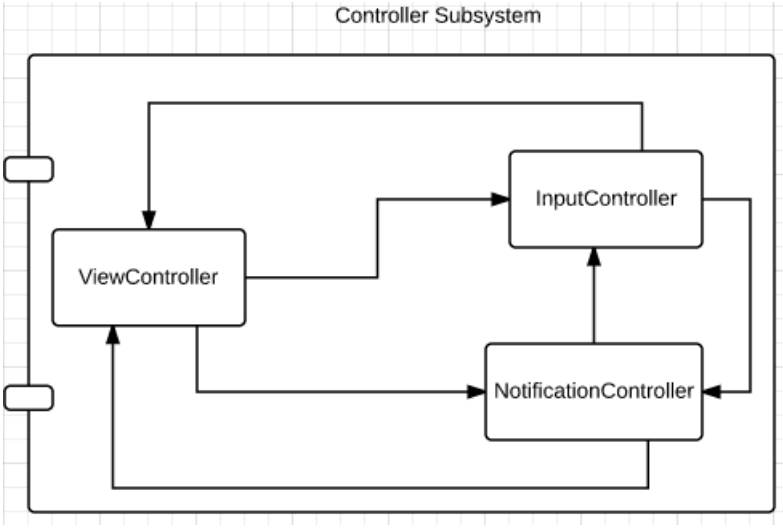

Figure 2.5 – The control subsystem decomposition

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:

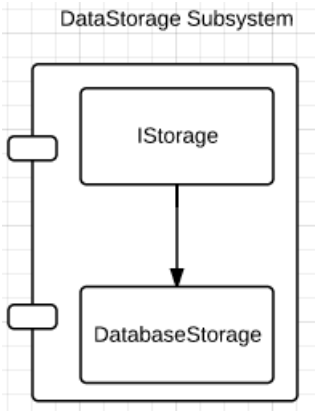

Figure 2.6 – The data control subsystem decomposition - not up to date

2.3 Hardware/software mapping

Hardware/Software Mapping

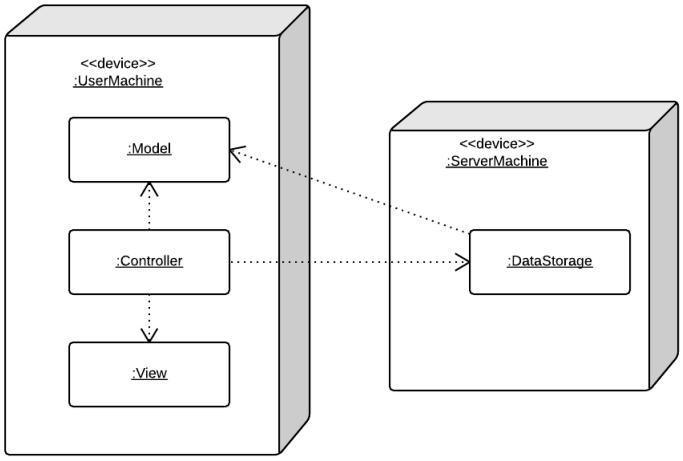

Figure 2.7 – Zoom in to see details - Diagram of the Hardware/softwarem apping

The three subsystems running on the UserMachine are Model, Controller and View. The Controller interacts both with the Model and the View, for logic and presentation. Upon saving an event in the CalendarSystem, the Controller will then communicate with the second device, the ServerMachine, and send/request data. Following this, the DataStorage returns the data to the Model and can be done with as pleased.

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

Access Matrix for CalendarSystem

Objects / Actors	Calendar	Notification	Event
User	createCalendar updateCalendar deleteCalendar userLogin	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. This limits the users access to other users personal information.
Furthermore it can bee seen from the matrix that the user only has access to certain data through certain objects. This is done to provide ease of use to the user. While the user can create, delete and update entries and modify their associations, the system handles the automated 'reminders'.

2.6 Global software control

The flow of our system would be a Proceduce-driven control, and the reason for this, is that the program will follow specific prodecures, depending on how the user is interacting with the system.

Upon starting the program, the user will be prompted with a login view, whereas the login will be used to receive a specific calendar, that is being stored in our database. The user can't access any calendar if he or she hasn't logged in, since the system wouldn't know which calendar that belongs to the user.

For login in, the InputController of our Controller Subsystem will be using the information entered into our login view to authenticate the information, and if the authentication is a success, our DataStorage Subsystem will access the database and receive the specific calendar belonging to the user, and our Model Subsystem will then observe the DataStorage and update its state, whereafter the CalendarView, from our View Subsystem, will be initialized and show the data received from the database.

