Create a system for note taking, management and synchronization. The design should be based on an analysis of existing solutions addressing their main shortcomings. It should clearly separate notes management and storage from user interface so different user interfaces could be built allowing multiple different user interfaces to access the same notes storage. The core functionality will include:

- notes storage and organization
- notes taking using markdown format together with rendering using appropriate graphical user interface
- notes indexing with full text search support
- indexing of to-do items within notes
- notes synchronization allowing one to access notes on multiple devices

The implementation should be realized with appropriate web technologies following the standard software engineering best practices including testing and documentation.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Shere - Notes and Document Management Application

Bc. Marek Foltýn

Department of Software Engineering
Supervisor: Ing. Filip Křikava, Ph.D.

May 8, 2018

Acknowledgements

I would like to thank my supervisor Ing. Filip Křikava, Ph.D. for his consultations and ideas during the work on the thesis. I want also thank my wife Veronika for her support, my daughter Štěpánka for preventing me from sleeping too long and my whole family. Last, but not least, I would like to thank all teachers at FIT CTU for their enthusiasm and knowledge that I could gain from them.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 8, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Foltýn, Marek. *Shere - Notes and Document Management Application.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato diplomová práce se zabývá tvorbou aplikace pro psaní poznámek. Součástí práce je analýza existujících aplikací pro tvorbu poznámek a rozbor jejich nedostatků. Návrh implementované aplikace je vytvořen na základě této analýzy a řeší nalezené nedostatky.

Klíčová slova poznámky, markdown, editor, parsování, indexování, fulltextové vyhledávání

Abstract

The main purpose of this thesis is to create a note taking application in order to improve note taking experience. The thesis contains the analysis of existing note taking software, the discussion of their shortcomings and also the implementation. Its design is based on the analysis and addresses the main shortcomings.

Keywords note taking, markdown, editor, parsing, indexing, full-text search

Contents

List of Figures

List of Tables

Introduction

People write down notes to free their minds. They need to keep ideas, tasks or other information in order to reach their work goals. People can write down their notes on paper or post-it sticky notes. When they require notes to be saved in a digital form, computers and mobile devices can be used. There are many applications usable for creating notes: simple programs such as Notepad, generic word processors (for example Microsoft Word) or specialized software for managing notes and documents such as Evernote. Note taking applications offer different features for writing and organizing notes, support different platforms etc. Because that users do not have the same note taking styles, they require different of features for work with notes and thus they feel comfortable with different applications.

This thesis is focused on specialized note taking applications. It analyses existing note taking software based on a set of defined criteria. The main goal of the thesis is to create a note taking application based on this analysis. Chapter 1 defines a set of criteria for evaluating note taking applications. Chapter 2 analyses existing note taking applications using the criteria, describes their features, advantages and disadvantages. The analysis will be used for creating a set of requirements for the software project *Shere* in Chapter 3 which describes the implementation of the project.

My personal motivation to this topic is that I could not find appropriate note taking application offering minimalist non-disturbing user interface, quick note formatting and easy notes organization, but efficient with large number of notes. I spent a long time comparing note taking software and searching for the best application. Based on the comparison and found disadvantages, I decided to create my own application.

Chapter 1

Note Taking Evaluation Criteria

To be able to choose or create a note taking application, it is necessary to describe how users work with notes and which features help them to manage their work efficiently. To create a note in general, various data types can be used: rich text documents, web page links, media files, tabular data, calendar events, tasks etc. Each type of data has its own purpose and format. In real world, data with different types can be related to each other, but also can be stored on different locations (local computer, web page, cloud, ...). Users need to store or to reference these documents to keep them easily reachable to stay oriented in their work. One of the features that differ note taking applications from writing notes on a paper is that they allow linking other notes and different data types. They serve as a *digital brain* for digital data. Some applications even extend this abilities with sharing the notes with other users which is difficult using paper or sticky notes.

When searching for the optimal note taking application, users need to consider many aspects of the software such has what data types can be stored in notes, how are note organized, which platforms are supported, how does the application look and how comfortable it is for them etc. Also, users have to consider the type of their work and their personal habits.

This chapter defines a set of evaluation criteria for comparing note taking applications. It compares software attributes of applications that can be limiting for some users, but beneficial to others and thus help to decide better. These criteria will be used for comparison of several existing applications in Chapter 2.

1.1 Note Workflows

The first aspect of note applications is the support of *note workflows*. I divided how users work with notes and documents into four workflows:

- *creative workflow*

- *data workflow*
- *agenda workflow*
- *sharing workflow*

This categorization helps to understand different note taking activities. As described further in 2, applications have different levels of support these workflows. Each workflow requires a different set of features, but multiple workflows are usually used together.

Creative workflow stands for creative activities. Users write down their ideas to keep them outside of their brain and thus freeing their mind. It helps them remember things about actual activity, stimulates creativity and helps think about difficult problems. Some complex ideas require some form of visualisation or linking among other ideas. Typical features suitable for creative workflow are:

- Text formatting (highlighting, colors, bold, ...)
- Drawings (diagrams, sketches, ...)
- Links to other notes

Data workflow is mainly related to knowledge. Notes store information gained from an external sources: files, web links, personal notes about a science book, etc. There is a difference between *including* and *linking to* the data. When including, the external data become the part of the note. A good example is inserting an image into a note or writing notes during a lecture. When linking to a PDF file with presentation of the lecture, the file's location is not changed. Both ways have advantages and disadvantages and are suitable for different use cases. Features supporting data workflow include:

- Links to other notes.
- Web links
- Images
- File attachments

Notes combining creative and data workflows are very common. Users often write ideas or knowledge and link them to other existing data.

Notes created using *agenda workflow* contain organizational content: tasks and events. For users using organization data only, a calendar app is more suitable than note taking app, but in a combination with idea and data workflows, agenda in a note taking application can become very powerful productivity tool. Tasks can be part of the note, they can be aggregated and efficiently linked to the data required for the tasks. The main used element is a to-do

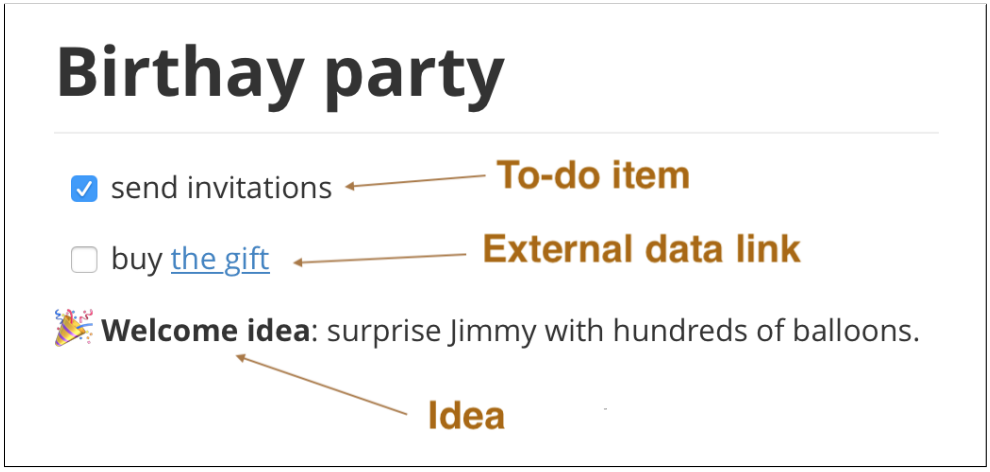


Figure 11: Note using creative, data and agenda workflow

item. A good example of a note combining first three note taking areas is the note in Figure 11.

The fourth workflow is *sharing workflow*. It contains activities and allowing sharing notes with other devices or people. We can split notes sharing into two different ways. The first is *personal sharing*: users want to access notes on multiple devices. This is commonly known as *synchronization*. The second way is sharing with other people. Users might want to share notes for example when they want to collaborate with team members, they want to publish it as a part of a project etc. Features that support sharing workflow include:

- Note synchronization to multiple devices
- Export a note to a file (PDF, HTML, Markdown, ...)
- Share a note via web link
- Realtime collaboration

Sharing workflow is conceptually orthogonal to the other workflows, because all data related to creative, data and agenda workflows can be shared.

Different applications have different support for note workflows. Some applications support one workflow more than others, some try to support each equally. The following list contains examples of applications that supports mainly one single workflow:

- *Ideas*: graphical mind-map application
- *Data*: wiki pages
- *Agenda*: calendar application

- *Sharing*: file synchronization software

Personal note taking applications usually support 2-4 workflows (with different emphasis on each). There are also applications that offer almost equal support for all workflows. This software is usually suitable for teams in companies rather than individuals: knowledge, organization and collaboration systems.

1.2 Notes Organization

When working with large number of notes, it can become hard to keep notes organized. Note taking applications offer different ways of how to organize notes. This criterion evaluates which of them are available. There are two main approaches to organizing notes, in this thesis they are called *tree-organization* and *tag-organization*.

Tree-organization is based on hierarchy similar to filesystem folders: each note belong to a particular folder and a folder can contain notes or other folders. Using folders and subfolders, hierarchical tree structure is created.

Tag-organization allows notes to have a set of some form of a tag: text label, a color tag etc. The main difference from tree-organization is that a note can have multiple tags and thus can belong to multiple sets of notes. Using tags is more flexible for gathering notes across different areas of work that have the same attributes (for example gathering all notes related to solved issues with Android programming across different projects).

There is also a combined way of notes organization, we can call it *nested-tag-organization*. This approach mixes both previous ways: a note can have multiple tags, which can be hierarchically organized. Nested-tag organization seems to be the most universal approach as it unifies two different ways of organization in a very elegant way. Users can choose more precise hierarchy or more

1.3 Search

Search functionality enables faster access to large number of notes. Sometimes users need to find specific information, but they do not remember which notes contain that information. Full-text search engines provide fast way how to retrieve data searching by words. More advanced search engines allows also querying and filtering based on other content attributes.

Generally, fast and precise searching is a very complex problem, there are many algorithms available: primitive sequential search, inverted index [1], advanced fuzzy search [2], attempts to use neural networks [3] and many more. In note taking applications, this evaluation criterion can be limited to check if the simple full-text search is present and if some advanced filtering options can be used.

1.4 Markdown

One of the common features appearing in text editors is text formatting. It allows users to change properties of the text (*e.g.* font size, weight) or create semantic elements such as lists, headings, links, etc. Text editors have different approaches for text formatting support. An example of an application with rich text formatting options is Microsoft Word. In contrast, editors without text formatting support use plain text only (for example Notepad). An approach between these two is to use formatted plain text using special text characters for quick formatting. In order to write formatted plain text, a specification of special characters and its syntax is required. This specification is called *markup language*. The main benefit of markup languages is that they enable quick and consistent formatting.

Markdown[1] is a lightweight markup language created in 2004 by John Gruber [4]. The main design goal of the language is to make text both easy to write and easy to read in plain text and to allow conversion of markdown text to HTML. The syntax is inspired by formatting conventions from email and usenet conversations. After years, many Markdown derivatives were created. They define ambiguous corner cases missing in the original specification or extend the specification with additional elements such as tables, task list items etc. The following list contains examples of popular markdown derivatives:

- CommonMark [5]
- GitHub Flavored Markdown [6]
- GitLab Flavored Markdown [7]
- Markdown Extra [8]

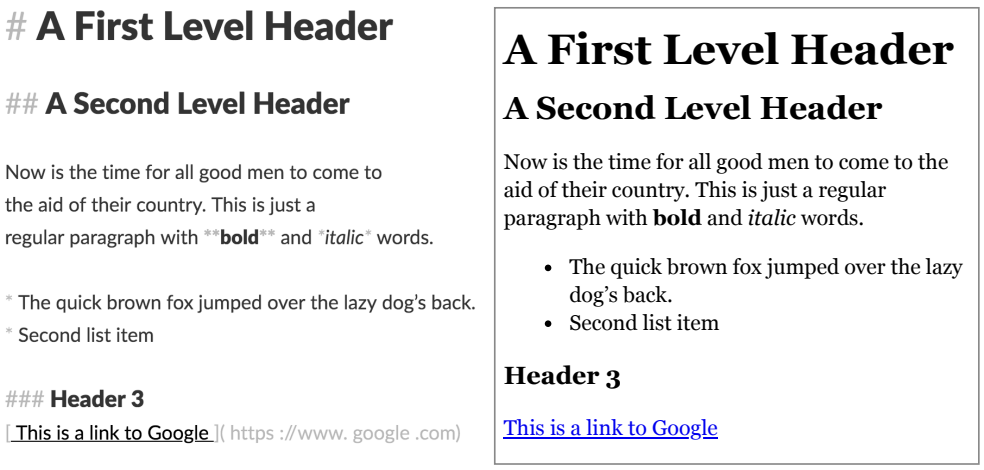
Markdown was initially developed as a text-to-HTML conversion tool for web writers [4]. Figure 12 shows plain text using markdown compared to the same text converted to HTML.

Nowadays, markdown is used by various applications including note taking software. There are different approaches to markdown editing and rendering. We can differentiate four types of markdown editor implementation approaches:

- Side-by-side views
- Syntax highlighting
- WYSIWYG[2] extension

[1] In this thesis, all Markdown derivatives are referenced as *markdown* (with lowercase *m*). Some documents and web pages use Markdown with capital *M* as the reference to the original specification by John Gruber [4].

[2] WYSIWYG is an acronym for *what you see is what you get.* [9]



(a) Markdown formatted text (b) Markdown text converted to HTML

Figure 12: Markdown formatting

- Seamless realtime preview

Side-by-side views is one of the most simple ways how implement markdown editor. There are usually two panes of an editor in an app, one containing plain text and the other showing the formatted text (usually converted to HTML). It is easily implementable as there are only three requirements: a plaintext field, web view and markdown-to-HTML convertor. However, from the user's point of view, seeing the text two times on both sides can be distracting and space-vasting. An example of using side-by-side approach is web application WriteMe [10] shown in Figure 13.

Syntax highlighting is more advanced way how to render markdown. There is only one text view that contain markdown text, which is formatted according to tokens used in the text, especially special characters are usually underemphasized, headings font size is increased and so on. All the special markdown characters are always visible. An example of application using syntax highlighting is StackEdit [11] shown in Figure 14. This application also offers combination of side-by-side view and syntax highlighting: the first pane shows syntax-highlighted text and the second pane shows the text converted to HTML.

WYSIWYG extension allows to use markdown in editors that show formatted text only. Usual behaviour is that after writing special characters, the characters disappear and the format is applied (for example: after writing `**bold text**`, the asterisks disappear and **bold text** is displayed). This approach is beneficial for users that want to see the formatted text only (without special characters). However, there is also an option to format text manually using UI buttons or keyboard shortcuts in case that users do not know markdown syntax.

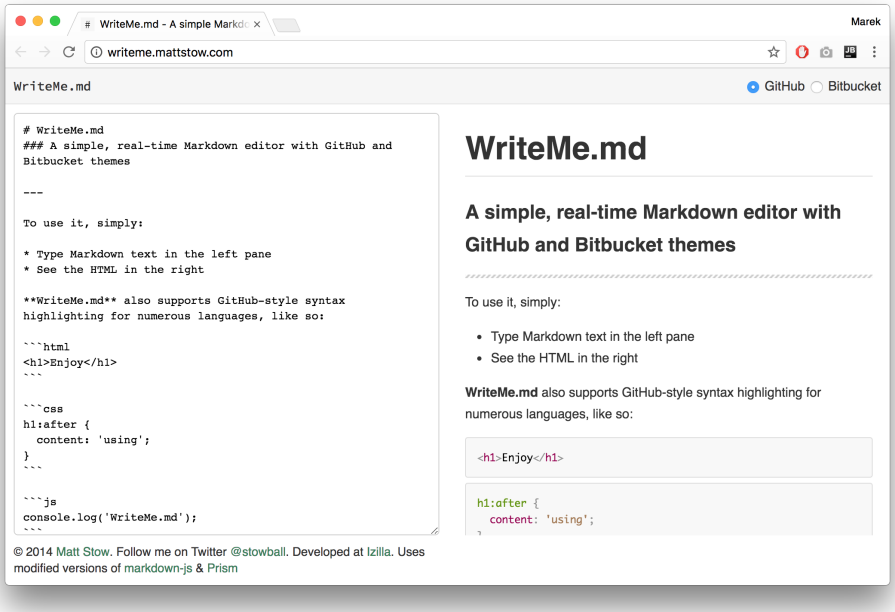


Figure 13: Example of side-by-side approach

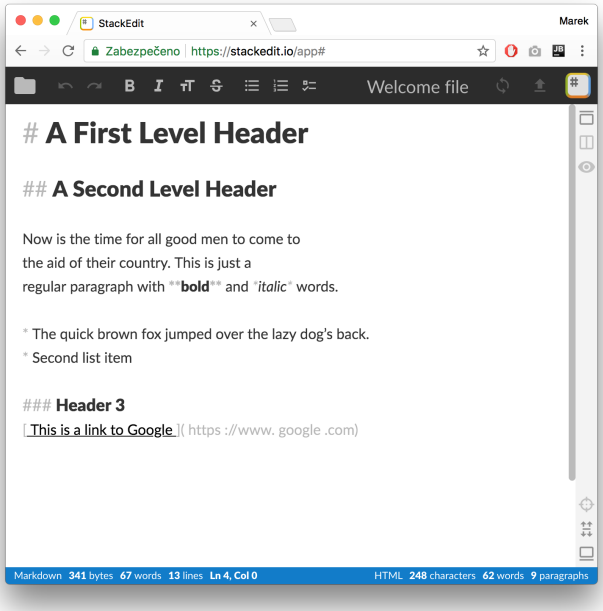


Figure 14: Example of syntax highlighting

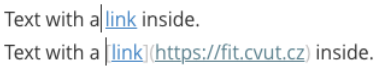


Figure 15: Markdown seamless preview example

Seamless realtime preview combines the last two approaches. Basically, the text is highlighted in the same way as in *syntax highlighting*. The difference is that after cursor is moved away from the text, special characters disappear and only formatted text is visible. Figure 15 illustrates the seamless preview in markdown editor Typora [12]: when the cursor approaches to the link, the markdown link token is expanded and can be edited. After the cursor leaves the link, the URL is hidden and the link text is displayed only.

1.5 Supported Platforms

Supporting more platforms is a big advantage. In combination with synchronization features, it allows users to access their notes using the same application on different devices. On the other hand, developing and maintaining multiple platforms requires significantly more effort. Some applications have different features available in different platforms. This criterion evaluates which platforms are supported and how an application differ on each platform. The following platforms are included into the analysis:

- Windows
- macOS
- Linux
- Android
- iOS
- Web

1.6 User Interface

A proper user interface design is an important part of the user experience with the application. For note taking purposes, this criterion evaluates the aspects of the UI that have impact on work with notes. Firstly, it is described if the UI support users to stay focused on notes. Showing too many UI elements can be overwhelming and can distract users when they need to be focused to write or read notes. Non-distracting interface highlights the actual note content rather than the number of features that the application offers. Secondly, compliance

with platform design recommendations helps users understand the interface by using UI paradigms they already know. Note taking applications are targeted to be used often or every day, so it is important for users to feel familiar with the interface. Thirdly, a part of this criterion is a subjective evaluation note taking experience.

