System design document for HabitHets

Elin Nilsson, Jacob Messinger, Tina Samimian, Norbets Laszlo, Oscar Helgesson

2019

Contents

1 Introduction

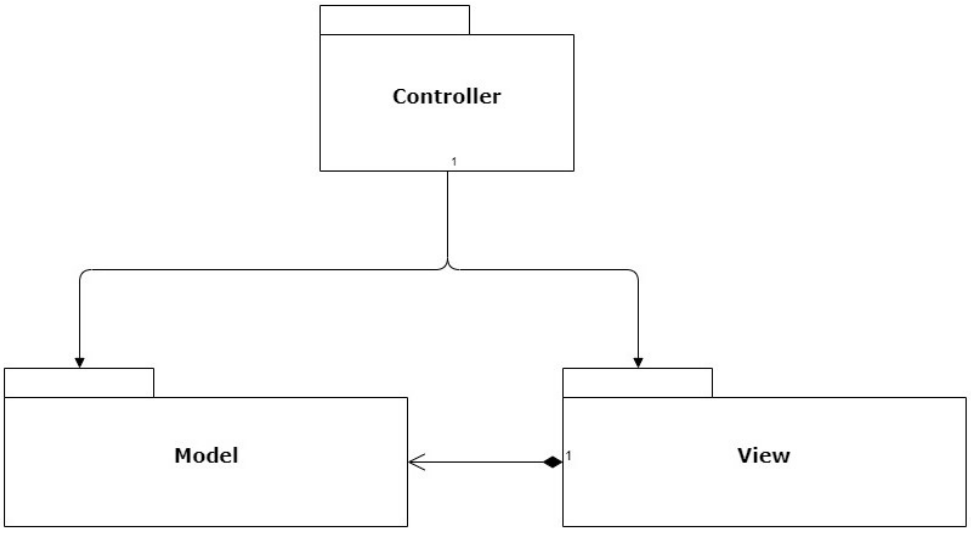
The application is a Calendar desktop application to simplify everyday planning and structure. With easy access to habits (recurrent activities), todos (activities the user want to do once but still be able to see when and if finished), notes (a quick way to get down thoughts one might have) and of course events to be placed in the calendar.

1.1 Definitions

To be able to write and design this program in a sufficient manner different design patterns and libraries has been implemented. These are mentioned in this document, but not specified what they are:

- **JavaFX**, is a set of graphic and media packages allowing a user to have graphical implementation of the Java code.**[1]**
- **LocalDateTime** class is an immutable date-time object in Java with a format of ”yyyy-mm-dd-hh-mm-ss”. **[2]**
- **JUnit** is a unit-testing framework for java, used to write and run tests for the java programming language.**[3]** Because it has a graphical interface, it is easy for the user to test source code.
- **MVC**, or Model View Controller, is an architectural Design Pattern which has been the foundation on which the application has been designed on. The design pattern separates the Model (which holds all logic), from the View and Controller, to not get unexpected dependencies.
- **Organizers** are classes defined for this project responsible for logic that concerns more than one object of their corresponding class. For example the HabitOrganizer is responsible for all logic which concerns more than one specifc Habit.

2 System architecture



The application implements a classic **MVC** pattern to achieve a modular design on a high level. The graphical representation (**View package**) and the computing representation (**Model package**) that holds all the logic, for example Calendar logic and data handling, has been separated. By implementing the **MVC** pattern together with a facade pattern (described in detail below) a low coupling has been achieved.

The application does not use external components. All data will be saved as text-files.

2.1 Flow

The first thing that occurs when running the application is that the **Main** method is run. The **Main** method instantiates the **JavaFX** application which initiates the **CalendarController** through the initilize menthod. In the **CalendarController** lies constructors for the database methods which in turn fetches the previously saved data. These methods are responsible for writing the fetched data to the corresponding **Organizers** and at the same time sets the IDs for each object in the **Factory** class to make sure an object is not overwritten by another object with the same ID.

After fetching and writing the data from the database, the different views are created and at the same time the **MasterTime** variable is set to the current time. This variable can be manipulated in order to make calculations for the calendar such as what happens if the user wants to view next week from now.

As the views have now been created, the **CurrentView** method gets set to the week view which is what the user will be met with at the start of the program. The **CurrentView** then changes as the user switches between views. There are also different setup methods being run responsible for things like setting up the time line. The time line (which marks the current time) then gets updated every 60th seconds through out the applications lifespan.

