Project document

25.4.2024

Roosa Ripatti 100666234

Studying second year of information networks

Table of contents

1. General description of the project

The idea of the calendar program is to provide the user with the ability to keep track of their own appointments and related details. The program offers a graphical user interface where the user can add and customize events. Each event includes a name, date, start time, and end time. The user can define categories for events, and each category can be monitored as its own group. Additionally, the calendar includes common holidays and can send reminders to the user for important tasks. Generally available calendar programs are also not very visually appealing, so I wanted to focus on the appearance and design of the program.

Even though my initial goal was to implement the task in a challenging way, the final product lands somewhere between the easy and intermediate task, but more on the intermediate side. Even though it doesn’t necessarily have all of the requirements from the intermediate assignment perfectly executed, I think my program is a lot more advanced than the easy assignment required. I have put especially much time and effort in the GUI, which wasn’t even a requirement in the easy task, and I hope that this will be taken into account in the grading.

I hope it will also be considered in the grading of this report that I am not a native English speaker, and I struggle with writing English. Writing this took me a considerable amount of time, and I didn’t have time to write down everything I would have wanted. I also don’t know programming terms in English since I have taken my programming courses (including the tasks of this course) in Finnish, and therefore might mix up some terms, apologies for that. I am happy to answer any questions if this report is unclear in any part.

2. User interface

The program is launched from the beginning of CalendarGUI and it starts from the weekly view (appendix 1) which shows the current week from monday to sunday. All of the weekdays are represented by a column, in which the events of the corresponding day are listed below each other. Additionally there is a column for the reminders of the week. The user can switch the week they're viewing from the "<" and ">" buttons one week forward or backwards. The shown events are highlighted with the colour of their category. The weekly view contains also a menu button, from which the user can choose to change the view to the daily view, filter events, add events, search for events with a keyword or delete events.

The weekly view contains a "+" button, from which the user can add an event. The add event view (appendix 3) contains text boxes where the user writes down the name of the event, starting date and ending date in format dd.mm.yyyy, starting time and ending time of the event in format hh.mm. If the event is all day, the user can write "allday" as the starting and ending time of the events. The user can choose from the ready categories (work, school or hobby) in a multiple selection box, but they can also choose "add own category" and write the name of that new category in the text box below, and this new category will be available next time when adding an event. There is also a checkbox where the user can choose whether they want to be reminded of the event. In this case, a text "remember to do the task x" will appear in the weekly view for the corresponding week. There is a "save" button which adds the event to the calendar and a "cancel" button that sends the user back to the weekly view. If the required fields are left empty or the input is invalid, the program sends an error message to the user (appendix 8). The view for deleting an event (appendix 4) is very similar to the add event view, except that it only asks for the name, starting and ending date of the event.

The daily view (appendix 2) can be accessed either from the menu or by clicking the corresponding day header from the weekly view. It shows the hours of the day as rows, and the wanted events from the hour they begin to the hour they end. There is also a box for all day events that show events marked as "allday", but also overnight events that last the whole day. And again, each event is highlighted with the color of its category. In this view there is also a go back button, from which the user gets back to the starting view. It is also possible to add events from this view by dragging or clicking with a mouse in the container representing the starting hour. The new event view will automatically appear containing the wanted starting date and hour.