1.7 Notes Storage

This criterion concerns how and where are notes stored. It discusses used data format and location of notes.

Open or proprietary data format can be used. Using open formats allow notes to be accessed and modified outside particular application by third-party software. Open storage structure can be for example: file and folder structure on a filesystem, an open-source database (*e.g.* SQLite) with published data model description. Using proprietary storage structures does not allow direct note storage access. On the other hand, it reduces risk that an external application corrupt the storage. There is always a trade-off between using open or proprietary storage, but using published storage format mitigates the risk of *vendor lock-in*.

The second part of the criteria concerns where the notes are stored. Usually, they are stored in a local filesystem, or in a cloud service. Saving notes in local filesystem allows to access them without internet connection. On the other hand, online storage enables notes to be instantly available everywhere on devices with working internet connection. However, using online-only storage limits users to work without internet connection. The optimal method is combination of online and offline storage: notes are stored locally and also synchronized to an online repository.

1.8 Pricing

Pricing is the last criterion used for evaluating note taking apps in this thesis. It compares pricing models of applications. If the applications offers a free plan, both free and paid plans are compared.

1.9 Summary

In this chapter I described a set of criteria for evaluating note taking applications. These criteria will be used for analysis existing note taking software in Chapter 2.

Chapter 2

Existing Applications

There are many existing note taking applications [13]. In this chapter, I will focus on five representatives that will be described in this thesis using the criteria defined in the previous chapter. Each selected application meets at least one of these requirements:

- It is widely used among users (based on the number of active users or downloads)
- There is an outstanding feature related to one or more evaluation criteria

Table 21 introduces the analysed applications, their authors and project web page.

Name	**Author**	**Web page**
Evernote	Evernote	evernote.com
Microsoft OneNote	Microsoft	onenote.com
Bear	Shiny Frog	bear-writer.com
Dropbox Paper	Dropbox	paper.dropbox.com
Typora	Abner	typora.io

Table 21: Analysed applications for taking notes

2.1 Evernote

Evernote [14] is one of the most popular note taking applications in the world. Besides note taking, it offers simple organizational features and efficient multi-user collaboration. Evernote was launched in 2008. In 2017, it reached 220 million users. [15] Today, Evernote ecosystem supports many platforms and 3rd-party extensions.

2.1.1 Note Workflows Support

Evernote supports all workflows but with different quality. The lowest support (but still high quality) is for *creative workflow*. The application lacks creative features as diagrams or other visualisation tools. However, text formatting is well supported. Editor allows to use classic features as bold, italic, lists, colors, font size etc. Advanced tables are also supported, including cell coloring and rich text content inside cells.

Evernote has also a wide range of features supporting *data workflow*, *e.g.* links, images and file attachments. There are also buttons for direct sound record, camera photo and attach a file from Google Drive. A big advantage is a web plugin allowing to convert a web page or its part into a note. Additionally, premium subscription offer business card scanning.

At default, *agenda workflow* support is basic only. There are only simple to-do tasks available. There are no advanced organizational capabilities as task deadline date or tasks overview. On the other hand, with 3rd-party applications that use Evernote API, new features can be added. There are applications that automatically export tasks from Evernote to specified calendar etc.

Workflow with the highest quality of support is *sharing workflow*. There are many tools for publishing notes and collaboration with other users:

- Share notes via *sharable link*
- Export notes to other formats (HTML, ENEX[3])
- Collaborative editing
- Comments in notes
- Chat with other users

2.1.2 Notes organization

There are three ways of how to organize notes in Evernote. The most important one is using *notebooks* and *stacks*. A notebook contains a list of notes. Multiple notebooks can be stacked together using a stack. This approach corresponds to *tree-organization*. However, there is no option to use more than two levels of hierarchy.

The second way to organize notes is using tags. Each note contains a list of text tags attached to it (outside the note content). Notes can be filtered by tag using search or from the sidebar.

When users often access some notes or notebooks, they can use *Shortcuts*. In the sidebar, there is a list of notes and notebooks that were added to Shortcuts which makes them quickly accessible.

[3]ENEX - Evernote XML note format

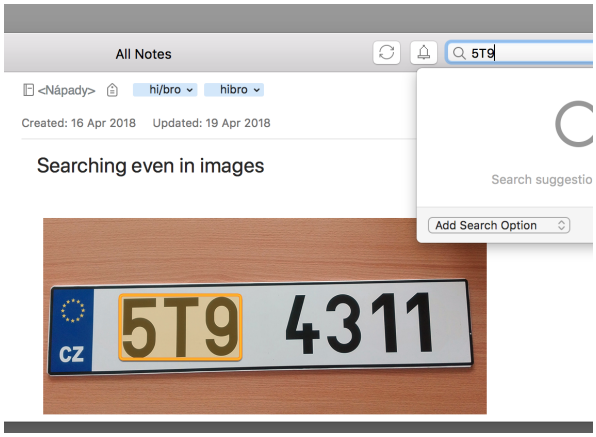


Figure 21: Search inside images in Evernote

2.1.3 Search

Searching through notes is very powerful in Evernote. There is support for full-text search, filtering results based on many parameters (note content, tags, notebooks, date and many more). To improve results relevance, there is even artificial intelligence involved in searching algorithms [15]. Besides note text indexing, text in attached files and even text found in images (using embedded OCR) is indexed. Figure 21 shows text retrieval from an image stored in a note.

2.1.4 Markdown

Markdown support is very limited. It is implemented as *WYSIWYG extension*. Only few elements are supported: lists, code block, line separator, task and link. Other elements as bold, heading, etc. have to be formatted using GUI buttons or keyboard shortcuts. The application does not offer import/-export of markdown text.

There exist alternative approaches how to use markdown in Evernote: using 3rd-party applications. These applications implement markdown editing on their own and then modify notes via Evernote API. An example of such app is Marxico [16]. However, using another software only for enabling markdown is not comfortable, because two separate applications for note taking are required.

2.1.5 Supported Platforms

Evernote supports all platforms analysed in this thesis except Linux. In past, even more platforms were supported (Windows Phone and Blackberry), but they were discontinued in 2017. [17]

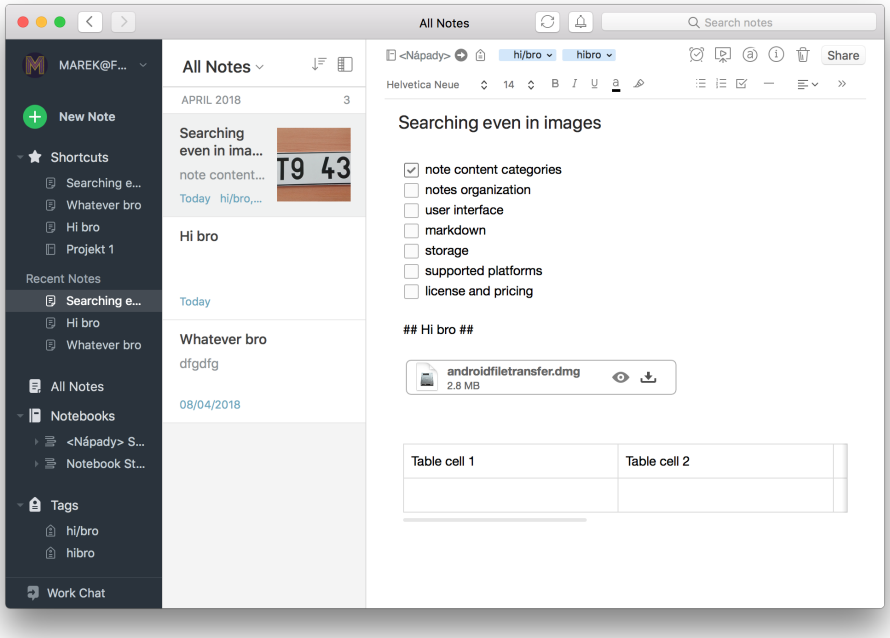


Figure 22: Evernote window on macOS

2.1.6 User Interface

Each supported platforms uses native UI following recommended design guidelines. There is a sidebar on the left side that contains created notebooks and the list of tags. Next to the sidebar is a note list containing notes from selected notebook or containing selected tag. The rest of the window is occupied by the editor.

However, during analysis I found several caveats:

- When working with tables on macOS app. I couldn't find how to delete entire row. I tried the same approach as in most applications working with tables: select the row by click on the header on the left side and click delete. This only cleaned the row content. Contextual click (right-click) did not work at all. However, the working solution was non-intuitive: the row can be delete from contextual menu of a single cell, as shown in Figure 23.

- Desktop application suffers from large number of UI elements, especially in editor view as shown in Figure 24. Some editing features are more suitable for rich text document rather than note taking application, for example many different font sizes, text alignment and different fonts.

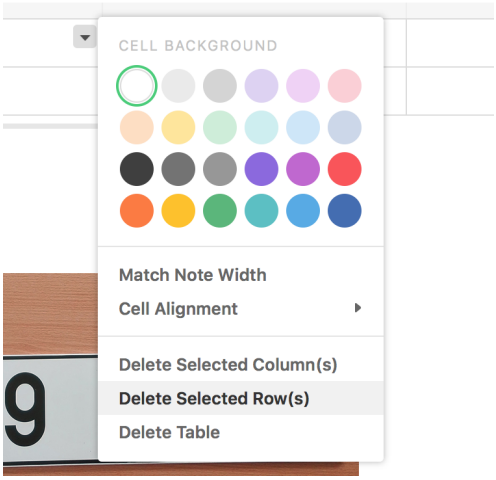


Figure 23: Non-intuitive row delete in Evernote

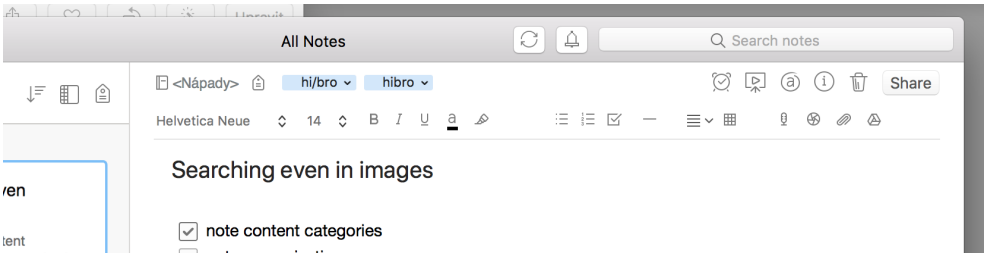


Figure 24: Distracting UI elements

There are also features that stand out above other applications. For example *presentation mode*. The window can be switched into fullscreen presentation showing opened note. This enables easy notes presenting without distractions by editing elements. Another advantage of the UI is its rich customization options: sidebar's content can be customized, there are many options how to show the note list (*e.g.* card view, snipped view) as well as many sorting options.

2.1.7 Notes Storage

Notes in Evernote are stored in a proprietary format. On desktop platforms, all notes are stored on disk and synchronized with Evernote servers. As mentioned before, there is possibility to export them to HTML or ENEX. Mobile platforms offer both online and offline ways of note storage depending on subscription plan.

Notes are available for external manipulation via Evernote API. However, it requires communication with Evernote servers and thus being online, so external note storage manipulation is not supported when being offline.

2.1.8 Pricing

Evernote is a closed-source application. It offers three subscription plans:

- *Free* - Usage without paid subscription has several limitations. There is possible to upload only 60MB of data per month. Synchronization can be active across 2 devices only.
- *Premium* - Limit on uploads is increased to 10GB, synchronization is allowed for unlimited number of devices. There are more features available including presentation mode, offline notebook access in mobile apps, and search inside PDF and Office documents. There is also customer support available via chat. The subscription price is 69.99 USD per year.
- *Business* - This plan is suitable for companies. It enables central user administration, better collaboration and priority business support. The subscription price is 12 USD per user per month.

2.1.9 Summary

Evernote is a very successful application with many note taking features. Wide range of supported platforms, rich collaboration functions and high quality search engine help with everyday work. Ecosystem around Evernote API is also an important advantage. On the contrary, user interface is sometimes non-intuitive and distracts while reading and writing. There is also very limited markdown support and the subscription plans have high prices.

2.2 Microsoft OneNote

OneNote is a note taking application created in 2003 by Microsoft [18]. Initially, it was a part of Microsoft Office suite. In 2014, OneNote was announced as a standalone application. [19] In Windows 10, OneNote comes pre-installed at default.

2.2.1 Note Workflows Support

The *creative workflow* has the biggest support in OneNote. The reason is that the editor does not use traditional page-layout, but there is an unlimited canvas where text and other elements can be placed freely. There are also features that support creative work: standard text formatting, drawing with pen, inserting shapes as rectangle, arrows, and other. In a combination with the canvas, these features are very useful for visualisation and creativity.

Data workflow is also well supported. OneNote supports images, files attachments, tables mathematical equations, sound record and embedding PDF files or YouTube videos.

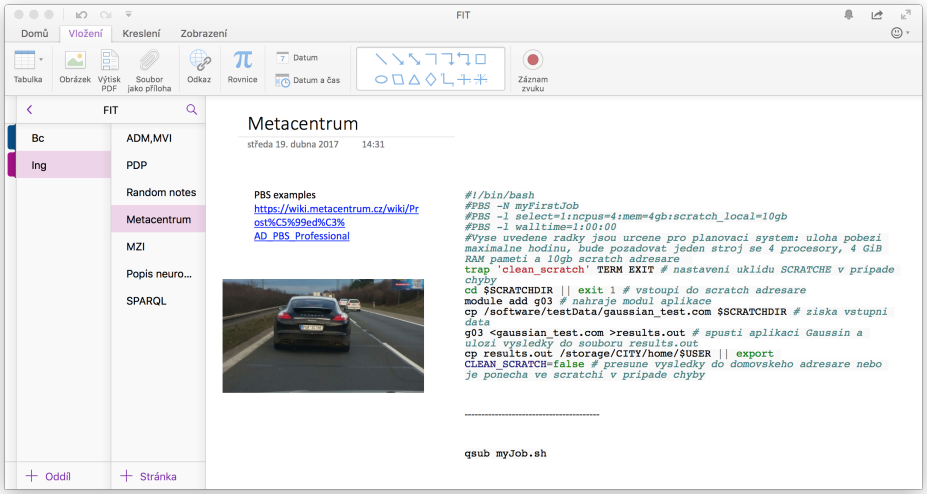


Figure 25: OneNote window in macOS

A big disadvantage is that OneNote lacks support for almost entire *agenda workflow*. Only simple tasks are available. The only option how to show all unfinished tasks is using *show all marks* command. However, this command is available in Windows version only[4].

Similarly as other Microsoft Office products, OneNote offers many ways of sharing notes with other users: export as PDF, create shareable link and since 2015, OneNote offers also collaborative editing. Considering all these features, the application offers rich support for the *sharing workflow*.

2.2.2 Notes organization

Only way how to organize notes is based on *tree-organization*. There are three types of note containers:

- Notebook
- Section group
- Section

A notebook contains sections or section groups. A section contains notes only. A section group can contain sections or another section groups and thus allowing infinite section group nesting.

[4]Only OneNote 2016 for Microsoft Windows supports *find tags* command. Note that there are two windows versions of OneNote, as described in Section 2.2.5

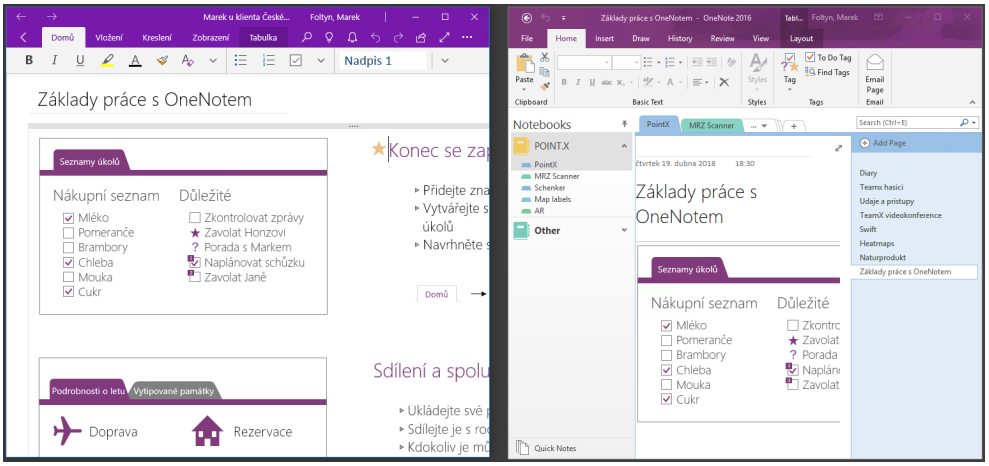


Figure 26: Comparison of OneNote and OneNote 2016 in Microsoft Windows

Lack of a non-hierarchical note organization is a disadvantage of OneNote. Notes can be tagged, but filtering by tag is done in a non-intuitive way and it is present in OneNote 2016 in Windows only.

2.2.3 Search

OneNote offers simple full-text search only. There are no advanced filtering options or special syntax that allows more precise querying. Only OneNote 2016 for windows offer also searching in text, similarly as in Evernote.

2.2.4 Markdown

Unfortunately, OneNote does not support markdown at all. Text can be formatted using UI buttons or keyboard shortcuts. This disadvantage is problably related to that OneNote was a part of Microsoft Office product family - no Office software supports markdown.

2.2.5 Supported Platforms

On every platforms listed in Section 1.5 except Linux, OneNote can be installed. For Microsoft Windows, there are even two versions: *OneNote* and *OneNote 2016*. The first one is a standalone application that comes preinstalled on Windows 10, whereas OneNote 2016 is a part of Microsoft Office. Visual comparison of these two alternatives shows Figure 26

However, not all features are available in all platforms. There are some differences. OneNote 2016 (for Windows) offers more features that other platforms, *e.g.* tag filtering, showing previous versions of notes, embedding Microsoft Excel spreadsheet and searching text in images. Although macOS ver-

sion supports mathematical equations, there are no sample equations available from the user interface as in Windows version.

Besides native applications, OneNote offers external interoperability using OneNote API. It is a part of *Microsoft Graph API*, which interconnects many Microsoft products. [20]

2.2.6 User Interface

On desktop platforms, user interface is consistent with other Microsoft Office products. The *Ribbon* is a dominant GUI element - a control bar that contain sections with buttons. As mentioned in Section 2.2.5, user interfaces of the two windows versions are different. OneNote 2016 looks similar as other Office 2016 products for Windows, whereas OneNote follows new design that appears for example in the web version of Microsoft Office. This new design is used also in mobile platforms. The main difference in OneNote is that notebooks, sections and pages are now stacked in left sidebars. In OneNote 2016, notebooks are in a sidebar on the left side, sections are above the editor and pages are in the right sidebar.