The user can then interact with the system in multiple ways. For instance, the user can create/update or delete an event. These activities will be handled by the InputController, which will register the users activity, update the state of the Model and then load it into the CalenderView. In the background, recent activity will be stored in the database, as the user is still using the system. If a change is commited, the model will update its own state, whereafter the CalenderView will be updated by the ViewController.

2.7 Boundary conditions

Boundary Conditions for CalendarSystem

Boundary Conditions:	
PersistentDataStorage	When the CalendarSystem is shut down, the CalendarSystem can convert the persistent storage from a flat file storage to a database storage or from a database storage to a flat file storage, depending on the current state of internet connection.
StartCalendar	The CalendarSystem invokes the Check Data Integrity use case on start-up, and depending on the state of connection from previous shutdown will load from either persistent storage.
ShutDownCalendar	The CalendarSystem checks for any changes being committed at the time of termination, and depending on the internet connection, will confirm saves to the respective storage current.
CheckDataIntegrity	CalendarSystem checks the integrity of the persistent data. For the textfile-based storage, this may include checking if the last logged transactions were saved to the textfile. For database storage, this may include invoking tools providing by the database system to re-index the tables.

2.8 Interfaces

IViews
An interface containing three central methods, common for all our views that implement this interface: Show(); Hide(); Clear();

IEvent
An interface made for the event class. This interface contains methods which hold the bare minimum that an Event should be able to do.

INotification
An inferface made for the notification class. This interface contains a method for setting the description of a notification, and a method for determining whether or not the notification is in an alarm state.

IStorage
An interface for the storage class. The interface has methods which makes it possible to retrieve and save events into the calendar, without knowing the actual implementation.

2.9 Invariants and pre- & post-conditions

Invariants

Event
context Event inv:
DateTime > (1990,1,1) && DateTime < (2100,1,1)

context Event inv:
this.ID < storage.MaxValue(ID)

context Event inv:
this.ID > -1

INotification
context INotification inv:
DateTime > (1990,1,1) && DateTime < (2100,1,1)

context INotification inv:
DateTime.now > _date == IsInAlarmState

IStorage
context IStorage inv:
GetAllEvents().Count() < GetMaxID()

context IStorage inv:
EventsBelongTo(events, userName)

context IStorage inv:
DateTime > (1990,1,1) && DateTime < (2100,1,1)

Pre- & Post-conditions

IStorage
context IStorage pre:
userName != null

context IStorage pre:
password != null

context IStorage::LoginAuthentication(u: userName, p: password) pre:

match(u, p)

context IStorage::LoginAuthentication(u: userName, p: password)pre:
exists(u)

context IStorage::SaveEvent(e: eventToSave) pre:
eventToSave != null

context IStorage::SaveEvent(e: eventToSave) pre:
GetEvent(e.ID) == null

context IStorage::SaveEvent(e: eventToSave) pre:
e.ID > 0

context IStorage::SaveEvent(e: eventToSave) post:
GetAllEvents().Count() == self@pre.getAllEvents().Count() + 1

context IStorage::SaveEvent(e: eventToSave) post:
GetEvent(e.ID) == e

context IStorage::UpdateEvent(e: eventToUpdate) pre:
e != null

context IStorage::UpdateEvent(e: eventToUpdate) pre:
GetEvent(e.ID) != null

context IStorage::UpdateEvent(e: eventToUpdate) post:
GetAllEvents().Count() == self@pre.GetAllEvents().Count()

context IStorage::UpdateEvent(e: eventToUpdate) post:
GetEvent(e.ID) == e

context IStorage::DeleteEvent(i: ID) pre:
GetEvent(i) != null

context IStorage::DeleteEvent(i: ID) pre:
i > -1

context IStorage::DeleteEvent(i: ID) post:
GetEvent(i) == null

context IStorage::DeleteEvent(i: ID) post:
getAllEvents().Count() == self@pre.GetAllEvents.Count() - 1

context IStorage::GetEventsBetweenDates(b: beginDateTime, e: endDate-Time) pre:
b < e

context IStorage::GetEventsBetweenDates(b: beginDateTime, e: endDate-Time) pre:
b > new Date(1900,1,1)

context IStorage::GetEventsBetweenDates(b: beginDateTime, e: endDate-Time) pre:
e > new Date(1900,1,1)

context IStorage::GetEventsBetweenDates(b: beginDateTime, e: endDate-Time) pre:
b < new Date(2100,1,1)

context IStorage::GetEventsBetweenDates(b: beginDateTime, e: endDate-Time) pre:
e < new Date(2100,1,1)

context IStorage::GetEventsBetweenDates(b: beginDateTime, e: endDate-Time) post:
return.TrueForAll(e => e._date.Value >= beginDateTime && e._date.Value <= endDateTime)

context IStorage::GetEventsBetweenDates(b: beginDateTime, e: endDate-Time) post:
return != null