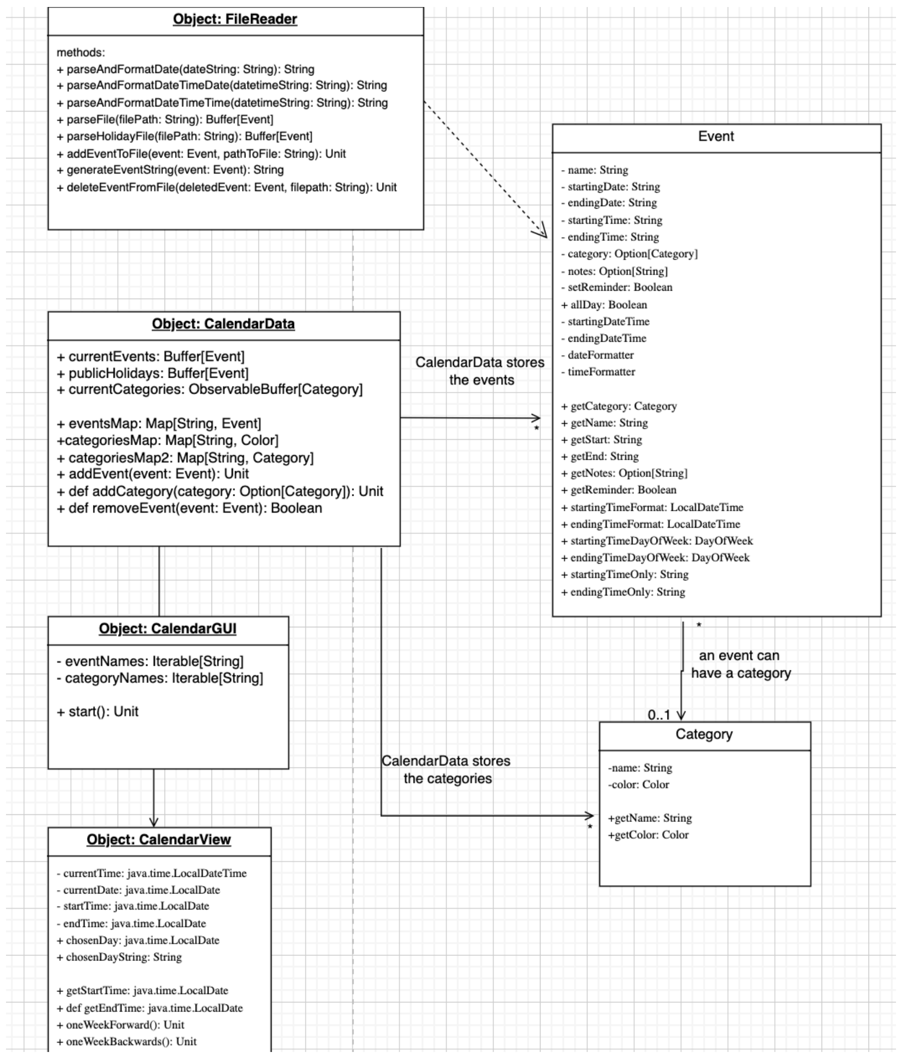
3. Program structure

Since I had never worked with scalaFX before, I didn't understand how to plan a graphical UI, and my initial class structure didn't really work out. The CalendarGUI (called CalendarApp in the technical plan) is the most important part of my program, since it gathers all the information of the other classes and transforms that into an UI. To create the weekly view, it uses the variables and methods in object CalendarView. All of the other data (events, categories) is stored in object CalendarData. When looking at my program structure now, the objects of CalendarView and CalendarData include some irrelevant information, and then again they are missing some information that could have been stored in them to prevent repetition. These two objects should also have been merged, since CalendarView also stores data, so it is kind of useless on its own.

The first class of my program is Event, which takes the parameters of name, startingDate, endingDate, startingTime, endingTime, category (which is of class Category), notes and setReminder. Each event is its own object, stored in the currentEvents or publicHolidays Buffers of CalendarData. The second class is Category, which takes only its name and color as parameters. Every time the user adds a category in the GUI, a new calendar object is created.

Object FileReader reads and writes to the userdata.ics file that stores the events that the user adds in the calendar. It has a method to parse the holiday file that is included in the program and a method to parse other kinds of ics files. These methods return a Buffer with events, which is stored in the currentEvents or publicHolidays Buffers of CalendarData. From there, the CalendarGUI uses these buffers when updating the daily or weekly view, adding or deleting an event, and filtering or searching events.

Please note that I have put a lot of time and effort into commenting on my code, and the comments explain the best what each part of the program does.


Object: FileReader
methods:
+ parseAndFormatDate(dateString: String): String
+ parseAndFormatDateTimeDate(datetimeString: String): String
+ parseAndFormatDateTimeTime(datetimeString: String): String
+ parseFile(filePath: String): Buffer[Event]
+ parseHolidayFile(filePath: String): Buffer[Event]
+ addEventToFile(event: Event, pathToFile: String): Unit
+ generateEventString(event: Event): String
+ deleteEventFromFile(deletedEvent: Event, filepath: String): Unit
Event
- name: String
- startingDate: String
- endingDate: String
- startingTime: String
- endingTime: String
- category: Option[Category]
- notes: Option[String]
- setReminder: Boolean
+ allDay: Boolean
- startingDateTime
- endingDateTime
- dateFormatter
- timeFormatter
+ getCategory: Category
+ getName: String
+ getStart: String
+ getEnd: String
+ getNotes: Option[String]
+ getReminder: Boolean
+ startingTimeFormat: LocalDateTime
+ endingTimeFormat: LocalDateTime
+ startingTimeDayOfWeek: DayOfWeek
+ endingTimeDayOfWeek: DayOfWeek
+ startingTimeOnly: String
+ endingTimeOnly: String
Object: CalendarData
+ currentEvents: Buffer[Event]
+ publicHolidays: Buffer[Event]
+ currentCategories: ObservableBuffer[Category]
+ eventsMap: Map[String, Event]
+categoriesMap: Map[String, Color]
+ categoriesMap2: Map[String, Category]
+ addEvent(event: Event): Unit
+ def addCategory(category: Option[Category]): Unit
+ def removeEvent(event: Event): Boolean
CalendarData stores the events
*
*
an event can have a category
0..1
Object: CalendarGUI
- eventNames: Iterable[String]
- categoryNames: Iterable[String]
+ start(): Unit
CalendarData stores the categories
*
Category
-name: String
-color: Color
+getName: String
+getColor: Color
Object: CalendarView
- currentTime: java.time.LocalDateTime
- currentDate: java.time.LocalDate
- startTime: java.time.LocalDate
- endTime: java.time.LocalDate
+ chosenDay: java.time.LocalDate
+ chosenDayString: String
+ getStartTime: java.time.LocalDate
+ def getEndTime: java.time.LocalDate
+ oneWeekForward(): Unit
+ oneWeekBackwards(): Unit

UML diagram explaining the classes and objects of my calendar

Since most of my code is inside the start function, the program is not covered fully in the UML. This is why I have made a list of the most important parts of my start function and what they do below.