2.2 Input flow

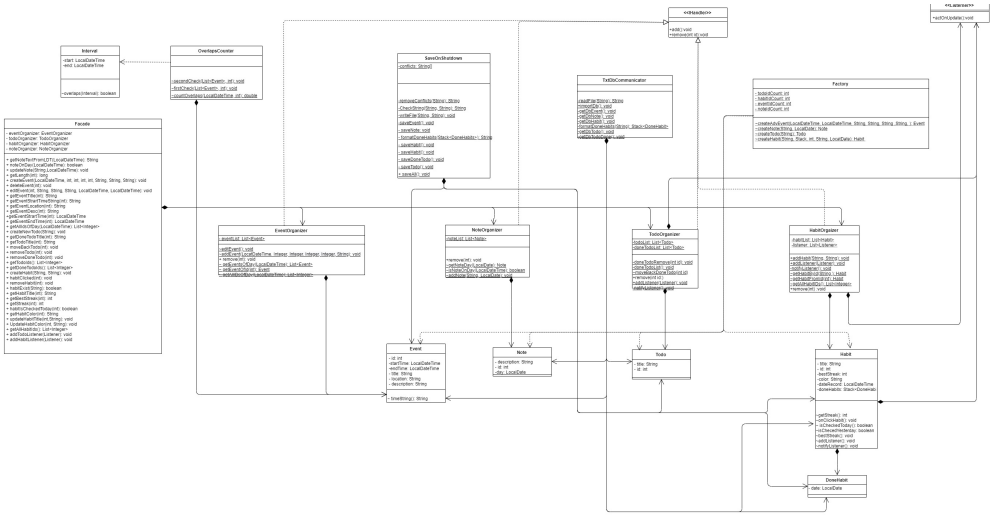
If the user wants to **edit an event** for example, the flow is as follows:

The event is pressed in the GUI. This in turn sends a signal to the **Controller** (via Listeners) which in turn has a method to fetch the data for that event using the ID. The fetched data then gets portrayed to the user, who can now manipulate the inputs. When the user is finished, pressing the ”Save” button in the GUI sends a signal to the facade to save the data from the inputs, and then the input interface gets closed.

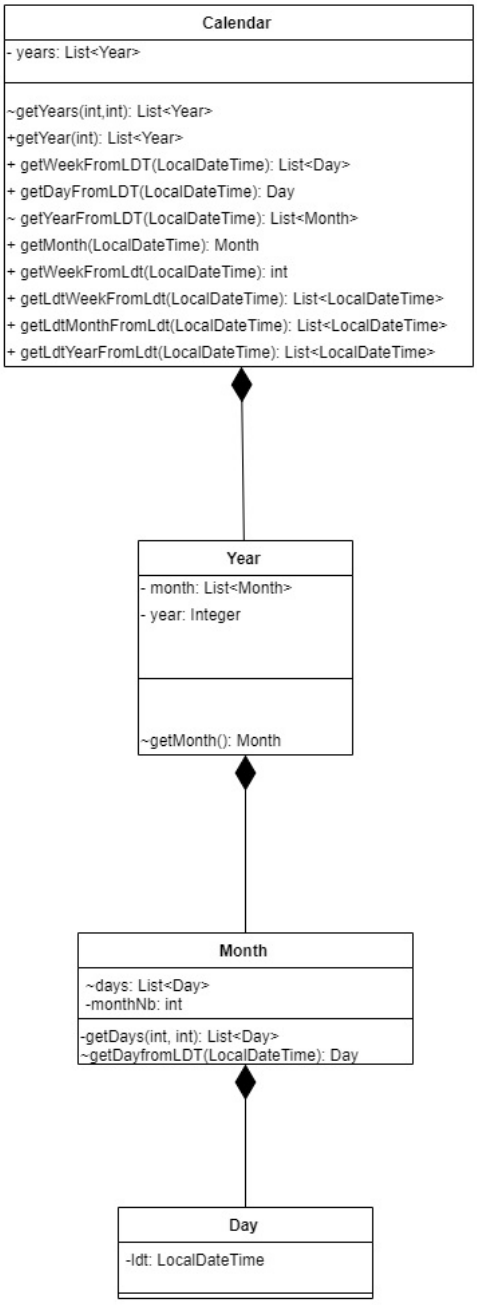
Another example is if the user wants to **Add a habit**. Then the flow is:

The button ”+” next to the title Habit gets pressed. The GUI now shows a ”Add Habit” box. The user is now free to input a title as well as deciding what color the Habit should be. When done, pressing the button ”Create” will call upon the **Factory** class to create a new Habit Object with the given inputs. When it has been created it gets added to the list of Habits in the **HabitOrganizer** class. The **HabitView** can now update the list of Habits (which it gets from the HabitOrganizer through the **Facade**) and the user can see the newly created Habit in the GUI.