OneNote's canvas-editing approach has great benefits when working with creative notes, because it allows elements to be non-linearly ordered. However, when using multiple different screen sizes, the canvas is not comfortable for wide long notes - user have to scroll both horizontally and vertically.

Mobile platforms efficiently combine the new design with platform-specific design guidelines. Notebooks and sections are in separate views, but the animation direction after going from notebook to a section to a note helps to understand that the leftmost are notebooks etc.

2.2.7 Notes Storage

The main unit of notes storage is a notebook. The whole notebook is stored in a single file with *.one* file extension. Using a single file for the notebook allows easy backup and note transferring. A big advantage of OneNote is that the notebook file format is published. [21] On the other hand, having the whole notebook in a single file is not suitable for complex tasks using external software. For this use cases, using OneNote API is better approach.

Notes are saved in local filesystem with ability to synchronize them in cloud using Microsoft OneDrive storage.

2.2.8 Pricing

OneNote for desktop (not OneNote 2016) and mobile platforms is a free closed-source application. Web application is a part of Office 365 which costs from 69.99 USD per year. OneNote 2016 is a part of Microsoft Office 2016, so the whole Office suite must be purchased for 149.99 USD.

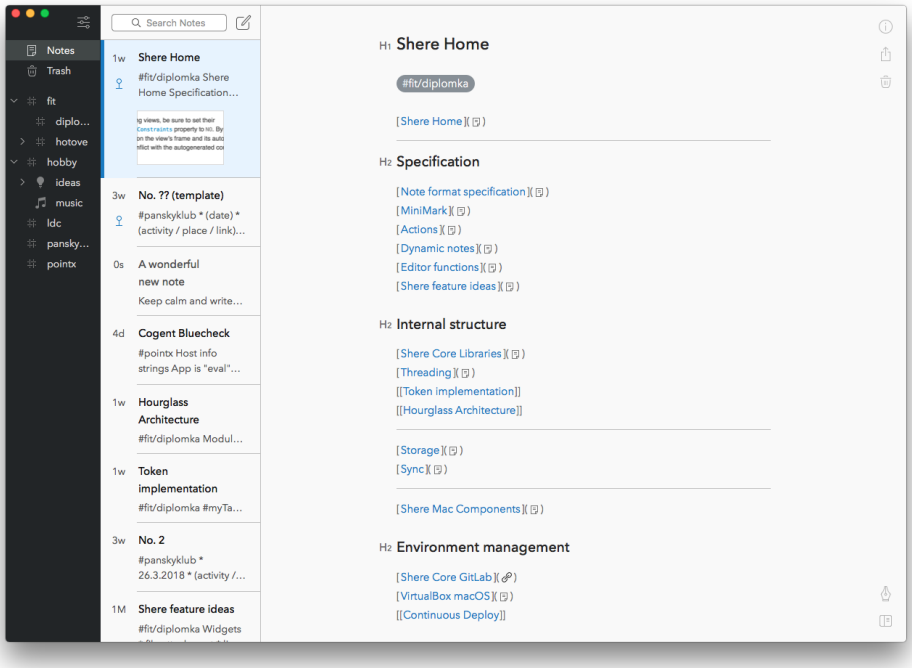


Figure 27: Bear user interface

2.2.9 Summary

OneNote is a complex note taking application that works best for creative and sharing workflow. Collaboration features support teamwork and many supported platforms enable using OneNote on multiple devices. On the other hand, the application lacks advanced *agenda* support and markdown support at all.

2.3 Bear

Bear is a lightweight note taking application created in 2016 by Italian company Shiny Frog [22]. Main differences from other applications are very clean user interface and unique notes organization approach.

2.3.1 Note Workflows Support

Bear editing is based on markdown. It offers classic features for *creative workflow* as standard text formatting. Bear application for iOS even offers embedding hand-written sketches into notes.

There is standard support for *data workflow*: images, links and limited support for file attachments, for example: files included in a note can't be

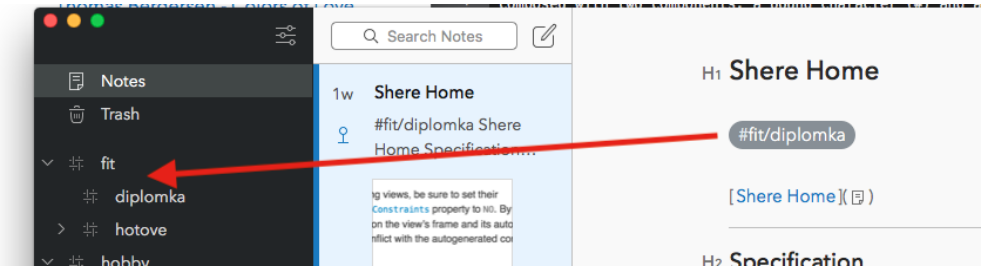


Figure 28: Bear tag usage

renamed.

For *agenda workflow*, there is similar support as in OneNote: to-do items are the only organizational elements available. Compared to the OneNote, Bear has better support of filtering to-do items in search, as described in Section 2.3.3.

For notes synchronization with multiple devices, iCloud API is used, so the application has to be linked to an iCloud account. For teamwork only note import/export can be used. There is no collaboration feature available.

2.3.2 Notes Organization

Bear is the only analysed application that organizes notes using *nested-tag* organization. This approach combines hierarchical organization with tagging. A note can contain any number of tags. These tags are composed with two components: a pound character # and a tag path. Tag paths are similar to filesystem paths: they contain words (tags) separated by slashes. Figure 28 describes how the tag relates to the hierarchical outline view in sidebar. This approach allows very elegant note organization. Since notes can have more than one tag, multiple custom hierarchies can be included. It is very efficient, when for example a note relates to multiple distinct areas (e.g. work and school).

2.3.3 Search

Bear allows full-text search with additional commands. There are two search command categories that extends search functionality:

- *Search operators*: the operator *exact word sentence* allows searching for exact phrase composed with multiple words. The desired sentence needs to be placed inside quotes, e.g.: `"John Appleseed was here"`. The second search operator is the *minus symbol*. It filters notes that do not contain search command after the minus character, e.g.: `recipes -broccoli` finds all notes containg word *recipes*, but not containing word *broccoli*.

- *Special Searches*: Bear has defined a set of search tokens that can be used for note filtering. All the tokens begin with `@` symbol: `@todo` finds all notes containing at least one unfinished tasks, `@files` shows notes containing a file. Token `@today` filters notes that was edited today. There are many other tokens: `@tagged`, `@untagged`, `@code`, `@done`, ...

Combining these possibilities, complex search queries can be constructed, for example: `#school/math @today -@images "Example of derivation"` filters notes from school subject math that were today modified, do not contain any images and contain phrase "Example of derivation".

2.3.4 Markdown

Markdown editing is the only way how to format notes. Bear implements customized markdown syntax called *Polar* [23]. It is a lightweight syntax inspired by CommonMark [5]. In user interface, markdown is implemented as modified *syntax-highlighting*: special markdown characters are visible all the time, but they have different (usually less visible) appearance. The modification means that some parts of tokens are hidden or have extended control, for example headings. Instead of showing number of pound characters, a single button containing header level info (H1, H2, ...) is displayed. In a link, the URL is also replaced with a button. URL modification can be changed after clicking on the button. In a task, a checkbox is displayed instead of the plain text characters.

2.3.5 Supported platforms

For notes synchronization, Apple's iCloud service is used. It implies that only Apple platforms are supported. There is an application for both macOS and iOS. Comparing these two versions, iOS application has support for handwritten sketches. Except that, both applications offer the same features.

2.3.6 User interface

One of the most important Bear advantages is that is has very clean and minimalist user interface. There are three main sections in a Bear window: a sidebar containing tag hierarchy, a note list and an editor view. The sidebar and the note list can be collapsed and thus non-distracting editing environment can be created. iOS version has almost the same design as the macOS.

2.3.7 Notes storage

Notes in Bear are stored in a proprietary way. During my analysis I found out that the note text is stored in a SQLite database. However, the structure is not officially documented and thus not suitable for external use. There is

no API provided which implies that there is no comfortable way how to work with notes using external software.

2.3.8 License and pricing

Bear is a closed-source application which offers both free and paid plans. There are several features unlocked with paid subscription:

- Note synchronization - this is the most important paid feature. In the free plan, there is no possibility to synchronize notes with other devices.
- More themes - there are 2 different themes available for free. Subscription adds 10 more themes.
- Export to more formats in addition to export to markdown: PDF, HTML, RTF, DOCX, JPG.

The price for Bear Pro subscription is 1.49 USD per month.

2.3.9 Summary

Bear is a lightweight and minimalist note taking application with innovative notes organization approach. It combines fast markdown editing with clean user interface. The main disadvantages are that only Apple platforms are supported and no external access to notes is available.

2.4 Dropbox Paper

Dropbox Paper is a document-editing application with minimalist interface and supporting rich collaboration. It was created in 2016 by Dropbox. The main goal of the project is to empower collaboration in a way that does not reduce creativity [24].

2.4.1 Note Workflows Support

Dropbox supports all workflows. For *creative workflow*, following features are available: text formatting, links to other notes and hashtags. Unfortunately, there is no advanced features like diagrams drawing or handwritten sketching.

However, this little disadvantage is compensated with *data workflow* support. Besides classic links and images, there is big support for embedding content from 27 external sites including, Facebook, YouTube, Trello, GitHub, Spotify and Dropbox. This feature greatly simplifies access to external content.

Agenda workflow is also well supported. As in other apps, the main unit of organizing is a task. In Dropbox Paper, tasks can have (in addition to text

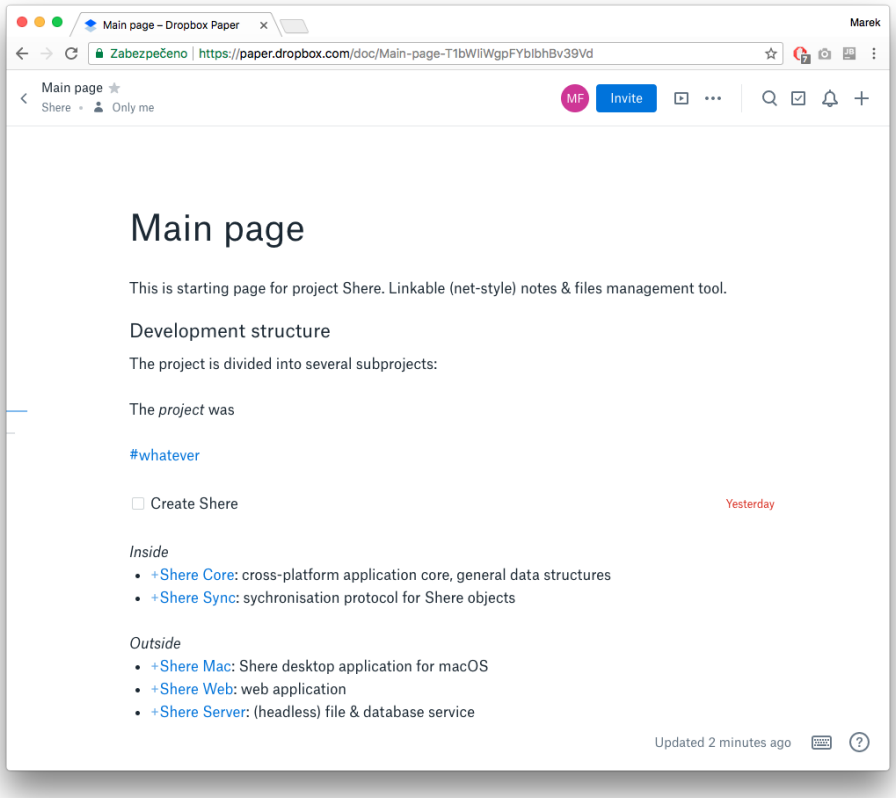


Figure 29: Dropbox Paper

description and done-status) specified deadline date and a list of people that are related with the task. These abilities make Paper very powerful in team organization. Tasks attached to the user can be aggregated from all notes and viewed from sidebar. There is also one more agenda feature: a note can have a calendar event attached. The calendar can be from Google or Office 365.

As mentioned above, *sharing workflow* is one the core features of Dropbox Paper. There are export options to markdown and DOCX and more important realtime collaboration. The collaboration is extended with ability to comment parts of documents which is important when discussing about the document content or the particular work activity.

2.4.2 Notes Organization

Notes are organized using *tree-organization* system that uses traditional folders. A folder can contain notes or other folders. In addition to this, a note or a folder can be *starred* for quick accessibility. List of starred items can be accessed from the main page.

2.4.3 Search

Dropbox Paper includes standard full-text search. There are few advanced search abilities: click on a hashtag to search notes with this hashtag or click on a mentioned team member to search notes created by this member.

2.4.4 Markdown

Markdown in Dropbox Paper is supported as *WYSIWYG extension*. Except links, all elements are supported including task extension, e.g. `[] buy food`. Except standard markdown, there two more features available. Very useful extension is LaTeXsupport, especially for mathematical expressions. The second feature is emoji support. After writing a colon, a pop-up window with many available emojis is displayed.

2.4.5 Supported platforms

In 2016, Dropbox Paper was introduced as a web application. Nowadays, native mobile platforms (iOS, Android) are also supported. Mobile platforms have one advantage when compared with the web application: they support *offline mode*: when a mobile device has no internet connection, user can still create new or access and edit starred and recently modified documents. However, this is usable only for short moments without the internet, because no existing (not opened) notes can't be loaded. For most of the time an internet connection is needed for smooth work.

2.4.6 User interface

User interface and work with the editor is what make Dropbox Paper different. The editor is minimalist and clean, yet easy to understand. There is very well-crafted UI for work with both mouse and keyboard-only: there are no formatting buttons visible in order to keep the UI clean. When a part of text is selected, a ribbon with formatting buttons is showed above the selection. This approach is useful for users that use keyboard shortcuts because it does not distract them and still elegant and accessible for users that want to use mouse and UI buttons.

2.4.7 Notes storage

Notes in Dropbox Paper are stored in a proprietary way on Dropbox servers. As mentioned above, in mobile platforms, only recent and new documents are stored locally for offline access.

However, there is an API available for document manipulation. It is a part of Dropbox API. It includes advanced functionality like full-text search, thumbnails and sharing. [25]

2.4.8 License and pricing

Dropbox Paper is tightly related to Dropbox application - file synchronization software. It is a closed-source application that has three different pricing plans including a free plan. Dropbox Paper is can be used in every pricing plan, only thing that differs (except Dropbox specific features that are not related to Paper) is maximum storage capacity. In the free plan, each user has 2GB available. This capacity can be increased by inviting other users via a web link up to 16GB. Both Dropbox Plus and Professional pricing plans offer 1TB of capacity. Dropbox Plus costs 8.25 EUR per month. Dropbox Professional costs 16.58 EUR per month.

2.4.9 Summary

Dropbox Paper is an excellent document-writing and organizational tool. Although it is focused mainly on team collaboration, it can be successfully used as a personal note taking application. Main advantages are rich support for embedding external content and very comfortable user interface. The disadvantage is that the documents can't easily be stored in computers.

2.5 Typora

The last application analysed in this thesis is Typora. [12] It is not a note taking application per se rather a generic markdown editor. The reason why I included it into the analysis is that it is the only editor found that implements *realtime seamless preview*. However, using the integrated directory explorer enables Typora to be used as a simple note taking application.

2.5.1 Note Workflows Support

Although Typora is just a markdown editor, it offers some advanced features for some workflows. For *creative workflow*, standard editing features are available, *e.g.* text formatting, tables, lists. However, there is also support for diagrams using *js-sequence* syntax [26]. Figure 210 code shows diagram syntax example with rendering of the diagram image.

For *data workflow* support, Typora includes links to other notes, images, tables and embedding LaTeX. However, there is no support for embedding or linking files. No additional content types as embedded videos are supported.

Agenda workflow has very basic support. Only feature available is using markdown tasks using GFM[5] syntax. These tasks can not be aggregated nor they have any additional parameters as deadline date etc.

Sharing workflow is limited to import/export documents. On the other hand, since Typora uses Pandora for importing and exporting, the number of

[5] GitHub Flavored Markdown

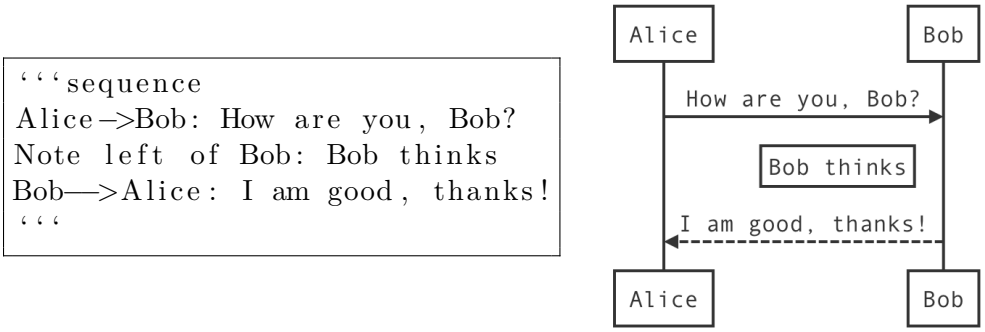


Figure 210: Diagram syntax and rendering in Typora

supported formats is large: PDF, HTML, DOCX, RTF, ePUB, LaTeX, Media Wiki and more.

Another way how to achieve note sharing is to synchronize the whole storage. More information about this approach is in Section 2.5.7.

2.5.2 Notes Organization

Typora does not uses any custom document organization system, it relies on the filesystem: any folder structure can be used as document-tree. Typora includes simple tree-view file browser in the sidebar so the notes can be created, opened and deleted directly from the Typora window. There is no support for tags or other organization methods.

2.5.3 Search

Unfortunately, Typora does not support searching through notes at all. Only search inside the opened note is available. Absence of such feature complicates work with large number of notes.

2.5.4 Markdown

Support for markdown is the most important feature of Typora. Editor renders markdown using *seamless realtime preview*. Complete Markdown syntax is supported including additional optional elements as subscript and superscript, strikethrough and emojis using two colons wrapping (e.g. `:smile:`). In addition to usual markdown elements, tables, diagrams and mathematical expressions are supported.