3.1 Methods and variables inside the start function of the CalendarGUI

All of these methods are inside the CalendarGUI.start() function, but since they are a key part of my program, I wanted to explai them a bit more in detail, even though they are not technically a part of the methods of the CalendarGUI object. More information about these can be found in the comments of my code.

- updateEventsOnWeeklyView: calls for the method “updateEventsOfWeek”, giving the wanted events as a parameter. These events can either be all of the events in the calendar, or filtered based on the category.
- updateEventsOfWeek: actually updates the wanted events to the GUI.
- categoryNamesPlusShow: returns a collection with all of the category names and a “show all” string for the choiceBox for when the user wants to filter events
- makeEventsList: when the user searches for events with a keyword, this function updates the list view based on that keyword
- updateDailyHeaderText: updates the header of the daily view based on the date and weekday that the user is looking at
- updateEventsOnDailyView: updates the daily hour slots to contain the labels of the correct events
- updateDailyView: calls updateEventsOnDailyView with the correct date that the user has chosen
- updateLabels: updates the weekday labels in the weekly view
- categoryNamesPlusNew: returns a collection with all of the category names and “no category” and “new category” strings for the dropdown box in the add event view
- updateAddEventWindow: clears the name, category text and checkbox of the add event view

All of the nodes of the GUI are stored in their own variables, which is the easiest way to keep in track what each part does, since these nodes are put in the right parts of the grid “manually” and the contents of these elements are changed based on the weekday, hour etc. These variables can represent container boxes, buttons, dialogs, labels or images, to name a few. Therefore I don’t think it is necessary to go through them one by one, especially since I have tried to comment on the code to explain what each part of the code does.

3.3 Justifications and alternatives

In the project plan my intention was to create a trait for the calendar’s view, so that the daily, weekly and monthly views would all be classes. This way each day/week would be represented by an object that would store the information that the view needs. However, when I started to write the code I realized that this would not be the most effective way to do it: I would have to make hundreds or even thousands of these daily/weekly view objects. After realizing this, I still left the CalendarView class, but then I understood that the best way to conduct the weekly/daily view was that there was only one view, whose labels are just updated based on the user's choices. After this CalendarView was turned into an object. When looking back, I think that the whole CalendarView is unnecessary, and the information it contains could be just stored in the CalendarData object.

In the original plan Event was not a class but a trait, there were also separate classes for Holidays and RepeatingEvents. However, as I didn't do the difficult task in the end, the RepeatingEvents (as well as MonthlyView) was left out of the program. Holidays then again were just normal all day events. Since these classes were not executed, I made Event a class instead of a trait. When reflecting back, the decision to make events take these parameters was a really good choice: this way new Event objects that contain all of the necessary information can be easily created from the file and imported to the file. Since all of my other objects / classes use or create Event objects, they also often need to be able to access or communicate that information

After my original plan for the class structure didn't work out, I should have spent more time to plan it over. Now I just started to write code in the start function, and in the end it became very "heavy". If I would do the project again, I would definitely break down the GUI to classes and smaller pieces instead of having everything inside the start function.

4. Algorithms

The algorithms of my program are not very mathematical, since the Java Time library parses the dates and times so well. The library also has methods for comparing times and therefore there are almost no mathematical formulas needed. My algorithms mainly involve iterating over events, filtering them based on criteria, and updating the UI accordingly. They also ensure data validation, correct display of events on different views, and handling user interactions (such as adding new events or searching for events by keyword). Other possible solutions could involve different data structures or algorithms for event filtering and UI updating, but the chosen approach seems efficient and straightforward.

When the user clicks “save event”, an algorithm (CalendarGUI, row 747) checks if any required fields are empty and validates the time format for start and end times. It ensures that the end date/time is not before the start date/time. Additionally, it adds a new category if specified and checks if there are notes for the event. Finally, it creates a new Event object with the provided information, adds the event to the file, updates the calendar data, and updates the views accordingly. This algorithm provides a straightforward and step-by-step process for saving events, ensuring data validity and consistency before updating the calendar data and views. However, the algorithm may become complex and lengthy if more validation rules or features are added, which could lead to maintenance challenges and decreased readability. An alternative approach could have been to break down the saving process into smaller, more modular functions or methods, each responsible for a specific validation or action, and I would have done this if I had more time.

When the user clicks the daily view open, an algorithm (CalendarGUI, row 306) clears all containers for each hour of the day and filters events based on whether they occur on the chosen day. It then iterates over those filtered events, determining their starting and ending times and setting the category color for the event label. Event labels are added to appropriate hourly containers based on their start and end times.

When the user searches for events based on a keyword, an algorithm (CalendarGUI, row 214) first clears the ”list view” from previous content, then adds the header and buttons back. Then it searches for events containing a specific keyword in their notes and displays a message if no events matching the keyword are found. For events matching the keyword, it creates labels and adds them to the list box.