3 System design



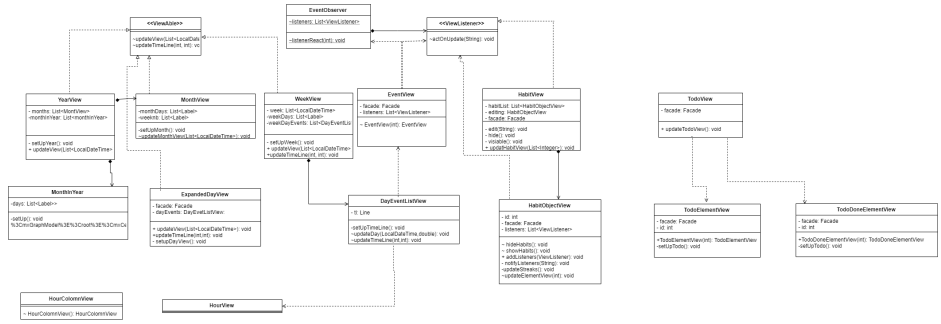
UML diagram is also attached in the zip-file



UML diagram is also attached in the zip-file

HabitHets model package is divided into two main components. The first part is responsible for all the data related to logic, which falls into either one of the following 4 categories: **Habit, Event, Note and Todo.** The only way to access this data is through the factory class. The reason behind this is to achieve a low coupling on a package level and high cohesion within the package.

The second part of the package is responsible for the calendar and it's logic that is used in the application. When the application is started this calendar generates days in an interval relative to today's date.



UML diagram is also attached in the zip-file

The **View** package aspire to be as modular as possible to enable smaller views to be reused in other parts of the application. The application's calendar and main view can be represented in four different ways:

- An **Expanded day view** with more detailed information about a single day. It presents the user with the possibility to add notes to that certain day.
- A **Week view** that summarises a whole week and acts more of a schedule.
- A **Month view** that shows a month's structure in form of weeks and days.
- A **Year view** that presents the user with a view of the years twelve months.

The week and expanded day view uses the same day components which uses multiple hour views and possible event views. This way the reusability of the smaller views in the larger views has been maximised.

All of these 4 views implement a **Viewable** interface which is responsible for the update of the content in the calendar views.

Apart from the calendar view HabitHets has two more views which are the **Habit View** and **Todo view**. These views are entirely independent and do not rely on other views except their own elements.

The **Todo view** has two different parts. One where the user can see the upcoming todos and one where the last 5 accomplished todos are visible under the title Done Todos.

Habits can be seen in two windows. One where all information about each habit is seen (title, chosen color of the habit, streak and best streak) and one collapsed window where only the color is visible. The collapsed window is the default window. In this window one can not edit nor delete a habit but the check-function is still available here. The check-function of both windows is created in a way that it is connected to the color. By checking a habit the color-circle gets filled with that same color.

By taking a look at the design model one can see connections to the domain model. By describing the most important aspects of the application before starting to code, it has been easier to follow the original plan for the project and not get distracted or confused as to what to prioritize. In the design model, it can clearly be seen that the main-class, which contains the start method, is called **HabitHets**. This is done to show that this is what every aspect of the program is connected to. There are also different **Organizer classes**, responsible for the logic associated to the parts in the domain model, such as Habits, Todos etc.

Looking at the design model one can see that there are two separate entities that are not in any way intertwined. These two model packages are actually working together. For example Note has only a date(LocalDateTime), which in the NoteView can be connected to a day in the calendar through that date. Since this is done in the View package, it is not visible in neither the design model, of the separate packages, nor the domain model.

3.1 Design Patterns

The **State-pattern** is used in the code. It is not wanted to update all of the views all the time. It is also desirable to know what view the user sees. These two can be achieved by creating an interface **ViewAble** that has the methods updateView() and updateTimeLine(), and also by making a variable ”currentView” of type ViewAble in Controller. ExpandedDayView, WeekView, MonthView and YearView implement ViewAble and inherits the methods updateView and updateTimeLine and have their own versions of the methods. Every time any of the methods of ViewAble are called in Controller, the corresponding version of the method is run in the view-classes. This means that every time the view changes, currentView is pointing at the view that the user sees.