2.5.5 Supported platforms

Typora supports all three major desktop platforms: Windows, macOS and Linux. There is no support for web or mobile platforms. Since Typora is

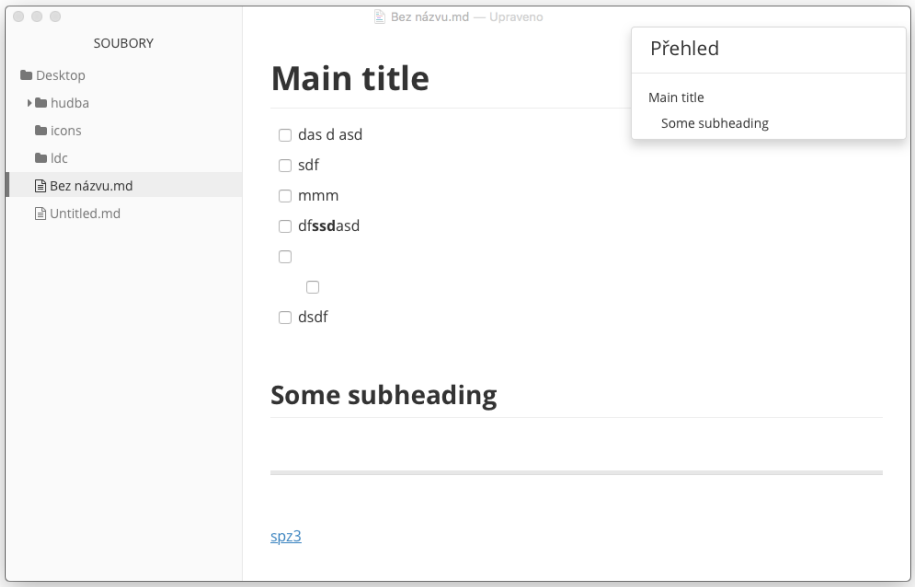


Figure 211: Typora window

written using cross-platform library Electron [27], the same features are available in all platforms.

2.5.6 User interface

Typora GUI is the same on all platforms. The only difference is the appearance of the window-buttons (exit, minimize and maximize) in the main window. Using Electron frameworks impacted consistency with the platform design recommendations - elements in HTML have different design, and the logical parts of the GUI are placed on unexpected places. For example, menubar in the sidebar is placed on the bottom in Typora, which is confusing in macOS. Elements have custom styles which do not correspond with any supported platform.

2.5.7 Notes storage

As mentioned in Section 2.5.2, Typora uses plain files and folders. This makes any folder usable as note storage. In Typora window, *open a folder* command is available that shows the folder content in the sidebar. Using this approach have several advantages:

- When the folder with notes is placed inside a folder synchronized by an external software like Dropbox, a simple note synchronization can be

achieved.

- Notes can be easily accessed and modified by external applications.
- Note organization structure is not dependent on used tools.

2.5.8 License and pricing

Typora is a free closed-source software. The project page [12] says that the application is free only during beta. However, no non-beta version has been released until the analysis.

2.5.9 Summary

Typora is a markdown editor with innovative rendering approach that makes markdown content both easy to write and easy to read. It does not offer advanced organizational, sharing or structure features, its main focus is in editing plain text files.

2.6 Comparison

Every application analysed in this chapter is included in the comparison table 22. For each evaluation criterion, a score number is provided. The score range is from 0 (no support / very bad quality) to 3 (rich support / excellent quality).

The comparison table quantifies the analysis and for each criterion, it shows which application has the best score. If there are more applications with the same best score for a certain criterion, the emphasized one means that this application is slightly better.

There is an interesting fact that can be seen in the table 22: each application is the best at least in one criterion. Also, even the applications differ in their best features, the total score is nearly similar for each app. It can mean that the applications have comparable quality, but they have different main focus.

2.7 Shortcomings analysis

After the analysis, we can point out several shortcomings that were found in note taking applications:

- *Open-source*: all analysed application are closed-source. Although there are many open-source note taking projects, none of them has such quality to be used for daily basis or at least to be analysed in this work. [13]

	Evernote	Microsoft OneNote	Bear	Typora	Dropbox Paper
Note workflows support	2	2	1	1	**3**
Notes organization	2	1	**3**	1	2
Search	**3**	1	2	0	2
Markdown	1	0	2	**3**	2
Supported platforms	**3**	3	2	2	2
User interface	2	2	3	1	**3**
Notes storage	1	2	1	**3**	1
Pricing	2	**2**	1	2	2
Total	16	13	15	13	**17**

Table 22: Score-based comparison of note taking applications

- *Cross-platform apps*: using the same application on multiple devices is a feature that is not supported by all analysed applications.
- *Note and storage format specification*: when saving notes into proprietary storage, users lose control over their data and without API, they can't easily use their notes via external applications or migrate to other note taking software. Using publicly specified note and storage format and accessible data storage solve these problems.
- *Markdown realtime preview*: there is no note taking application that seamlessly previews markdown except Typora (which is an editor only). Using this approach is very suitable for taking notes, because it is both easy to read and easy to write.

These disadvantages are one of the reasons why I chose this topic as my master's thesis. I believe that creating an application resolving these will improve user experience of taking notes and allows them have more control about of their personal data.

2.8 Summary

In this chapter, I worked with the evaluation criteria that were used to analyse and compare existing note taking applications. We can see that each appli-

cation has different set of high quality features. Based on the analysis, most important shortcomings were discussed.

Chapter 3

Shere development

This chapter contains description of the practical part of this thesis: a note taking application called *Shere*. At the begining of the chapter, software requirements for the project are defined. Next parts contain description of software architecture, software modules, discussion of testing, documentation and future development possibilities.

3.1 Requirements

The goal of software requirements is to specify the properties of the project and to define the functions necessary for the purpose of the system. [28]

Requirements for Shere are based on the analysis of the existing applications and the shortcomings listed in Section 2.7.

3.1.1 Functional Requirements

Functional requirements are specified in the following list:

1. Application allows creating and editing text notes.
2. Application uses *nested-tag* notes organization.
3. Application implements markdown editing (the syntax can be customized) using *seamless realtime preview*.
4. Markdown syntax supports to-do items
5. Markdown syntax supports tags and sub-tags for notes organization.
6. Full-text search is supported.
7. To-do items can be aggregated from all notes.
8. Application uses note storage structure with published specification.

3.1.2 Non-functional Requirements

Non-functional requirements complement the functional. They specify not what the system will do, but how and other properties, e.g. reliability, portability, performance etc. They are usually not easily measurable and thus evaluated subjectively [28]:

1. Application-logic code is portable allowing to be easily reused in multiple platforms.
2. User interface is minimalist and non-distracting.
3. Application is published as an open-source project.

3.2 Architecture

Software architecture describes structures used in an application, their properties and relations among others. It is a high-level view of the application. In Shere, the architecture is separated into the two components: shared code architecture and GUI architecture. Considering all the requirements, it turned out that two aspects the most significantly influenced the GUI architecture: cross-platform approach and the GUI programming language support. Both aspects are described in the following sections.

3.2.1 Cross-platform Approach

Because the application logic must support multiple platforms, a cross-platform approach needs to be selected. We can differentiate three basic cross-platform approaches [29]:

1. *Native*: separate development for each platform
2. *Universal*: Using a cross-platform toolkit with a single codebase
3. *Hybrid*: A part of the code is cross-platform and is reused in separate projects for each platform.

Each approach has advantages and disadvantages. Using native application for each platform enables using all platform-specific features, but it requires the most effort and the logic code is duplicated on every platform. Using universal cross-platform toolkits allows writing code single time that is used in every supported platform. On the other hand, supported features are limited by the used toolkit and sometimes native look and feel is affected. Using hybrid approach shares a part of the code that is used in separated native project. It allows a part of the code to be written only once and in platforms as well as using native features and GUI. Hybrid approach combines the advantages

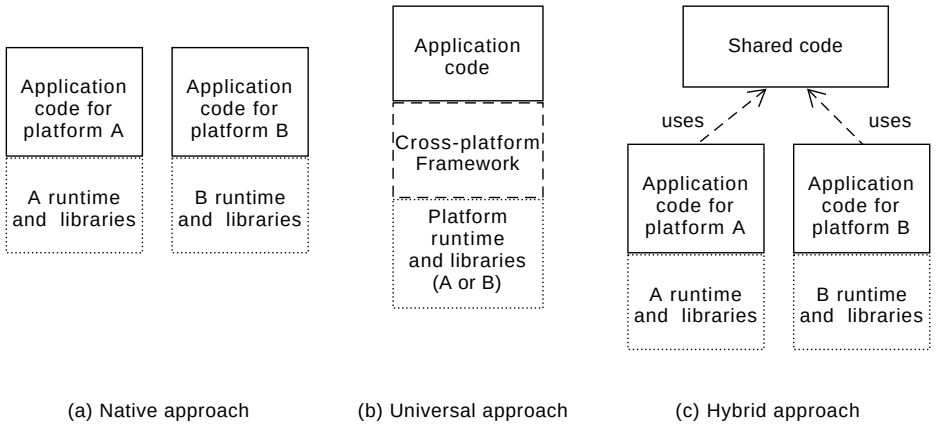


Figure 31: Differences of cross-platform approaches

of both previous approaches, but increases complexity with connecting the shared code. Also, separate projects still have to be created. Table 31 compares the approaches based on a subset of the criteria specified in [29]. Each approach also contains examples of technologies using this approach.

	Native	**Universal**	**Hybrid**
Ease of development	Slow, each platform separately	Fast, single code works everywhere	Moderate, separate projects with shared code
Look and feel	Best as using native GUI libraries	Limited to used technology, often non-native	Best as using native GUI libraries
Supported platforms	Every platform with separate project variant	Limited to used technology	Every platform with ability to use the shared code
Maintainability	Bad, repeating changes on all platforms	Very good, changes made only once	Moderate, shared code changed once, the rest is duplicated
Technology examples	Cocoa, WinAPI, Android	Electron, Qt, GTK+	Djinni, C/C++ bindings to other languages

Table 31: Cross-platform approach comparison

Technical differences of the approaches are illustrated in Figure 31. For the purpose of Shere, native approach can be excluded. Creating a separate application for each platform is not efficient, it would require big effort and

even then it would lead to bad code maintainability. Universal approach works well for fast development requirement. Since there is only single codebase for all platforms, the maintainability is very good. However, using universal frameworks limits using only features implemented in the used framework. There can be required features that can not be implemented (or it is difficult to implemented them in a cross-platform way). When the whole project depends on a certain GUI framework, it can complicate the development in a future. Also, the application cannot be extended to other platforms, if the framework supports only some of them. The most flexible solution seems to be the hybrid approach. It combines the best from the previous approaches. However, more effort than using universal approach is needed, because the shared code linking must be linked to the separate parts and separate projects (usually with additional code) must be maintained. So, for Shere project I selected the hybrid approach. A cross-platform library containing application logic and shared functionalities will be implemented. This library will be then used in platform-specific projects.

It is efficient to share as much code as possible to avoid code duplicities. The ideal is to achieve that the shared code contains all common features except platform specifics. The biggest difference among platforms is the GUI - systems provide different libraries with different features and formats for layout specification. This leads to the following approach: implement all functionalities and logic in the shared part of the code except the GUI. User interfaces for each platform will be created separately using its native tools and libraries. The interfaces will be then linked to the shared code. This is the approach that will be used in Shere.

3.2.2 GUI Programming Language Support

For note taking application, supporting all major platforms (Windows, macOS, Linux, iOS, Android and web) is a great benefit. This fact has to be also considered before architecture specification. Because Shere is planned to be developed even after thesis finish, used architecture and the programming language should not exclude any of the mentioned platforms. However, sharing code with these platforms reduces the number of available programming languages. Each of the platforms supports a limited number of languages that can be used to create native GUI. Table 32 compares frameworks provided by different platforms and programming languages they support.

From the table we can see, that there is no language that can be directly used in every platform. However, some languages are interoperable with C or C++ so C and C++ can be candidates to be used for the shared code. In Microsoft Windows, both C and C++ native libraries are provided, so including the shared source (or linking a library) is enough.

In macOS and iOS, C code can be used in Objective-C code, because Objective-C is a superset of the C language. This applies for Objective-C++

	Framework	**Supported languages**
Windows	Windows API	C
	Microsoft Foundation Classes	C++
	Windows Forms	C#
	Windows Presentation Foundation	C# and XAML
macOS	Cocoa	Objective-C, Objective-C++, Swift
Linux	Xlib	C
Android	(embedded UI framework)	Java
iOS	Cocoa Touch	Objective-C, Objective-C++, Swift
Web	(none)	HTML, CSS, JavaScript

Table 32: Native GUI libraries and supported languages

and C++ in the same way [30].

Linux is a bit different, because it does not have a single GUI library[6]. However, there are many GUI libraries written in C or C++ that can be used [32].

In Android, things get more complicated. GUI programming is available in Java only, so there is necessary to use Java Native Interface [33]. JNI is a technology that enables interoperability between Java and C/C++ using C interface. Using JNI, the shared code in Shere can be used in Android. The disadvantage of this technique is that it requires additional code for linking the languages. However, this code can be automatically generated using tools as Swig [34] or Djinni [35].

The most problematic platform is web. Only languages usable for creating GUI is HTML combined with CSS and JavaScript, because these are the only languages supported in web browsers. Using C/C++ code can be done using a library that bridges between these languages, for example CppCMS [36] or Wt [37]. Because the code runs on a server, there can be delays caused by network layer between the client and the server. Another alternative is to use C++ in a server only and provide an API allowing use the shared code. A separate web application would then communicate with the server using the API. Both approaches increase complexity of the application, but it is still possible to reuse the shared code.

[6]In Linux, the graphical user interface is not a part of the operating system. The graphical user interface found on most Linux desktops is provided by software called the X Window System, which defines a device independent way of dealing with screens, keyboards and pointer devices. X Window defines a network protocol for communication, so any program can use it. There is a C library called Xlib that makes it easier to use this protocol, so Xlib is kind of the native GUI API. [31]

Based on this analysis, it is possible to used C or C++ for shared code. Difficulty of the usage with other languages is not equal on every platform, but it is possible. For the shared code in Shere, I selected C++ because of these reasons: performance is not critical (so it is not necessary to use low-level language as C), C++ has more features than C and it allows faster development.

3.2.3 GUI Architecture candidates

From the two previous sections, the following requirements helped design appropriate application architecture:

- *Use hybrid cross-platform approach:* there will be a shared codebase containing the logic and separate implementation of the GUI for each platform.
- *Shared codebase written in C++*: in order to remain portable to all major platforms, the logic will be implemented in a portable way using C++.

One of the widest architectural pattern used in GUI applications is Model-View-Controller (MVC) [38]. It separates the data representation, application logic and the views in order that a change in one of them should have minimal impact on the others. The wide usage among applications is the reason why I considered using this pattern for the overall Shere architecture. To figure out whether the MVC pattern is suitable for Shere, it must be decided which components will be included in the shared code and which in the platform-specific code. Including Model only in shared code, it would require custom controllers for each platforms and thus large part of the application would be duplicated across platforms. Sharing controllers in every platform is not reusable, because different platforms use different views and thus needs different controllers. These complications caused that using MVC pattern for the overal application architecture is not suitable for hybrid cross-platform application.

As mentioned in Section 3.2.1, the more code is possible to reuse among platforms, the better is the code maintainability. The best case is that GUI code only will be separated from the shared codebase. So the second approach how to design the architecture was to reach the following goal: the shared code is independent on used native views and controllers. The native GUI is separated from the logic and communicates with the shared code in a way that is not dependent on the used GUI elements, e.g. specific views (as buttons, tables) and layouts, rather than in a domain-specific way. The following example illustrates the difference: In a note taking application, when a new note is created, a new GUI element is appended into the list of notes in the sidebar. So when an user clicks o na button *new note*, the application calls the shared code

logic as illustrated in the following pseudocode: `sharedCode.createNote()`. When the note was successfully created, some code processing the success should be executed. In the approach dependent on used GUI elements, the callback `sharedCode.onUpdateSidebar` would be called and as an argument, a list of notes would be specified. This callback would be directly related to the GUI element Sidebar and for example `sidebar.updateList(noteList)` would be called. The second (more abstract) approach has the callback called `sharedCode.onNoteCreated` which does not tells anything about used GUI elements. This approach is more flexible, because it allows use different GUI elements in different platforms (not all platforms now are not forced to contain a sidebar for example). Secondly, it enables coordination with multiple elements. Returning back to the example: after a note is created, the sidebar is not the only element that might be updated. Also other GUI element might be modified, *e.g.* an empty editor view might be shown to be able to edit the created note. Here goes the non-trivial bridging: when a new note is created, a new entry in the sidebar is placed and an empty editor view is shown. The logic that creates the notification about the note creating is separated from the logic that adds a new element into the sidebar. This approach is described as a Presentation-Abstraction-Control (PAC) pattern. [38] In context of PAC, terms *agents* and *layers* are used. An agent is a software component that can have its own state, functionality and relations to other agents. Using the *create note* example once more, there are four agents: `sharedCode` agent containing the note manipulation and storage functions, `sidebar` and `editor` view agents and `onCreateNoteCoordinator` which is the bridging between the shared code and the views. These agents are hierarchically organized in layers that the agents have transitive access to agents in higher layers (the `onCreateNoteCoordinator` can access the views, but the views do not know anything about the coordinator). Using PAC in Shere, the lowest-level agent would contain the shared code and its main task would be to link agents one layer higher. The agents in non-shared code would be only GUI-related, e.g. views and bridge-classes that coordinate multiple views. This architecture seems more promising than MVC, because it allows easy decoupling of different user interfaces and the logic.

There is also a third approach of the architecture. In PAC pattern, data flow from the top-level agents (native GUI views) through lower levels to the shared code and vice versa. Usually two or more layers would be required in the non-shared code (the views and the bridges). During research if these layers could be reduced, an architecture using *Observer* [38] pattern and separate agents was considered: the shared code would be encapsulated in an observable object. GUI elements as sidebar, editor etc. would be independent components communicating with the shared code via sending commands and listening (observing) for changes they are interested in and behave according to them. The following example illustrates using Observer pattern in Shere: `sharedCode.onNoteCreated.addObserver(sidebar)`. Inside the sidebar ob-

ject, a method dispatching successful note creation would be implemented. Using observer and separate GUI elements reduce the coordination layers as in PAC, because each element handles its own behaviour independently on others. However, this architecture has also several limitations:

- The independent GUI elements contain state information, for example: a sidebar with notes organization structure (e.g. tags) contains information about actually selected tag. There are situations that are dependent on a certain state. If a component containing a list of notes wants to load the list, it has to know, which tag in sidebar is selected and show notes containing this tag. One solution can be communication between the sidebar and the note list, but this would introduce dependency between the components which is not desirable. Another solution can be communication through the shared code. However, similarly as using MVC, this would introduce dependency to the GUI into the shared code and thus it limits usage on multiple platforms with different user interface design.

- The architecture using Observer pattern seems that it reduces bridging layers between the shared and the native code. However, it would only split these layers into smaller parts included in each independent GUI element. Each of them would still require some way of linking with the shared code: if a new note is created, both the sidebar and the editor view must have sort of a controller that adds a new entry into the sidebar, resp. shows the empty editor. It would be the similar amount of code required.

Considering the limitations, it turned out that they prevent using Observer pattern from being used in Shere architecture. It would unnecessarily complicate cooperation with multiple GUI elements and the code amount optimization would be negligible.

