Supporting the Documentation of Architectural Decision Records in IDEs: A VS Code Plugin for the ADR Manager

Steven Chen
Abstract

Software architecture is largely shaped by decisions made during the software development life cycle. These architectural decisions contain information about the underlying architecture that is relevant for creating a unified view of the architecture across all stakeholders of the software project. Disregarding the documentation of architectural decisions leads to a multitude of problems during the maintenance of the software.

One way of capturing architectural decisions along with their rationale is using so-called architectural decision records (ADRs). Tools for managing ADRs already exist, but they often operate in a context that feels unfamiliar to software developers or requires them to switch to a different tool. This thesis explores the possibility of providing ADR tooling as an IDE plugin, making ADR management features conveniently available for developers.

The result of the research is the ADR Manager extension for the Visual Studio Code IDE, available on the Visual Studio Code Marketplace. This extension is based on the web-based tool with the same name and provides features for creating, editing, and managing ADRs on a local repository in a Markdown format. In addition to providing two form-based ADR editors with varying levels of detail, the extension enhances the functionality of the raw Markdown editor of the IDE by providing linting support and quick insertion of ADR templates in the form of snippets.

In an empirical user study with seven software professionals, the ADR Manager extension received resounding feedback from evaluation participants regarding its usability in the context of Visual Studio Code. The participants praised the simple yet effective extension functionality, stating that it fits nicely in the IDE environment. On the other hand, they wished that the extension was at times more concise in communicating which inputs are available to the user and how the extension expects the user to enter them such that they receive their desired result.

Still being a prototype, there are multiple ways the extension can be improved in the future. Examples include enhancing existing functionality, extending the functionality to appeal to more users, or implementing the support for more ADR formats.
# Contents

1 Introduction 15  
1.1 Motivation ........................................... 15  
1.2 Objective ........................................... 16  
1.3 Structure ............................................ 16  

2 Fundamentals 19  
2.1 Architectural Decisions .............................. 19  
2.2 Integrated Development Environments ................. 24  

3 Related Work 27  
3.1 Existing Integrated Development Environment (IDE) Plugins .......... 27  
3.2 ADR Tools Supporting MADR .......................... 28  
3.3 Comparison with the ADR Manager Extension ................. 30  

4 Methodology 33  
4.1 Requirements Phase .................................. 34  
4.2 Design Phase ........................................ 37  
4.3 Implementation Phase ................................ 40  

5 ADR Manager Extension 43  
5.1 Design ............................................... 43  
5.2 Implementation ...................................... 45  

6 Evaluation and Analysis 59  
6.1 Evaluation Phase .................................... 59  
6.2 Analysis Phase ....................................... 63  
6.3 Results ............................................... 66  

7 Discussion 73  
7.1 Usability of the ADR Manager as an IDE Plugin .................... 73  
7.2 Threats to Validity .................................. 75  

8 Conclusion 79  
8.1 Summary ............................................. 79  
8.2 Outlook ............................................. 79  

Bibliography 81  
A Appendix 85
# List of Figures

2.1 MADR 1.0.0 format proposed by Kopp et al. [KAZ18] .......................... 22

3.1 Graphical user interface of the ADR Manager, showing the professional editor mode 29

4.1 General study process for the ADR Manager extension ............................... 33
4.2 Snippet of the Google Sheets spreadsheet used for requirements validation .... 36

5.1 General architecture of the extension ....................................................... 44
5.2 Main webview in the UI Mock-up .......................................................... 52
5.3 Main webview in the VS Code extension ............................................... 52
5.4 Basic Editor webview of the ADR Manager extension ............................... 53
5.5 Professional editor in the UI mock-up .................................................. 54
5.6 Professional editor in the VS Code extension ......................................... 55

6.1 Statements grouped with the constant comparative method ....................... 68
# List of Tables

3.1 Overview of selected ADR tools .................................................. 30
4.1 Overview of the focus group participants ...................................... 34
4.2 Categorization scheme for requirements extracted from using the web-based ADR Manager .......................................................... 35
4.3 Assessment of different alternatives for the requirement “Editing existing ADRs”(shortened) ...................................................... 38
6.1 The System Usability Scale by Brooke [Bro96] ................................. 61
6.2 The Sauro-Lewis curved grading scale [Lew18] ............................... 64
6.3 Overview of the evaluation participants ........................................ 66
6.4 Task completion times of each participant ...................................... 67
6.5 Number of statement mentions for each label ................................ 68
6.6 Statements mentioned more than once across all participants .......... 70
6.7 SUS scores of all participants ........................................................ 71
A.1 List of validated requirements for the extension ............................ 86
# List of Listings

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Example of using CSS variables for automatically picking VS Code theme colors</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Source code for creating a basic ADR from specified fields</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Snippet of the package.json file of the ADR Manager</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Content of the file web/pages/main.ts, creating a MainView component</td>
<td>51</td>
</tr>
<tr>
<td>5.5</td>
<td>Source code for specifying the content of a webview (shortened)</td>
<td>51</td>
</tr>
</tbody>
</table>
Acronyms

AD  Architectural Decision. 15
ADR  Architectural Decision Record. 5, 15
API  Application Programming Interface. 25
GUI  graphical user interface. 25
IDE  Integrated Development Environment. 5
MADR  Markdown Architectural Decision Record. 5, 20, 21
SUS  System Usability Scale. 60
UI   user interface. 26
VS Code  Visual Studio Code. 16
1 Introduction

This chapter establishes the motivation behind creating a plugin for the ADR Manager and illustrates the objective of this thesis. At the end of this chapter, the general structure of this thesis is outlined.

1.1 Motivation

When designing a software system, several choices are to be made that have a significant impact on the underlying software architecture, for example, the selection of a programming language or the application of a software design pattern for a specific component of the architecture. Such choices are also known as Architectural Decisions (ADs), and their underlying information, as well as their implications on the architecture, are often implicitly embedded in the architecture itself due to the lack of documentation [TA05]. This results in stakeholders having multiple different understandings of the software architecture, thereby running into difficulties when working with the architecture throughout its life cycle, such as costly maintenance, excessive complexity, and erosion of the software architecture [TA05][JB05].

To circumvent these issues, ADs need to be captured by additional artifacts that describe and justify their rationale [FCK08]. One approach to accomplishing this task is utilizing the concept of Architectural Decision Records (ADRs) introduced by Michael Nygard [Nyg11]. According to Nygard, an ADR is “a short text file” that describes a single AD in brevity, making these documents “easier for all stakeholders to consume” and bringing value by “being kept up to date” [Nyg11].

While there are tools available¹ that help in the creation and management of ADRs, they are often used in a way that may hinder the workflow of a software developer. For instance, some ADR tools use a web browser to provide their features. A developer must switch between the IDE and web browser if they insist on using said tools: they are interrupting themselves from their primary task by switching between two disjoint environments, which may hurt the task completion time or increase the number of errors the developer produces [BK06].

By integrating ADR tooling directly into an IDE, the efficiency and convenience of managing ADRs can be further improved, thereby centralizing information about the software architecture in a single familiar environment.

¹ADR tools listed by the GitHub ADR Organization, https://adr.github.io/#tooling
1.2 Objective

This thesis focuses on the development and evaluation of an IDE plugin (in the following called “plugin”) that provides features based on an existing ADR tool, enhancing the functionality of the IDE with support for documenting and managing ADRs. Users should benefit from a more intuitive feel and a more shallow learning curve compared to ADR tools running outside of an IDE without compromising on the adequacy of ADR management features. For this, both the IDE and the ADR tool in question were chosen in advance:

Choice of ADR Tool  The plugin will be based on the ADR Manager, taking its core functionality, workflow, and feel and attempting to adapt it to fit the context of an IDE. Users experienced with the ADR Manager should be able to recognize the connection to the original, web-based version and adapt to the plugin without much difficulty. In light of this, the plugin will also go by the name “ADR Manager”.

In a previous study of the web-based ADR Manager conducted by the Institute of Software Engineering at the University of Stuttgart, several takeaways for ADR tools in general and the shortcomings of the web application and its features are presented. The plugin aims to address some of the ideas of the study, although it should be examined whether the results of the study are even applicable in the context of an IDE.

Choice of IDE  The IDE of choice is Visual Studio Code (VS Code), not only because of its popularity – reflected in the number of Google searches compared to other IDEs – but also due to its extensibility and livid extension ecosystem hosted on the Visual Studio Code Marketplace. Additionally, multiple participants from the previous ADR Manager study requested a plugin for the ADR Manager specifically for VS Code.

Once developed, the ADR Manager plugin (from now referred to as “the ADR Manager extension” or just “the extension”; to stay true to the VS Code naming convention) will be evaluated in terms of its perceived usability.

1.3 Structure

Excluding this first introduction chapter, this thesis is divided into seven chapters:

Chapter 2 lays the foundation and defines the fundamental concepts this thesis builds upon.

Chapter 3 examines work that is related to this thesis.

---

2 ADR Manager, https://adr.github.io/adr-manager
3 Evaluation results of the original ADR Manager, https://github.com/adr/adr-manager/tree/main/docs/evaluation/user-study/results
5 PYPL Popularity of Programming Language Top IDE Index, https://pypl.github.io/IDE.html
7 As of writing, the Top 60 VS Code extensions have over five million installs each
Chapter 4 describes the design and general approach followed for the development of the extension.
Chapter 5 presents the result of the development process, the ADR Manager extension.
Chapter 6 illustrates the preparation, execution, and results of the evaluation study performed with the extension.
Chapter 7 discusses the usability of the ADR Manager as an IDE plugin and critically looks at the limitations of this thesis.
Chapter 8 summarizes this thesis and gives pointers to future follow-up work.
2 Fundamentals

This chapter defines the terminology and background knowledge required for this thesis and intends to narrow the scope of this thesis even further.

2.1 Architectural Decisions

Architectural Decisions (ADs) are design decisions regarding a software architecture that determine its fundamental aspects, such as the structure or the interaction between its components [Zim11]. ADs also include the reason why a solution has been chosen over other alternatives, as well as the consequences of the decision on the system [Zim11]. They are typically difficult to make [Fow03], expensive to change [Vli08], and address so-called *architecturally significant requirements*. These are requirements (functional or nonfunctional) whose effects on the software architecture can be identified and measured [CAN13].

There are many templates, models, and tools proposed to document ADs [vAH12]. The following sections will focus on Architectural Decision Records as a technique to capture ADs.

2.1.1 Architectural Decision Records

This section of the thesis is based on a blog post dating back to 2011 by Michael Nygard where he coins the term Architectural Decision Record (ADR) as a tool to systematically document ADs [Nyg11].

Characteristics of an ADR

In his blog, Nygard defines an ADR as a short, formatted text file with the following characteristics:

1. An ADR describes “a set of forces and a single decision in response to those forces”.
2. ADRs are “numbered sequentially and monotonically”, where each ADR has its unique number.
3. ADRs are written in a lightweight markup language like Markdown\(^1\) or Textile\(^2\).
4. ADRs are project-specific and are kept in the project repository under the subdirectory `doc/arch/adr-NNN.md` (where N stands for a number between 0 and 9).

---

\(^1\)[Markdown, https://daringfireball.net/projects/markdown](https://daringfireball.net/projects/markdown)

5. Once created, an ADR must not be deleted and is visible to every present and future stakeholder of a project. If an ADR is reversed, it will be marked as “superseded”.

Nygard justifies the last characteristic by stating that new ADRs often build upon the consequences of preceding ADRs and that stakeholders need to be able to track and comprehend decision changes and the rationale behind them. As a result, the evolution of the underlying software architecture becomes more transparent and explicit.

Sections of an ADR

In addition to the characteristics mentioned above, Nygard also proposes a format for documenting ADRs that consists of the following five sections:

- **Title** – This section defines the name of the ADRs. A name should be “a short noun phrase”, for example, “ADR 1: Deployment on Ruby on Rails 3.0.10” or “ADR 5: Persistence on a MySQL Database”.

- **Context** – This section objectively specifies the relevant technological, political, social, and project local forces of the ADR in a declarative manner.

- **Decision** – This section describes the response to the forces in the previous section of the ADR. The decision is written in whole sentences and with an active voice, i.e., “We will ...”.

- **Status** – This section depicts whether the ADR is currently “proposed” (stakeholders have not yet come to an agreement), “accepted” (stakeholders have agreed upon this ADR), or “deprecated” / “superseded” (ADR has been modified or reverted by a newer ADR). In the latter case, a reference to its replacement will be included.

- **Consequences** – This section describes the resulting context after the decision has been applied. Positive as well as negative consequences should be listed.

Nygard also mentions additional rules for ADRs, like writing whole sentences and paragraphs (“as if it is a conversation with a future developer”) and suggests that each document should span between one and two pages. The format is intentionally kept short so that the artifacts can be digested and kept up to date more easily.

Experience with ADRs

At the end of his blog, Nygard writes about his practical experience of using ADRs in his projects. He and his team perceive ADRs as “especially useful for capturing long-term intentions” and “appreciate the degree of context they received by reading them”.

There are other lightweight templates for decision capturing that have been derived from the concept of ADRs, namely *Y-statements* [ZCTZ13] and *Markdown Architectural Decision Records (MADRs)* [KAZ18]. The latter template will be the main focus of not only the following section but for the rest of this thesis.
2.1 Architectural Decisions

2.1.2 Markdown Architectural Decision Records

Markdown Architectural Decision Record (MADR) is a lean format proposed in a paper by Kopp et al. for documenting ADs [KAZ18]. It intends to treat ADs similarly to other developer tools and artifacts by 1) strictly choosing Markdown as the markup language to capture ADs, and 2) saving MADRs in the version-controlled folder `docs/adr` of code repositories, whereby each MADR file is named after the format `NNNN-lowercased-with-dashes.md`, with N being a number between 0 and 9 and the rest of the file name written in lower kebab-case.

In [KAZ18], Kopp et al. state the following reasons for making the above decisions:

1. Markdown is chosen as the markup language of choice because it “eases writing”, it is used as “the markup language for comments by users within GitHub” and it allows leveraging “already available rendering tools”.

2. The folder `docs/adr` was chosen for enabling “rendering in GitHub pages” out of the `docs` folder using the Jekyll site generator, making MADRs “accessible on the World-Wide Web”.

Similar to Nygard’s ADRs, Kopp et al. also propose a structured, section-based format for MADRs. This format has changed throughout the years since its first release in 2017.

The following subsections aim to illustrate the general MADR format and highlight the key changes between each major version of MADR. All updates to the MADR format are published to the MADR repository and can be tracked in the “CHANGELOG.md” file in that repository.