By using the **Factory-pattern** internal implementation can be hidden, for example the constructors. This happens in the class Factory where the methods that creates objects are defined. This results in avoiding classes getting instantiated when they are not needed to. This also result in the code following the Dependency Inversion Principle. In the future when some functionality might be extended, the changes only need to happen in Factory instead of changing all calls to the constructors spread out in the code. This follows the open-closed-principle.

In order to decrease the number of dependencies in and between the packages, the **Observer pattern** has been implemented. One example is the class HabitView that listens to changes being made in HabitObjectView (which implement the Interface ViewListener). Another example of this is where Controller listens to TodoOrganizer. When changes are being made in TodoOrganizer the Controller gets notified and can then tell the TodoView to update. This way there is no dependencies between the view and model.

To make this the model easily accessible it has two aggregate classes with methods within that returns different parts of the calendar on call. By using the **Facade pattern** it makes it possible to hide complexity from the view and not show unnecessary logic which the view does not need to know. This leads to a more secure code with low coupling. Two different facades have been set up, Calendar and Facade. One for the Calendar, and one for the rest. By doing this the two are separable, making it easy to handle them as completely separate entities. This means that in the future, if someone wants to use only one of them, there is no need to implement

the other.

4 Persistent data management

As far as persistent data goes there are four kinds of persistent data in the sense of not being deleted at restart of the program. These are Todos, Habits, Notes and Events. All data is being saved as list items in corresponding Organizer classes. These are responsible for holding logic for corresponding classes in order to create a more modular design and to hold data.

The ability to switch to a different database in the future is something that has been thought through. That's why toady's implementation does depend directly on any information storing solution. Information can always be inserted into the organizers by independent services. Thanks to the modular design the data storing implementation could be changed with ease.

4.1 On shutdown

In the main-method, **saveAll()** (in class SaveOnShutDown) is called. When the application is stopped, this method makes sure that everything that has happened during the applications lifespan is saved. The process of saving data is the following:

saveAll() is called and calls on separate saving-methods for each part of the application. There are two methods that are called in the separate saving-methods. These two methods checks that the data that is about to be saved, do not contain strings that will cause conflicts on future reads. For example if the user writes

$$< end >$$

these methods will replace the string with

$$< -end >$$

In **SaveOnShutDown**, there is a List of the strings that are to be replaced. There is a write-method that creates a new text-file with the same name as the previous, and in that way overwrites the old text-file with the new data.

5 Quality

The Definiton of Done regarding the User Stories says: ”To be considered done, all User Stories should first be unit tested to a sufficient degree of satisfaction.” This has led to a thorough testing of each implemented logical functionality.

Using JUnit as our testing platform, each method has been tested in such a way that every possible outcome has been tested for in a specific method. For example the Method **IsCheckedToday()** in Class **TestHabit** has been tested in order to see if the habit has been checked or not. This specific method can be tested in various ways. For example when a habit is unchecked and the CheckedButton is pressed the first test-method tests if a habit then is checked today. The second test-method **CheckedTodayTest()** confirms that a checked habit that gets unchecked shows that the stack is empty. And a third test that confirms that miss matched data returns false.

Known issues include:

- When three or more events overlap there seems to be a graphical bug where instead of filling the entire width of the day, it only fills half of the width.
- Since text-files are currently used as a database alternative, some security problem aspects are present. Such as manually being able to override the id counter.
- Because of lack of time, the only time Mutate-by-copy is implemented is in the method **copyMasterDate()** in class **ControllerCalendar**. By using this, the security of the code would increase a lot, hence this should be used in more places in the code.
- On application start the existing events might have issues with displaying there full background. This issue is partly solved by updating the view a few seconds after start.

6 References

In the list of references relevant articles will be found explaining external libraries used in the project, such as **JUnit** for testing purposes, **JavaFX** for graphical purposes and **Java Date/Time** for Calendar logic purposes.

1. M. Clark , ”JUnit FAQ - What is JUnit”, JUnit, 25 Nov, 2018 [Online], Available: ”https://junit.org/junit4/”, [Accessed 23 Oct, 2019].

2. M. Pawlan, ”What is JavaFX?”, JavaFX, Apr, 2013 [Online], Available: ”https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm”, [Accessed 23 Oct, 2019].

3. E. Paraschiv, ”Introduction to the Java 8 Date/Time Class”, Java Date/Time, Available: ”https://www.baeldung.com/java-8-date-time-intro”, [Accessed 23 Oct, 2019].