When the weekly view is updated (for example when the user wants to see the previous / following week, after an event has been added / deleted or after the user has filtered events), an algorithm (CalendarGUI, row 126) iterates over all events, determining their starting and ending dates, as well as the category color for each event. It then iterates through each day between the starting and ending dates of the event, adding event labels to the appropriate day of the week container. If the event has a reminder which has not been added before, a reminder label is also added.

These three algorithms (updating the daily and weekly views and searching the event for keywords) have all the same upsides and downsides. For a small program, these implementations are easy to read and understand, but they would become problematic if the volume of events would increase, since the program could become very slow if it would always have to go through and filter all of the events from a buffer. Alternatively, implementing a more efficient data structure could improve the performance, but unfortunately I don't yet know how to use them.

5. Data structures

I use four types of collections in my code. The most important collections of my code are Buffers that store the events that are currently saved in the calendar. When the program is opened, Buffers are used to store events parsed from the calendar files, but almost all of the functions that add/change/delete events use these Buffers. Therefore I chose to use Buffers for storing events because they are mutable and they don't have a fixed size, making it suitable for storing a variable number of events parsed from the file. The number of events is changing constantly as the user can add and delete events as they please.

Alternatively, Arrays or Vectors could have been used to store events, providing fast access to elements. However, due to their fixed size, they would have posed challenges when adding or removing events constantly. Lists could serve as a dynamic alternative to Buffers for event storage, offering flexibility in size adjustments. However, my familiarity with Lists is limited compared to Buffers, so I decided to use Buffers.

The second data structure that I use is Map. It is used so that with the name of a category as a key, it is possible to get hold of the actual category / the color of the category. This is needed when new categories are added and especially when the GUI needs to retrieve the color of labels based on category name. There is also a Map where the key values are a string with the name, start date and end date of an event and the matching values are the actual event objects. This is used to delete events based on their names and times. I selected to use this structure for its key-value pairing, which allows efficient retrieval of categories / events based on Strings that the GUI uses.

I also use Arrays to store the containers for daily view in the GUI, and the functions that update the GUI with events / reminders use these Arrays. I used them for their simplicity and ease of access, since they provide a collection of fixed size, which suits the containers since there are always 25 of them (24 hour containers + 1 all day container). This way I didn't have to manually copy and paste the code for 25 containers.

In my FileReader I also use Lists to store lines of the ics files for efficient modification and updating. Each line of the iCalendar file is saved as an element to the List as a string. The methods of FileReader then retrieve these lines, finding the index of a specific line, and patching the list to insert or replace lines. Here I used Lists due to its mutability and efficiency indexing for changing the lines during file updates. I tried at first to use a Buffer here, since I am more familiar with using them, but for some reason it didn't work so I found List to be the best mutable alternative.

6. Files and Internet access

My calendar program deals with iCalendar (ics) files, which are basically text files but in a specific format. In my calendar program, the data of the calendar is always stored to userData.ics, from which the FileReader object parses the file and creates Event objects that are seen in the GUI. When an event is created in the program, the FileReader object automatically creates a String representing an event in ics format and stores it in userdata.ics.

An ics file always starts with `BEGIN:VCALENDAR` and ends with `END:VCALENDAR`. After the starting line, there are lines that specify the basic information of the whole calendar program, such as `VERSION:2.0`, `PRODID:-//OS2//CalendarbyRoosa//EN`, `CALSCALE:GREGORIAN` and `METHOD:PUBLISH`. After these lines, events can be created.

Each event starts with `BEGIN:VEVENT` . and ends with `END:VEVENT`. In between, there are a couple lines that every event has to have so that the ics file is valid: `UID:` represents the user's ID. In my calendar, the user id is always user1234, since this is required. Another required line is `DTSTAMP:`, which stores the time when the event was added. However, neither of these values are ever used in my calendar, so they could be anything. On top of these, the iCalendar files can have tens of different qualities. The ones that my program handles are DTSTART, DTEND, SUMMARY, CATEGORIES & DESCRIPTION.

The line that starts with DTSTART contains the date and time when the event starts. This can be of multiple different formats, however my program only handles the most commonly used forms. If the event is allday, the line should start with DTSTART;VALUE=DATE:and be of format yyyyMMdd. If the event is regular, the starting time should be of format yyyyMMdd'T'HHmmss'Z' (so for example 12.4.2024 at 12.30 would be 20240412T123000Z). The same logic applies for DTEND, which represents the end of the event. The line SUMMARY contains the name of the event, CATEGORIES the possible name of the category and DESCRIPTION the possible additional notes of the event.

I am aware that handling the ics files as text files and parsing it as Strings sets certain limitations to what it is possible to do with my program. In some iCalendar files, the start / end texts can be of different formats, and my program doesn't currently take that into account.. I heard that some other students were using the iCal4j Java library that helps reading and writing iCalendar data streams. It would have provided an API for parsing, generating, and manipulating iCalendar data, making it easier to work with different kinds of calendar information. However, at this point I had already got my FileReader to work, and it would have been too risky to change it that late in the process. This is too bad, since I believe that using the iCal4j would also have spared me a lot of time and effortl since writing the algorithms for parsing the files was very difficult and time consuming.