MADR 1.0.0

Figure 2.1 shows the MADR format originally proposed in [KAZ18]. It consists of mandatory as well as optional sections and fields, where each major section is initiated by Markdown headings and subheadings (denoted by “#” and “##” respectively). Lists use “*” as the marker for list items and placeholders in this template are signaled by the double pair of enclosing characters “*[...]*”.

MADR 1.0.0 defines the general MADR template that has been updated throughout the years.

The following parts are mandatory for a MADR in the version 1.0.0:

- **Title (line 1)** – The title of a MADR is placed at the top of the Markdown file and is the only heading in the whole document.

- **Context and problem statement (line 5)** – This field describes the context the MADR plays in, similar to the “Context” section in Nygard’s ADR template.

---

3 Every word is written in lowercase and words are separated by dashes
4 GitHub, https://github.com
5 GitHub Pages, https://pages.github.com
6 Setting up a GitHub Pages site with Jekyll, https://docs.github.com/en/pages/setting-up-a-github-pages-site-with-jekyll
7 MADR 1.0.0 was already released on GitHub in 2017, https://github.com/adr/madr/releases/tag/1.0.0
2 Fundamentals

1 # *[short title of solved problem and solution]*
2 3 **User Story:** *[ticket/issue-number]* <--> optional -->
4 5 *[context and problem statement]*
6 *[decision drivers | forces]* <--> optional -->
7 8 ** Considered Alternatives
9 10 * *[alternative 1]*
11 * *[...]* <--> numbers of alternatives can vary -->
12 13 ** Decision Outcome
14 15 * Chosen Alternative: *[alternative 1]*
16 * *[justification. e.g., only alternative, which meets k.o. criterion decision]*
17 * *[driver | which resolves force force | ... | comes out best (see below)]*
18 * *[consequences. e.g., negative impact on quality attribute, follow-up decisions required, ...]* <--> optional -->
19 20 ** Pros and Cons of the Alternatives <--> optional -->
21 22 ** [alternative 1]*
23 24 * `+` *[argument 1 pro]*
25 * `-` *[argument 1 con]*
26 * *[...] <--> numbers of pros and cons can vary -->

Figure 2.1: MADR 1.0.0 format proposed by Kopp et al. [KAZ18]

- **Considered Alternatives (lines 8 - 11)** – This section is denoted by a subheading (line 8) and lists every alternative that is being evaluated for this decision as list items. This includes the chosen alternative as well as the neglected alternatives for this decision.

- **Decision Outcome (lines 13 - 19)** – This subheading section states the chosen alternative (prefaced by “Chosen Alternative: ”) and the reasoning behind choosing that alternative over the others.

Optional parts of MADR 1.0.0 include:

- **User Story (line 3)** – This field is placed under the title heading and further elaborates on the context and problem statement of the MADR, for example by stating a ticket or issue number that this MADR is referring to.

- **Decision drivers / Forces (line 6)** – This field can be used to explicitly address criteria that need to be considered when choosing an alternative, like the goals pursued with this decision or concerns that need to be paid attention to.

- **Consequences (lines 18 - 19)** – This field is placed inside of the “Decision Outcome” section, after the justification of the chosen alternative. It describes the positive, but also the negative consequences that come with the decision to go with the chosen alternative.
• **Pros and Cons of the Alternatives (lines 20 - 26)** – This subheading section lists the positive and negative arguments of each alternative in the “Considered Alternatives” section. For each alternative, a sub-subheading section is created (denoted by “###”), under which the positive and negative arguments are depicted as list items. To mark the disposition of each argument, all arguments start with either “+/−” or “−/−”.

### MADR 2.0.0

The second major version of the MADR format, MADR 2.0.0, was published on GitHub in 2018. Between MADR 1.0.0 and MADR 2.0.0, the format has gone through several additions and changes regarding the structure and the terminology used in the template. The following outlines some noteworthy changes:

**Additions**  
MADR 2.0.0 introduced three optional fields “Status”, “Deciders” and “Date”. They are placed under the title heading to determine the current state of the MADR (like the “Status” section in Nygard’s ADR), the people involved in this decision, and the date on which the MADR was last updated. Additionally, a new optional subheading called “Links” has been added at the end of the document where links to other information like other MADRs can be listed as list items.

**Changes**  
MADR 1.2.0 only included a minor change in the template where the placeholders were changed to use only one enclosing pair of square brackets (“[..]”) instead of the original pairs of asterisks and square brackets.

MADR 1.3.0 brought many changes to the template syntax: positive and negative consequences are now their own bullet lists in the “Decision Outcome” section, the “+/−” that were preceding positive and negative arguments in the “Pros and Cons of the Alternatives” section have been replaced by “Good, because ...” and “Bad, because ...” respectively, and “alternatives” have been renamed to “options”, resulting in the modified subheadings “Considered Options” and “Pros and Cons of the Options”. The motivation behind the latter change was to bring the MADR template closer to the format of Y-statements [ZCTZ13].

In MADR 2.0.0, the previous fields “Context and problem statement” and “Decision drivers” have become separate subheadings (and thus are now written in title case), the field “User Story” has been changed to “Technical Story”, and the chosen option is now surrounded by double quotes to highlight the name of the option in the “Decision Outcome” section.

---

9 In Markdown, text enclosed in a pair of backticks (“`...`”, also known as grave accents) marks code and is usually rendered in a block that stands out from other Markdown content.
MADR 2.1.2

Although there is only one noteworthy change between MADR versions 2.0.0 and 2.1.2 (“Positive consequences” and “Negative consequences” changed to being separate sub-subheadings as of MADR 2.1.0), MADR 2.1.2 deserves a special mention here because the web-based ADR Manager supports MADRs in the version 2.1.2, as it was the newest version of the MADR format at the time ADR Manager was released\(^\text{10}\).

MADR 3.0.0-beta

In 2022, a beta version of the next major release has been published on the MADR repository. The MADR 3.0.0-beta release is the first update of the MADR format since the release of MADR 2.1.2 and includes relatively many changes to the MADR format\(^\text{11}\).

The most notable change to MADR, however, is the shift of the meaning behind the acronym “MADR”: MADR now stands for “Markdown Any Decision Record” to reflect the idea that the concept of ADRs can be used for documenting not only ADs, but all types of decisions in general, such as organizational decisions, managerial decisions or even everyday decisions. This change of notion is further expanded on in a blog post by Zimmermann \cite{Zim21}.

At this point, this thesis will not go any further into detail with MADR 3.0.0-beta, as it is a conscious decision for the ADR Manager extension to be based on MADR 2.1.2, just like the web-based tool. Thus, from the next chapter onward, this thesis will “only” focus on the MADR format and its changes up until MADR 2.1.2. Additionally, from now, any mention of ADR will refer to the MADR format specifically.

2.2 Integrated Development Environments

An Integrated Development Environment (IDE) is a software application that comes with a set of tightly integrated features to boost the productivity of developers throughout the entire software development process, for example by automating common tasks or providing quality-of-life features \cite{Zei17}. IDEs can support multiple programming languages or specialize in a single programming language and its paradigms.

Modern IDEs commonly include a source code editor with features like syntax highlighting, code completion and automated refactoring, a debugger, build automation tools, and a way to run programs directly inside of the IDE. Some of today’s most popular IDEs are VS Code, Eclipse\(^\text{12}\), and IntelliJ IDEA\(^\text{13}\). IDEs also often come with integrated version control clients, enabling developers to work with remote repositories using version control systems like Git\(^\text{14}\) or Subversion\(^\text{15}\).

\(^{10}\)ADR Manager was released in late 2021, \url{https://github.com/adr/adr-manager/releases}

\(^{11}\)Changelog for MADR 3.0.0-beta, \url{https://github.com/adr/madr/blob/main/CHANGELOG.md#300-beta--2022-05-17}

\(^{12}\)Eclipse IDE, \url{https://www.eclipse.org/ide}

\(^{13}\)IntelliJ IDEA, \url{https://www.jetbrains.com/idea}

\(^{14}\)Git, \url{https://git-scm.com}

\(^{15}\)Apache Subversion, \url{https://subversion.apache.org}
A typical workflow for developers is to initialize a repository on the local computer that is based on a remote repository. Developers will then make changes to their local repositories by leveraging the IDEs features and push these changes onto the remote repository once they wish to do so. This allows developers to simultaneously work on the same project by working on local copies of the remote repository without worrying about breaking existing software.

### 2.2.1 Plugins

Many IDEs provide additional tools to help developers in writing their own software that extends the core features of the IDE. Such pieces of software are also known as plugins, add-ons, or extensions.

Plugin developers use an Application Programming Interface (API) provided by the IDE developers to interact with the IDE itself. This allows them to build upon native features of the IDE, for example by adding new commands or elements to the graphical user interface (GUI) that execute custom actions within the context of the IDE.

To distribute plugins, IDE developers often host platforms where plugins can be uploaded to. These platforms are known as marketplaces and users oftentimes can conveniently search for plugins and install them directly in the IDE.[16]

### 2.2.2 Visual Studio Code

Visual Studio Code (VS Code) is a free IDE developed by Microsoft[17]. It is written in TypeScript[18], a superset of JavaScript with static typing, and utilizes the Electron[19] framework: a framework for writing cross-platform desktop applications running on Windows, macOS, and Linux using HTML, JavaScript, and CSS. VS Code natively supports both JavaScript and TypeScript, and plugins (called extensions in VS Code) provide additional support for other languages such as Java, C++, Python and more. VS Code extensions are published and hosted on the Visual Studio Code Marketplace which can be accessed from a web browser or directly in VS Code.

**Workspaces**

One important concept of VS Code is the so-called workspace. According to the VS Code documentation[20], a workspace is a “collection of one or more folders that are opened in a VS Code window”. This allows the user to configure settings for each folder individually, which also applies to extensions that the user has installed: extension developers can provide features or change the behavior of the extension based on the current state of the user workspace.

---

[18]TypeScript, [https://www.typescriptlang.org](https://www.typescriptlang.org)
2 Fundamentals

Extension API

Like the IDE itself, extensions for VS Code are also written in TypeScript by using the Extension API\(^2\), a set of APIs which enables extension authors to enhance almost every aspect of VS Code.

The Extension API includes the native VS Code API\(^2\), allowing developers to use core features of VS Code from within the extension and customize them to the needs of the extension.

Webview API

The Webview API\(^3\) is another part of the Extension API that allows extensions to render custom HTML, CSS, and JavaScript content inside of panels known as webviews. This makes the Webview API a powerful and versatile tool to create complex user interfaces (UIs) that are not realizable using the native VS Code API, albeit having a higher resource cost. Because webviews essentially generate whole websites inside of VS Code, it is also feasible to use front-end frameworks like Vue\(^4\) or React\(^5\) to create the interface for the webview.

The communication between a webview and the extension which created the webview is realized via message passing, where each party constantly listens to incoming messages from the other party. The sender attaches data to a message and sends it to the receiver that will do specific actions based on the content of the received message. This especially means that a user can trigger the extension to use parts of the VS Code API when interacting with a webview, like executing commands or modifying files and folders in the workspace.

\(^2\)Extension API, https://code.visualstudio.com/api
\(^3\)Webview API, https://code.visualstudio.com/api/extension-guides/webview
\(^4\)Vue, https://vuejs.org
\(^5\)React, https://reactjs.org
3 Related Work

There already exist multiple tools that help the user with documenting and managing ADs which have been compared and analyzed, for example by Capilla et al. [CJT+16] or Liang and Avgeriou [LA09]. This chapter briefly presents some existing tools for capturing architectural decisions. More specifically, it highlights some existing IDE plugins, but also looks at a selection of tools that explicitly support the MADR format. The last section of this chapter finally discusses how the ADR Manager extension differentiates itself from all the tools mentioned in this chapter.

3.1 Existing IDE Plugins

3.1.1 Archie

Archie\(^1\) is a plugin for the Eclipse IDE created by Mirakhorli et al. that allows the user to create and manage links between ADs and the source code of a project [MFG+14]. With this, Archie aims to raise the awareness of software developers regarding design decisions and the impact of code implementations on the underlying ADs. Some features of Archie include a “detection engine” for identifying code sections that implement an AD, a highlighter for automatically highlighting and annotating code that has been classified by the plugin as architecturally significant, and the ability to manually mark parts of the code as architecturally significant. Archie also notifies the user when a critical part of the code is being modified and provides a feature to visualize connections between decisions and code in a graph.

3.1.2 DecDoc

DecDoc\(^2\) is another plugin for the Eclipse IDE [HKR16]. It was developed by Hesse et al. and focuses on the collaborative documentation of ADs, describing the Problem, Solution, Context, and Rationale for each AD. Decisions are modeled with the model-based knowledge management tool UNICASE\(^3\) and are persisted using the versioned model repository EMFStore\(^4\).

\(^1\)GitHub repository of Archie, https://github.com/ArchieProject/Archie-Smart-IDE

\(^2\)According to [HKR16], DecDoc should be available at http://svn.ifi.uni-heidelberg.de/unicase/0.5.2/ures/decdoc-features, but it is not available as of August 31, 2022.

\(^3\)UNICASE on the Eclipse Marketplace, https://marketplace.eclipse.org/content/unicase

\(^4\)EMFStore, https://www.eclipse.org/emfstore
3 Related Work

This allows for an incremental decision-making process where decisions are fleshed out by multiple stakeholders throughout the entire software development process, with each stakeholder contributing domain-specific knowledge to the decision model [HKR16]. DecDoc also features code annotations and integrates with the Eclipse Papyrus\(^5\) modeling tool to provide a way to associate decisions with UML entities [HKR16].

3.2 ADR Tools Supporting MADR

3.2.1 Log4brains

Log4brains\(^6\) is a command line interface for creating new ADRs in the MADR 2.1.2 format and rendering existing ADRs on a static website.

When creating a new ADR, Log4brains asks the user for a title and if the new ADR will supersede an existing ADR in the project. Then, it creates a new Markdown file in the ADR directory (specified in a configuration file) with the specified title and a template with the rest of the essential MADR fields along with placeholders. If the user wants to edit an ADR, they will have to use a raw Markdown editor, as Log4brains does not provide a feature for editing ADRs.

Log4brains can also generate a log of all ADRs and display them chronologically in a timeline, starting from the newest ADR.

Unlike MADR, Log4brains uses another format for naming the Markdown files of newly created ADRs: instead of using the NNNN-lowercased-with-dashes.md format, it uses YYYYMMDD-lowercased-with-dashes.md as the naming format, with YYYYMMDD being the date on which the Markdown file has been created. This change has been documented with an ADR\(^7\).