Figure 32 visualises the architecture candidates. As mentioned above, the architecture using PAC pattern was selected for Shere GUI, because it offers a good trade-off between the data flow control and the amount of non-shared code.

3.2.4 Shared Code Architecture

For the shared code architecture, it was only stated that it should contain as much non-GUI code as possible. It should also provide information necessary for showing GUI, but in an UI-independent way. In order to design the architecture properly, it is necessary to outline modules which will be required based on the software requirements. I divided the shared code into three sections:

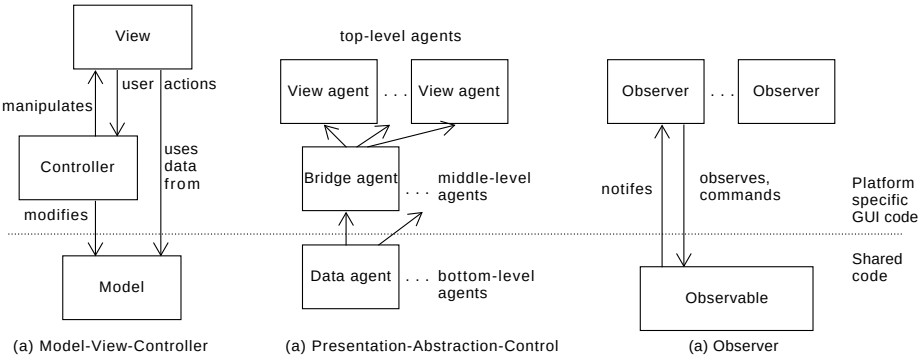


Figure 32: Architecture candidates

- *Actions* provide a public interface of the shared code. *Actions* are methods that can be called to initiate a task (from both inside and outside the shared code). They cover tasks related to the process logic of taking notes, e.g. create a note, find notes containing a specific tag, show all unfinished tasks etc. Actions are the main application-logic units. They coordinate functional modules to manage their tasks and call callbacks or other actions.

- *Callbacks* enables to run custom code when an certain event occurs, e.g. a note was created, search results are available and so on. The reason why I separated the initiation of an action from getting the result is that the code that calls the action does not have to check errors and the results explicitly. This helps to hide the logic into the share code, because the GUI only starts and action and process a result. For example: when a note is successfully created, `onNoteCreated` callback is called. When an error occurs (for example a new file could not be created due to limited write access), `onNewNoteFailed` callback is called containing the error information. If the GUI has set up the error callback, for example a dialog describing the error is displayed.

- *Functional modules* contain all non-logic functionalities, for example storage, parsing and indexing features. The modules are hierarchically organized in a way that bottom-level module does not have access to higher-level modules.

Given these three sections, we can built the architecture as follows: the top-level modules are the Actions and Callbacks. The user lower-level Functional modules that are also hierarchically layered.

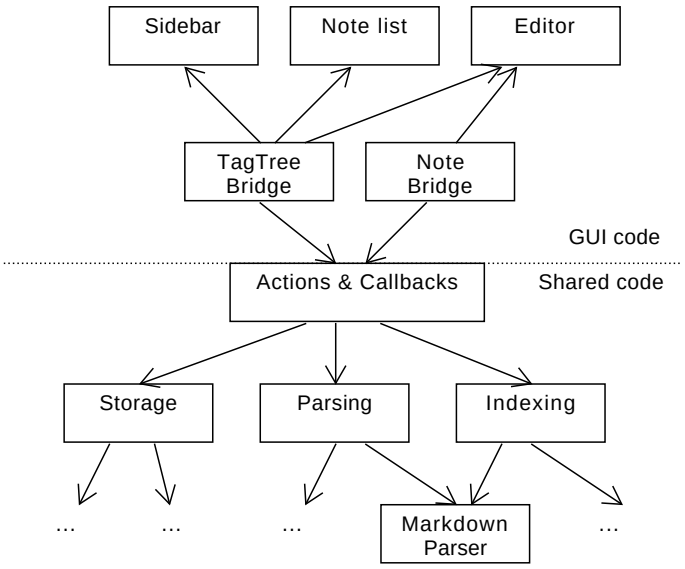


Figure 33: Shere hourglass architecture

3.2.5 Summary

When the shared code multi-layer architecture is joined with the PAC architecture for the GUI, the structure corresponds to the *hourglass architecture* as shown in Figure 33.

3.3 Development Introduction

After the architecture explanation, the development process and the implementation details will be described. Firstly, I will describe the development environment and used tools. Secondly, the Shere Core library will be introduced describing the overall concepts as well as individual modules used in the Core. After that, macOS application and its GUI components will be described. The last sections will contain other development information: testing, documentation and a discussion of future development.

Since my primary computer is MacBook Pro, I decided that Shere will be developed for macOS in this thesis. The GUI code is built using the Cocoa Framework [39]. The shared code is created as a C++ library called *Shere Core* (or shortly *Core*) that will is linked by the GUI application.

The GUI was created using Xcode. It is the default IDE created by Apple. The Core was created using CMake, an open-source tool for building, testing and packaging software, and CLion, a C/C++ IDE created by JetBrains [40].

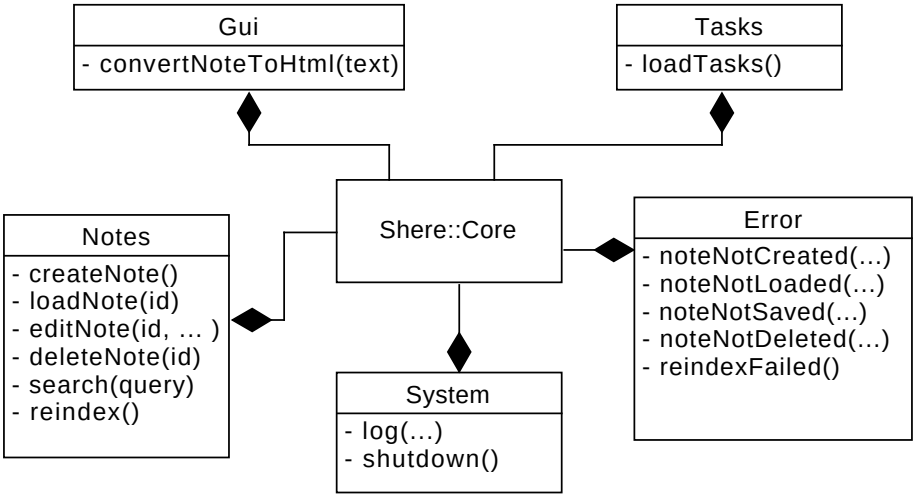


Figure 34: Shere Core Actions

3.4 Core Library Fundamentals

The first part of the development description is dedicated to the Core library description. As already described, it is a C++ library that contains note taking application logic and all functionalities required for working with notes. Its interface is provided in header files so the library could be easily linked.

There are three fundamental objects provided in the public library interface: `Core`, `NoteInfo` and `iTagTree` interface. `Core` class encapsulates all the functionalities using Actions and Callbacks. In runtime, an single instance of this class is used. `NoteInfo` is a simple class containing basic info about a single note (e.g. an identifier of the note, a title, ...). `iTagTree` is an interface providing methods for querying about the notes organization data structure (`TagTree`).

3.5 Actions

Actions provide a way how to initiate an application-logic task. They are implemented as public methods inside the `Core` containing zero or more arguments and returning `void` value. Actions in Core are divided into five sections according to the area of usage. Figure 34 lists all the actions and its hierarchy.

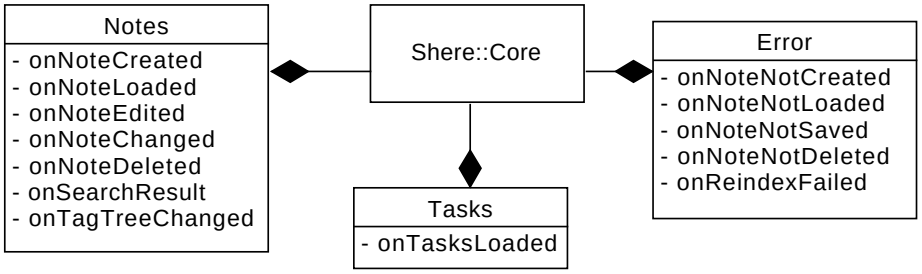


Figure 35: Shere Core Callbacks

```
// Set up callback
core.notes.onNoteCreated = [&](string noteId)
{
    std::cout << "Created: " << noteId << std::endl;
};

// Execute the action
core.notes.createNote();
```

Figure 36: Example of Action-Callback usage

3.6 Callbacks

Callbacks represent result values of Actions (or an event that occurs without the Action invocation). They are implemented in Core as member variables having `std::function` type. This implies that only one function can be bound to as the callback for a specific task. Unlike the Observer patter (which could be used here - registering multiple functions for the same callback), using a single function keeps the callback dispatching in a single place and helps to control data flow and simplifies debugging. In Core, the Callbacks are divided in the same was as Actions. Figure 35 lists all the callbacks and its hierarchy.

Combining Actions and Callbacks tasks can be performed and the results can be processed. For example, if an user wants to created a new note, the following sequence is performed: `Shere.notes.createNote()` is called. If the creation was successful, `Core.notes.onNoteCreated` callback is called. In case an error occurs during the note creating, `Core.error.noteNotCreated` callback is called. This process visualises Figure 37. Note that the initiation of creating new note is separated from the handling the result. However, this approach makes relations between the Actions and Callbacks not obvious. To explain how the Actions and Callbacks are used in code, Figure 36 shows a code example.

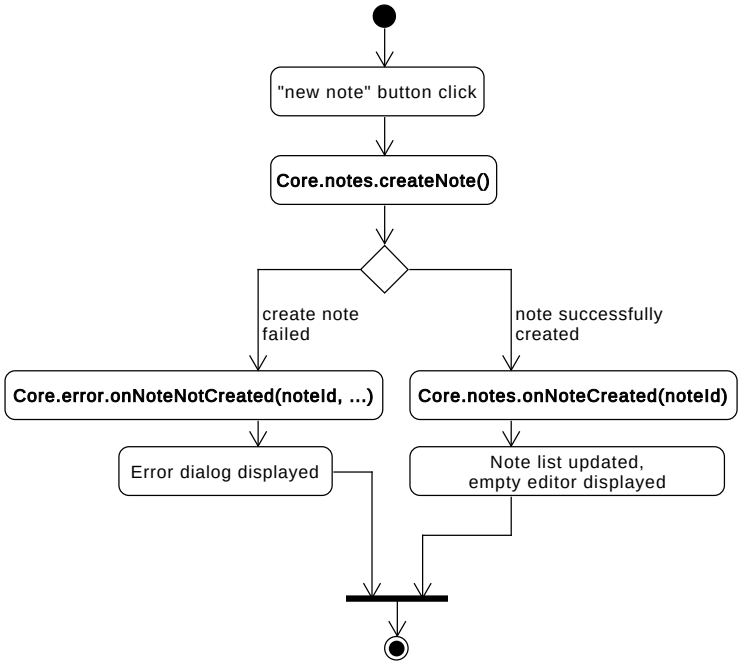


Figure 37: Using Actions and Callbacks in Core

Because of the non-explicit relation between Actions and Callbacks, this pattern is not suitable for situations when an Action have many different possible results (and thus many different callbacks required). It would make the Action-Callback processes too complex and hardly readable. In Shere, simple tasks are designed that many of them are understandable even without checking the implementation or documentation. They usually have one or two possible callbacks (success and error).

3.7 Core Backend

Actions and Callbacks are not the only part of the `Core` class. There also private components not accessible from the external code: the functional modules and the event loop. The functional modules are described in the following sections. The event loop is a mechanism that dispatches the actions and callbacks: when an action is initiated from the GUI, it is not processed directly in the public method in Core, but it is queued and processed in a background thread separately from the GUI. Also the callbacks are executed in the Core thread. Executing the logic separately from the GUI does not block the main thread which is benefitial for user experience in case of long-running tasks. The Core event loop is implemented around *asio* library. It is a cross-platform C++

library for network and low-level I/O programming that provides a consistent asynchronous model [41].

Using asynchronous model in Core corresponds to the overall architecture since it separates GUI threading and application logic threading. On the other hand, in order to use the same data in multiple threads, some form of data synchronization must be implemented to avoid data races. Two main approaches can be used: locking the data using mutexes or sharing immutable data copy. In Shere, the second approach is optimal, because one thread in Shere usually produces the data and the second thread then consumes them without modification. For example: when a note is loaded, a `std::string` instance containing the note text is created. This string is then sent into the callback `onNoteLoaded` and no further modified. The background thread does not need this string anymore. In fact, the background task finished after a callback is finished, so there is no reference to the string anymore. So this string is safe to be used in the GUI thread. However, even that, two data synchronizations are needed: when enqueuing a task into the queue, it can be done from a GUI thread (e.g. after a button click) and simultaneously, the background thread can take a task from the queue. So more than one thread accesses the same data variable and thus a synchronization is required. In Shere, this is already implemented in asio in `io_service` class. The second synchronization is needed on the other side - in the GUI. If a Core callback needs to update the interface, is is usually not possible to do it from a background thread. In macOS Cococa GUI framework, an error is shown in the debugger when trying to modify the GUI from background thread. To be able to modify views from callbacks, we have to ensure that the modifying code will be executed in the main thread. This can be done in a similar way as in asio, because user interfaces use the same queue concept (usually called *event loop*). To sum it up, data synchronization between threads in Shere are constructed using two concurrent queues containing runnable tasks with immutable data.

3.8 Storage

Notes created in Shere are stored on disk. As stated in the functional requirements, this storage must be published so other applications could be built using the same note storage. In this section, the note storage specification will be defined. This specification is then implemented in Shere in the Storage module.

3.8.1 Note Storage specification

This section contains the specification of the notes are stored on a disk. It is built around plain files and directories so any external application could easily work with notes without additional dependencies except filesystem access. The

structure is inspired by the Git objects structure [42]. The main structure in the specification is a note. Notes have two attributes:

- *Identifier* is a 20 byte long sequence represented as a text string in Base64 encoding. It uniquely identifies the note. It is created as a SHA-1 hash of the time the note was created.

- *Content* is the text that users need to store. It is formatted using the *Shere Markdown* syntax (described in Section 3.9). Note content is stored directly in a plain text file.

All notes are stored in a user-specified directory. This directory contains a subdirectory `notes`. To avoid very large number of files in this directory, the following directory structure is defined: there are three levels of subdirectories created from the note identifiers. First two characters define the name of the first subdirectory and the next two characters define the name of the second subdirectory. The rest of the characters define the name of the third subdirectory. In the third subdirectory, file `_note.md` containing the text note is placed. In future, also file attachments and images will be placed in the folder so all the files belonging to one note are in the same directory. Figure 38 shows where a note with identifier `a94a8fe5ccb19ba61c4c0873d391e987982fbbd3` is stored using the specification.

Note that the folder structure is not the same as the user-defined note organization structure. In applications that use tree-organization this can be easily done, because the notes organization structure can always correspond to the notes storage structure. When using nested-tag organization, this is not possible, because one note can belong to multiple tags. It would have to be duplicated or soft-links would have to be used. The first solution is data-inefficient and increases complexity when for example deleting a note. The second solution introduces dependency on filesystem types that support soft-links.

One of the advantages of using files and directories is that it is possible to use data synchronization software independent on the note taking applications. When the storage directory is synchronized via generic file synchronization software (as Dropbox, Google Drive, etc.), the notes will be available across the synchronized devices.

In future, including images and files into notes is planned, so this NSS would have to be extended to enable attachments retrieval and storage. An idea is to append the attachment filename after the note identifier separated by a slash: `a94a8fe5ccb19ba61c4c0873d391e987982fbbd3/dog.jpg`. However, using note attachments was not implemented yet and thus will not be discussed in this thesis.

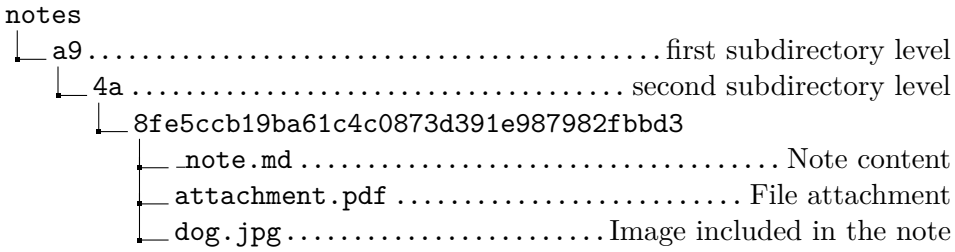


Figure 38: Example of a note stored in the NSS structure

3.8.2 Storage classes

Note Storage Specification in Shere is implemented in a Storage module. The note storage can be simply viewed as a key-value store. The key is the note identifier and the value is the note text. Based on this observation, `iStorage` interface containing all methods required for note storage manipulation was created. The interface is independent on a real storage, which allows to create multiple implementations using different storage. In Shere, three classes implementing the `iStorage` interface was created:

- `RamStorage`: it was the first implementation of the interface. All the data are stored in RAM only. This causes data lost after the application is terminated so this class could be used for development purpose only. The real implementation was very easy: simply `std::map<string,string>` was wrapped around the `iStorage` interface.

- `Storage`: this class implements filesystem access using `boost::filesystem` library. Each modification is immediately saved into the filesystem.

- `CachedStorage`: to avoid writing the entire note on the disk after a single character changes, a cached variant of the storage was created. It both previous implementations. If a note is modified, it is not immediately written on a disk, but it is temporarily keeped in RAM. After a certain period of time, all modified notes are saved. This reduces writing and optimizes performance of recently modified notes. Saving notes can be also forcen to happen immediately, which is required when for example the application terminates and there are still unsaved notes. Cached storage and the timing was implemented using the asio library.

Figure 39 shows the `iStorage` interface and its hierarchy. Note that `CachedStorage` does not inherits from `Storage`, but it includes its instance, because it reuses filesystem functions and extends them with a caching layer.