7. Testing

The most significant part of the tests were made and run during the programming. Always when adding a feature, I tested it as thoroughly as possible both in REPL and in the GUI if possible. Especially when testing parts where the user can write any fre input, I tested the new features first as they were intended to be used, but always also with invalid inputs. This way I learned how my program reacts to other ways of using it than intended. Here I also learned that I should make good error messages, which I implemented in the later stage.

I had a separate class for testing, which I used especially in the start when I trained on how to parse the Java Time dates and times to the right format. I also had test files which I used to see how my calendar would react to importing ics files from my Google / Apple calendar. I used unit testing especially when testing the FileReader object. These tests can still be found from the end of that file, and they were maybe the most useful when debugging the problems in my FileReader.

To ensure the usability and get some "outside the box" opinions of my calendar application, I did user testing sessions with friends and family members. I observed as they used the calendar, noting their behaviors, feedback, and especially the challenges they encountered. By involving users who don't understand anything about programming, I identified some usability issues, such as confusing interface elements: after these tests I changed the reminder box to a different color than the weekday containers, and I also removed unnecessary elements from the menu bar.

The planned testing in my project plan involved continuous testing during code development, unit tests for critical methods and system testing through the user interface and test classes. The focus on the testing plan was to ensure functionality, error handling, and usability through user feedback. However, in execution, testing was mainly integrated into the programming process, with testing of new features in the REPL and GUI. While unit testing and file management testing were conducted as planned, user testing was added, which was useful but also fun to do. Overall, while I did the testing pretty much according to the plan, the execution was definitely more iterative: originally, I had planned to spare a lot of time in the end just for testing, but in reality this testing happened mainly continuously at the same time as the programming. Additionally, broader testing of file import functionality and broader could have provided deeper understanding potential improvements.

8. Known bugs and missing features

Unfortunately, due to time running a bit out, some issues and bugs in the program are still unresolved. One significant problem is that reminders are not saved to the file, which means that when importing the ics file to other calendars or when restarting the calendar application, the reminders are not shown. I believe that this would have been very straightforward to implement by using the alert property in the.ics files, but I just didn't have time to implement it.

Additionally, the filtering feature only displays the pre-existing categories (work, school and hobby) and does not include categories added later by the user, which limits the customization options of the user. I also tried to implement comprehensive error handling / error messages, but it is still possible to add events with end times earlier than their start times, which indicates a bug somewhere in the added event validation algorithm. The error message handling is implemented with various if loops, so I think that these if loops are still missing some possible error scenarios.

The bug that makes me the most upset is that importing calendar files can still be a bit unpredictable, even though I have spent so much time on the FileReader object. At times all of the events are imported successfully to other calendars, while other times it's only a part of them. Like I have mentioned previously, the program only handles specific types of ics files, limiting its compatibility with other formats, and I think that this is the problem here. Integrating the iCal4j reader could help this problem. But because my FileReader handles the lines in the file as normal strings, trying to address all kinds of scenarios would make the code very complex as I would have to add so many different if loops.

Lastly, the functionality to add events by selecting and dragging over time slots does not work properly, as events are currently added by clicking rather than dragging in the daily view. I added this functionality as the last, so when it didn't work, I didn't have time to debug it properly.

9. 3 best sides and 3 weaknesses

The best side of my code is probably the design of my GUI: I have put a lot of effort into designing and implementing the weekly and daily views, and done a lot of extra work with that: for example importing the icons and images, custom fonts and colors. Even though this was "extra work" I found it really rewarding and interesting, as I am interested in design and usability.

Another good addition to my program is the error messages that I implemented. While the instructions only mentioned error handling, I customized five different error messages for different kinds of invalid inputs, which improves the usability of my calendar a lot. Also, unlike many others who used the day picker provided by ScalaFX, I opted for text input fields as the instructions stated, despite the additional effort required. I hope this decision will be acknowledged as it aligns with the course instructions.

The first weakness of my code is that it contains a lot of repetition especially in the CalendarGUI and I acknowledge that. Especially in the early stages of development, when scalafx was new to me, I added functionality manually and repetitively to help myself understand the task at hand. I intended to refactor and condense the code later on, but ran out of time. This would be the first thing that I would change if I had the chance to improve my code. I would use loops to condense for example lines 572-653. Luckily I managed to already refactor some of my code: for example, I originally created all 25 "dailyTimeBoxes" and "dailyContentBoxes" individually, but later condensed them into lines 282-286. Another area in my code with too much repetition is the FileReader class, where I initially implemented a method for reading holiday files as a practice exercise in file handling. However, when I implemented the function that reads a basic ics file, I couldn't use it directly but I didn't want to risk accidentally "breaking" my well-working code so I chose to make a very similar function with a lot of repetition.

Another weakness in my code is the heaviness of the start function, which is really packed with code, making it challenging to extend or develop further functionalities. Despite this apparent overcomplexity, this issue wasn't ever addressed during sprint meetings, where I would have liked to hear more feedback if "critical" or bigger structural problems were noticed. Breaking down the start function in further development could make code readability, maintainability, and scalability better, enabling easier development in the future.