3.2.2 eADR

Embedded Architectural Decision Records (or eADR\(^8\)) adds annotations to Java which allow the user to create ADRs directly in Java code and can be used in projects via the build tools Gradle\(^9\) or Maven\(^10\). It supports Y-statements (using the @YStatementJustification annotation) and MADR 1.0.0 (using the @MADR annotation). The content of the decision is specified directly in the annotation itself. The user can also point to ADRs that exist in the ADR directory of the project by using the @ADR annotation.

\(^5\)Eclipse Papyrus, https://www.eclipse.org/papyrus
\(^6\)Log4brains, https://github.com/thomvaill/log4brains
\(^7\)ADR about changing the naming format for ADRs, https://github.com/thomvaill/log4brains/blob/master/docs/adr/20201016-use-the-adr-slug-as-its-unique-id.md
\(^8\)eADR, https://github.com/adr/e-adr
\(^9\)Gradle, https://gradle.org
\(^10\)Maven, https://maven.apache.org
3.2.3 ADR Manager

The ADR Manager is a client-side web application created at the Institute of Software Engineering at the University of Stuttgart for managing ADs in the MADR format. By using the authentication protocol OAuth 2.0\(^\text{11}\), it can connect to GitHub repositories to create new ADRs, edit and delete existing ADRs as well as push these changes to GitHub directly from the web application using the GitHub API.

Figure 3.1: Graphical user interface of the ADR Manager, showing the professional editor mode

ADR Manager provides two form-based editor modes: the Basic Editor Mode for quickly documenting ADRs only includes the essential fields of an ADR (i.e., Title, Context and Problem Statement, Considered Options, and Decision Outcome), while the Professional Editor Mode displays all fields of MADR 2.1.2 for refining and extending ADRs (as shown in Figure 3.1). Additionally, it provides a raw Markdown editor to manually edit an ADR.

For detecting existing ADRs in a repository, ADR Manager looks for Markdown files in the subdirectories that meet one of the following criteria\(^\text{12}\):

---

\(^{11}\) OAuth 2.0, [https://oauth.net/2](https://oauth.net/2)

Table 3.1: Overview of selected ADR tools

<table>
<thead>
<tr>
<th>ADR Tool</th>
<th>Environment</th>
<th>Key Feature</th>
<th>Supports MADR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Archie</td>
<td>IDE (Eclipse)</td>
<td>Management of relations between decisions and code</td>
<td>No</td>
</tr>
<tr>
<td>DecDoc</td>
<td>IDE (Eclipse)</td>
<td>Management of ADRs</td>
<td>No</td>
</tr>
<tr>
<td>Log4brains</td>
<td>Command line, web browser</td>
<td>Rendering of ADR logs</td>
<td>Yes</td>
</tr>
<tr>
<td>eADR</td>
<td>Any</td>
<td>Annotation of source code (Java exclusive)</td>
<td>Yes</td>
</tr>
<tr>
<td>ADR Manager</td>
<td>Web browser</td>
<td>Management of ADRs</td>
<td>Yes</td>
</tr>
<tr>
<td>ADR Manager Extension</td>
<td>IDE (VS Code)</td>
<td>Management of ADRs</td>
<td>Yes</td>
</tr>
</tbody>
</table>

1. The subdirectory includes one of the following strings: “/docs/adr/”, “/docs/adrs/”, “/docs/AoDR/”, “/doc/adr/”, “/docs/decisions/”, “/docs/design/”, “/technical-overview/adr/”
2. The subdirectory starts with “adr/”

For each Markdown file, it then tries to parse the content of the file using a JavaScript parser generated with ANTLR4. If the file content cannot be parsed without errors, the ADR Manager will display the option to change the raw Markdown such that the ADR can be parsed and the editor modes can be used.

3.3 Comparison with the ADR Manager Extension

Table 3.1 shows an overview of the tools that were highlighted in this chapter along with their environment, key feature, and support for the MADR format.

Among the tools that are realized as IDE plugins, the ADR Manager extension is the only tool that supports the MADR format. Furthermore, when compared to DecDoc for example, the ADR Manager extension does not rely on external plugins, making the ADR Manager extension more maintainable. For instance, the model repository EMFStore used by DecDoc will be declared end-of-life by December 31st, 2022, forcing DecDoc to migrate to another solution, which comes with considerable effort and risk for introducing bugs.

The web-based ADR Manager and its VS Code extension are also the only tools in this selection that have “Management of ADRs” listed as their key feature and support the MADR format as well. Although the web-based ADR Manager and the VS Code extension are strongly related, the different contexts of these tools will inevitably lead to noticeable differences between these two tools.

13ANTLR4, https://github.com/antlr/antlr4
To summarize, the ADR Manager extension for VS Code sets itself apart from the other tools in this chapter by aiding users in capturing ADs in the MADR format directly in the IDE. On top of that, the extension can leverage the popularity of VS Code among developers, reflected in VS Code being ranked as the number one IDE in the Stack Overflow Annual Developer Survey\textsuperscript{14} since 2018 by a considerable margin.

\textsuperscript{14}Stack Overflow Annual Developer Survey, \url{https://insights.stackoverflow.com/survey}
4 Methodology

This chapter illustrates the general approach that was followed to develop the extension and gives detailed insights into the different activities during the various development phases.

Figure 4.1: General study process for the ADR Manager extension

Figure 4.1 visualizes the study process in its entirety. It is divided into five sequential phases, each with its own activities:

1. Requirements Phase (indicated by the blue activities)
2. Design Phase (indicated by the green activities)
3. Implementation Phase (indicated by the yellow activity)
4. Evaluation Phase (indicated by the purple activity)
5. Analysis Phase (indicated by the orange activity)

To keep track of the progress during development, a definition of done was established for each phase in the form of a goal or milestone, oftentimes represented as an output item of a phase. The results of each phase affected all subsequent phases, such that the milestone of a phase had to be reached before continuing with the next phase.

Focus group

For the first three phases of the study process, a focus group was established as a method for collecting early feedback. According to Krueger and Casey, a focus group is a set of participants who discuss a topic that has been chosen by the moderator [KC15]. The discussion takes place “in a permissive and nonthreatening conversation” and does not aim for consensus, but rather “to gain participant insights on the topic of discussion” [KC15].
4 Methodology

<table>
<thead>
<tr>
<th>Role</th>
<th>Years of Experience</th>
<th>Familiar with MADR</th>
<th>Familiar with original ADR Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Researcher (Industry)</td>
<td>15 years</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Cloud Architect</td>
<td>10 years</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Professor for software quality and architecture</td>
<td>12 years</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4.1: Overview of the focus group participants

The focus group consisting of three people gave feedback and their opinion on the current results at various points during the activities of the three phases. The members of the focus group are experienced in working with ADRs and have extensive knowledge about using VS Code. They were recruited using a short document listing the requirements and tasks that needed to be done as part of the focus group. Table 4.1 shows an overview of the three participants of the focus group.

The following sections of this chapter will go into detail about the activities in the first three phases of the study process as these phases revolve around the initial development of the extension. Chapter 6 will take a deeper look at the last two phases of the study process: the evaluation and the analysis of the evaluation results.

4.1 Requirements Phase

The Requirements Phase was the first phase of the study and focused on the elicitation and validation of requirements for the extension. The milestone for this phase was a list of requirements that the extension aimed to satisfy. These requirements were also validated by the members of the focus group in terms of their necessity and implementation details.

4.1.1 Eliciting Requirements

As the web-based ADR Manager is the basis of the extension, the core functionality of the extension should not drift too far from the original. The first set of requirements was thus extracted from the web-based tool itself, deriving requirements just from using the ADR Manager. For instance, the requirement “Automatically detect ADRs in repositories” was derived from the fact that the ADR Manager automatically finds and displays ADRs that already exist in a repository. Requirements that were elicited in such a way will be referred to as extracted requirements.

---

4.1 Requirements Phase

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Description</th>
<th>Type</th>
<th>Verdict</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>Provide MADR template</td>
<td>The user can add and edit ADRs (…)</td>
<td>functional</td>
<td>keep as is</td>
<td>This is a core functionality (…)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>9</td>
<td>Add GitHub repositories</td>
<td>The user can add GitHub (…)</td>
<td>functional</td>
<td>drop</td>
<td>The user can use the IDE to (…)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4.2: Categorization scheme for requirements extracted from using the web-based ADR Manager

Another set of requirements was collected from the results of the previous evaluation study with the ADR Manager\(^3\), where the participants voiced requests for new features that were not implemented in the tool. Some additional requirements that could not be derived from tool usage or the original study were found by using ideation methods like brainstorming or reverse brainstorming and were also added to the list of requirements.

Because the web-based ADR Manager and the extension function in different environments, there were further considerations to be made for extracted requirements: some of them could be adopted for the extension without any changes, while others needed to be changed to fit in the extension context. Some extracted requirements were even omitted entirely because they became obsolete in the IDE environment. For this, all requirements were structured using a categorization scheme, as shown in Table 4.2.

Aside from having a title, each requirement was given a unique ID and a description explaining what the user should be able to do if the requirement was satisfied by the extension, and also giving general technical implementation details. Next, a type was added that labeled each requirement as a “functional” or a “non-functional” requirement. On top of that, each extracted requirement had a verdict, signaling how the requirement will be adapted to the extension, with possible values being “keep as is”, “modify but keep” and “drop”. Accompanying the verdict, a reason was added to each extracted requirement which briefly explains the given verdict, hence making the verdict more transparent.

Requirements that were not implemented in the web tool (i.e., requirements that were not extracted directly from the ADR Manager) were also categorized with this scheme, with the verdict being left out and the reason clarifying why a certain requirement should be considered in the VS Code extension.

4.1.2 Validating Requirements

To assess the quality of the elicited requirements, the members of the focus group (the reviewers) were asked to review and validate the requirements iteratively. An iteration of requirements validation consisted of the following steps:
1. The current, most up-to-date list of requirements is shared with the reviewers
2. Each requirement is examined by each reviewer to see if the given description is reasonable for the extension and if the given verdict and reason are justified
3. For every requirement, each reviewer decides on whether the requirement will be accepted or should be modified in any way. The reviewer can add a comment to each requirement to either explain their decision or give additional remarks on the requirement
4. Each reviewer may propose new requirements that are not present in the list of elicited requirements
5. The study conductor tries to consolidate the results of each reviewer such that the majority of stakeholders (including the study conductor himself) are content
6. The study conductor updates the list of requirements according to the consolidations from the previous step.
7. If there are any new requirement proposals, the study conductor will decide on whether to accept a new proposal, propose changes to a newly proposed requirement or decline the requirement entirely. The decisions are also accompanied by comments from the study conductor.

<table>
<thead>
<tr>
<th>Title</th>
<th>Description</th>
<th>Type</th>
<th>New?</th>
<th>Verdict</th>
<th>Reason</th>
<th>Reviewer Decision</th>
<th>Comment on Reviewer Decision</th>
<th>Notes from the Study Conductors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.2:** Snippet of the Google Sheets spreadsheet used for requirements validation

The validation took place asynchronously using Google Sheets\(^2\), through which each reviewer was given access to their own spreadsheet by the study conductor. Each spreadsheet contained a sheet with the list of requirements as well as a sheet for proposing new requirements. Every iteration of requirements validation was initiated through email and a deadline was set for each iteration. The reasoning for performing the validation in this manner was 1) to give the reviewers the most flexibility in choosing when to do the validation, 2) to let the reviewers express their thoughts in their own words, and 3) to prevent the reviewers from influencing each other.

In total, the requirements validation for the extension took two iterations to complete, after which the stakeholders came to a general consensus, resulting in 12 validated requirements. Table A.1 shows the full list of validated requirements along with their ID, title, description, and their type.

With this, the milestone for the Requirements Phase had been reached, meaning that it was time to move on to the next phase in the development process.

\(^2\)Google Sheets, [https://www.google.com/intl/de_de/sheets/about/](https://www.google.com/intl/de_de/sheets/about/)
4.2 Design Phase

The Design Phase was the second phase in the development process and tackled the challenge of transforming the validated requirements into a possible implementation within the context of a VS Code extension. For this phase, the milestone was to have an interactive prototype that can be used as a reference for the future implementation of the extension.

4.2.1 Designing the Prototype

Initially, the process of designing the prototype was split into three tasks that had to be solved:

1. Assess different alternatives to realize each validated requirement with the Extension API
2. Create a simple sketch in the form of a paper prototype or other comparable method, according to the “best” implementation alternative from the previous task
3. Implement a mock-up in VS Code based on the sketch from the previous task

Assessing Alternatives to Realize Requirements

Coming up with different ways for realizing a requirement was a creative process that was mostly limited by the capabilities of the VS Code Extension API. Knowledge about the Extension API was important to gauge the feasibility of an implementation alternative.

Building on that, there may be some alternatives that can be implemented more easily or that fulfill the same requirement better than other alternatives. As such, the easier an alternative could be implemented (while still adequately fulfilling a requirement), the more suited that alternative is for realizing the underlying requirement. Besides the complexity and the degree of fulfillment, the aspect of usability also plays a non-negligible role when comparing different alternatives for realizing the same requirement: the user should enjoy using the extension while also achieving their goals.

Thus, alternatives that satisfy the same requirement were grouped together and analyzed based on their suitability, where an alternative could be well suited, moderately suited, or not suited for satisfying a particular alternative.

Similar to the process of requirements validation, alternatives were also given an ID that is unique for each requirement, a short description illustrating how the user can use the alternative to fulfill the requirement, and a reason explaining how the suitability classification of an alternative came to be. As an example, the assessment of alternatives for the validated requirement “Editing existing ADRs” can be seen in Table 4.3.
<table>
<thead>
<tr>
<th>ID</th>
<th>Alternative</th>
<th>Description</th>
<th>Suitability</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Using a command to render a webview where an ADR can be chosen to be edited</td>
<td>The user executes a command to open a webview where they can choose to edit ADRs that exist inside a specific path on the workspace.</td>
<td>well suited</td>
<td>By using a webview, the UI of the ADR Manager can be reused (or at least be based on it), making the implementation easy and reminiscent of the original ADR Manager.</td>
</tr>
<tr>
<td>A2</td>
<td>Adding a new submenu when right-clicking on a Markdown file with the option to edit using the extension</td>
<td>The user can right-click on a Markdown file to select an option to edit the ADR using the extension.</td>
<td>well suited</td>
<td>This option could be implemented in parallel to another alternative listed in this table as it is intuitive and common to select the file directly from the workspace.</td>
</tr>
<tr>
<td>A3</td>
<td>Using a command to edit specific fields of an ADR</td>
<td>The user executes a command which prompts them to select an existing ADR in the workspace and then specify a certain field of the ADR that he would like to change. The user is then shown the current value of the requested field and is asked to update the value.</td>
<td>not suited</td>
<td>The user should be able to see all fields of an ADR and their corresponding values at once.</td>
</tr>
<tr>
<td>A4</td>
<td>Using the Markdown editor provided by VS Code to manually edit ADRs</td>
<td>The user opens the existing ADR that they want to edit using the built-in Markdown editor to make and save changes.</td>
<td>moderately suited</td>
<td>There may be users who are not familiar with writing raw Markdown. Additionally, formatting and inconsistency issues may occur.</td>
</tr>
</tbody>
</table>

Table 4.3: Assessment of different alternatives for the requirement “Editing existing ADRs” (shortened)

After assessing all alternatives, there was at least one alternative for each requirement chosen to be sketched and implemented in the prototype.
4.2 Design Phase

Creating the Mock-Up

During the design process, the second and third tasks of the prototype designing process were executed in parallel with the help of Figma\(^3\), a UI design tool with the ability to create interactive prototypes that can be easily shared and accessed via the web browser. The look and feel of VS Code could be simulated directly in Figma by using the official Visual Studio Code Toolkit\(^4\) template from Microsoft, allowing the user to not only use the original colors used in VS Code but also place different UI elements like buttons, input fields or icons directly in the sketches.