CHAPTER 3

Subsystem Services

Controller Subsystem

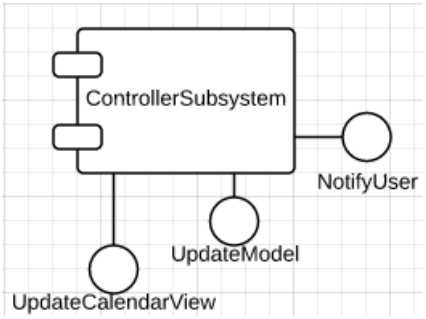

Figure 3.1 – Diagram of the Controller subsystem service

The ControllerSubsystem is a subsystem that contains 3 different controllers:

- ViewController
- InputController
- NotificationController

The purpose of our viewController is to change the view, depending on the interaction between the user and the system. There are different kinds of views, such as a view for login, event creation, the calendar as a whole and so on. Our UpdateCalendarView service will use the viewController to update the calendar view whenever there has been made a change in the calendar, for example if there has been added an event to the calendar.

The purpose of our InputController is to use the input that the system gets and then do something with the input, depending on the current event. The UpdateModel service will use the InputController to update the model, and thereby change the state of the model, and thereafter the UpdateCalendarView service will be used to update the calendar view with the new state of the model.

The purpose of our NotificationController is to notify the user, of a specific notification that has been set by the user. The NotifyUser service will be using the NotificationController to notify the user, which then will create a NotificationView that will be showed by the viewController to the user.

Model Subsystem

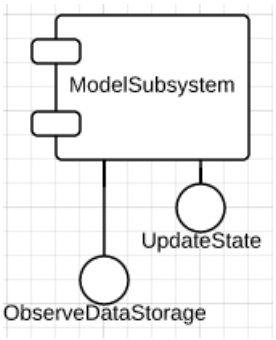

Figure 3.2 – Diagram of the model subsystem service

The ModelSubsystem consists of two different services: UpdateState and ObserveDataStorage.

Our UpdateState service is meant to update the state of the model, that the current system is using. When the UpdateModel service is being used by the ControllerSubsystem, the ModelSubsystem will receive whatever data that the UpdateModel service has to give, and then update its own state with the new data, received by the ControllerSubsystem, by using the UpdateState service.

Our ObserveDataStorage service is meant to observe and receive data from our persistent data storage, which is our database. When the user

starts up our system, our ModelSubsystem will be using the Observe-DataStorage service to check if there is any updated data, that the current system doesn't contain, and if there are any updated data, it will receive that data from the database, store it locally and show it by using the ViewController.

View Subsystem

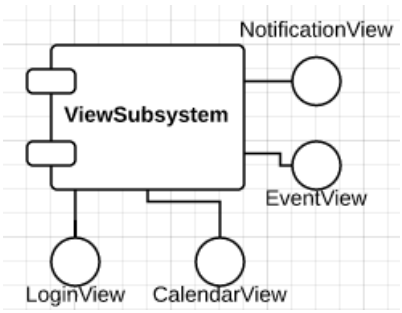

Figure 3.3 – Diagram of the view subsystem service

The ViewSubsystem contains four different services: NotificationView, EventView, CalendarView and LoginView.

As the ViewSubsystem services in themselves are views, the Subsystem services' purpose will be to create the different views, depending on the current event.

DataStorage Subsystem

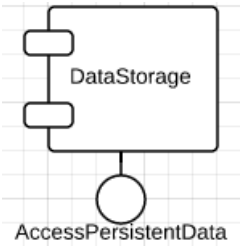

Figure 3.4 – Diagram of the data storage subsystem service

The DataStorage Subsystem contains a single service, which will be used

to access the persistent data, that has been put into our database. This service will be used by our ObserveDataStorage service of our Model-Subsystem, to observe and receive data from the database, though our DataStorage Subsystem.