The third weakness of my code has to do with the reading and writing of files, since my program only deals with certain types of iCalendar files, but this I have discussed more thoroughly in previous chapters.

10. Deviations from the plan, realized process and schedule

My third period was really intense, as I am taking a lot of courses and working part-time. This is why I didn't do anything but the first draft of the GUI in March. At the start of the fourth period I left for a two-week holiday where I didn't do any school. This is why I have basically made the whole project during the last three weeks, and this is why I have run a bit out of time. My original plan for the schedule was very idealistic, with a lot of time for testing and perfecting. I would have loved it to work but I have been too busy for that. This is also why I had to drop my plan to do the difficult task, as I did not have time to implement those extra features that I had planned.

However the order of executing the tasks was kept about the same: first, I did a dragt of the GUI, then I implemented my first classes, then the most important functions, then the rest of the methods, and in the end I did testing and debugging. What differed from the original plan was that I did the file handling only after adding most of the functions, variables and data structures. This was because I didn’t know anything about file handling and it felt really scary, so I wanted to start with something familiar. This worked out surprisingly well, and I didn’t have to change too much of the original code after implementing the file handling. However, the biggest change from my plans was that the project turned out to be a lot more iterative than I had thought.

I hope my poor time management skills won't negatively impact the evaluation. Despite my failure to stick to my plans, I made both of the plans carefully and with time: my biggest issues were my overly optimistic and perfectionist visions. If I redid the project, I would prioritize starting early and planning more effectively and leave more time for finalizing and testing the project. However, I have learned a lot about coding throughout the project, particularly in finding information on my own and assistance when “running into a wall” and navigating solo project management.

11. Final evaluation

I hope that the final program demonstrates the considerable effort and learning that I have made throughout its development. The strengths lie in the comprehensive error handling, thoughtful user interface design, and iterative approach to testing and refinement.

The weaknesses of my program emerge especially in the heavy and somewhat cumbersome structure of the code of my CalendarGUI, particularly in the start function, which could do some further development. Additionally, while the iterative nature of the project allowed for flexibility and adaptation, the execution deviated a lot from the original plan, which made the balance between different tasks and the overall workflow somewhat impossible.

In the future, the program could be improved by refactoring the codebase to reduce redundancy and enhance maintainability. Simplifying the start function and restructuring the class hierarchy could streamline development and facilitate easier extension of features. Moreover, adopting more efficient data structures and solution methods, such as using iCal4j for handling calendar files, could enhance the program's flexibility and compatibility with different file formats.

If I were to start the project again, I would prioritize better time management and planning to ensure a smoother workflow. I would plan the structure of the program a lot better now that I understand how programming a graphical user interface actually works. Beginning with a clearer roadmap and breaking down tasks into smaller, more manageable chunks would help minimize the risk of heavy code and enable more effective testing and debugging processes. Additionally, I would aim to explore alternative solution methods and data structures earlier in the development process, because it is hard to change these things when so much code has already been written.

12. References and code written by someone else

Especially in the beginning of the project, when learning for the first time about these topics (iCalendar files, Java / Scala time, scalafx and javafx) I spent a lot of time just googling these topics and surfing different websites searching for information. However I didn't copy any code straight from these websites, since it didn't really work in my context without modifying. However, I will still link the websites where I found models and inspiration below.

In my project, I used Java time a lot for managing dates and times. Specifically, I used Java's LocalDate and LocalTime classes to represent dates and times without time zone information, which suited the requirements of my use. These classes allowed me to parse, format, and manipulate dates and times easily throughout the program. Additionally, I used DateTimeFormatter to specify custom date and time formats when parsing strings into LocalDate and LocalTime objects, ensuring consistency and accuracy in date and time representation. Overall, Java's time-related classes provided an easy and convenient framework for handling time data in my project.

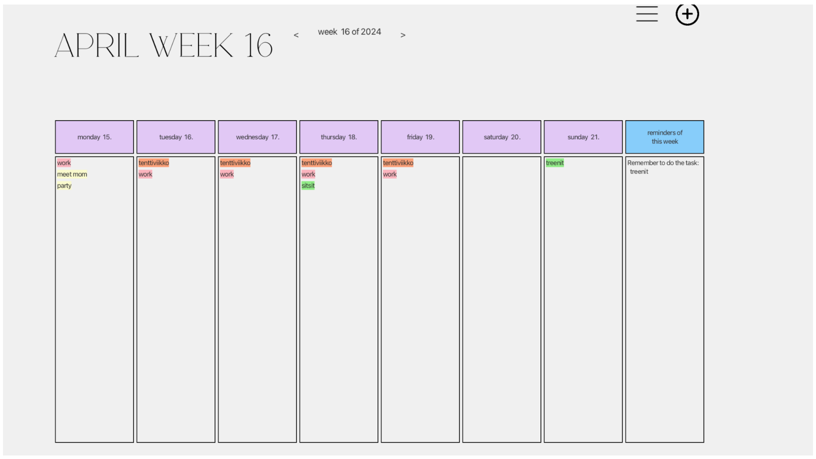
I used a lot of ChatGPT for debugging and ideation purposes. If my code didn't work and I couldn't figure out why, I fed it to ChatGPT and explained the context and asked it to give me ideas on where the problem could be. Often this worked, especially if the problem was a typo or if my code was in the wrong format etc. However, I never made it do any code for me: as a programming assistant myself, I can pretty quickly see if code is produced by ChatGPT, and I believe that I am capable of writing better code with my own brain.