Based on the chosen implementation alternatives, seven different sketches were created in Figma to cover every validated requirement:

**Accessing Extension Features**  This sketch showcases the ways the user will be able to use the features of the extension, mainly through the use of commands and the context menu when right-clicking a file in the file explorer of VS Code.

**Main Webview**  This sketch shows a webview that is intended to be the central view of the extension. From this view, the user has access to all major features of the extensions, like adding new ADRs or editing and deleting existing ADRs.

**Short ADR Template Webview**  This sketch depicts another webview that renders a form-based MADR editor akin to the basic editor mode of the original ADR Manager where ADRs can be created or modified.

**Long ADR Template Webview**  This sketch shows a webview that is similar to the “Short ADR Template Webview”, but instead renders a more detailed editor that is similar to the professional editor mode of the web-based ADR Manager.

**Convert ADR Webview**  This sketch displays a webview that shows up when the user tries to edit an ADR with the ADR editor provided by the extension, even though the ADR in question does not conform to the MADR format. This sketch is also based on the original ADR Manager.

**Delete Confirmation Webview**  This sketch displays a popup over the “Main Webview” that asks the user for confirmation before deleting an ADR from the webview.

**Raw Markdown Editor Features**  This sketch presents some basic features of the extension that are available directly in the built-in VS Code Markdown editor, such as linting or using auto-completion.

\(^3\)Figma, [https://www.figma.com](https://www.figma.com)
4 Methodology

Architecture Diagram

In addition to the mock-up, a diagram was created which illustrates the general architecture of the extension and the interactions between components. This diagram also visualizes the data flow within the context of the IDE. It was used to get a better understanding of how the extension would transform user input into the desired output, and was also a general reference when implementing features that involved data stored on the local computer.

4.2.2 Discussing the Prototype

After the sketches were made, they were turned into an interactive prototype [Che22] that a human can interact with by clicking on certain elements of a sketch to navigate to different sketches. Each sketch was also accompanied by a brief explanation.

This prototype was then shared with the members of the focus group for feedback, which was also documented using a Google Sheets spreadsheet.

Some noteworthy changes that resulted from the feedback were:

1. The “Convert ADR Webview” view has been removed because with the built-in Markdown editor from VS Code, there is no need for an additional view where the user can edit the raw Markdown to make an ADR MADR-conform.
2. The custom popup on the “Delete Confirmation Webview” sketch will not be included in the extension. Instead, a built-in VS Code popup will be used to ask for confirmation.
3. The user can now switch between the two different editor modes when creating or updating an ADR. This change aims at making the workflow of the extension more familiar for users who are used to the workflow of the original tool.
4. The terms “short ADR” and “long ADR” have been renamed to “basic ADR” and “professional ADR” respectively to adhere to the naming convention of the original, web-based ADR Manager.

With the prototype created and the feedback in mind, the actual implementation of the extension was initiated.

4.3 Implementation Phase

In the third phase of the extension development process, the Implementation Phase, the functionality of the extension was implemented in VS Code using the Extension API. The milestone to reach in this phase was to have an extension that was “ready for the final user evaluation”. Due to the time constraints of this study, it was agreed upon that the extension was ready for evaluation if at least all major features of the original ADR Manager were implemented in the extension. As a consequence, some requirements have not yet been implemented in the current version of the VS Code extension. Specifically, the ADR converter from requirement 4 and requirement 16 from Table A.1 have not been implemented yet, which may be a basis for future work.
4.3 Implementation Phase

4.3.1 Implementing the Prototype

For implementing the prototype in VS Code, an iterative prototyping approach was chosen. For this, the sketches from the previous phase were fully implemented one by one using the Extension API, thereby prioritizing sketches that focused on providing features from the original ADR Manager. This led to the following prioritization of sketches:

1. Accessing Extension Features (as a prerequisite for accessing the other sketches)
2. Main Webview
3. Basic ADR Template Webview
4. Professional ADR Template Webview
5. Raw Markdown Editor Features

For each sketch that had been implemented, a synchronous meeting with the supervisor was held in which the new implementation was showcased and discussed. The goal of this approach was to generate detailed feedback that was directed at specific parts of the extension instead of “only” receiving superficial feedback regarding the extension as a whole. Thus, this approach followed the principle of failing fast and discovering problems and uncertainties in the extension as early as possible. Additionally, when under time constraints, Dow et al. observed a higher performance when using an iterative approach compared to a single iteration [DHK09].

To get a holistic impression of the extension, the focus group of this study was asked to try out the extension once all sketches had been implemented and discussed internally. For this, the extension was published on the Visual Studio Code Marketplace\(^5\) so that the members of the focus group had easy access to the extension and could evaluate the extension in their free time\(^6\). The feedback came in the form of free text comments via email, upon which the extension was improved. This procedure was performed twice and with that, the extension was deemed ready for evaluation.

Summary

This chapter described the whole study process up until the ADR Manager extension had reached a stage at which it was mature enough to proceed with a final user evaluation. To summarize, the development process started with eliciting the extension requirements by analyzing the original ADR Manager and using different ideation methods. The elicited requirements were validated and then used to create an interactive UI mock-up of the extension, which aimed to fulfill these requirements. Lastly, the mock-up was taken as a reference for the actual implementation which was discussed in detail before moving on to the next phase of the study process.

The result of this process, the *ADR Manager VS Code Extension*, will be presented in the following chapter.


\(^6\) The time of developing the extension fell right into the summer break in Germany, so it would have been difficult to find a free time slot for the evaluation
5 ADR Manager Extension

This chapter introduces the ADR Manager VS Code Extension in its current form, focusing on design decisions regarding the UI. This chapter also looks at how the extension features have been implemented.

The extension is available for download on the Visual Studio Marketplace\(^1\). The source code of the extension is available on the GitHub repository\(^2\) of the extension. The current version of the source code has been archived via Zenodo\(^3\).

There is also an introductory GitHub repository\(^4\) which explains all currently implemented features of the extension in the style of a tutorial.

5.1 Design

This section briefly looks at some of the major design choices that affect numerous parts of the implementation.

5.1.1 Multiple Ways of Accessing the Extension Features

Figure 5.1 shows the architectural diagram designed for the extension. It displays two generic ways for the user to access the features of the extension: they can 1) interact with the webview panel that was created by the extension, or 2) use other common features of the IDE such as commands or the file explorer.

User inputs prompt the extension to execute certain functions of the Extension API that allow the extension to read or write local data on the computer (e.g., fetching existing ADRs or creating new ADRs). Based on the response, the extension can directly update the VS Code UI or send a message including a payload for the webview to render.

As suggested in the sketch “Accessing Extension Features”, the extension offers more than one approach to using its features (excluding the use of webviews). The reason for this is to provide multiple paths that lead to the same goal, which gives users the freedom to choose their preferred way of working instead of forcing them to adhere to a workflow that they potentially do not enjoy.

---

\(^1\)ADR Manager Extension, https://marketplace.visualstudio.com/items?itemName=StevenChen.vscode-adr-manager
\(^2\)GitHub repository of the ADR Manager extension, https://github.com/adr/vscode-adr-manager
\(^3\)DOI to archived source code, https://doi.org/10.5281/zenodo.7092738
\(^4\)Introduction to the ADR Manager Extension, https://github.com/adr/vscode-adr-manager-introduction
5.1.2 Color Choice

One of the main goals of the extension is to make the extension as unobtrusive as possible by having the extension integrate nicely into the context of VS Code. But because of the large number of customization options that VS Code offers, it becomes hard – if not impossible – to pinpoint a specific VS Code environment. In particular, the option to choose from a big selection of color themes makes it difficult to design a UI that feels native to the user.

VS Code solves this problem for webview extension authors by providing the colors used in the currently active color theme as CSS variables: instead of defining colors that cannot be changed by the user, CSS variables can be used to automatically pick fitting colors regardless of the color theme of the user. All available CSS color variables are listed in the Extension API reference for theme colors.

Listing 5.1 demonstrates how CSS variables can be used. Here, a webview HTML body is defined with a background color that is the same as the background color of VS Code. The font color will also have an adequate amount of contrast to the background.

---

5 As of writing, there are 8196 themes available for download in the Visual Studio Code Marketplace.

6 Reference for VS Code Theme Colors, [https://code.visualstudio.com/api/references/theme-color](https://code.visualstudio.com/api/references/theme-color)
5.2 Implementation

Listing 5.1 Example of using CSS variables for automatically picking VS Code theme colors

body {
  color: var(--vscode-foreground);
  background-color: var(--vscode-editor-background);
}

5.1.3 Resemblance to the original ADR Manager

Besides having the UI mock-up as a reference for the actual implementation in VS Code, the general design of the webviews deliberately bears a resemblance to the web-based ADR Manager. By having a streamlined design for both the IDE version and the web version, users experienced in at least one version of the tool should feel a sense of familiarity when using any other version of the ADR Manager.

5.2 Implementation

This section presents details about the general structure of the underlying source code and demonstrates how the requirements have been realized in the extension.

5.2.1 Source Code Structure

The source code of the extension can be roughly divided into the following main sections:

- The src directory contains all JavaScript and TypeScript source code for the logic of the VS Code extension. Examples include the ADR parser or functions of the Extension API to interact with local data on the computer.
- The web directory includes code for defining the contents of a webview, i.e., HTML, TypeScript, and CSS content that are passed to the extension to render inside of a webview.
- The package.json file (also known as the Extension Manifest)\(^7\) not only specifies metadata or dependencies of the extension but also defines the features that extend the base functionality of the IDE, e.g., new commands or new menu entries.

The following segments take a deeper look into the most important files and subdirectories of these three sections of the source code.

\(^7\)VS Code Extension Manifest, https://code.visualstudio.com/api/references/extension-manifest
The src Directory

The src directory contains files that implement the main features of the extension:

**extension.ts** The extension.ts file is the main entry point of the extension and defines the activate() and deactivate() functions. They are called when the user enables or disables the extension in the IDE. While the activate() function sets up the extension, for example by specifying new commands and their actions, the deactivate() function frees up resources used by the extension and removes the features of the extension from the IDE.

**extension-functions.ts** The extension-functions.ts file can be labeled as “the brain” of this extension, as it contains all functions that interact with the Extension API. In this file, functions like fetching all ADRs from a specified folder or creating a new ADR as a Markdown file are implemented. They are then exported to other files located in the src directory. As an example, Listing 5.2 shows the source code for saving a basic ADR with the specified data.

**WebPanel.ts** The WebPanel.ts file implements the WebPanel class which contains the webview of the extension. The WebPanel class handles the rendering of the webview as well as the logic for exchanging messages with the webview. There may only be a maximum of one active instance of the WebPanel class at any given moment.

**classes.ts** The classes.ts file exports the class ArchitecturalDecisionRecord. Instances of the ArchitecturalDecisionRecord class represent ADRs in the MADR format as JavaScript objects, making ADRs more convenient to handle in the code.

**parser.js** The parser.js file implements the md2adr() function. This function utilizes the ADR parser to convert ADRs from their string representation (e.g., the content of a Markdown file containing an ADR) into an equivalent JavaScript object of the type ArchitecturalDecisionRecord. The file also defines the adr2md() function to convert an ArchitecturalDecisionRecord object back into an ADR string.

The reason why these functions are written in JavaScript instead of TypeScript is that the parser (which is located in the directory src/plugins/parser/) has been generated using ANTLR4, which does not fully support a TypeScript implementation at the current time. The parser.js file and also the ANTLR4 parser itself have been reused from the original ADR Manager parser, albeit slightly modified.

---

Listing 5.2 Source code for creating a basic ADR from specified fields

```javascript
/*
 * Creates a new ArchitecturalDecision object with the minimum required fields (basic ADR) and
 * saves the ADR as a Markdown file in the ADR Directory.
 * @param fields The fields of the new short ADR
 */
export function createBasicAdr(fields: {
  yaml: string;
  title: string;
  contextAndProblemStatement: string;
  consideredOptions: {
    title: string;
    description: string;
    pros: string[];
    cons: string[];
  }[];
  chosenOption: string;
  explanation: string;
}) {
  const adrFields = {
    yaml: fields.yaml,
    title: fields.title,
    contextAndProblemStatement: fields.contextAndProblemStatement,
    consideredOptions: fields.consideredOptions,
    decisionOutcome: {
      chosenOption: fields.chosenOption,
      explanation: fields.explanation,
      positiveConsequences: [],
      negativeConsequences: [],
    },
  };
  const newAdr = getAdrObjectFromFields(adrFields);
  // Convert ADR object to Markdown and save it in the ADR Directory
  const newMD = adr2md(newAdr);
  saveMarkdownToAdrDirectory(newMD, newAdr.title);
}
```

The web Directory

The web directory contains all files that are relevant to the actual content that is being rendered by the webview. It is structured into multiple subdirectories, with the most important ones being:

**components** The components directory contains elements of the webview that can be reused in multiple views or other components. As the UI is made with the framework Vue, the components within this directory are saved in .vue files. The .vue files contain HTML, CSS and TypeScript for each component.
mixins  The mixins directory stores so-called mixins in TypeScript files. Mixins are a feature of Vue and are used to define reusable code that is used throughout multiple components. The mixins adr-data.ts and save-adr.ts include functions to manipulate and save ADRs data within Vue components. To pass messages between the webview and the extension, the postMessage() function has to be imported. This is done via the mixin vscode-api-mixin.ts.

views  The views directory includes .vue files that represent the root components of a webview. Root components themselves are comprised of child components that are located in the components directory.

pages  The pages directory contains a TypeScript file for each view in the views directory. They are used to inject the contents of a specific view into the webview. This mechanism will be explained further down in this chapter.

static  The static directory contains CSS files that are shared across all components. The vscode.css and reset.css files use CSS variables provided by VS Code to define colors for common HTML elements such as headings, input fields, and buttons that fit with the currently active color theme.