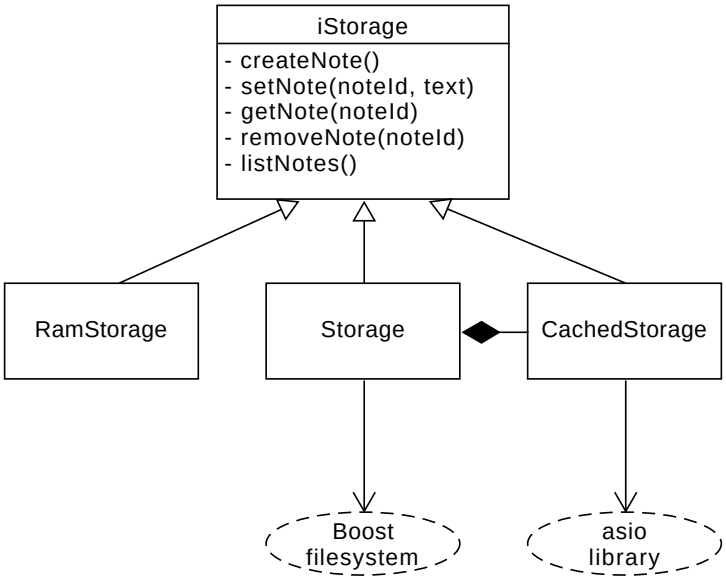


Figure 39: Storage class hierarchy

3.9 Markdown

There are several requirements for Shere related to markdown. The application has to implement markdown rendering using *seamless realtime preview* technique, the syntax have to support to-do items and tags and the to-do items can be aggregatable. These requirements imply that the implementation of markdown functionalities is one of the fundamental parts of the Shere development. It is also one of the biggest difficulties. Many parts have to be designed: which components will be used, where they will be placed in code (GUI or Core) which libraries will be used and how the tags and to-do items will be extracted from the notes (for notes organization and tasks aggregation). Using seamless realtime preview also increases complexity of the implementation, because the markdown special characters must be dynamically hidden and displayed based on the cursor position. To understand the decisions made in Shere development, the analysis foregoing to the used approach has to be described.

3.9.1 Architecture analysis

During the research about markdown parsing and rendering, it turned out that several aspects affect the design of the markdown functionalities. The first thing I analysed was how markdown notes are displayed in the existing applications; how the approaches differ in cross-platform and non cross-platform

applications and which technologies and libraries are used. It turned out that all applications mentioned in Chapter 2 that support markdown (Evernote, Bear, Typora and Dropbox Paper) render the note content using HTML. It is not surprising in Typora, because it is written using web technologies only. However, even that Evernote and Bear and mobile versions of Dropbox Paper are native applications, they contain an embedded web view that renders the note content using HTML, JavaScript and CSS. There also exist libraries for converting markdown text into platform-specific views, for example: *Cocoa-Markdown* [43] is a library for markdown parsing and rendering for iOS and macOS. It transforms a markdown string into `NSAttributedString`, which is a structure for rich text in Apple operating systems. However, the advantage of using web views is obvious: HTML rendering allows cross-platform text styling, allows using existing web frameworks and libraries and (unlike rich-text only approaches as NSAttributedString) any content as in a classical web pages can be included in the note: custom-designed elements, images, videos, forms,... Because of the flexibility and the fact that embedded web views can be used in all platforms[7], writing an editor view and using in all platforms in their respective web views seems to be the efficient approach how to avoid editor code duplicates and still be able to adapt the UI for the each platforms separately. In fact, the whole application can be written using web technologies only. However, it would complicate using native design guidelines so the optimal trade-off is to create editor as a web view component (because the editor may look the same on all platforms) and the rest of the GUI using native frameworks.

The second aspect that needs to be considered relates to code placement of the component that converts the plaintext into HTML. One approach is to use JavaScript parser inside the web view. The editor view would receive a plain-text string and it would convert it into HTML. The second approach is to include the parsing code inside the Core. The editor component would receive a string containing the note converted into HTML. Using JavaScript library inside the web view seems as an efficient solution, because and there are many extendable javascript parsing libraries and the same JavaScript code could be used in every platform. [44]

The third aspect of markdown usage analyses how the text would be edited. This supports using JavaScript parser, because the rendering, parsing and editing logic could be integrated inside the editor view. It would allow create the editor as a standalone web component with simple interface that receives plain-text as an input and provides plain-text representing the modified note.

Without to-do and tagging requirements, using JavaScript for all markdown-related features is ideal. However, the application has to be able to index to-do items and tags. Both of this elements are a part of the Shere markdown syn-

[7] In macOS, iOS and Linux, `WKWebView` from WebKit is available; Microsoft Windows provides `MSHTMLWebView` and Android offers a `WebView` class

tax. For indexing purposes, the application has to be able to extract tags and to-do items. A simple solution seems to be extending the used JavaScript markdown parser to include to-do and tagging tokens. A disadvantage of this solution is that the Core would be dependent on the JavaScript parser which would break the entire architecture. Additionally, these features relate to application logic. These facts cause that the markdown parsing has to be implemented in the Core.

Considering the aspects from the previous paragraphs, the following concept is created: the editor component is a web component that receives a HTML string (converted markdown note) and implements the realtime seamless preview technique. The HTML string is created in Core by the markdown parsing and conversion module. This module can be also used for extracting tags and to-do items for indexing purposes. The last thing that needs to be designed is the Core module responsible for markdown functions. This module is responsible for conversion a markdown-formatted string to HTML and extraction of tags and to-do items from the string.

The last thing that needs to be prepared before the actual implementation is to select a markdown parsing library and to design the extension of the parser. Because new elements are required, the library has to allow extending markdown syntax with new tokens. Even there are many markdown parsing libraries [44], few of them is written in C or C++ and none of them is written in a modular way to easily support custom extensions (instead of some libraries written in JavaScript). After this research I decided that I will implement markdown parser on my own.

3.9.2 Parsing-expression Grammar

During analysis which algorithm is suitable for parsing markdown, the initial idea was to specify context-free markdown grammar and use an existing parser to parse the text using this grammar. [45] However, it turned out that the output using CFG parser would be ambiguous. For example: during parsing `***this text***` it would be not possible to decide in which order should bold and italic tokens be parsed. This is the reason why many parsers use an modified alternative to context-free grammars called *parsing-expression grammar*. In computer science, a parsing expression grammar, is a type of analytic formal grammar, i.e. it describes a formal language in terms of a set of rules for recognizing strings in the language. The formalism was introduced by Bryan Ford in 2004 [46] and is closely related to the family of top-down parsing languages introduced in the early 1970s. Syntactically, PEGs also look similar to context-free grammars (CFGs), but they have a different interpretation: the choice operator selects the first match in PEG, while it is ambiguous in CFG. This is closer to how string recognition tends to be done in practice, e.g. by a recursive descent parser. Formally, a parsing expression grammar consists of:

- A finite set N of nonterminal symbols.
- A finite set Σ of terminal symbols that is disjoint from N.
- A finite set P of parsing rules.
- An expression e_S termed the starting expression.

Each parsing rule in P has the form $A \to e$, where A is a nonterminal symbol and e is a parsing expression. A parsing expression is a hierarchical expression similar to a regular expression, which is constructed in the following fashion:

1. An atomic parsing expression consists of:
   - any terminal symbol,
   - any nonterminal symbol, or
   - the empty string ε.
2. The Given any existing parsing expressions e, e1, and e2, a new parsing expression can be constructed using the following operators:
   - Sequence: $e_1 e_2$
   - Ordered choice: e_1 / e_2
   - Zero-or-more: $e*$
   - One-or-more: $e+$
   - Optional: $e?$
   - And-predicate: $\&e$
   - Not-predicate: $!e$

An example of constructing rules for markdown heading look like this:

$L \to$	`'##?#?'`	heading levels 1-3
$S \to$	`' '`	single space
$N \to$	`'\n'`	new line
$A \to$	$(!N)*$	sequence of any characters except new line
$H \to$	$LSAN$	complete heading structure

The fundamental difference between context-free grammars and parsing expression grammars is that the PEG's choice operator is ordered. If the first alternative succeeds, the second alternative is ignored. Thus ordered choice is not commutative, unlike unordered choice as in context-free grammars. This makes PEG suitable for markdown parsing. Each token will be represented as a grammar rule and the order of the rules would solve ambiguities.

3.9.3 Parser architecture

The basic idea of markdown parsing is to generate an abstract syntax tree using the markdown PEG and convert the syntax tree to HTML. In Shere, markdown parsing is implemented in Parser module. This module contains four types of classes:

- *Token*: this classes represent the markdown elements: bold, heading, link, etc. Each specific token has its own attributes: for example heading contains the information about the heading level and the text, the link contains the text and the URL.
- *Tokenizer*: the purpose of tokenizers is to found its respective token in the provided text and generate a *Token* instance. This classes implement resolution of the expressions from the grammar.
- *Parser*: it is a wrapper that contains a list of tokenizers and implements the generic parsing algorithm: for each tokenizer a part of the text is provided. If the tokenizer finds a token, the loop is repeated starting from the first tokenizer. The list of tokenizers is the fundamental part of the PEG implementation, because the order of tokenizers in the list implies the priorities of tokenizers and thus corresponds to the PEG rule priorities.
- *Convertor*: after the abstract syntax tree is created, convertors are used for traversing the tree and generating HTML (or other data) from it.
- *Parsing context*: It contains all the information for tokenizers during parsing: the original text, the abstract syntax tree and a *cursor* - a number specifying the position of the first unparsed character in text.

Using these class types, we can design the complete markdown parsing module architecture and create specific classes. Firstly, an abstract code skeleton can be created even without specific tokens and tokenizers using class `GenericParser` and abstract class `aToken` and interface `iTokenizer`. `GenericParser` implements the *parsing loop*. This loop is the fundamental part of markdown parsing. Figure 310 shows the code snippet of the loop.

The design of markdown parsing in Shere is very flexible. Having the abstract algorithm implemented, specific markdown tokens can be developed independetly using the following procedure:

1. `aToken` subclass must be created, e.g.: `BoldToken`, `TagToken`, ...
2. A tokenizer implementing `iTokenizer` interface must be implemented (for example `TagTokenizer`, ...). It has to properly find and create its token.
3. The tokenizer has to be added into tokenizer lists in the module.

```
shared_ptr<aToken> GenericParser::parseSubstring(
    const string & text,
    unsigned long start,
    unsigned long len)
{
    ParseContext ctx(text);
    ctx.cursor = start;
    auto cursorEnd = start + len;

    while(true)
    {
        for(const auto & tokenizer : tokenizers)
        {
            if(ctx.cursor >= cursorEnd)
                return ctx.firstToken();

            bool tokenFound = tokenizer->parse(ctx);
            // Token is already appended into in AST
            // (responsibility of tokenizers)

            if(tokenFound) break;
        }
    }
}
```

Figure 310: Generic parsing loop

Note that the last step relates to more than one tokenizer lists. It has not been not mentioned yet, but it relates to the specific token types. There are three basic types of tokens implemented in Shere: *inline*, *container* and *block* tokens. Container tokens represent a text separated by a new line character. It can be a text paragramph a list entry and so on. Container tokens contains subtokens - inline tokens, for example italic text, a link or a plain text. Block tokens can span more than one line (e.g. code block). In Shere, there are planned to be implemented in future. Container tokenizers have their own tokenizer list that contains inline tokenizers. If a new inline tokenizer is created, it has to be appended in all container tokenizers that should recognize this token. Using this procedure, markdown tokens in table 33 were implemented.

Using tokens, tokenizers and the parsing loop, an abstract syntax tree can be constructed. To be able to convert the syntax tree to HTML, extract tags and to-do tokens from it, *convertor* classes has to be implemented. A convertor is a class that traverses the syntax tree and for each token a method

Token	Syntax	Additional info
ItalicToken	`*italic text*`	Standard italic text
BoldToken	`**bold text**`	Standard bold text
HeadingToken	`## heading`	There can be 1-3 pound characters indicating the heading level
InlineCodeToken	`‘inline code‘`	The code inside the backticks uses monospaced font
UnorderedListToken	`* Unordered list`	Standard unordered list
TaskToken	`- todo item`	Unfinished todo item. If the task is finished, plus sign is used instead of minus sign.
TagToken	`#tag/subtag`	Tag path. Each sub-tag is divided by a slash.
LinkToken	`[link text](link URL)`	Link to a note or a web page.
TextToken	`plain text`	Unformatted text
ParagraphToken	(none)	Container token for text.

Table 33: Markdown tokens implemented in Shere

based on the token type is invoked which imply that the visitor pattern [38] is a suitable approach to convertors design. The most simple convertor in Shere is `aConvertor` which provides default (empty) visit methods for every token. In order to create a custom convertor, `aConvertor` class must be subclassed. Using method overriding, custom methods for tokens can be created. In Shere, four convertors were implemented:

- `HtmlConvertor` converts the abstract syntax tree to HTML code suitable for rendering in the web view.
- `TagExtractor` ignores every token type except `TagToken`. After traversing the whole syntax tree, a set of tags is returned.
- `TaskExtractor` ignores every token type except `TaskToken`. It returns a list of tasks.
- `WordExtractor` returns a set of words that are in the note text. A word is recognized as a sequence of characters separated by a space. The purpose of this class is further described in Section 3.11.

The whole markdown parsing functionality is wrapped in `Parser` class. The main method for parsing is `shared_ptr<aToken> parseNote(const string & text);`. It returns a pointer to the root of the abstract syntax tree (the first token). In order convert the syntax tree to HTML, method `string convert(shared_ptr<aToken> firstToken);` from class `HtmlConvertor` can be used.

3.10 Note Editing

According to the architecture specified in Section 3.9.1, the web component responsible for the seamless realtime markdown preview receives already converted note in HTML. Its responsibility is to show the note content in HTML and to implement special characters displaying and hiding based on the cursor position (the seamless realtime preview). Because of the fact that the web view does not work with plain-text note content, it is not possible to edit the text in the web component only. Another fact is that when an user edits the text, the formatting has to be reparsed, because the changed text can have different meaning (for example deleting the space from the `# heading` should create a tag token `#heading`. This forces note editing features to be inside the Core. Since the note content is in plain-text, the only data required for successful editing is the selection indices and the edit string. The selection indices are two numbers describing the range selected by the GUI caret. The plain-text selection can be computed from the HTML converted text, because we know, how many characters the elements have in plain-text. The computation of the selection indices is more explained in Section 3.14. The edit string contains the information about the text change that the user performed. Generally, there are two main types of change: insertion or removal. The insertion is represented by the string to be inserted. If the users wants to perform a removal, the edit string contains backspace or delete character only (in ASCII, characters with code 8 or 127).

With note editing features, the complete markdown editing functionality can be achieved. When an editing occurs, `Core.notes.editNote` action is called with all the necessary information for editing (note identifier, selection indices and the edit string), the note is modified and the new content is parsed and rendered back in the web view. The overall editing process and all markdown-related components are illustrated in Figure 311.

3.11 Indexing

Indexing is a technique that reduces search time by extracting metadata from the data and storing them in data structures optimized for fast retrieval. It is usually used for large datasets where the scanning of the entire dataset takes a long time. In Shere, there are three types of data that have to be indexed:

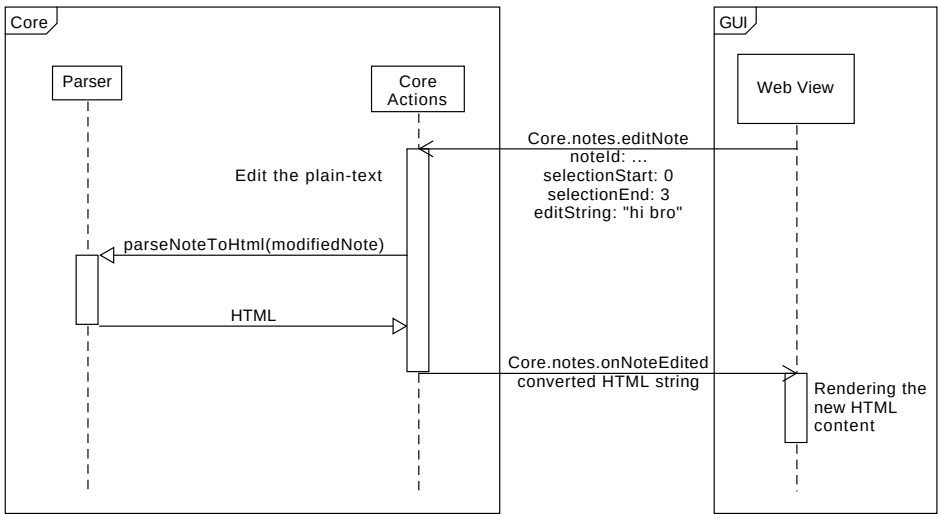


Figure 311: Markdown editing process

to-do items, tags and words. To-do items indexing allows to show all tasks from the entire collection of notes without repeated loading of all these notes. After the initial scan, the tasks are keeped in RAM and quickly accessible. For each note with tasks, a list of its tasks is stored in RAM. Indexing tags is required for fast navigation through notes organization using nested tags. The index is stored in the opposite way as in the note storage: for each tag, its notes and sub-tags are stored in RAM. Indexing words is used for full-text searching, as described in Section 3.12.

The entry to the indexing features in Shere is class `NoteIndex`. It provides methods for reindexing notes, removing a note from index and getting indexed data. `NoteIndex` class contains three classes: `TaskIndex`, `TagIndex` and `FulltextIndex`. They are responsible for indexing their respective metadata from notes. The indexing in these classes uses *convertors* for extracting the data from notes as described in Section 3.9.3.

All indexing classes are built around the `NoteInfo` class. It is a simple data structure dat encapsulates basic information about a note. For the memory efficiency, there is always a single `NoteInfo` instance for each note. These instances are shared among the indices. Each index then contains a form of a key-value data structure where the values are `NoteInfo` instances. `TagIndex` contain a `TagTree` structure. This tree structure keeps for each node its name (name of the tag), list of `NoteInfo` instances (notes that belong to the tag) and a set of sub-tags. `TaskIndex` contains a simple key-value map that for each note identifier (with at least one task) stores a list of its tasks. And the `FulltextIndex` contains a prefix tree [1] that for each word contains a set of notes that contain this word. The structure of the indexing module in Shere

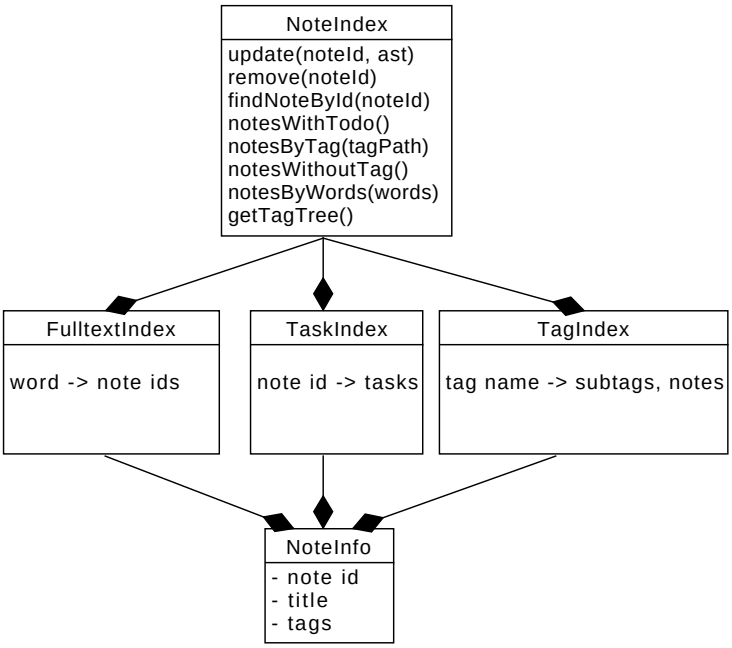


Figure 312: Simplified structure of indexing module

is visualised in Figure 312.