12.1 Websites and references

- ScalaFX: Dialogs and Alerts https://www.scalafx.org/docs/dialogs_and_alerts/

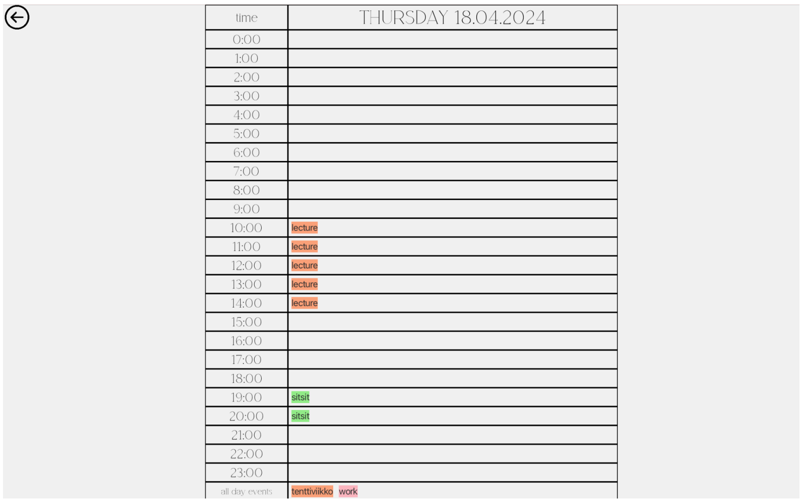
- iCalendar website: https://icalendar.org/Home.html
- Baeldung. (30.11.2023). Working With Dates and Times in Scala. Available at: https://www.baeldung.com/scala/date-time
- Wikipedia. (30.11.2023). iCalendar. Available at: https://en.wikipedia.org/wiki/ICalendar

13. Appendices

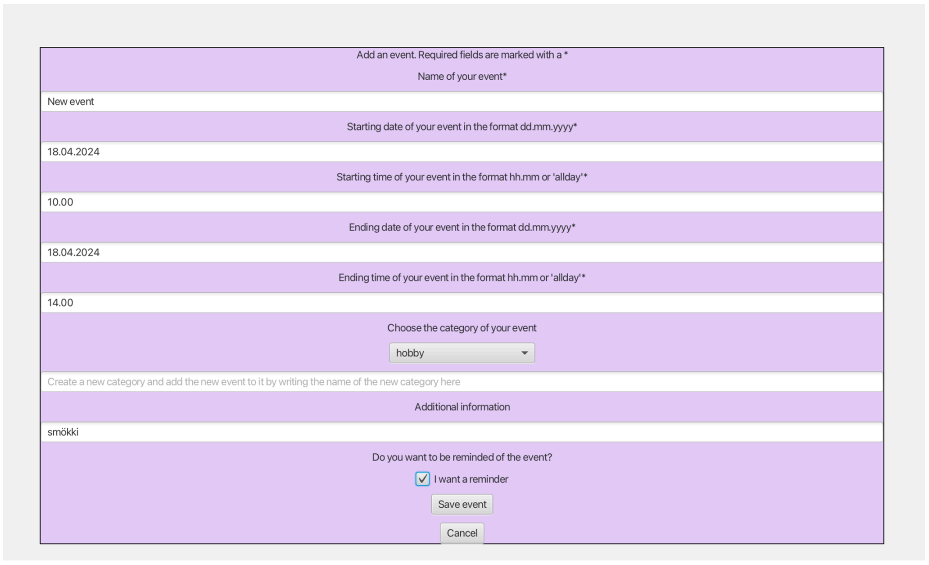


APRIL WEEK 16 < week 16 of 2024 >

monday 15.	tuesday 16.	wednesday 17.	thursday 18.	friday 19.	saturday 20.	sunday 21.	reminders of this week
work meet mom party	tenttiviikko work	tenttiviikko work	tenttiviikko work sitsit	tenttiviikko work		treenit	Remember to do the task: treenit



time	THURSDAY 18.04.2024
0:00	
1:00	
2:00	
3:00	
4:00	
5:00	
6:00	
7:00	
8:00	
9:00	
10:00	lecture
11:00	lecture
12:00	lecture
13:00	lecture
14:00	lecture
15:00	
16:00	
17:00	
18:00	
19:00	sitsit
20:00	sitsit
21:00	
22:00	
23:00	
all day events	tenttiviikko work