The Extension Manifest

The Extension Manifest (or the package.json file) is a JSON file located in the root directory of the extension. It holds all information that describes the extension, such as the name, the current version number, or the extension author. The Extension Manifest also specifies some functional aspects of the extension, like actions that trigger the activation of the extension (also known as activation events) or its contributions to the functionality of VS Code. These contributions to the base functionality of VS Code are called contribution points and include commands, menus, settings, snippets, and more. Listing 5.3 shows a snippet of the package.json file of the ADR Manager.

The descriptive data from the Extension Manifest is also displayed on the ADR Manager overview page on the Visual Studio Marketplace.

Packaging the Extension

When publishing the extension to the Visual Studio Code Marketplace, not the whole source code will be published. Instead, the source code will be combined into only a few JavaScript and CSS files to compress the size of the extension as much as possible. For this, the extension uses the module

---

9 JSON format, https://www.json.org
10 VS Code Activation Events, https://code.visualstudio.com/api/references/activation-events
5.2 Implementation

Listing 5.3 Snippet of the package.json file of the ADR Manager

```json
1
2
3
4
(...) 5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
(...)
27
```

bundlers Webpack\textsuperscript{12} and Rollup\textsuperscript{13} with the configurations specified in the files webpack.config.js and rollup.config.js respectively. More specifically, Rollup bundles the webview contents located in the web directory, while Webpack bundles the rest of the source code.

The bundled files will be saved in the dirt directory (not available in the GitHub repository). The dirt directory will be used to create a file in the .vsix file format. This file format represents VS Code extension files and can be uploaded to the Visual Studio Code Marketplace.

5.2.2 Implementation in VS Code

This section presents how the extension – along with its requirements – has been realized inside of VS Code and compares the extension to the UI mock-up.

\textsuperscript{12}Webpack, https://webpack.js.org
\textsuperscript{13}Rollup, https://rollupjs.org
ADR Directory Concept

The VS Code ADR Manager provides its features by utilizing the workspace concept of VS Code. To use the extension features, the user has to open at least one folder as a root folder in the workspace of a VS Code instance.

Building onto that, the user has to specify an ADR Directory: a path to a directory inside of the workspace that is relative to a root workspace folder. This also implies that there can be multiple ADR Directories if there are multiple root folders in the workspace. Features like adding ADRs or fetching ADRs will happen inside this specific directory. By default, the ADR Directory is set to docs/decisions (relative to a workspace root folder).

Depending on the number of folders opened as root folders in the same workspace, the extension will behave differently:

- **Single-root workspace** If the user has opened only one folder in the workspace as a root folder, then there can only be exactly one ADR Directory upon which the extension will act when the user uses its features.

  **Example:** The user has set the ADR Directory to docs/decisions and opens the repository vscode-adr-manager as the only root folder in VS Code instance. Then, the extension will only use the folder vscode-adr-manager/docs/decisions/ for fetching or adding ADRs.

- **Multi-root workspace** If the user has opened multiple folders in the workspace as separate root folders, then there may be multiple ADR Directories. The extension will then try to take every possible ADR Directory into account when a user utilizes its features.

  **Example:** The user has set the ADR Directory to docs/decisions and opens the repositories vscode-adr-manager and adr-manager in the workspace as separate root folders. Then, the resulting ADR Directories will be vscode-adr-manager/docs/decisions/ and adr-manager/docs/decisions/. Thus, the extension will fetch ADRs from both of those directories and ask in which specific ADR Directory the user wants to save a newly created ADR.

- **Special case: Single-root workspace with only subdirectories** This special case serves as an alternative to the multi-root workspace. If the user has opened a single folder in the workspace that contains only other folders (i.e., no files or links are in this folder), then the extension will treat the subfolders as if they were separate root folders in the workspace. This implementation makes it more convenient for users who do not want to use the multi-root concept of VS Code, e.g., by having a single folder in which they keep multiple repositories.

  If the user does not want to use this feature, they can disable it in the settings.

  **Example:** The folder repos contains only the folders vscode-adr-manager and adr-manager. The user has set the ADR Directory to docs/decisions and opens the repos folder as the sole root workspace folder. Then, the resulting ADR Directories will be vscode-adr-manager/docs/decisions/ and adr-manager/docs/decisions/. Thus, the extension will fetch ADRs from both of those directories and ask in which specific ADR Directory the user wants to save a newly created ADR.
Implementing Webviews

The extension features multiple different views that each serve their own purpose and are rendered inside of the same webview panel. To transition from one view to another, the extension has to specify which view will be loaded inside of the webview after the user performs an action like clicking a button or executing a command.

For this, there are TypeScript files in the `web/pages/` directory that each create an instance of a specific Vue component contained in the directory `web/views/`. These Vue components each represent the root component of a certain view and thus contain the entire content of a view. Listing 5.4 shows the creation of a view that is specified in the file `web/views/MainView.vue`.

**Listing 5.4 Content of the file web/pages/main.ts, creating a MainView component**

```typescript
1 import { createApp } from "vue";
2 import Main from "./views/MainView.vue";
3
4 createApp(Main).mount("#app");
```

In the next step, Rollup transforms the Vue component into a pair of JavaScript and CSS files, which are saved in the dist directory. These files are then passed to the webview panel of the extension and injected into the webview. Specifically, the Vue component will be injected into the HTML element with the ID “app”.

Listing 5.5 displays the function `_getHtmlForWebview()` in the `WebPanel` class. This function specifies the content that is rendered in the webview. The view that will be loaded is specified by passing a string key as an argument to the function. The content will be injected into the `div` element on line 15.

**Listing 5.5 Source code for specifying the content of a webview (shortened)**

```typescript
private _getHtmlForWebview(webview: vscode.Webview, page: string) {
    // Local path to main script run in the webview
    const SCRIPT_URI = vscode.Uri.joinPath(this._extensionUri, "dist/web", `${page}.js");
    // URI to load the script in the webview
    const SCRIPT_WEB_URI = webview.asWebviewUri(SCRIPT_URI);

    // Local path to css styles
    const STYLE_URI = vscode.Uri.joinPath(this._extensionUri, "dist/web", `${page}.css");
    // URI to load styles into webview
    const STYLE_WEB_URI = webview.asWebviewUri(STYLE_URI);

    // The content that will be loaded is specified by passing a string key as an argument to the function.
    // The content will be injected into the div element on line 15.
    return `<!DOCTYPE html>
    (...)<body>
        <div id="app"></div>
        (...)</body>
    </html>`;
}
```
The following sections present the different views that are implemented in the extension and briefly show the capabilities of each view.

Figure 5.2: Main webview in the UI Mock-up

Figure 5.3: Main webview in the VS Code extension
Main Webview  Figure 5.2 and Figure 5.3 show the UI mock-up and the actual implementation of the main webview. Under a prominent ADR Manager logo\(^\text{14}\), the webview renders all potential ADRs in the current ADR Directories of the workspace, along with the title of the ADR and the location within the workspace. For each ADR in this list, the user has access to two buttons: the “View” button prompts the extension to transition to the ADR editor, while the “Delete” button lets the user delete ADR from within the webview. If the content of an ADR does not conform to the MADR format, the border of the ADR will appear in red, and an error message will be displayed below the ADR container. Under the ADR list, a green “Add ADR” button leads the user to the ADR editor. There, the user can create a new ADR and save it in an ADR Directory.

To prevent the user from accidentally deleting an ADR by misclicking on the “Delete” button, a VS Code warning will appear upon clicking on the button. The warning will ask the user to confirm the deletion or to cancel the deletion process. If the user manages to also unintentionally confirm the deletion, the extension does not permanently delete the ADR. Instead, it moves the ADR from the workspace folder to the trash bin on the computer.

While the implementation does not differ much from the UI mock-up, the web-based ADR Manager does not feature an entire screen dedicated to displaying the available ADRs. Instead, the ADRs are shown in a side panel along with their corresponding GitHub repositories.

This webview can be accessed via the VS Code command “Open ADR Manager” or via the context menu when right-clicking on an ADR Directory.

![Basic Editor webview of the ADR Manager extension](image)

**Figure 5.4:** Basic Editor webview of the ADR Manager extension

Basic ADR Editor  Figure 5.4 shows the implementation of the basic ADR editor of the extension. This webview is rendered when the user wants to create or edit an ADR. It displays input fields for every mandatory field of an ADR. Additionally, the editor features tooltips that, when hovered, display additional informative texts for a particular field. It also validates the user input: an invalid input is indicated by a red border around an input field and an error message below that input field. The button to create or edit the ADR is disabled unless all mandatory fields have been filled in by the user.

\(^{14}\text{taken from the original ADR Manager, https://github.com/adr/adr-manager/blob/main/src/assets/logo.png}\)
At the top of the webview, the user can go back to the main webview of the extension. They can also open the ADR using the built-in Markdown editor or switch the editor mode from “basic” to “professional”.

While this design bears a resemblance to the web-based tool, there is one glaring difference: to choose an option, the user has to click on the container of the option they want to choose. This implementation aims to make the process of choosing an option quicker and more intuitive for the user. They just point and click on an option, just as one would do in real life by pointing at something. Extending this metaphor, the user can also rearrange the order of the options via drag and drop.

This webview is accessible through the VS Code command “Add New ADR”, by clicking on a “View” button in the main webview or by switching from the “Professional ADR Editor” webview.

![Figure 5.5: Professional editor in the UI mock-up](image)

**Professional ADR Editor**  Figure 5.5 and Figure 5.6 show the mock-up and the implementation of the “Professional ADR Editor” webview, which is a more detailed version of the “Basic ADR Editor” webview. To accommodate for the higher level of detail in the template, this webview displays all fields of an ADR and also allows the user to click on options to reveal more input fields.

Unlike in the UI mock-up, the user does not have to manually add input fields for each positive or negative argument of an option: the webview will automatically add a new input field if the user enters an argument to either list. The extension also does not provide the ability to enter neutral arguments, as the extension supports the MADR 2.1.2 format, which does not consider neutral arguments.
5.2 Implementation

Figure 5.6: Professional editor in the VS Code extension

Similar to the “Basic ADR Editor” webview, this webview can be accessed through the VS Code command “Add New ADR”, by clicking on a “View” button in the main webview or by switching from the “Basic ADR Editor” webview.

Features for the Markdown Editor

The ADR Manager extension also adds some features to the built-in Markdown editor of VS Code. These features help users who are more comfortable with writing raw Markdown instead of using webviews to manage ADRs. For now, the extension only provides some basic features for the raw Markdown editor:

Diagnostics  The extension offers linting rules (diagnostics) that help the user with detecting inconsistencies within the Markdown content of their ADRs. Problematic sections detected by the extension are underlined in the editor using the two severity levels warning (yellow underlining) and error (red underlining). Currently, diagnostics for the following problems have been implemented in the extension:

- Missing title header (error)
- Missing subheader for any other required field of an ADR, i.e., missing subheaders for “Context and Problem Statement”, “Considered Options”, or “Decision Outcome” (warning)
- Empty required field even though the subheader for the required field is present (warning)
- Heading or subheading is not written in title case (warning)
- The chosen option is not listed in the list of considered options (error)
This feature was implemented to showcase the basic linting features of the built-in Markdown editor of VS Code. In the future, more diagnostics will be implemented, for example for detecting duplicates in the list of considered options or when trying to add positive and negative arguments to an option that is not listed in the list of considered options.

**Snippets** For creating new ADRs with raw Markdown, the extension provides two snippets of the ADR template that can be inserted by typing a certain keyword. The basic ADR template only contains the mandatory fields of an ADR and can be inserted by typing the keyword `basic-madr`. If the user wants to generate a template with all ADR fields, they can insert the professional ADR template by using the keyword `professional-madr`.

After inserting a template, the user can quickly fill in its placeholders by using the “Tabulator” key to jump to the next placeholder.

**Extension Settings**

For further customization, the extension provides the following settings that can be configured by the user:

**ADR Directory** The user can change their preferred ADR Directory to another path. The path is a relative path starting from a root directory in the workspace and must not include the special characters that are prohibited in file or folder names. The default ADR Directory is `docs/decisions`. This setting can also be changed via the command “Change ADR Directory”.

**Show Diagnostics** The user can choose whether they want the extension to display diagnostics when opening ADR files with the raw Markdown editor.

**Treat Single Root as Multi Root** The user can choose whether they want to treat certain single-root workspaces as if they were multi-root workspaces (as described in the “special case” of the ADR Directory concept).

**Add ADR Editor Mode** The user can decide which ADR editor mode the extension initially renders upon clicking on the “Add ADR” button in the main webview. They can choose between “basic” and “professional”. The default value for this setting is “basic”.

**View ADR Editor Mode** The user can decide which ADR editor mode the extension initially renders upon clicking on a “View” button in the main webview to display an existing ADR. They can choose between “basic”, “professional” and “sufficient”, with “sufficient” meaning that the extension only chooses the basic ADR editor if the existing ADR does not contain any optional fields. The default value for this setting is “sufficient”.

---

15More specifically, the extension does not allow the following characters: ? * : " < > |
Summary

This chapter introduced the ADR Manager VS Code Extension in its current form. It explained the rationale behind some of the design choices and gave insights into the structure of the source code. Finally, the features of the extension were briefly laid out.

The next chapter handles the final two phases of the study process: the Evaluation Phase and the Analysis Phase. There, the methodology behind evaluating the ADR Manager VS Code Extension and the results of the evaluation are presented.
6 Evaluation and Analysis

This chapter covers the last two phases of the study process, the Evaluation Phase, and the Analysis Phase.

The first section of this chapter looks at the Evaluation Phase and outlines the purpose of the evaluation, the preparation leading up to the interviews of the evaluation, as well as the process of conducting the interviews. Following that, the Analysis Phase is presented by illustrating the procedure for analyzing the raw evaluation data collected from the interviews with the participants. Then, the results of the analysis are summarized and compared to a goal that was set before the evaluation.

For the most part, the methodology behind the evaluation and the analysis was deliberately based on the evaluation study of the original ADR Manager\(^1\), in hopes of making the results of the two studies more comparable.

The evaluation artifacts can be found in the GitHub repository of the extension\(^2\).