3.12 Search

The last module implemented in the Core is the search module. It filters the indexed data by the search query. In a simple full-text search engine, the search query contains only words separated by space. Using an advanced search engine, special search tokens can be used for more precise query specification. For example, as descripted in Section 2.3.3, several special tokens can be used in Bear: `@todo`, `@image`, etc. To implement this feature, splitting the search query by space is not enough, the query needs to be parsed to recognize the special tokens. In Shere, I decided to create similar search feature as in Bear. In this thesis only three special tokens was created:

- `@todo`: search only notes that have at least one unfinished to-do item.
- `@untagged`: search only notes without tags
- `#tag/subtag`: search only notes containing this tag (or its sub-tags))

The searching process then has to have to phases: firstly, the search query must be parsed to recognize the words and special tokens. Secondly, the indexed data must be filtered and to get the search result.

```
struct Shere::SearchQuery
{
    bool todo;
    bool untagged;
    std::vector<std::string> tags;
    std::vector<std::string> words;
    std::vector<std::string> phrases;
};
```

Figure 313: Data structure for parsed search query

Since the generic parsing code is already written, it can be reused to parse the search query. The extension procedure (as described in Section 3.9.3) can be used. New `aToken` subclasses `WordQueryToken`, `TodoQueryToken` and `UntaggedQueryToken` were created as well as their tokenizers. `SearchEngine` class was constructed, which contains a `GenericParser` instance with the search tokenizers to be able to parse search query. It also contains a reference to the `NoteIndex` instance to have access to the indexed data.

After the search query is parsed, the notes are filtered to retrieve only notes that fits into the query parameters. For this purpose, the search query has to be transformed into a data structure more suitable for data filtering than an abstract syntax tree. Using the visitor pattern again, the abstract syntax tree of the query is traversed and a `SearchQuery` instance is created. `SearchQuery` is a simple object that holds all the information about the searching, as described in Figure 313.

Using the prepared query information, the indexed data can be filtered. For each indexed `NoteInfo` structure, comparing the query parameters with the indexed data decides if the note belongs to the search result set. Using this algorithm a set of successful `NoteInfo` instances is created as a search result.

Note that this algorithm scans each indexed `NoteInfo`. For many cases this would not be necessary, for example: if there is a `@todo` token part of the query string, we are interested only in notes with to-do items. The starting set could be then only to-do notes. If the number of all notes in the storage is orders of magnitude bigger than the number of notes with unfinished to-do items, this optimization can speed up the searching. However, this optimization was not implemented yet, but is is planned in future, as mentioned in Section 3.19.

Search feature is accessible from the `Core.notes.search` Action. It receives a search query string. The search result is provided as a callback `Core.notes.onSearchResult` containing a set of found notes. The overall architecture of the search module is shown in Figure 314.

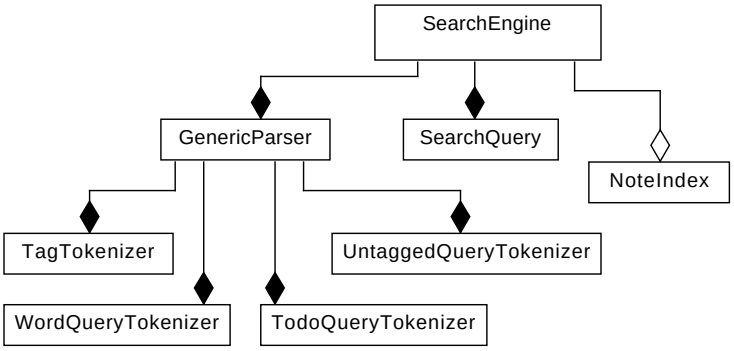


Figure 314: Structure of search module

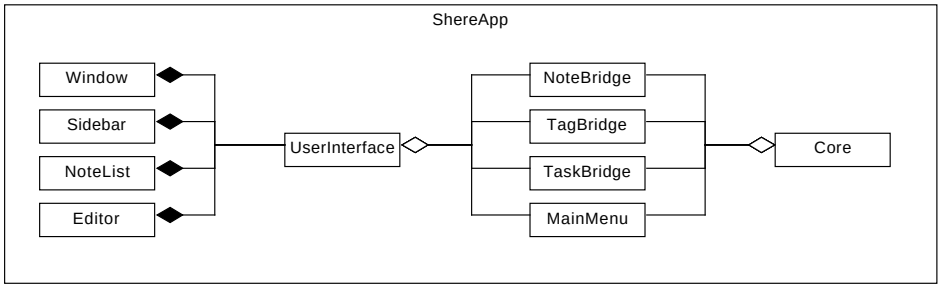


Figure 315: Key classes in GUI application

3.13 MacOS Application Architecture

Next part of the development is to create the GUI application using the Core library. In this thesis, I decided to create an application for macOS using its GUI framework called *Cocoa*. It supports Swift and Objective-C languages. The language used for writing the GUI code was chosen Objective-C++, because it allows interchangeability between C, C++ and Objective-C and thus makes the easiest to link the Core with the GUI.

The key classes in the GUI are depicted in Figure 315. It is clearly visible that the *bridge classes* connect the Core with the user interface. They are the only classes that implement the logic related to both Core and the user interface. All classes contained in `UserInterface` class are not dependent on Core and. They only contains logic for showing provided data to the user. Using this architecture then allowed me to create highly modular code without complex dependencies.

In the following sections, the fundamental components of the GUI are described. For each component, its responsibilities are defined as well as input and output interface. The user interface is designed as a single-window

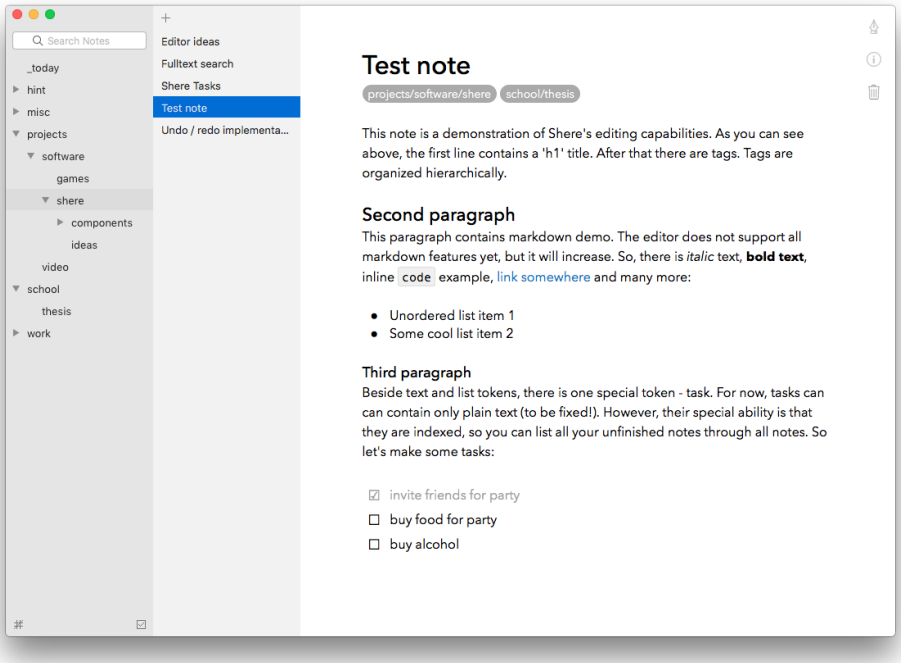


Figure 316: Shere window

application. The window containing all the GUI elements is shown in Figure 316.

3.14 Editor View

The most important graphical component is the note editor. Editor view implements the seamless realtime preview of markdown text. Despite of the name, it can not edit the markdown text directly, but it provides *edit-events* to the Core. As previously described in Section 3.9.1, the GUI editor view implements read-only access to the markdown text converted to HTML. In the following section, these two functionalities are described: *seamless realtime preview* and *note editing*.

3.14.1 Seamless realtime preview

Seamless realtime preview hides all markdown special characters (as asterisks around a bold text, brackets around links and so on) and shows only special characters around the actual caret position. Showing plain-text markdown around the caret allows to edit notes with the *pure markdown* feeling while

the rest of the note keeps easy to read, because the special characters are hidden.

Showing certain parts of the markdown text formatted in HTML requires that the editor can decide if the HTML element represents special characters or a normal text. If the HTML would be used for syntax highlighting feature only, using simple HTML tags would be enough. For example markdown text `# Heading\n**hello**` would be converted to

```
<h1># Heading</h1>
<p>
  <b>**hello**</b>
</p>
```

and so on. However, for seamless preview, this approach is not enough. The editor does not recognize the asterisks as special characters, they are only a part of the bold text. More complex HTML structure needs to be designed in order to be able to distinguish special characters. For this purpose, `HtmlConvertor` class in Core needs to be modified: instead of wrapping the whole token into a single HTML element, the token has to be divided into separate elements based on their type - *special* or *normal*. The example has to be converted into more complex structure:

```
<h1>
  <span class='heading-prefix'># </span>
  <span class='heading-text>Heading</span>
</h1>
<p>
  <b>
    <span class='bold-asterisk'>**</span>
    <span class='bold-text'>hello </span>
    <span class='bold-asterisk>**</span>
  </b>
</p>
```

Using separate classes for special and normal character allowed to programatically hide all special characters. In the example, all elements with class `bold-asterisk` can be treated as special characters. However, during development it turned out that many tokens share the similar functionality of showing and hiding the special tokens. It is not efficient to explicitly enumerate which classes should be hidable. To generalize this attribute of an element, class `hidable` was created. The editor was implemented that each HTML element having this class would be hidden if the caret is not near it. It allowed to implement the hiding feature independently on specific tokens. Using the example again, the converted code has to be modified:

```
<h1>
  <span class='heading-prefix hidable'># </span>
```

```
  <span class='heading-text>Heading</span>
</h1>
<p>
  <b>
    <span class='bold-asterisk hidable'>**</span>
    <span class='bold-text hidable'>hello</span>
    <span class='bold-asterisk hidable>**</span>
  </b>
</p>
```

The second part of the seamless preview functionality is the caret handling. The first thing needed is to enable caret functionality in HTML, because at default, there is no caret in HTML documents. To enable it, `contenteditable="true"` attribute has to be added to the top-level HTML element. The next part of the caret handling is plain-markdown position computation. Because is note possible to get the caret offset from the top HTML element (`window.getSelection()` returns the selected node the plain-text offset only), the markdown caret offset has to be computed manually. The other problem is that some HTML elements have different number of characters that the same elements in markdown. For example, a task prefix in markdown is composed from the minus characters and a space. In HTML, the task prefix is converted into a checkbox spanning one character. To be able to determine the number of characters in plain-markdown in a general way independently on specific markdown elements, additional attribute `data-width` can be added to HTML elements. Using this HTML structure, the computation is able to compute the plain-markdown caret position based on the HTML-markdown caret position. The caret computation algorithm traverses the DOM tree backwards starting from the selected node and increases the offset with the length of the previous nodes. The algorithm is illustrated in Figure 317.

However, `contenteditable` attribute enables direct editating of the HTML content which is not desirable in Shere. To disable it, keydown events had to be prevented using JavaScript. Using these algorithms and workaround it was possible to implement seamless rendering with caret position mapping to plain markdown.

Having the realtime preview implemented, it is possible to compare it with *syntax highlighting* technique. The same markdown text was used in Shere with both seamless preview and syntax highlighting (hiding special characters was deactivated). In Figure 318, both approaches are shown. It is clearly visible, that the realtime preview is more readable and non-disturbing.

3.14.2 Note editing

The second part of the editor functionality is sending edit events. When an user wants to change a note, for example by removing the character before

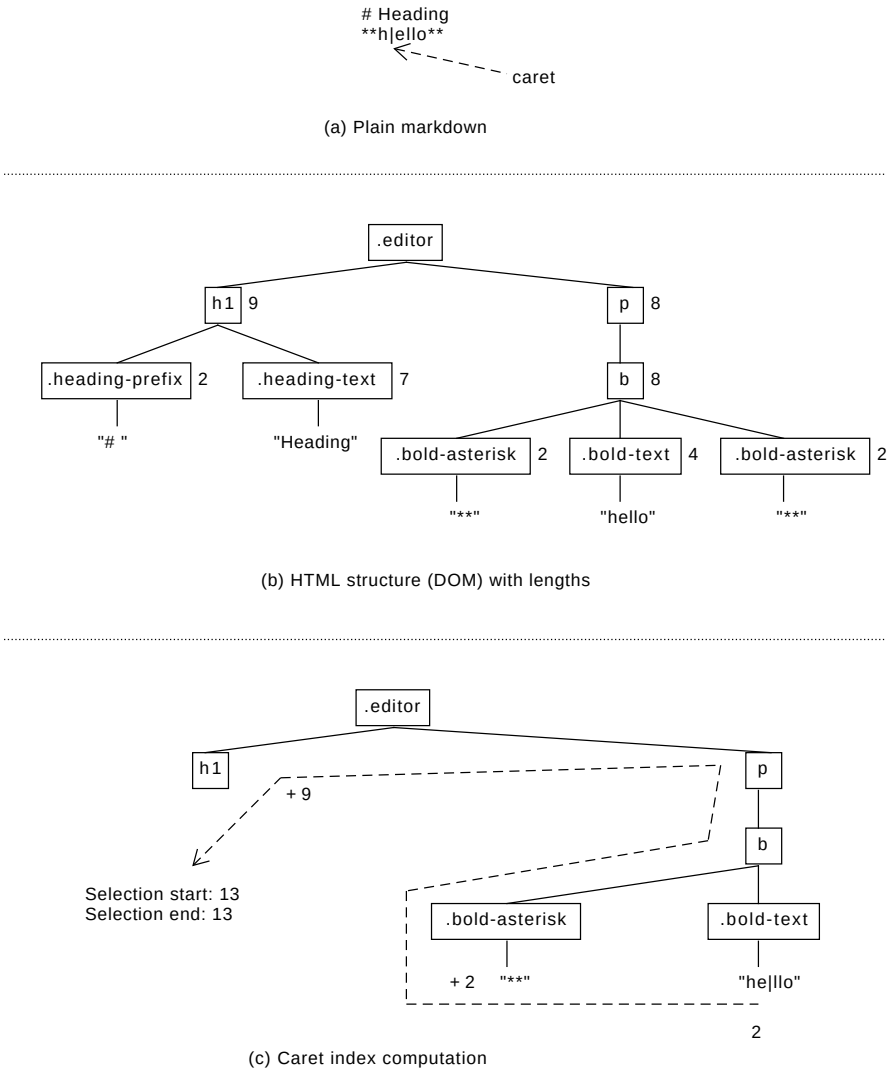
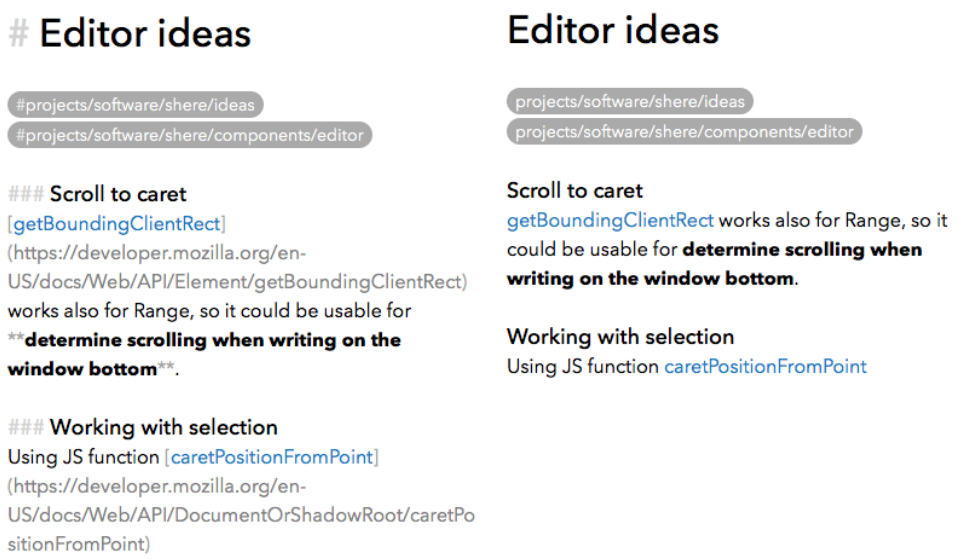


Figure 317: Caret handling



(a) Syntax highlighting only (b) Realtime seamless preview

Figure 318: Comparison between different rendering techniques

the caret, the *backspace* click is detected and all information required for the editating is sent to the Core. As introduced in Section 3.10, there are three information required for the note editating:

- Note identifier
- Selection index (or indices, if the selection spans over multiple characters)
- Edit string

Note identifier is a string that is set when the note content is loaded. The selection indices can be obtained using the traversing algorithm described in previous section. The last piece of the required data is the editing string. This string is created when the user clicks a key on the keyboard or wants to cut or paste a selected text. This actions are in web view handled via JavaScript events `keydown`, `paste` and `cut`. The code that prevents these events to edit the HTML content directly (as mentioned in the previous section) has to be extended to catch the keys and the content to be pasted or cut. When such an event occurs, all the pieces of required data are gathered and sent to Core as already explained in Figure 311.

3.14.3 Web view in macOS

The last important thing related to note editing that needs to be described is the approach, how the web view communicates with the Core. Initial

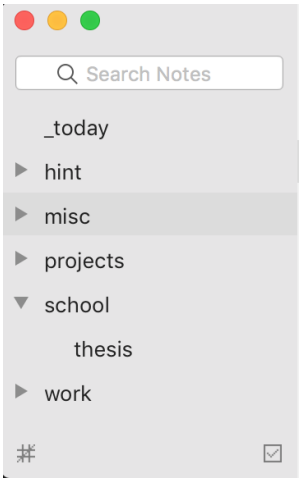


Figure 319: Shere sidebar

idea was to bridge the components using WebSockets, because both of the languages supports it and it would be cross-platform. During studying Cocoa framework and WebKit functionalities in macOS I found out, that the web view in macOS (`WKWebView` class) supports direct communication between JavaScript code in the web view and Objective-C code. It is easier than using WebSocket, only single setting up is required in Objective-C using `addScriptMessageHandler` method to enable calling Objective-C code from JavaScript. The second way, calling JavaScript code from Objective-C can be done using `evaluateJavaScript` function in `WKWebView`.

A disadvantage of using native communication abilities instead of cross-platform WebSocket is that it has to be implemented separately for each platform. However, because the WebSocket implementation would require additional WebSocket library and difficult connection handling, more efficient solution seemed to use separate (but very easy to implement) platform-specific methods.

3.15 Sidebar

Other GUI components complement the editor and provide functionalities with tag tree. As shown in Figure 316, other components in Shere window are the sidebar and the note list. The sidebar has three parts: search field, tree view showing the tag tree and buttons on the bottom. More detailed view of the sidebar is shown in Figure 319.

The tag tree structure is shown using `NSOutlineView` class which is the optimal choice for showing hierarchical data. The bottom part of the sidebar contains two buttons: the first allows to show list of notes that have no tag. The second button shows the aggregated tasks from all notes.

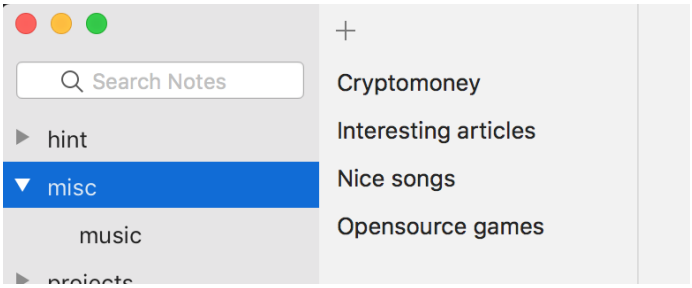


Figure 320: Sidebar and note list

3.16 Note list

The third main GUI component is the note list. It shows notes containing the tag selected in the sidebar or the results based on the search query. The list is implemented using Cocoa's `NSTableView`. Above the table view, an *new note* button is placed. In the current implementation, when a new note is created, the content is empty. In future, creating note with the same tag as the currently selected tag is planned. The note list is shown in detail in Figure 320.

3.17 Testing and Documentation

This section describes supporting parts of the development process: *testing* and *documentation*. In this thesis, these parts are described after the implementation. However, both of them was done concurrently with the implementation.

3.17.1 Testing

Automated testing is the fundamental activity in Shere, because many Core modules was developed before they were used in GUI, so without unit test it would not be easily possible to verify the proper functionality. Tests provide check the basic component usage and corner cases.

For Core testing, *Catch2* unit testing library was used. [47] Its main advantages are easy integration (single header file only), easy syntax and integration with CLion. The code 322 shows an example of testing the indexing functionality. During the development, hundreds of assertions were created which simplified debugging in many cases.

Another advantage of using Catch2 is the integration with CLion IDE. It is easy to run the tests using custom target configurations. A big advantage is to set up temporary filters to avoid running all the test during development a single component. The integration is illustrated in Figure 321.

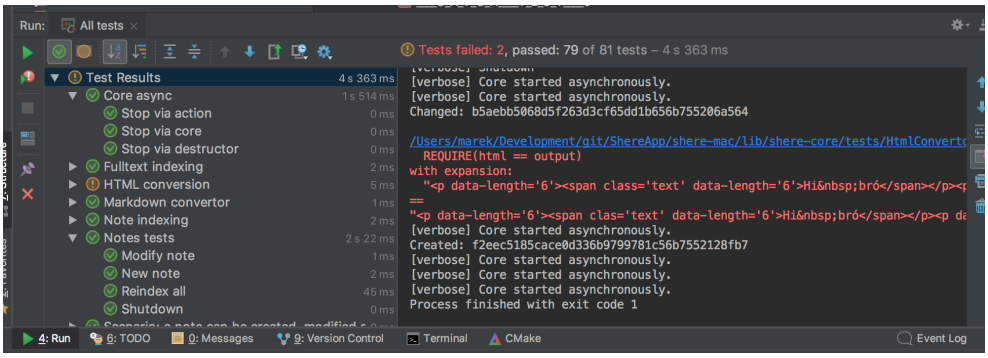


Figure 321: Catch2 integrated into CLion

```
TEST_CASE("Note indexing", "[index]")
{
    NoteIndex index;
    Parser parser;

    auto ast1 = parser.parseNote("Some title\nHi bro");
    auto ast2 = parser.parseNote("Other\note");
    index.update("note1", ast1, "Some title");
    index.update("note2", ast2, "Other");

    SECTION("Retrieve")
    {
        auto indexEntry = index.findByNoteId("note1");
        REQUIRE(indexEntry != nullptr);
        REQUIRE(indexEntry->noteId == "note1");
        REQUIRE(indexEntry->title == "Some title");
    }
}
```

Figure 322: Unit test example using Catch2

3.17.2 Documentation

Documentation is an integral part of the software product. It is a set of texts and other data that are relevant to the whole project development. In Shere, the most important parts of the documentation are the requirements, information diagrams of the architecture and individual components and source code documentation. In addition to this, compilation and launch instructions are also part of the documentation.

The requirements are described in Section 3.1. It contains description of the fundamental functionalities.

Source code comments improve the understandability of the code for developers. In definition of methods and interfaces, it explains the purpose of their usage and helps, for example, to understand algorithms or complex parts of code.

In addition to the text description, it is also important to illustrate the relationships between the components and their cooperation. A properly designed diagram can help to understand the code better than a large amount of text. All the diagrams in the this thesis are thus part of the project documentation.

3.18 Evaluation criteria

To be able to decide if Shere has some benefits, the evaluation criteria from previous chapter can be used. In this section, these criteria are used to evaluate Shere. It is necessary to mention, that most of the existing applications has many developers and it is not in the scope of this thesis to implement all desired features. However, many ideas are planned in future development.

3.18.1 Note Workflows

For now, Shere offers *creative workflow* that is limited to text format only and links to other notes. Advanced features as diagrams or drawing are not supported, but it is possible to extend the application to support them.

To support *data workflow*, links to notes and webpages are implemented. Images and files are not supported in GUI, but the note storage specification is prepared for images and file attachment extension.

Agenda workflow is supported with to-do item feature. To-do items can appear anywhere in note contents. Tasks can be aggregated and displayed all from the whole note repository. An important future extension is to allow set a deadline to a task and task-filtering features.

Sharing workflow offers the same capabilities as plain files sharing: using a dedicated file sharing software. Other options as note exporting to various formats as are planned.

3.18.2 Notes Organization

Notes in Shere are organized using nested tags allowing both hierarchical and non-hierarchical ways of organization in a simple way. Notes are appended into a tag by writing the tag into the note content.

3.18.3 Search

In order to support search, full-text search engine is implemented. In addition to filtering by words, special search tokens are available allowing filter for example only notes with unfinished tasks or notes with specified tag.

3.18.4 User Interface

User interface of Shere uses native GUI library for macOS. It is minimalist and shows only the most necessary features. It allows non-distracting both writing and reading.

3.18.5 Markdown

Markdown is the only way how to format text, so it is well supported. Shere uses custom syntax inspired by Polar syntax from Bear. For rendering the text, *realtime seamless preview* technique is used, so the text is both easy to read and easy to write. In future, more markdown tokens are planned to be implemented.

3.18.6 Notes Storage

Notes in Shere are stored using plain text files and directories. The directory structure is specified and published as a part of this thesis which allows the storage to be used by other applications.

3.18.7 Supported Platforms

For now, only macOS is supported. However, the application logic is written as cross-platform C++ library, so it is possible to efficiently create GUI for other platforms. In future, Windows, Linux, iOS and Android and web application are planned.

3.18.8 Pricing

Both the Shere parts (`shere-mac` and `shere-core`) are published on GitLab [48] with permissive MIT license [49] that requires only preservation of copyright and license notices. Licensed works, modifications, and larger works may be distributed under different terms and without source code.

3.18.9 Summary

As already mentioned, a lot of features in Shere are planned in the future. Even that the project were developed for more than six months, it was not possible to implement all desired features to create an application comparable with note taking applications that are developed by many developers. In this thesis, the main focus was to implement the fundamental concepts that are further extendable, such as cross-platform core, markdown rendering web view and modular parsing and indexing modules.

3.19 Future Development Ideas

As in every project, there is almost always a possibility to improve the software by fixing errors or adding new features. In this section, the most important areas of further development are listed and eventually more described. The most important things to improve are related to notes content and manipulation. Besides that, there are many features decreasing user experience. Some of them are caused by lacking standard functions provided in most text editor, others are ideas for improving the user interface or extending the note content abilities:

- *Undo / redo support*: because the text editing does not rely on the embedded `contenteditable` functionality, the default command history can not be used and the undo/redo system has to be manually implemented.
- *Moving tasks and items*: many editors offer ability to move a list item of a task by mouse which prevents to cut and paste the desired notes and thus simplifies work with items.
- *Add more markdown tokens*: there are some markdown elements that are planned to be extended or added into the markdown syntax: code block, horizontal line, set deadline to tasks, ordered list, quote element, indent tasks and list elements and add images and files to notes.
- *Open multiple notes*: one of the future requirements related to GUI is to be able to open multiple notes at once, which is not possible yet.
- *To-do items sorting and filtering*: the task aggregation view should be able to select a subset of unfinished tasks, based on a tag, deadline date and so on.
- *Platform support*: Supporting more platforms is also one of the features required in future. Firstly, implementations for desktop platforms (Windows, Linux) are planned and after that mobile platforms and web.

Additionally, to enable scripting notes manipulation, command line interface could be added to Shere.

- *Save indexed data*: to avoid slow reindexing the whole storage, the indexed data could be saved on disk and loaded on launch to be able quickly navigate through the notes. item *Export to other formats*: abilities to convert notes to other format as PDF, HTML, DOCX or other markdown flavors.

There are many more ideas and issues that can be created or fixed. In this thesis, only the most important were mentioned.

Conclusion

In this thesis I defined the criteria set for note taking application evaluation, analysed existing note taking software, their main benefits and shortcomings. Based on the analysis I created a note taking application for macOS which addresses the found shortcomings. Its main features are minimalist UI and text rendering, markdown formatting, flexible nested-tag notes organization and open storage format. The application logic can be easily used in other platforms with native GUI. The code was covered by unit tests and documented. The project can be further developed by adding new features and supporting more platforms.

During the work on the thesis I used knowledge gained through the study on CTU, especially from software and web engineering. My personal benefits were that I gained a lot of knowledge related to text parsing, indexing, rich text formatting, GUI creating and many more. In future, I would like to continue developing Shere to become a comfortable note taking open source application.

Bibliography

[1] Knuth, D. E. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998, ISBN 0-201-89685-0.

[2] Bast, H.; Celikik, M. *Efficient Fuzzy Search in Large Text Collections*. Albert Ludwigs University, [quot. 2018-04-18]. Available from: `http://ad-publications.informatik.uni-freiburg.de/TOIS_fuzzy_BC_2013.pdf`

[3] Craswell, N. Neural Models for Full Text Search. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, New York, NY, USA, 2017, ISBN 978-1-4503-4675-7, pp. 1–2.

[4] Gruber, J. *Daring Fireball: Markdown*. [quot. 2018-04-08]. Available from: `https://daringfireball.net/projects/markdown/`

[5] MacFarlane, J. *CommonMark Spec*. [quot. 2018-04-12]. Available from: `http://spec.commonmark.org/0.27/`

[6] MacFarlane, J.; GitHub. *GitHub Flavored Markdown Spec*. [quot. 2018-04-12]. Available from: `https://github.github.com/gfm/`

[7] GitLab. *GitLab Flavored Markdown*. [quot. 2018-04-12]. Available from: `https://docs.gitlab.com/ee/user/markdown.html#gitlab-flavored-markdown-gfm`

[8] Fortin, M. *Markdown Extra: Syntax*. [quot. 2018-04-12]. Available from: `https://michelf.ca/specs/markdown-extra/`

[9] Klatsky, S. A. WYSIWYG. *Aesthetic Surgery Journal*, volume 23, no. 4, 2003: pp. 274–275, doi:10.1016/S1090-820X(03)00150-X, [quot. 2018-04-10]. Available from: `http://dx.doi.org/10.1016/S1090-820X(03)00150-X`

[10] Matt Stow. *WriteMe.md - A simple Markdown editor*. [quot. 2018-05-08]. Available from: http://writeme.mattstow.com/

[11] Benoit Schweblin. *StackEdit – In-browser Markdown editor*. [quot. 2018-05-08]. Available from: https://stackedit.io/

[12] Abner. *Typora - a markdown editor, markdown reader*. [quot. 2018-04-23]. Available from: https://typora.io

[13] Wikipedia. *Comparison of notetaking software - Wikipedia*. [quot. 2018-04-18]. Available from: https://en.wikipedia.org/wiki/Comparison_of_notetaking_software

[14] Corporation, E. *Evernote*. [quot. 2018-04-04]. Available from: https://evernote.com/

[15] Shankland, S. *Evernote now has 220 million users even after raising prices - CNET*. [quot. 2018-04-04]. Available from: https://www.cnet.com/news/evernote-raised-prices-got-more-of-us-to-sign-up/

[16] Gock, G. *Marxico - Markdown Editor for Evernote*. [quot. 2018-04-19]. Available from: http://marxi.co/

[17] Evernote. *Discontinued support for Evernote for BlackBerry and Windows Phone*. [quot. 2018-04-19]. Available from: https://help.evernote.com/hc/en-us/articles/212280518-Discontinued-support-for-Evernote-for-BlackBerry-and-Windows-Phone

[18] Zobec, M. *Microsoft OneNote — Digitální poznámková aplikace pro vaše zařízení*. [quot. 2018-04-19]. Available from: http://www.michalzobec.cz/microsoft-office-onenote-2013-pro-windows-zdarma-2760

[19] Zobec, M. *Microsoft Office OneNote 2013 pro Windows zdarma*. [quot. 2018-04-19]. Available from: http://www.michalzobec.cz/microsoft-office-onenote-2013-pro-windows-zdarma-2760

[20] Microsoft. *Overview of Microsoft Graph*. [quot. 2018-04-19]. Available from: https://developer.microsoft.com/en-us/graph/docs/concepts/overview

[21] Microsoft. *[MS-ONE]: OneNote File Format*. [quot. 2018-04-20]. Available from: https://msdn.microsoft.com/en-us/library/dd924743(v=office.12).aspx

[22] Shiny Frog. *Hello, I'm Bear*. [quot. 2018-04-21]. Available from: https://blog.bear-writer.com/hello-im-bear-1d0476e957d2

[23] Shiny Frog. *Polar Bear markup language*. [quot. 2018-04-21]. Available from: http://www.bear-writer.com/faq/Markup%20:%20Markdown/Polar%20Bear%20markup%20language/

[24] Dropbox Inc. *Dropbox — Company Info*. [quot. 2018-04-21]. Available from: https://www.dropbox.com/news/company-info

[25] Dropbox Inc. *Developers - Dropbox*. [quot. 2018-04-23]. Available from: https://www.dropbox.com/developers/paper-api-alpha

[26] Abner. *Draw Diagrams With Markdown*. [quot. 2018-04-23]. Available from: http://support.typora.io/Draw-Diagrams-With-Markdown/

[27] GitHub Inc. *Electron — Build cross platform desktop apps with JavaScript, HTML and CSS*. [quot. 2018-04-23]. Available from: https://electronjs.org/

[28] Foltýn, M. *Interaktivní ovládání PC hry pomocí chytrého telefonu*. Bakalářská práce, Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

[29] Heitkötter, H.; Hanschke, S.; et al. *Comparing Cross-platform Development Approaches for Mobile Platforms*. 2012, [quot. 2018-04-25]. Available from: https://www.wi1.uni-muenster.de/pi/veroeff/heitkoetter/Comparing-Cross-Platform-Development-Approaches-for-Mobile-Applications.pdf

[30] Apple Inc. *About Objective-C*. [quot. 2018-04-25]. Available from: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

[31] Gettys, J.; Scheifler, R. W.; et al. *X Window System Standard*. [quot. 2018-04-25]. Available from: https://www.x.org/releases/X11R7.6/doc/libX11/specs/libX11/libX11.html

[32] Wikipedia. *List of widget toolkits - Wikipedia*. [quot. 2018-04-25]. Available from: https://en.wikipedia.org/wiki/List_of_widget_toolkits

[33] Oracle and/or its affiliates. *Java^{TM} Native Interface*. [quot. 2018-04-25]. Available from: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/

[34] David Beazley. *Simplified Wrapper and Interface Generator*. [quot. 2018-04-25]. Available from: http://www.swig.org/

[35] Goundan, K.; Grue, T.; et al. *Djinni*. [quot. 2018-04-25]. Available from: https://github.com/dropbox/djinni

[36] Beilis, A. *CppCMS — High Performance C++ Web Framework*. [quot. 2018-04-26]. Available from: http://cppcms.com/wikipp/en/page/main

[37] Emweb bvba. *Wt, C++ Web Toolkit - Emweb*. [quot. 2018-04-26]. Available from: https://www.webtoolkit.eu/wt

[38] Buschmann, F.; Meunier, R.; et al. *Pattern-Oriented Software Architecture, A System of Patterns*. Pattern-Oriented Software Architecture, Wiley, 2013, ISBN 9781118725269.

[39] Apple Inc. *About the Cocoa Document Architecture*. 2012, [quot. 2018-04-24]. Available from: https://developer.apple.com/library/content/documentation/DataManagement/Conceptual/DocBasedAppProgrammingGuideForOSX/Introduction/Introduction.html

[40] JetBrains s.r.o. *A cross-platform IDE for C and C++*. [quot. 2018-04-26]. Available from: https://www.jetbrains.com/clion/

[41] Kohlhoff, C. M. *Asio C++ library*. [quot. 2018-04-26]. Available from: https://think-async.com/

[42] Git community. *Git Repository Layout*. [quot. 2018-04-27]. Available from: https://git-scm.com/docs/gitrepository-layout

[43] Sadler, J. *indragiek/CocoaMarkdown: Markdown parsing and rendering for iOS and OS X*. [quot. 2018-04-27]. Available from: https://github.com/indragiek/CocoaMarkdown/tree/master/CocoaMarkdown

[44] Sadler, J. *Markdown Implementations - Markdown Community Group*. [quot. 2018-04-30]. Available from: https://www.w3.org/community/markdown/wiki/MarkdownImplementations

[45] Ramirez, F. *Writing a Markdown Compiler – Part 2*. [quot. 2018-05-01]. Available from: https://blog.beezwax.net/2017/08/10/writing-a-markdown-compiler-part-2/

[46] Ford, B. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. *SIGPLAN Not.*, volume 39, no. 1, Jan. 2004: pp. 111–122, ISSN 0362-1340, doi:10.1145/982962.964011. Available from: http://doi.acm.org/10.1145/982962.964011

[47] GitHub Inc. *catchorg/Catch2: A modern, C++-native, header-only, test framework for unit-tests, TDD and BDD*. [quot. 2018-05-07]. Available from: https://github.com/catchorg/Catch2

[48] Foltýn, M. *Shere / shere-mac · GitLab*. [quot. 2018-05-07]. Available from: `https://gitlab.com/shere/shere-mac`

[49] GitHub Inc. *MIT License — Choose a License*. [quot. 2018-05-07]. Available from: `https://choosealicense.com/licenses/mit/`

Appendix A

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `bin` ..... the directory with executables
- `src` ..... the directory of source codes
- `thesis` ..... the directory of LaTeX source codes of the thesis
- `thesis.pdf` ..... the thesis text in PDF format