Add an event. Required fields are marked with a *

Name of your event*

New event

Starting date of your event in the format dd.mm.yyyy*

18.04.2024

Starting time of your event in the format hh.mm or 'allday'*

10.00

Ending date of your event in the format dd.mm.yyyy*

18.04.2024

Ending time of your event in the format hh.mm or 'allday'*

14.00

Choose the category of your event

hobby

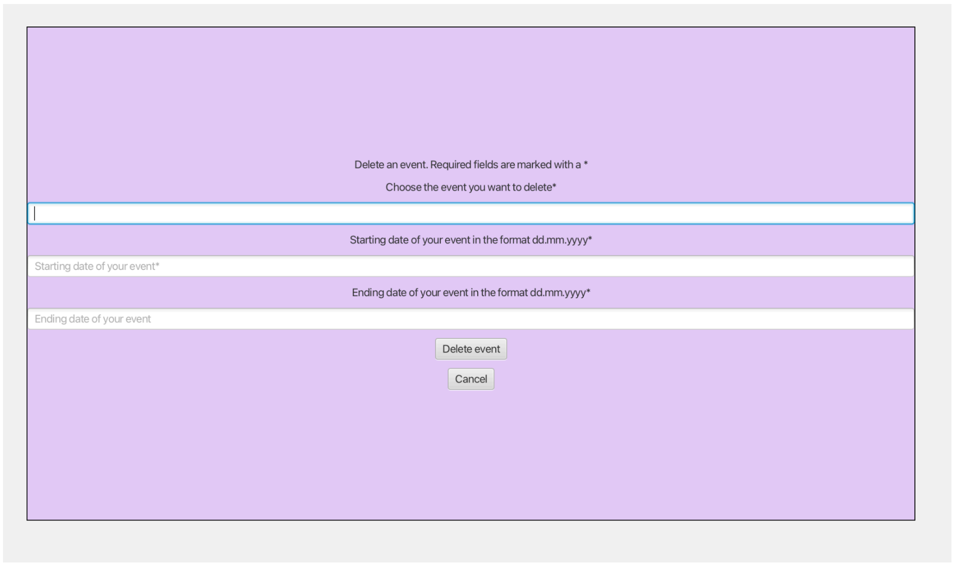
Create a new category and add the new event to it by writing the name of the new category here

Additional information

smökki

Do you want to be reminded of the event?

[✓ I want a reminder]



Delete an event. Required fields are marked with a *

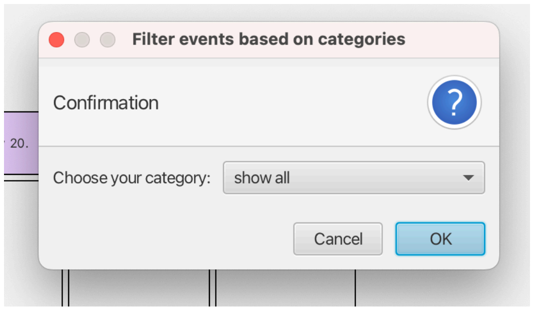
Choose the event you want to delete*

Starting date of your event in the format dd.mm.yyyy*

Starting date of your event*

Ending date of your event in the format dd.mm.yyyy*

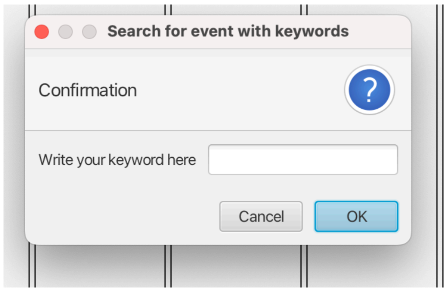
Ending date of your event



Filter events based on categories

Confirmation

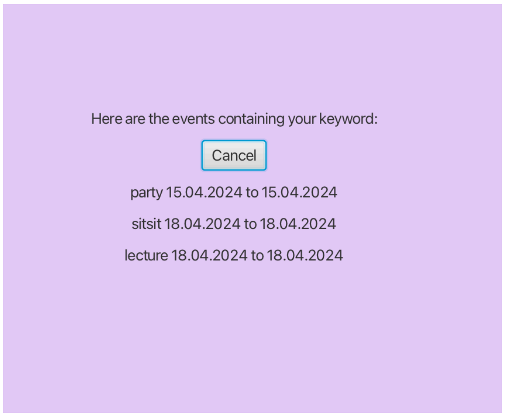
Choose your category: show all



Search for event with keywords

Confirmation

Write your keyword here

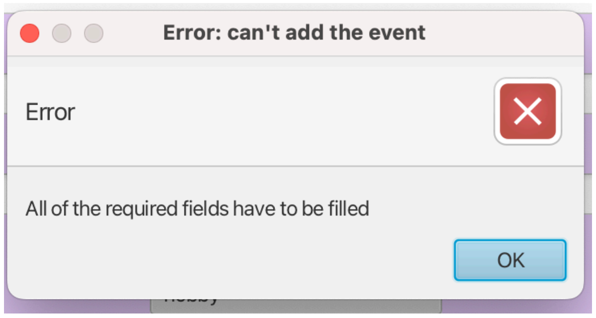


Here are the events containing your keyword:

party 15.04.2024 to 15.04.2024

sitsit 18.04.2024 to 18.04.2024

lecture 18.04.2024 to 18.04.2024



Error: can't add the event

Error

All of the required fields have to be filled