### 6.1 Evaluation Phase

The Evaluation Phase was the fourth phase of the study process. In this phase, the ADR Manager extension was tested by multiple participants to generate feedback regarding its usability and functional suitability. This feedback was collected by conducting interviews with each of the participants. The milestone for this phase was to have a collection of aggregated, anonymized data from all interviews that can be used for further analysis in the next phase of the study process.

#### 6.1.1 Purpose

Aside from receiving user feedback on the functional aspects of the extension, the main intention of the evaluation was to examine the perceived usability of the extension. In particular, the evaluation should inspect whether the participants would deem the extension to be usable in the context of VS Code.

\(^{1}\)Evaluation study of the original ADR Manager, [https://github.com/adr/adr-manager/tree/main/docs/evaluation/user-study](https://github.com/adr/adr-manager/tree/main/docs/evaluation/user-study)

\(^{2}\)Artifacts of the extension evaluation, [https://github.com/adr/vscode-adr-manager/tree/master/docs/evaluation](https://github.com/adr/vscode-adr-manager/tree/master/docs/evaluation)
6 Evaluation and Analysis

6.1.2 Preparation

To prepare for the evaluation, the following tasks needed to be completed:

Structuring the Interviews

The first step in creating a structure for the interviews was to establish the data that would be collected during the interviews. For the evaluation, data with regards to three different domains were needed:

• **Data regarding the participants**, including their *role*, their *professional experience* as well as their *experience with ADRs, MADR, and ADR tools*. This data was collected by asking the participant direct questions about these topics.

• **Data regarding the functionality of the extension**, ranging from feedback on existing features to suggestions for new features that could be implemented in the future. This data was collected by letting the participants solve several small tasks using the extension. The participants were also prompted to think aloud while solving the tasks in an attempt to capture their thought processes.

• **Data regarding the perceived usability of the extension**, which was captured using the *System Usability Scale (SUS)*, a “questionnaire (...) that could be used to take a quick measurement of how people perceived the usability of computer systems on which they were working on” [Bro13]. The SUS is shown in Table 6.1 and produces a usability score ranging from 0 to 100. Participants were asked to rate each item in the SUS with a score from 1 to 5, where a 1 indicates that they strongly disagree with a particular statement and vice versa. Additionally, participants were asked to give their opinion on the user experience of the extension in their own words.

This led to the following interview structure that was loosely followed during all interviews:

1. Asking questions regarding general information about the participant
2. Solving several tasks in the context of the extension while thinking aloud
3. Interviewing the participant about their user experience with the extension
4. Going through the SUS with the participant
6.1 Evaluation Phase

<table>
<thead>
<tr>
<th>ID</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SUS_1$</td>
<td>“I think that I would like to use this system frequently.”</td>
</tr>
<tr>
<td>$SUS_2$</td>
<td>“I found the system unnecessarily complex.”</td>
</tr>
<tr>
<td>$SUS_3$</td>
<td>“I thought the system was easy to use.”</td>
</tr>
<tr>
<td>$SUS_4$</td>
<td>“I think that I would need the support of a technical person to be able to use this system.”</td>
</tr>
<tr>
<td>$SUS_5$</td>
<td>“I found the various functions in this system were well integrated.”</td>
</tr>
<tr>
<td>$SUS_6$</td>
<td>“I thought there was too much inconsistency in this system.”</td>
</tr>
<tr>
<td>$SUS_7$</td>
<td>“I would imagine that most people would learn to use this system very quickly.”</td>
</tr>
<tr>
<td>$SUS_8$</td>
<td>“I found the system very cumbersome to use.”</td>
</tr>
<tr>
<td>$SUS_9$</td>
<td>“I felt very confident using the system.”</td>
</tr>
<tr>
<td>$SUS_{10}$</td>
<td>“I needed to learn a lot of things before I could get going with this system.”</td>
</tr>
</tbody>
</table>

Table 6.1: The System Usability Scale by Brooke [Bro96]

Creating Tasks for the Interviews

To get as much feedback as possible regarding the functionality of the extension, the participants should solve tasks that cover the majority of the extension features. For this, two types of tasks were created that aim to showcase different features of the extension:

1. Creating an ADR according to a fictional meeting summary. The goal of this type of task was to demonstrate the ADR editors of the extension and how they could be used to create and edit ADRs.

2. Finding and fixing an error in an already existing ADR. This type of task addressed the ability of the extension to help with detecting inconsistencies in ADRs.

Two tasks from each task type were created, where one task was meant to be solved without the help of the extension, and the other was intended to be solved with the extension. Thus, each participant had to solve four tasks in total.

Similar to the original ADR Manager study, the created tasks revolved around the following fictional scenario:

“The U.S. American company Snackato Inc. is active in the food industry and produces a variety of snacks using potatoes. Snackato Inc. has decided to document the business decisions of each business department in a central repository using the MADR format.”

The tasks were documented in two different task lists, the `task-list-without-extension.md` and `task-list-with-extension.md`. The task completion times were tracked for comparison, even though task duration was not a central variable for this study.
Writing a Call for Participation

The call for participation is a document that was created for recruiting people to take part in the evaluation of the extension. This document includes the purpose of the evaluation, the requirements for each participant, the procedure of the interview, and a privacy policy describing how the collected data from the interviews will be handled and processed.

The specific requirements for participants were 1) a “basic understanding and experience in the use of ADRs”, and 2) being “familiar with the core features and workflow of the VS Code IDE”. Participants were not required to have experience with MADR. The requirements were intentionally non-restrictive such that people were not deterred from them.

Additionally, the potential participants were asked to familiarize themselves with the extension before the interview by taking a look at the introduction repository of the ADR Manager extension.

The call for participation also stated that the interview will be recorded, although the data from the interview “will be treated as strictly confidential and will not be disclosed to any third party not involved in this study”. The recordings were used to extract statements regarding the extension and to create an anonymized case description for each participant. The case descriptions describe the general course of the participants solving the tasks and their feedback for the extension.

The privacy policy affirmed that only aggregated, anonymized data will be published and used for further analysis. The recordings were also destroyed upon having extracted and anonymized the data from them.

Recruiting Participants

Using the call for participation, the supervisor helped with the recruitment by contacting several peers from both academia and industry and inviting them to the evaluation study. This resulted in seven people that were willing to participate in the evaluation study, most of whom have also participated in the original ADR Manager evaluation. As the interviews were held online using the video conferencing tool WebEx, meeting invites were also sent to the participants via email.

6.1.3 Execution

After the interviews were prepared, each participant was interviewed over a one-on-one WebEx meeting. At the beginning of each meeting, the participant was reminded of the privacy policy of the evaluation. The interview and the recording were only initiated if the participant accepted the terms and conditions of the privacy policy. After the recording was started, the interview followed the general structure that was previously laid out.

---

3 Call for Participation for the extension evaluation, https://github.com/adr/vscode-adr-manager/blob/master/docs/evaluation/material-before-study/call-for-participation.md


5 WebEx, https://www.webex.com/
After answering the general questions about themselves, participants received the tasks they had to solve in the next step of the interview, either via the data transfer functionality of WebEx or via email. This way, the participants had no chance to think of a solution beforehand.

Every participant was then introduced to the fictional scenario surrounding the tasks. Afterwards, they were prompted to first solve the tasks without the help of the extension before moving on with the tasks that needed to be solved with the extension. The task order was chosen such that the time between using the extension and giving feedback regarding the user experience was minimized.

While solving the tasks, any intervention was avoided unless the participant needed help to progress with the tasks. The participants were given the freedom to solve the tasks in a way that was comfortable for them.

After the tasks had been solved, the participants were given a chance to express their feedback in their own words before going over the SUS, which marked the end of each interview.

Because the interviews had been recorded, there were no additional notes taken during the entire interview process to stay focused on the thoughts and actions of the participant.

The seven participants were interviewed in the span of two weeks, with interviews lasting approximately 30 to 90 minutes. The length of an interview was thereby largely determined by the task completion time as well as the level of detail in the discussion after solving the tasks. In between the interviews, the case descriptions had been created by using the interview recordings. Each recording was reviewed from start to finish and all statements regarding the extension and actions of a participant were anonymized and documented. The case descriptions are also available on GitHub.

With the case descriptions of every interview available, the Evaluation Phase was concluded and the analysis of the evaluation data had started.

6.2 Analysis Phase

The **Analysis Phase** was the last phase of the study process and was centered around interpreting the generated feedback from the participants regarding the functionality and usability of the extension. The milestone for this phase was to not only obtain an idea of the overall sentiment towards the functionality and usability of the extension but also deriving potential next steps for the extension.

6.2.1 Goal

For motivational purposes, a goal was set at the beginning of the study process concerning the perceived usability of the extension. At first glance, the SUS score offered itself well for setting a goal for this study. But the problem was that the SUS score on its own does not inherently provide a way to compare the quality of usability with other computer systems.

---

To solve this issue, the *Sauro-Lewis curved grading scale* was applied to the SUS score. This curved grading scale was constructed with data from “over 5000 completed SUS questionnaires” and “provides an empirically grounded approach to the interpretation of mean SUS scores” [Lew18]. The scale was used to transform the numerical SUS score into a letter grade ranging from F to A+, with the latter being the best achievable grade.

Table 6.2 shows the SUS score ranges according to the Sauro-Lewis curved grading scale, along with their corresponding letter grade and percentile range relative to the SUS scores that were analyzed in the creation of this grading scale.

<table>
<thead>
<tr>
<th>SUS Score Range</th>
<th>Grade</th>
<th>Percentile Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>84.1 - 100</td>
<td>A+</td>
<td>96 - 100</td>
</tr>
<tr>
<td>80.8 - 84.0</td>
<td>A</td>
<td>90 - 95</td>
</tr>
<tr>
<td>78.9 - 80.7</td>
<td>A-</td>
<td>85 - 89</td>
</tr>
<tr>
<td>77.2 - 78.8</td>
<td>B+</td>
<td>80 - 84</td>
</tr>
<tr>
<td>74.1 - 77.1</td>
<td>B</td>
<td>70 - 79</td>
</tr>
<tr>
<td>72.6 - 74.0</td>
<td>B-</td>
<td>65 - 69</td>
</tr>
<tr>
<td>71.1 - 72.5</td>
<td>C+</td>
<td>60 - 64</td>
</tr>
<tr>
<td>65.0 - 71.0</td>
<td>C</td>
<td>41 - 59</td>
</tr>
<tr>
<td>62.7 - 64.9</td>
<td>C-</td>
<td>35 - 40</td>
</tr>
<tr>
<td>51.7 - 62.6</td>
<td>D</td>
<td>15 - 34</td>
</tr>
<tr>
<td>0.0 - 51.6</td>
<td>F</td>
<td>0 - 14</td>
</tr>
</tbody>
</table>

Table 6.2: The Sauro-Lewis curved grading scale [Lew18]

The goal set for the extension was to achieve an average SUS score of at least 74.1 across all participants of the evaluation. This corresponds to the letter grade B or better.

**6.2.2 Analysis Procedure**

The following section describes how the interview data was analyzed and briefly touches on the purpose of each analysis. Only the anonymized data from the case descriptions were used throughout the entire procedure.

**Task Completion Times**

The task completion times for each participant were aggregated based on whether the extension could be used for a particular task. This resulted in two separate task completion times for each participant. These times were then rounded to the nearest minute, and the average times across all participants were calculated. The average task completion times were compared with each other to potentially hint at differences between working with and without the extension.
As this evaluation was not focused on comparing unassisted and assisted management of ADRs, this comparison should merely be viewed as an observation, especially since the participants were using the extension for the first time and also had to think aloud while doing so; a separate study needs to be conducted to further investigate this topic.

**Statements Regarding the Extension**

The statements from the case description regarding the extension were categorized using the constant comparative method [Gla65], giving each statement an initial, low-level label. These labels – along with their statements – were then recursively grouped into higher-level categories, until the categories on the highest level could not be grouped anymore. Additionally, each statement was marked as a positive, negative, or neutral statement about the extension. A neutral statement is a comment about the extension that does not rate the extension in any way.

This analysis does not only structure the statements based on their context but also provides an overall impression of the functionality and usability of the extension. As statements may be mentioned multiple times by different participants, the number of mentions can also be used to weigh the importance of a statement. In this case, the weight of a statement can, for instance, be used for prioritization when updating the extension in the future.

**System Usability Scale**

Every participant responded to each of the 10 SUS items with a score ranging from 1 to 5, where a 1 was given as a response to an item if a participant strongly disagreed with a particular statement, and vice versa for a response of 5. With these individual item scores, the overall SUS score for a participant was calculated in the following way:

Each item score was first normalized to be in the range of 0 and 4: starting the count from 1, the score of every odd-numbered questionnaire item was subtracted by 1, and the score of every even-numbered questionnaire item was subtracted from 5.

Then, the normalized item scores were added together and multiplied by 2.5, leading to a final SUS score that ranges from 0 to 100.

The following equation shows the calculation of the final SUS score for an individual participant, where $SUS_X$ stands for the individual item score of the X-th questionnaire item in the SUS:

$$SUS\ score = 2.5 \times ((SUS_1 + SUS_3 + SUS_5 + SUS_7 + SUS_9 - 5) + (25 - (SUS_2 + SUS_4 + SUS_6 + SUS_8 + SUS_{10})))$$

After calculating the SUS scores for every evaluation participant, the SUS scores were averaged. This average SUS score was finally graded using the Sauro-Lewis curved grading scale and used to check whether the goal has been met. Furthermore, the averages of all individual item scores were also calculated as they offer insights into specific aspects of the perceived usability of the extension, such as the learnability (e.g., covered by $SUS_4$ or $SUS_7$) or the satisfaction (e.g., covered by $SUS_1$ or $SUS_8$).
6.3 Results

After laying out the analysis procedure, this section presents the results of the ADR Manager extension evaluation. They can also be found in the GitHub repository.

6.3.1 Overview of the Participants

<table>
<thead>
<tr>
<th>ID</th>
<th>Role</th>
<th>Years of Exp.</th>
<th>Years of ADRs Exp.</th>
<th>Familiar ADR Formats</th>
<th>Exp. with MADR</th>
<th>Exp. with ADR Tooling</th>
<th>Exp. with original ADR Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Professor</td>
<td>12</td>
<td>4</td>
<td>MADR</td>
<td>4 years</td>
<td>Text editor</td>
<td>Yes</td>
</tr>
<tr>
<td>P2</td>
<td>Professor, Software architect</td>
<td>27</td>
<td>21</td>
<td>Y-statements, MADR</td>
<td>Involved in creation</td>
<td>ADMentor, text editor</td>
<td>Yes</td>
</tr>
<tr>
<td>P3</td>
<td>PhD student</td>
<td>6</td>
<td>0</td>
<td>None</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>P4</td>
<td>Scientific researcher, PhD student</td>
<td>8</td>
<td>0</td>
<td>Nygard template, Y-statements</td>
<td>No</td>
<td>Research on current ADR tools</td>
<td>Yes</td>
</tr>
<tr>
<td>P5</td>
<td>Project manager</td>
<td>4.5</td>
<td>5</td>
<td>MADR</td>
<td>5 years</td>
<td>ADR rendering tool</td>
<td>No</td>
</tr>
<tr>
<td>P6</td>
<td>Software architect</td>
<td>7</td>
<td>7</td>
<td>Y-statements</td>
<td>No</td>
<td>Decision Architect</td>
<td>Yes</td>
</tr>
<tr>
<td>P7</td>
<td>Cloud architect</td>
<td>10</td>
<td>7</td>
<td>MADR</td>
<td>5 years</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 6.3: Overview of the evaluation participants

Table 6.3 presents an overview of all evaluation participants. Each participant was given a unique ID and is listed with information about their profession and general experience with ADRs, MADR, ADR tooling, and the original ADR Manager.
6.3 Results

<table>
<thead>
<tr>
<th>ID</th>
<th>Task Completion Time (without extension)</th>
<th>Task Completion Time (with extension)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>13 minutes</td>
<td>14 minutes</td>
</tr>
<tr>
<td>P2</td>
<td>8 minutes</td>
<td>13 minutes</td>
</tr>
<tr>
<td>P3</td>
<td>13 minutes</td>
<td>16 minutes</td>
</tr>
<tr>
<td>P4</td>
<td>15 minutes**</td>
<td>12 minutes**</td>
</tr>
<tr>
<td>P5</td>
<td>9 minutes</td>
<td>7 minutes</td>
</tr>
<tr>
<td>P6</td>
<td>12 minutes</td>
<td>8 minutes</td>
</tr>
<tr>
<td>P7</td>
<td>14 minutes</td>
<td>11 minutes</td>
</tr>
<tr>
<td>Average</td>
<td>11.5 minutes</td>
<td>11.5 minutes</td>
</tr>
</tbody>
</table>

** time not included in the calculation of the average task completion time

Table 6.4: Task completion times of each participant

6.3.2 Task Completion Times

Table 6.4 shows the aggregated task completion times for each participant. The task completion times from P4 are not included in the average task completion times. The reason for this is that additional support was given to P4 while solving the tasks because it was the first time using VS Code for P4. This only became apparent during the interview.

Looking at the average task completion times (P4 excluded), all participants took – on average – the same amount of time to solve the tasks to be solved without the extension as they did for solving the tasks with the help of the extension.

At this point, there is no further interpretation of this result as no significant difference could be identified between these times. There is also a chance that an otherwise positive effect of the extension on the task completion time was nullified by an initial learning curve of the extension or by the participants intentionally taking more time to give more detailed feedback on the extension. Either way, comparing task completion times did not play a major role in the evaluation.

6.3.3 Statements Regarding the Extension

In total, 53 distinct statements regarding the extension have been collected from the interviews. After using the constant comparative method, 7 statements were classified with the label Efficiency, 13 statements with the label Understandability, 14 statements with the label User Interface (Webviews), 14 statements with the label Existing Features, and 5 statements with the label Feature Suggestions. Then, the labels Efficiency, Understandability, and User Interface (Webviews) have been further

---

grouped into the category *Usability*, while the labels *Existing Features* and *Feature Suggestions* have been categorized under *Functionality*. Figure 6.1 illustrates the categorization of the statements as a simple arrow diagram.

![Figure 6.1: Statements grouped with the constant comparative method](image)

Table 6.5 lists the mentions of positive, neutral and negative statements for each label across all participants:

<table>
<thead>
<tr>
<th>Label</th>
<th># of positive statements mentioned</th>
<th># of neutral statements mentioned</th>
<th># of negative statements mentioned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficiency</td>
<td>20</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Understandability</td>
<td>3</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>User Interface (Webviews)</td>
<td>13</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Existing Features</td>
<td>16</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Feature Suggestions</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>52</strong></td>
<td><strong>6</strong></td>
<td><strong>30</strong></td>
</tr>
</tbody>
</table>

*Table 6.5: Number of statement mentions for each label*

When comparing the number of positive statements mentioned to the number of negative statements mentioned, 59% of the total mentions were positive mentions. Notably, the mentions for the labels *Efficiency*, *User Interface (Webviews)*, and *Existing Features* were mostly positive, where the positive mentions make up 95%, 68%, and 76% of the total mentions in their respective labels. This indicates that overall, the participants were satisfied when it comes to these particular aspects of the extension.

Conversely, over 85% of the mentions for the label *Understandability* were negative. Five out of the seven participants felt lost at some point when using the extension: they did not know how to make the extension do what they have in mind or what the extension expects them to do. For instance, the negative statement with the most mentions was the statement “Unsure where to specify the location to save a new ADR”. On the other hand, one could argue that this was to be expected as this was
the first time for the participants to use the extension, which also comes with some learning curve. In the future, the extension should nonetheless still prioritize being concise with expressing their expected input and guiding the user to their desired outcome.

Among the 5 statements with the label Feature Suggestions, the participants wanted to have additional validations regarding the user input, namely a duplicate warning for detecting options that have been added multiple times to an ADR and an error for having only one option listed in an ADR. These features could be implemented both in the ADR editors and in the raw Markdown editor in the form of diagnostics. Also, a participant suggested that the main webview would benefit from having a tree-like structure for displaying ADRs, where the different ADR Directories can be expanded and collapsed to show or hide the ADRs of a specific ADR Directory. Another feature request for the main webview was the option to filter ADRs based on specific criteria like the status or the deciders of an ADR. Issues regarding existing and new features of the extension have been created for the GitHub repository.

Looking at the statements that were mentioned more than once (shown in Table 6.6), the overall sentiment of the participants towards the extension has been mostly positive. While the criticism was mostly directed at the understandability of the extension, the participants praised the extension for being easy to use, having intricate functionality, and having an enjoyable UI.

6.3.4 System Usability Scale

Table 6.7 lists the individual SUS item scores, as well as the final SUS score for each participant. Each SUS score is also accompanied by its corresponding letter grade according to the Sauro-Lewis curved grading scale. Additionally, the averages of the individual SUS items (rounded to two decimal points) and the average of the overall SUS score (rounded to one decimal point) across all participants are displayed.

The SUS scores of the participants indicate overwhelmingly positive feedback regarding the perceived usability of the extension. The SUS scores of the participants range from 82.5 to 95.0, with the average SUS score being 88.2. This corresponds to an average grade of A+ on the Sauro-Lewis curved grading scale, thus comfortably achieving the goal of an average SUS score of at least 74.1, which corresponds to a letter grade of B or better. This also puts the extension in the 96-100 percentile range of the Sauro-Lewis curved grading scale, indicating that the extension received a better SUS score than more than 96% of the systems that were analyzed in the construction of this grading scale.

Taking a look at the average scores for the individual items and the relative number of score responses for each item, the participants unanimously disagreed on the extension being “very cumbersome to use”. This is reflected by item SUS_8 receiving the best possible score of 1 from all participants. Similarly, most participants strongly agreed that the extension can be learned...

---

8 ADR Manager extension repository issues, https://github.com/adr/vscode-adr-manager/issues?q=is%3Aissue+is%3Aopen+sort%3Aupdated-desc

9 with SUS_8 being an even-numbered item, the individual item score for that item is 5 - 1 = 4, with 4 being the best score for an individual item.
### Statement Label Connotation # of mentions

<table>
<thead>
<tr>
<th>Statement</th>
<th>Label</th>
<th>Connotation</th>
<th># of mentions</th>
</tr>
</thead>
<tbody>
<tr>
<td>One can work well with the extension, it is easy to use</td>
<td>Efficiency</td>
<td>positive</td>
<td>7</td>
</tr>
<tr>
<td>Manageable, well considered functionality</td>
<td>Efficiency</td>
<td>positive</td>
<td>7</td>
</tr>
<tr>
<td>The UI is clear, pleasing, engaging</td>
<td>User Interface (Webviews)</td>
<td>positive</td>
<td>5</td>
</tr>
<tr>
<td>Explicitly highlighting errors using a linter is a nice feature</td>
<td>Existing Features</td>
<td>positive</td>
<td>4</td>
</tr>
<tr>
<td>Unsure where to specify the location to save a new ADR</td>
<td>Understandability</td>
<td>negative</td>
<td>4</td>
</tr>
<tr>
<td>It is easy to familiarize themselves with the UI</td>
<td>User Interface (Webviews)</td>
<td>positive</td>
<td>3</td>
</tr>
<tr>
<td>Tooltips for ADR fields are good</td>
<td>Existing Features</td>
<td>positive</td>
<td>3</td>
</tr>
<tr>
<td>Unclear that each list item has their own input field</td>
<td>Understandability</td>
<td>negative</td>
<td>3</td>
</tr>
<tr>
<td>Arrows for expanding options should be reversed</td>
<td>Understandability</td>
<td>negative</td>
<td>3</td>
</tr>
<tr>
<td>ADR editors could be used nicely in meetings</td>
<td>Efficiency</td>
<td>positive</td>
<td>2</td>
</tr>
<tr>
<td>ADR editors are preferred over raw Markdown</td>
<td>Efficiency</td>
<td>positive</td>
<td>2</td>
</tr>
<tr>
<td>Basic Editor: Message regarding optional fields with values is nice</td>
<td>Existing Feature</td>
<td>positive</td>
<td>2</td>
</tr>
<tr>
<td>Features of the extension are very well integrated</td>
<td>Existing Features</td>
<td>positive</td>
<td>2</td>
</tr>
<tr>
<td>Extension should give more information on where/how to enter information</td>
<td>Understandability</td>
<td>negative</td>
<td>2</td>
</tr>
</tbody>
</table>

**Table 6.6:** Statements mentioned more than once across all participants

quickly by most people ($SUS_7$), even without the help of a technical person ($SUS_4$). Like with the statements, the participants highlighted the *ease of use* ($SUS_3$) again and *felt confident* when using the extension ($SUS_9$). Many participants also feel like using the extension in the future ($SUS_1$).

One outlier that needs to be addressed is an individual item score that was given by participant P4: when confronted with the statement “I needed to learn a lot of things before I could get going with this system.” ($SUS_{10}$), P4 answered with a score of 5. As this is an unusual score among
6.3 Results

<table>
<thead>
<tr>
<th>SUS Item</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUS(_1) (Frequent Use)</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4.29</td>
</tr>
<tr>
<td>SUS(_2) (Unnecessary Complexity)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1.43</td>
</tr>
<tr>
<td>SUS(_3) (Ease of Use)</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4.43</td>
</tr>
<tr>
<td>SUS(_4) (Need for Support From a Technical Person)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.14</td>
</tr>
<tr>
<td>SUS(_5) (Good Integration of Functions)</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4.14</td>
</tr>
<tr>
<td>SUS(_6) (Inconsistency of the System)</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1.71</td>
</tr>
<tr>
<td>SUS(_7) (Learnability)</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4.86</td>
</tr>
<tr>
<td>SUS(_8) (Cumbersome Use)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.00</td>
</tr>
<tr>
<td>SUS(_9) (Confidence)</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4.71</td>
</tr>
<tr>
<td>SUS(_{10}) (Need to Learn a Lot Before Use)</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.86</td>
</tr>
<tr>
<td>SUS Score</td>
<td>87.5</td>
<td>87.5</td>
<td>87.5</td>
<td>82.5</td>
<td>90.0</td>
<td>87.5</td>
<td>95</td>
<td>88.2</td>
</tr>
<tr>
<td>Grade</td>
<td>A+</td>
<td>A+</td>
<td>A+</td>
<td>A</td>
<td>A+</td>
<td>A+</td>
<td>A+</td>
<td>A+</td>
</tr>
</tbody>
</table>

Table 6.7: SUS scores of all participants

the participants for this specific item, P4 justified that score by mentioning that it was their first time using VS Code. Being that the extension is integrated into the functionality of VS Code, it makes sense that the participant had to learn relatively many things before they were able to use the extension. Therefore, this outlier should not be taken too much at face value as it was influenced by an external factor.

Another interesting aspect to point out is that the results of the SUS seem to contradict the findings discovered from analyzing the statements of the participants: while the statements suggest that the participants had a hard time understanding several aspects of the extension, the SUS scores show no sign regarding the seemingly low understandability of the extension. The participants felt that they would not need the support of a technical person\(^{10}\) (covered by SUS\(_4\)) and everything other

\(^{10}\)In this case, a person who is familiar with the extension
than cumbersome to use (covered by $SU_9$). A potential explanation for this is that the negative statements regarding the usability of the extension go into so much detail that they do not have a significant influence on the overall perceived usability.

### 6.3.5 Summary

All in all, the extension left a positive impression on the participants regarding its usability and functionality. Based on the evaluation data, key characteristics of the extension include its ease of use, its pleasing UI, and its well-thought-out features that can be used without needing a comprehensive introduction to the extension.

What the extension lacks in some places is understandability: the extension needs a precise way of conveying its intended use and directing the user to their goals across all features.

And with that, the milestone for the Analysis Phase had been reached, and thus, the study process had come to an end.
7 Discussion

This chapter discusses the results of the study with regard to the initial objective of this thesis. Furthermore, some aspects that may impact the validity of this study will be pointed out.

7.1 Usability of the ADR Manager as an IDE Plugin

For this thesis, the objective was to create an IDE plugin based on the web-based ADR Manager. The plugin should allow users to document and manage ADRs in a way that does not intervene with the common workflow of an IDE. Moreover, users should perceive the plugin as enjoyable and convenient to use.

Looking at the results of the evaluation, one can argue that the objective of this thesis has been met. The above-average SUS scores hint at a great overall user experience that fits well in the context of an IDE. With the participants commending the manageable and adequate features, the extension also managed to avoid sacrifices regarding its core functionality. In my opinion, the development and evaluation of the ADR Manager VS Code extension was a success.

Nevertheless, I also want to address some observations that I noticed during the study process in hopes of providing some food for thought for other authors of IDE plugins to take away from this thesis:

1. The perceived usability of an IDE plugin is closely tied to the proficiency in using the IDE itself.

Because IDE plugins build upon the base functionality of the underlying IDE, the user still requires knowledge about working with the IDE in itself. The goal of an IDE plugin is to enhance the existing workflow of the IDE: even if it is integrated beautifully into the IDE, a plugin still loses the majority of its value if the user has not established a workflow in the first place.

When looking at the ADR Manager extension, for instance, one major design choice for efficiently accessing the extension features is using the command palette of VS Code. If the user is not used to commands\(^1\), then the whole rationale behind that design choice will hold less value. The extension combats this issue by providing access to features in common ways, like by right-clicking on a file or a directory (e.g., for opening the webviews of the extension), or by using the settings menu to configure a system (e.g., for changing the ADR Directory).

\(^1\)or even worse: if the user does not want to use commands
2. **It is hard to satisfy every type of user with a single IDE plugin.**

Due to the already large amount of features a modern IDE offers out-of-the-box, a user often has multiple ways to achieve a specific goal. This creates a variety of user groups that have their unique preferences and workflows. In a context where time is limited, trade-offs have to be made between the different user groups during the development process of the IDE plugin. This may lead to a non-marginal variance in the perceived usability between users with varying preferences, even when the underlying features and requirements remain the same. Focusing on a specific target audience may help narrow the scope of the IDE plugin and improve the perceived usability for that specific user group.

For the ADR Manager extension, two major user groups emerged during the early stages of development. One group preferred using the form-based GUI while the other group was more comfortable with using the raw Markdown editor. This would lead to a large variety of features that had to be implemented completely separate from each other even though they would still share the same overall context of VS Code. Given a limited time frame (like for this thesis), it would have been too much work to implement features for both the form-based GUI and the Markdown editor such that both parties would be adequately satisfied. For the extension, I decided to focus on the features of the GUI since the extension is ultimately based on the web-based ADR Manager which also provides its features using form-based editors instead of raw Markdown.

3. **Just because you can implement a feature, it does not mean you should implement it.**

When developing an IDE plugin, it might be tempting to implement as many features as possible, such that more use cases are covered by the plugin. Offering a large number of features may harm perceived usability because of the risk of overloading the user with too much information. Having many features also comes with the problem of only implementing shallow functionality that feels out of place or is not even wanted by the user. Identifying and expanding on the key features of an IDE plugin helps in retaining high-quality features and satisfied users.

An example from the ADR Manager is the feature for only treating Markdown files as ADRs if they exactly follow the naming convention of MADR\(^2\). While this feature helps to enforce the MADR format, some users want to have some degree of freedom when using ADRs to document their decisions. Forcing them to work in a certain way would hurt the perceived usability of the extension. While the naming restriction for ADR files has been loosened once\(^3\), it is still a topic of discussion for the ADR Manager extension.

---

\(^2\) i.e., `NNNN-lowercased-with-dashes-md`

\(^3\) ADRs may now contain uppercase and lowercase letters, and they may be written in kebab-case, snake_case, or a combination of these two cases.
7.2 Threats to Validity

The following section mentions some factors that may diminish the validity of this study.

7.2.1 Internal Validity

According to Wohlin et al., internal validity deals “with factors that may affect the dependent variables without the researcher’s knowledge” [WHH03]. In this case, the perceived usability and the functional suitability of the extension may be influenced by the following factors:

Task Choice

As the tasks for the evaluation were created by myself, there might be a possibility that the tasks were unknowingly constructed in such a way that it would be hard for the extension to perform poorly. Because of the nature of the constructed tasks, the participants may have never had the chance to reveal the shortcomings of the extension during the evaluation. This may nudge the results regarding the perceived usability in favor of the extension.

Even though I cannot deny the possibility of this affecting the results, the tasks still showcased the key features of the extension, which made an overall positive impression on the participants.

Experience with MADR and VS Code

Going into the evaluation, four out of the seven participants already had several years of experience with the MADR format. Knowledge about MADR may reduce the workload of the given tasks as the concept and the practice behind documenting ADRs in that particular format already seems familiar. This may then lead to higher perceived usability of the extension which could be reflected by higher SUS scores, for example for SUS items regarding learnability (SUS7), confidence (SUS9), or the need to learn things beforehand (SUS10).

Analogously, the experience in the use of VS Code may also play a role when determining the perceived usability of the extension. The item scores for SUS10 (“I needed to learn a lot of things before I could get going with this system.”) would also hint at the connection between the perceived usability and the experience in the tools used by the extension.

History of the Participants

Taking a look at Table 6.3, almost every participant from this evaluation study also took part in the previous study with the original ADR Manager. Because the extension is mostly based on the web-based tool, there is a chance that the participants did not need to put in as much effort in familiarizing themselves with the extension. In turn, this may have had a positive impact on the ease of use or the learnability of the extension, both of which play a considerable role when evaluating the usability of the extension. The results may not have looked as good if the participants did not have any prior experience with the ADR Manager.
It should also be noted that the idea of developing a VS Code extension itself was suggested by many participants of the original ADR Manager study. This may have also led to a biased evaluation of the extension seeing that the extension was their own request.

Having participants from the original ADR Manager study also has the advantage that the participants can make an informed comparison between the web-based version and the VS Code version of the tool. And seeing that the participants generally seemed to prefer using the VS Code extension over the web-based version, it would speak in favor of a successful adaptation of the web tool to the IDE context.

**Hawthorne Effect and Social-desirability Bias**

As the participants were under observation while evaluating the extension, they might have been affected by the Hawthorne effect, where participants change their natural way of acting when being observed, or the social-desirability bias, where participants answer surveys in a way that makes them look good in the eyes of others. Both of these effects lead to a distorted response that does not relate to the “real” thoughts of a participant.

While the presence of these effects cannot be excluded, I believe that these factors did not have a big impact on the results presented in this thesis. This may be hinted at by the fact that the participants openly voiced their criticism regarding several aspects of the extension. Also, I am confident that the topic of this evaluation was not sensitive to the point where these effects would have massively impacted the results of this study.

### 7.2.2 External Validity

Wohlin et al. describe external validity to be “related to the ability to generalise the results of the experiments” [WHH03]. The following points may represent a threat to the external validity of this study:

**Participant Sample**

In total, this evaluation was conducted with seven software professionals. This number is too small for the results to be representative of all people working in software or software-related fields. Even though participants came from both industry and academia and had different roles and levels of experience, their heterogeneity was limited, as they were recruited based on recommendations and partly out of convenience, having contacts already established from the original ADR Manager study.

Nevertheless, I believe the sample is adequate for a first study with the extension. An empirical study on a larger scale would be needed to confirm the results from this study.
7.2 Threats to Validity

Artificial Scenario

The participants were asked to solve tasks that were based on an entirely fictional scenario. There is a chance that the detailed circumstances of the artificial scenario do not accurately reflect real life, thereby limiting the generalizability of the results of this study. For illustration, meeting summaries may not be nearly as detailed and organized as they were in this controlled study environment. This issue arose from the personal dichotomy of wanting to make the tasks more interesting and tangible for the participants, but at the same time lacking the professional experience to derive a scenario that could be realistic. A study where participants would use the extension in their day-to-day life would result in data that may be more generalizable.

Generalization Outside of MADR and VS Code

In my opinion, the findings of this study are not limited to only the MADR format or VS Code. The observations of this study are aimed at the general overarching concepts of ADRs and IDEs, and thus may be applied to other ADR formats or IDEs, as long as they are grounded by the same principles as MADR and VS Code.
8 Conclusion

This chapter summarizes the thesis and gives pointers for future work on this topic.

8.1 Summary

This thesis explored the question of whether ADR tooling could be integrated into an IDE such that the user benefits from improvements regarding the usability of the ADR tool. It described the process of developing and evaluating a plugin for the VS Code IDE that allows the user to conveniently document and manage ADRs in the MADR format.

First, the requirements for the VS Code extension were elicited and validated by a focus group. Then, a UI prototype of the extension was created, which was used as a reference for the actual implementation of the VS Code extension using the VS Code API. After reaching a mature state, the extension was evaluated by seven software professionals in terms of its perceived usability and functional suitability.

The evaluation results indicated an overall positive perception. Among other things, the extension was praised for being easy to use, quick to learn, and having well-thought-out features that fit well in the context of an IDE. The usability of the extension was perceived to be above average, achieving an average SUS score of 88.2 across all seven participants.

On the other hand, the biggest criticism of the extension was directed at the understandability of the extension. Participants sometimes wished for more conciseness when it comes to the extension guiding the user to the goal they want to achieve.

8.2 Outlook

As the ADR Manager VS Code extension is still in a prototypical stage, the current features may be improved in the near future. The feedback from the evaluation can thereby be used as a general guideline and prioritization for the order of implementation.

Another option worth exploring is to expand the functionality of the extension such that it appeals to more users. In its current form, the extension heavily focuses on features that are solely accessible through the GUI that is rendered by the webview of the extension. Adding new features or extending the current features for the raw Markdown editor may help with attracting users who prefer to write raw Markdown.

A big topic for future work is adding support for other ADR formats. Either updating the extension to support the newest version of MADR or supporting new formats like Y-statements would be feasible in the future of the extension.
Bibliography


All links were last followed on September 19, 2022.
## Appendix

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Adding new ADRs to the local repository</td>
<td>The user can add new ADRs to a GitHub repository that will be saved via the UI. The user can fill out form fields, upon which the ADR Manager will create an ADR. The created ADRs will be saved in a Markdown file using a predetermined template (MADR). If there are multiple folders/repositories opened in the workspace, the extension will ask the user in which folder/repository he wants to save the new ADR.</td>
<td>functional</td>
</tr>
<tr>
<td>2</td>
<td>Editing existing ADRs</td>
<td>The user can edit fields of existing ADRs in a (GitHub) repository that are saved in docs/decisions via the UI. More specifically, the user can edit form fields and even rearrange options based on their priority.</td>
<td>functional</td>
</tr>
<tr>
<td>3</td>
<td>Provide MADR template</td>
<td>The user can add and edit an ADR by using the predefined MADR template provided by the UI. The user only has to fill out required fields instead of writing raw markdown.</td>
<td>functional</td>
</tr>
<tr>
<td>4</td>
<td>Provide an ADR parser and converter</td>
<td>The user can convert existing ADRs that don’t fit the MADR template such that it fits the MADR template. The UI should show the resulting ADR after the conversion and show differences and areas of interest in the ADR that will be converted in order to save potentially important data.</td>
<td>functional</td>
</tr>
<tr>
<td>5</td>
<td>Provide basic linting for ADRs</td>
<td>The user can see whether an editor form has been filled out correctly. If there are errors, then the UI will let the user know.</td>
<td>functional</td>
</tr>
<tr>
<td>6</td>
<td>Automatically detect ADRs in opened folders</td>
<td>The user can see existing ADRs saved in the folders that have been opened in the workspace in the UI, from which he can choose to edit or delete them. If the user opens multiple folders/repositories at once (multi-root workspace), the extension will try to detect ADRs in each of the opened folders.</td>
<td>functional</td>
</tr>
</tbody>
</table>

continued on next page...
<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Provide Basic Editor Mode &amp; Professional Editor Mode</td>
<td>The user can switch between Basic Editor Mode and Professional Editor Mode via the UI. In Basic Editor Mode, only required fields are shown (i.e., title, context and problem statement, considered options, decision outcome), while in Professional Editor Mode, optional fields can also be filled out (i.e., last update date, status, deciders, technical story, decision drivers, description and pros/cons of options, positive and negative consequences, links)</td>
<td>functional</td>
</tr>
<tr>
<td>8</td>
<td>Delete existing ADRs</td>
<td>The user can delete existing ADRs from a (GitHub) repository that are saved via the UI. Alternatively, he may delete ADRs directly by using the IDEs feature of deleting files.</td>
<td>functional</td>
</tr>
<tr>
<td>16</td>
<td>Supporting images in an ADR</td>
<td>The user should be able to choose pictures in png/jpeg format to embed while creating or editing an ADR.</td>
<td>functional</td>
</tr>
<tr>
<td>18</td>
<td>Provide VS Code commands</td>
<td>The user should be able to use several VS Code commands to quickly use the extension’s functionality.</td>
<td>functional</td>
</tr>
<tr>
<td>19</td>
<td>Provide an option to change the location where to save ADRs</td>
<td>The user should be able to customize the path where the extension will save newly created ADRs and where the extension will scan for already existing ADRs. This may be done by using a VS Code command. There should be a default path if the user has not set their preference.</td>
<td>functional</td>
</tr>
<tr>
<td>20</td>
<td>Provide quick access to the extension’s functionality</td>
<td>The user should be able to use the extension’s functionality within 3 user actions. Examples for user actions are using a VS Code command, filling out an input field, a click on a webview (custom rendered HTML content) etc.</td>
<td>non-functional (usability)</td>
</tr>
</tbody>
</table>

**Table A.1:** List of validated requirements for the extension
Kurzfassung

Die Softwarearchitektur wird weitgehend durch Entscheidungen geprägt, die während des Lebenszyklus der Softwareentwicklung getroffen werden. Diese Architekturrentscheidungen enthalten Informationen über die zugrundeliegende Architektur, die für die Schaffung einer einheitlichen Sicht auf die Architektur für alle Beteiligten des Softwareprojekts von Bedeutung sind. Die Vernachlässigung der Dokumentation von Architekturrentscheidungen führt zu einer Vielzahl von Problemen bei der Wartung der Software.

Eine Möglichkeit, Architekturrentscheidungen zusammen mit ihrer Begründung festzuhalten, ist die Verwendung von so genannten Architectural Decision Records (ADRs). Werkzeuge zur Verwaltung von ADRs gibt es bereits, aber sie arbeiten oft in einem Kontext, der für einen Softwareentwickler ungewohnt ist oder ein zusätzliches Werkzeug für die Verwendung benötigt. In dieser Arbeit wird die Möglichkeit untersucht, ADR-Werkzeuge in Form eines IDE-Plugins bereitzustellen, das Entwicklern ADR-Verwaltungsfunktionen auf bequeme Weise zugänglich macht.


In einer empirischen Nutzerstudie mit sieben Softwareexperten erhielt die ADR-Manager-Erweiterung von den Evaluationsteilnehmern ein herausragendes Feedback hinsichtlich ihrer Benutzerfreundlichkeit im Kontext der Visual Studio Code IDE. Die Teilnehmer lobten die einfach, aber effektive Funktionalität der Erweiterung und gaben an, dass sie sich gut in die IDE-Umgebung einfügt. Auf der anderen Seite wünschten sie sich, dass die Erweiterung manchmal präziser kommunizieren würde, welche Eingaben dem Benutzer zur Verfügung stehen und wie diese eingegeben werden sollten, sodass dieser sein gewünschtes Ergebnis bekommt.

Da es sich noch um einen Prototyp handelt, gibt es viele Möglichkeiten, wie die Erweiterung in Zukunft verbessert werden kann. Beispiele hierfür sind die Verbesserung der bestehenden Funktionalität, die Implementierung neuer Funktionen, um mehr Benutzer anzusprechen, oder die Implementierung für die Unterstützung weiterer ADR-Formate.
Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature