Vergleich der Extension-APIs in Visual Studio Code und IntelliJ IDEA

Philipp Seiringer

BACHELORARBEIT

eingereicht am

Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Februar 2024

Betreuung:

Dr. Josef Pichler

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt. Die vorliegende, gedruckte Arbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Hagenberg, am 1. Februar 2024

Philipp Seiringer

Inhaltsverzeichnis

Kurzfassung

Abstract

This should be a 1-page (maximum) summary of your work in English.

Kapitel 1

Einleitung

1.1 Motivation

SoftwareentwicklerInnen arbeiten täglich mit verschiedensten Werkzeugen und Entwicklungsumgebungen, sogenannten IDEs (=Integrated Development Environment). Diese Plattformen bieten teils sehr unterschiedliche Funktionalitäten, die die Softwareentwicklung erleichtern sollen. Dabei bieten sie Unterstützung für verschiedenste Programmiersprachen und Technologien und binden zahlreiche Werkzeuge für spezifische Anwendungsfälle ein. Aufgrund des immer rascher werdenden Entstehens von neuen Technologien bieten mehr und mehr IDEs Möglichkeiten zur Entwicklung von eigenen Plugins, welche dann auch an andere EntwicklerInnen bereitgestellt werden können. So können in kürzester Zeit neue Technologien unterstützt werden und EntwicklerInnen haben selbst die Macht darüber zu entscheiden welche Plugins sie nutzen möchten und welche nicht.

Vor der Entwicklung solcher Plugins ist es wichtig zu entscheiden für welche IDE das Plugin erstellt werden soll. Dabei spielen Aspekte wie zum Beispiel die Einfachheit und Flexibilität in der Entwicklung, der Umfang an angebotener Funktionalität, die Möglichkeit die Nutzerinteraktion und somit die User Experience zu steuern und viele weitere eine Rolle. Diese Bachelorarbeit versucht in diesen Bereichen einen Überblick zu schaffen und vergleicht hierfür die Plugin Entwicklung in zwei der momentan beliebtesten IDEs, Visual Studio Code und IntelliJ IDEA. Durch den Vergleich der beiden Produkte und dem Herausarbeiten und Aufbereiten der Unterschiede wird es anderen EntwicklerInnen erleichtert diese Entscheidung zu treffen.

1.2 Ziel

1.3 Aufbau

Kapitel 2

Grundlagen der Plugin Entwicklung

2.1 Entwicklungsumgebungen

2.1.1 Visual Studio Code

Die erste offizielle Version von Visual Studio Code, häufig abgekürzt auch als VS Code, wurde im April 2016 [17] von Microsoft veröffentlicht. Die Idee hinter VS Code war einen möglichst einfachen Code Editor anzubieten, welcher nur die wichtigsten und besten Funktionen für EntwicklerInnen beinhaltete. Es hob sich somit von anderen IDEs wie der Visual Studio Reihe von Microsoft ab, da es ein sehr leichtgewichtiger Editor war, welcher trotzdem mit einer großen Menge an Programmiersprachen arbeiten konnte und für diese auch Microsofts code completion namens „IntelliSense“ unterstützte. Weiters war Visual Studio Code das erste Produkt der Visual Studio Familie welches Cross-Plattform für Windows, Linux und OSX angeboten wurde [19].

Aus den Stack Overflow developer surveys der vergangenen Jahre kann der rasche Aufstieg von VS Code beobachtet werden. Während es im Jahr der Veröffentlichung nur von etwa 7,2 Prozent der EntwicklerInnen genutzt wurde, war es zwei Jahre später bereits (wenn auch knapp) das meistgenutzte IDE mit 34,9 %. In der aktuellsten Umfrage von 2023 war es der klare Sieger und wurde vom 73,71% der Abstimmenden aktiv genutzt[14, 15].

Ein Grund für diesen Erfolg mag vermutlich die Möglichkeit zur Entwicklung und zum Anbieten von Plugins sein. Durch die direkte Einbindung es Visual Studio Marketplace in VS Code bildete sich über die Jahre eine große Community die eine enorme Anzahl von Plugins entwickelt, verbessert und betreut. Durch solche, meist community-erstellte, Plugins kann VS Code auch eine enorme Anzahl von Programmiersprachen unterstützen.

2.1.2 IntelliJ IDEA

IntelliJ IDEA wurde erstmals im Januar 2001 [7, 8] von dem Unternehmen JetBrains veröffentlicht. Im Gegensatz zu Visual Studio Code handelt es sich bei IntelliJ um ein vollausgetattetes „Integrate Development Environment“ (IDE) welches speziell auf die Entwicklung von Programmen in den Programmiersprachen Java, Kotlin und Groovy ausgelegt ist. IntelliJ IDEA wird in einer frei zu verwendenden, open source „Community

Edition“, sowie in einer kommerziellen Form als „IntelliJ IDEA Ultimate“ angeboten [1].

Aufgrund der Spezialisierung auf JVM kompatible Sprachen unterstützt die IntelliJ Community Edition nur eine relativ kleine Auswahl an Sprachen, Frameworks und Build Tools. Während IntelliJ IDEA Ultimate den Umfang an Features schon deutlich erweitert, bietet JetBrains auch noch weitere (kommerzielle) IDEs an. Diese sind alle für unterschiedliche Programmiersprachen oder Sprachfamilien ausgelegt. Einige der bekanntesten sind dabei CLion für die Sprachen C und C++, Rider für die .NET Sprachen, PhpStorm für PHP, WebStorm für JavaScript und viele weitere. Zum aktuellen Zeitpunkt sind es insgesamt elf verschiedene IDEs die von JetBrains angeboten werden und die alle auf der IntelliJ Platform basieren. Das bedeutet nicht nur, dass sich all diese IDEs in der Verwendung und im Aussehen sehr ähnlich sind, sondern auch, dass ein Plugin, welches für die allgemeine IntelliJ Platform entworfen wurde, relativ problemlos auch für mehrere IDEs dieser Form veröffentlicht werden kann [10].

Im Gegensatz zu Visual Studio Code ist IntelliJ ein eher schwergewichtiger Editor, der sehr viel Funktionalität schon von Grund auf eingebaut hat. Die EntwicklerInnen sind hier nicht so stark auf Plugins angewiesen. Dies lässt sich auch durch die Anzahl von Plugins erkennen, die auf dem JetBrains Marketplace angeboten werden. Für die IntelliJ Platform gibt es aktuell etwas über 7500 Plugins die in die IDE integriert werden können [9]. Für Visual Studio Code sind es hingegen inzwischen über 51000 Plugins [18].

2.2 Programmiersprachen

2.2.1 TypeScript

Die TypeScript Programmiersprache wurde erstmalig am 1. Oktober 2012 [5] von Microsoft in Form eines open-source Projekts veröffentlicht. Designed wurde sie von Anders Hejlsberg, der auch an der Entwicklung von C# beteiligt war.

Die grundsätzliche Idee der Sprache ist, eine typsichere, kompilierte, und somit bessere Version von JavaScript zu sein. JavaScript ist aufgrund des Erfolgszugs des Internets zu einer sehr wichtigen Sprache geworden und war auch schon 2012 aus den TOP Listen für Programmiersprachen nicht mehr wegzudenken [12, 15, 16]. Webseiten setzen heute sehr stark auf JavaScript, um durch interaktive Elemente die User Experience zu verbessern oder um neue Funktionalität anbieten zu können. Durch das Node.js runtime environment kann JavaScript nicht mehr nur im Browser verwendet werden, sondern es können auch Desktop, Server oder Mobile Anwendungen in JavaScript entwickelt werden. Durch diesen großen Umfang an Möglichkeiten die JavaScript dadurch bietet werden natürlich auch immer größere Projekte damit entwickelt. Und hier kommen die großen Schwächen von JavaScript immer mehr zu tragen. Je größer die Projekte werden und je mehr EntwicklerInnen an einem Projekt mitarbeiten, desto mehr Fehler entstehen aufgrund der fehlenden Typsicherheit und des fehlenden Compilerschrittes. Diese Schwachstellen versucht TypeScript nun auszubessern.

TypeScript code wird mithilfe des TypeScript Compilers „tsc“ in einfache JavaScript Dateien transpiliert. Dadurch kann auf die Popularität und Verbreitung von JavaScript aufgebaut werden und TypeScript ist überall dort verwendbar, wo JavaScript ausführbar ist. Weiters ist TypeScript ist ein Superset von JavaScript. Es gilt also: „Any valid .js

file can be renamed .ts and be compiled with other TypeScript file." [2].

Jedoch bietet TypeScript eine Menge von Vorteilen gegenüber ihrer Basissprache.

- Durch den Kompilierschritt mit dem tsc Compiler wird der Code vor der Ausführung automatisch auf Validität geprüft. Es entfällt also die Notwendigkeit für einen zusätzliches Linting Tool wie JSLint. Dieser Compile-Schritt kann natürlich auch in eine CI/CD Pipeline eingebunden werden, um auch bei Merges Feedback über die Validität des Codes zu erhalten.
- Durch die statische Typisierung können Missverständnisse über die Verwendung von Variablen vermieden werden. Auch die Unterstützung durch verschiedene IDEs, zum Beispiel mittels IntelliSense kann durch die Typen verbessert werden. Dies ist nicht nur bei der Zusammenarbeit hilfreich, sondern kann auch die Arbeit jeder einzelnen EntwicklerIn beschleunigen.
- In TypeScript können Klassen erstellt werden, deren Properties mit Zugriffsmodifikatoren (private/public) versehen sind.
- TypeScript unterstützt Vererbung, Interfaces und generische Programmierung.
- In TypeScript können bereits bestehende JavaScript Bibliotheken wiederverwendet werden. Weiters ist es möglich durch zusätzliche Dateien Typinformationen zu den bestehenden Bibliotheken zu liefern.

2.2.2 Java

Die Entwicklung der Programmiersprache Java begann im Jahr 1991 und sie wurde von den James Gosling, Mike Sheridan und Patrick Naughton designed [4]. Java wurde erstmals im Jahr 1995 von Sun Microsystems veröffentlicht. Im Januar 2010 wurde Sun Microsystems dann von der Oracle Corporation übernommen, welche seitdem auch Java weiterentwickelt.

Das Design und vor Allem die Syntax der Sprache war stark von C und C++ inspiriert [3], um anderen Entwicklern einen leichten Umstieg auf das neue Java zu ermöglichen. Allerdings versuchte Java die teils sehr komplexen (wenn auch effektiven) Sprachfeatures von C++ etwas zu vereinfachen. Java sollte eine simple, objektorientierte und robuste Sprache werden. Die Funktionalität die Java zu dem großen Erfolg verhalf, den sie später hatte, war das

> "write once, run anywhere"

(WORA) Prinzip, wie Sharan und Davis beschreiben [3]. Im Gegensatz zu den zuvor gängigen Programmiersprachen muss Java nämlich für bestimmte Hardwarearchitekturen kompiliert werden. Java Programme werden zu einer Art Zwischensprache, dem sogenannten Java Bytecode kompiliert. Dieser Bytecode kann dann von einer Java Virtual Machine (JVM) ausgeführt werden. Diese JVM ist im Grunde ein eigenständiges Programm welche mit dem Java Runtime Environment (JRE) mitgeliefert wird. Ein einmal kompiliertes Java Programm kann also auf allen Geräten ausgeführt werden, auf denen ein passendes JRE installiert ist. So ist es zum Beispiel auch möglich Java für die Entwicklung von Android nativen Apps auf Mobilgeräten zu benutzen.

Ein weiterer Vorteil gegenüber älteren Sprachen wie C++ ist die automatisierte Speicherverwaltung. Diese funktioniert mithilfe eines sogenannten „garbage collectors" welcher nicht mehr benötigten Speicher am Heap bereinigt und freigibt. Man kann also

beliebig neue Objekte im Speicher allokieren und muss sich nicht um die deallokierung der zuvor erstellten Objekte kümmern. Auf diese Weise können häufige Programmierfehler wie Memory Leaks fast vollständig unterbunden werden.

Java unterstützt sowohl das objektorientierte, das prozedurale als auch das funktionale Programmierparadigma. Der Fokus liegt allerdings stark auf der Objektorientierung. Dabei bietet Java Funktionalitäten zur Abstraktion durch Verwendung von Klassen, Information Hiding mithilfe von Zugriffsmodifikatoren (public/private/protected/package), Vererbung, Interfaces, Polymorphismus, Überladen von Methoden, generischer Programmierung, Exception Handling und vieles mehr.

2.3 Aufbau der Plugin API

2.3.1 Visual Studio Code

Visual Studio Code bietet für Plugins zwei Arten der Interaktion, welche zusammenspielen um Plugins zu ermöglichen. Das Extension Manifest und die eigentliche API.

Extension Manifest

Das Extension Manifest befindet sich in der „package.json“ Datei. In dieser werden statische Einstellungen vorgenommen und Metainformationen über das Plugin bekannt gegeben. So kann hier unter anderem Name, Beschreibung, Herausgeber, Lizenzvereinbarungen und so weiter eingestellt werden. Weiters definiert das Manifest eine sogenannte „main“ JavaScript oder TypeScript Datei und dazu passende „Activation Events“ und „Contribution Points“.

Activation Events bestimmen den Zeitpunkt an dem das Plugin zum ersten Mal aktiviert wird. Dabei wird die „activate“ Funktion der zuvor definierten main Datei ausgeführt. Der Aktivierungszeitpunkt sollte immer so spät wie möglich gewählt werden, um VS Code möglichst wenig zu verlangsamen und das Plugin erst on demand zu Laden. Allerdings muss die Aktivierung natürlich passieren bevor die erste Funktionalität des Plugins erwartet wird. Typische Aktivierungsevents sind zum Beispiel „onCommand“ , „onDebug“, „onView“ oder „onStartupFinished“. Wurde das Plugin einmal aktiv, bleibt es auch aktiv bis VS Code wieder geschlossen wird oder das Plugin entfernt oder deaktiviert wird. Hierfür gibt es optional noch eine „deactivate“ Funktion in der main Datei, welche für etwaige Aufräumarbeiten genutzt werden kann.

Contribution Points legen fest welche Funktionalität das Plugin anbietet und somit auch welche zusätzlichen Elemente dem Nutzer in VS Code angezeigt werden sollen. Hier ist es beispielsweise möglich Visual Studio Code mit neuen Befehlen („Commands“), Menüs und Submenüs, Views für das Anzeigen von Plugindefiniertem Content, Keyboard Shortcuts, Unterstützung für neue Sprache, und vieles mehr auszustatten.

Visual Studio Code API

Die eigentliche VS Code API kann im TypeScript Code (sowohl in der main, als auch in anderen Dateien) genutzt werden. Hierfür wird einfach das „vscode“ Modul importiert. Dieses beinhaltet eine vollständige definition der angebotenen Schnittstelle, auf welche programmatisch zugegriffen werden kann.

```
1    import * as vscode from 'vscode';
2
3    export function activate(context: vscode.ExtensionContext) {
4      vscode.window.showInformationMessage('Hello World!');
5    }
6
```

Über diese API kann dann zum Beispiel festgelegt werden, durch welchen Code die zuvor definierten Contribution Points implementiert werden sollen. Der Plugin Code wird in Visual Studio Code nicht im selben Prozess wie das Hauptprogramm ausgeführt, sondern abgekapselt in einem seperaten „extension host process“. Dadurch kann verhindert werden, dass Plugins die Performance und die Interaktivität von VS Code negativ beinflussen [6, 11].

Ablauf

Visual Studio Code analysiert zuerst das Extension Manifest des Plugins. Je nachdem welche Activation Points definiert sind, wird zu einem bestimmten Zeipunkt die activate Funktion aufgerufen. In dieser können dann mithilfe der API Event Handler registriert werden. Die registrierten Handler werden dann während der Ausführung und Verwendung von Visual Studio Code aufgerufen und können so beliebigen Code ausführen. Siehe Abbildung 2.1.

2.3.2 IntelliJ IDEA

Der Aufbau der Plugin Architektur wirkt bei IntelliJ im ersten Moment genau gleich wie bei Visual Studio Code. Es gibt nämlich auch hier gibt es ein Plugin Configuration File, sowie ein Modul mit API Schnittstellen. Der große Unterschied liegt allerdings in der Funktionsweise und der Interaktion mit den Plugins und der Art wie der auszuführende Code angegeben wird.

Plugin Configuration File

Die Konfiguration eines Plugins liegt in der „plugin.xml“ Datei und beinhaltet, equivalent zum Extension Manifest in VS Code, alle für das Plugin notwendigen Meta-Informationen. So können auch hier Werte wie der Name, eine Beschreibung, die aktuelle Versionsnummer und so weiter angegeben werden. Für die Funktionen die das Plugin mitbringt gibt es Actions, Extension Points und Listener. Hier ist anzumerken, dass es sich sowohl bei den Extension Points, als auch den Listenern, immer direkt um eine Zuordnung eines Interfaces (meist definiert von IntelliJ) zu einer Implementierung (definiert durch das Plugin) handelt. Weiters ist es nicht nötig einen speziellen Aktivierungszeitpunkt festzulegen, da die zuordnung der auszuführenden Klassen sowieso durch die Konfigurationsdatei festgelegt wird. Eine Besonderheit an IntelliJ ist, dass Plugins

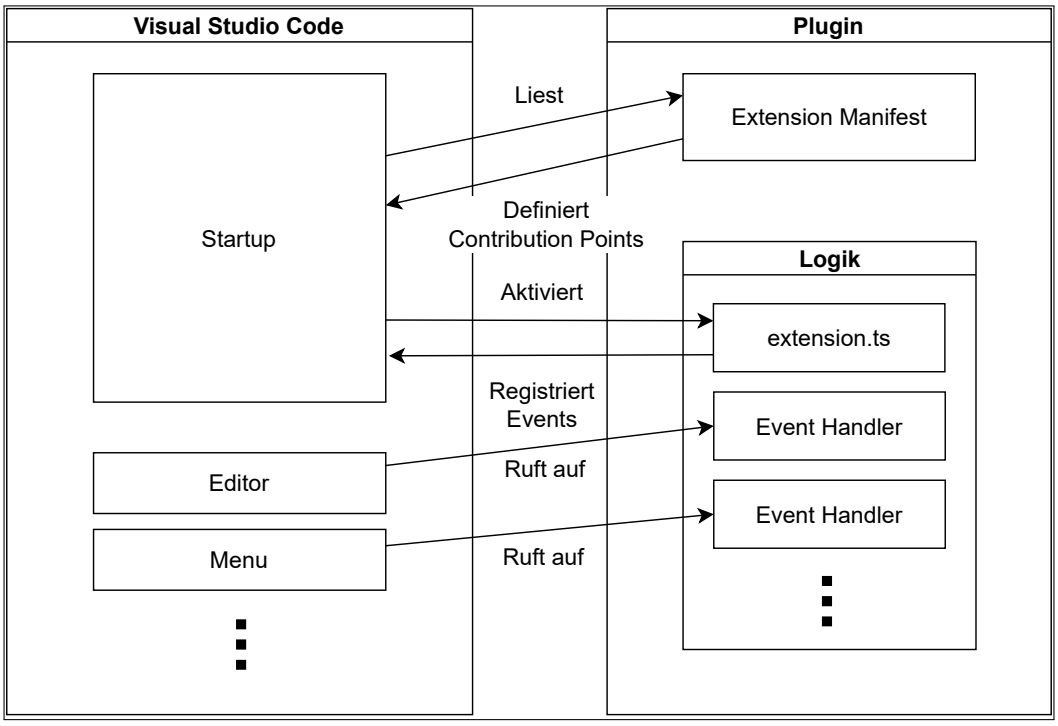


Abbildung 2.1: Übersicht über den Ablauf eines VS Code Plugins.

auch eigene Extension Points definieren können, um weiteren Plugins das erweitern des ursprünglichen Plugins zu erlauben.

IntelliJ Platform SDK

Die API für IntelliJ Plugins ist in mehreren Paketen des IntelliJ Platform SDK enthalten. Diese API enthält auch die unterschiedlichen Interfaces, welche dann in Form von Extension Points oder Listenern implementiert werden können.

Ablauf

IntelliJ analysiert zuerst das Plugin Configuration File. Je nachdem welche Funktionalität vom Plugin angeboten wird, werden von IntelliJ automatisch die entsprechenden Event Handler auf die unterschiedlichen Extension Points registriert. Die registrierten Handler werden dann während der Ausführung und Verwendung von IntelliJ aufgerufen und können so beliebigen Code ausführen. Siehe Abbildung 2.2.

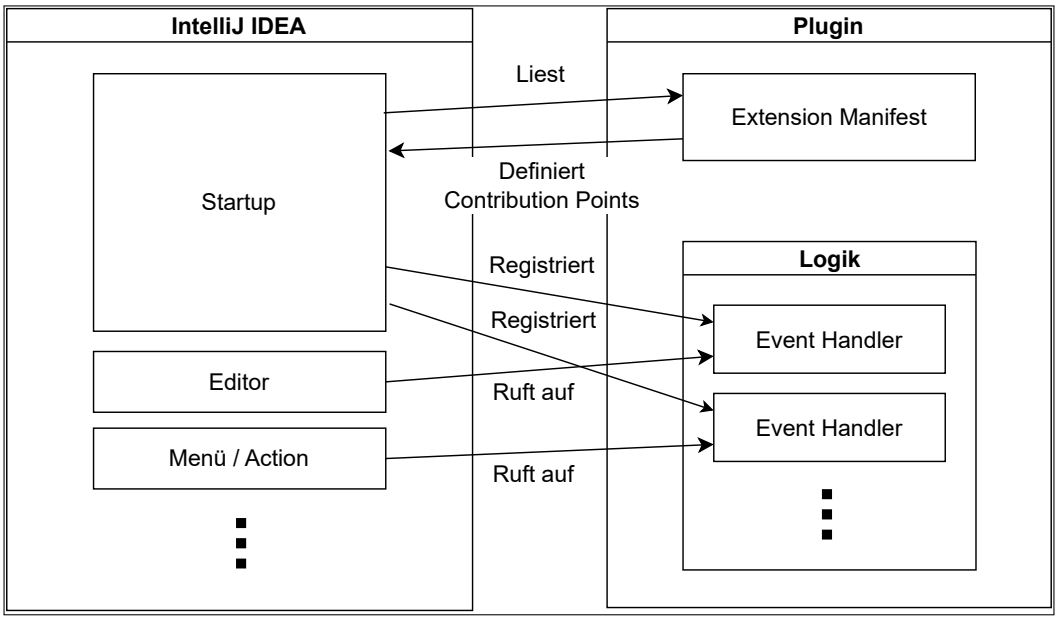


Abbildung 2.2: Übersicht über den Ablauf eines IntelliJ.

2.4 Funktionalität der Plugin API

2.4.1 Visual Studio Code

2.4.2 IntelliJ IDEA

2.4.3 IntelliJ Flora Plugins

In der Plugin Dokumentation von JetBrains wird zu Beginn empfohlen sich noch einmal gründlich zu überlegen, ob man für die von einem gewünschte Funktionalität wirklich ein vollwertiges Plugin benötigt. Häufig kommt es nämlich vor, dass nur bestimmte kleine Tasks innerhalb des IDEs automatisiert werden sollen [10]. Hierfür schlägt JetBrains einige leichtgewichtige Alternativen vor. Eine nennenswerte Alternative ist das „Flora Plugin“ für das IntelliJ IDEA.

Flora kann über die Einstellungen des IntelliJ IDEA im Abschnitt „Plugins“ installiert werden.

Das Plugin sucht dann in den geöffneten Projektverzeichnissen nach ausführbaren JavaScript oder Kotlin Script „micro plugin“ Dateien. Diese müssen sich in einem Ordner namens „.plugins“ befinden und auf „.plugin.js“ oder „.plugin.kts“ enden [13]. Innerhalb diese Plugin Dateien kann über die Variable „ide“ auf die angebotene Schnittstelle zugegriffen werden. Diese erlaubt es unter anderem Actions, Keyboard Shortcuts, Services und ToolWindows zu erstellen.

Flora Plugins bieten sich vor allem dann an, wenn eine projektspezifische Aufgabe automatisiert werden soll. Hier sind vor Allem die Leichtgewichtigkeit der Plugins und die Schnelle, mit der ein einfaches Plugin entwickelt werden kann, von großem Vorteil. Weiters spricht für diesen Anwendungsfall, dass der Plugin Code direkt im Projektordner abgelegt wird und somit auch in einem Version Control System wie Git mit abgelegt

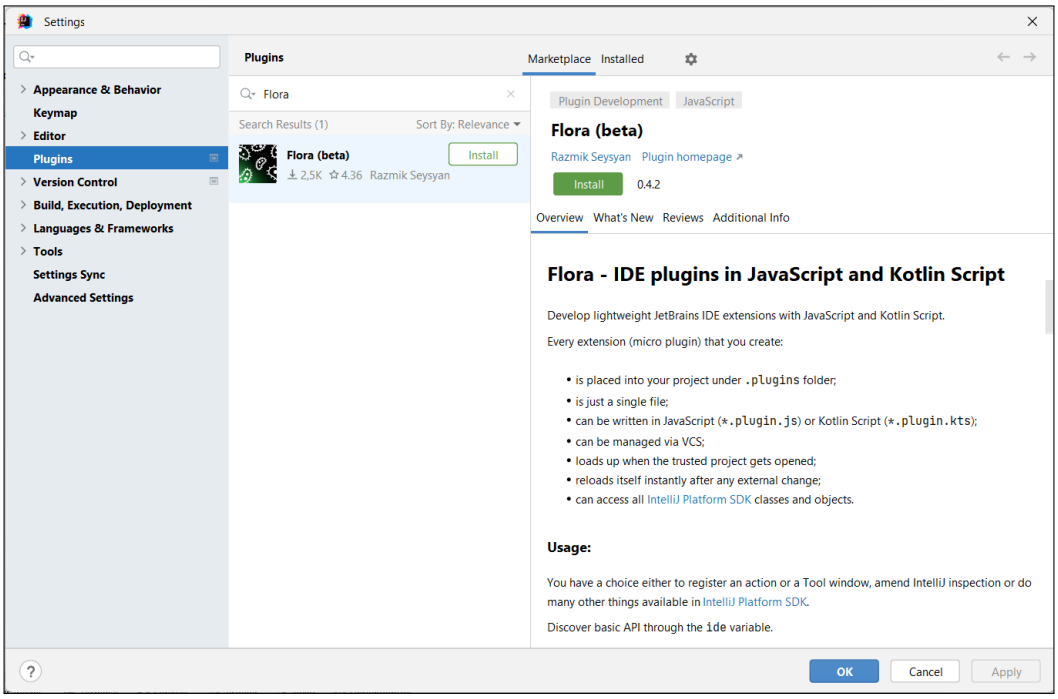


Abbildung 2.3: Flora Plugin im IntelliJ Plugin Marketplace.

werden kann.

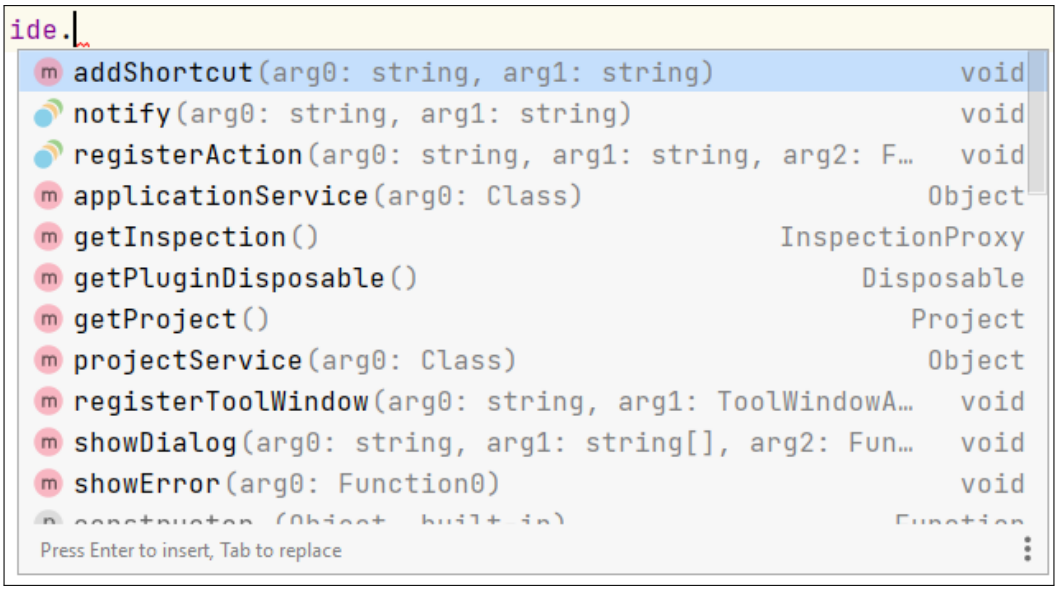


Abbildung 2.4: Übersicht über die API des Flora Plugins.

Kapitel 3

Anforderungen an den Prototyp

3.1 Aufbau

Kapitel 4

Entwicklung des Prototpys für Visual Studio Code

4.1 Design

4.2 Implementierung

4.2.1 Aufsetzen des Projektes

Aufbau der Ordnerstruktur

4.2.2 Entwicklung

4.3 Tests

4.4 Publishing

4.5 CI/CD

Kapitel 5

Entwicklung des Prototpys für IntelliJ

5.1 Design

5.2 Implementierung

5.2.1 Aufsetzen des Projektes

Aufbau der Ordnerstruktur

5.2.2 Entwicklung

5.3 Tests

5.4 Publishing

5.5 CI/CD

Kapitel 6

Bewertungskriterien

6.1 Popularität der Entwicklungsumgebung

6.1.1 Visual Studio Code

6.1.2 IntelliJ IDEA

6.2 Performance

6.2.1 Visual Studio Code

6.2.2 IntelliJ IDEA

6.3 Feature Umfang

6.3.1 Visual Studio Code

6.3.2 IntelliJ IDEA

6.4 Intuitivität der API

6.4.1 Visual Studio Code

6.4.2 IntelliJ IDEA

6.5 Dokumentation der API

6.5.1 Visual Studio Code

6.5.2 IntelliJ IDEA

6.6 Testbarkeit des Plugins

6.6.1 Visual Studio Code

6.6.2 IntelliJ IDEA

6.7 Möglichkeiten des Publishings

6.7.1 Visual Studio Code

6.7.2 IntelliJ IDEA

6.8 Installationsprozess des Plugins

Kapitel 7

Vergleich der Kriterien

7.1 Popularität der Entwicklungsumgebung

7.1.1 Visual Studio Code

7.1.2 IntelliJ IDEA

7.1.3 Vergleich

7.2 Performance

7.2.1 Visual Studio Code

7.2.2 IntelliJ IDEA

7.2.3 Vergleich

7.3 Feature Umfang

7.3.1 Visual Studio Code

7.3.2 IntelliJ IDEA

7.3.3 Vergleich

7.4 Intuitivität der API

7.4.1 Visual Studio Code

7.4.2 IntelliJ IDEA

7.4.3 Vergleich

7.5 Dokumentation der API

7.5.1 Visual Studio Code

7.5.2 IntelliJ IDEA

7.5.3 Vergleich

7.6 Testbarkeit des Plugins

7.6.1 Visual Studio Code

7.6.2 IntelliJ IDEA

Kapitel 8

Conclusion

Anhang A

Technische Informationen

Quellenverzeichnis

Literatur

[1] Ted Hagos. *Beginning IntelliJ IDEA : Integrated Development Environment for Java Programming*. eng. 1st ed. 2022.. 2022 (siehe S. 3).

[2] Dan Maharry. *TypeScript Revealed*. eng. Berkeley, CA: Apress (siehe S. 4).

[3] Kishori Sharan. *Beginning Java 17 fundamentals : : object-oriented programming in Java 17*. eng. Third edition.. 2022 (siehe S. 4).

[4] Doug Winnie. *Essential Java for AP CompSci: From Programming to Computer Science*. eng. Berkeley, CA: Apress L. P, 2021 (siehe S. 4).

Online-Quellen

[5] *„TypeScript“ on CodePlex. Archived from the original on 3 April 2015*. URL: https://web.archive.org/web/20150403224440/https://typescript.codeplex.com/releases/view/95554 (besucht am 06. 10. 2023) (siehe S. 3).

[6] Rens Hijdra u. a. *VSCode - From Vision to Architecture*. URL: https://2021.desosa.nl/projects/vscode/posts/essay2/ (besucht am 07. 10. 2023) (siehe S. 6).

[7] *IntelliJ IDEA Wikipedia*. URL: https://en.wikipedia.org/wiki/IntelliJ_IDEA (besucht am 06. 10. 2023) (siehe S. 2).

[8] *IntelliJ IDEA. Archived from the original on 28 January 2001*. URL: http://web.archive.org/web/20010128152900/http://www.intellij.com:80/idea/features.jsp (besucht am 06. 10. 2023) (siehe S. 2).

[9] *IntelliJ Marketplace*. URL: https://plugins.jetbrains.com/ (besucht am 06. 10. 2023) (siehe S. 3).

[10] *IntelliJ Platform SDK Documentation*. URL: https://plugins.jetbrains.com/docs/intellij/welcome.html (besucht am 06. 10. 2023) (siehe S. 3, 8).

[11] *Our Approach to Extensibility*. URL: https://vscode-docs.readthedocs.io/en/stable/extensions/our-approach/ (besucht am 08. 10. 2023) (siehe S. 6).

[12] *PYPL PopularitY of Programming Language index*. URL: https://pypl.github.io/PYPL.html (besucht am 06. 10. 2023) (siehe S. 3).

[13] Razmik Seysyan. *Flora (beta) Plugin on JetBrains Marketplace*. URL: https://plugins.jetbrains.com/plugin/17669-flora-beta- (besucht am 06. 10. 2023) (siehe S. 8).

[14] *Stack Overflow Developer Survey 2023*. URL: https://survey.stackoverflow.co/2023/ (besucht am 06. 10. 2023) (siehe S. 2).

[15] *Stack Overflow Insights - Survey Data*. URL: https://insights.stackoverflow.com/survey (besucht am 06. 10. 2023) (siehe S. 2, 3).

[16] *TIOBE Index*. URL: https://www.tiobe.com/tiobe-index/ (besucht am 06. 10. 2023) (siehe S. 3).

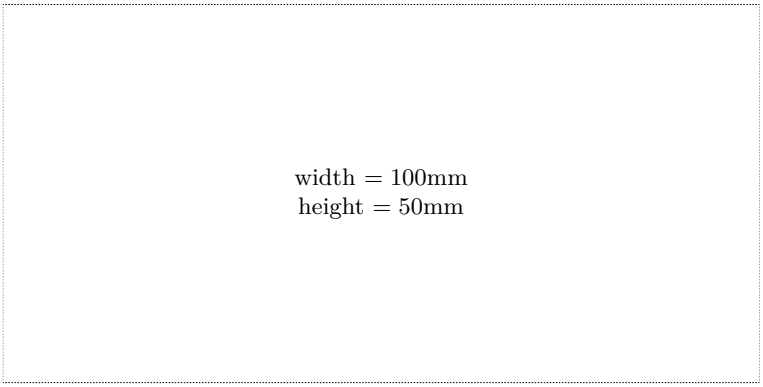
[17] *Visual Studio Code editor hits version 1, has half a million users*. URL: https://arstechnica.com/information-technology/2016/04/visual-studio-code-editor-hits-version-1-has-half-a-million-users/ (besucht am 06. 10. 2023) (siehe S. 2).

[18] *Visual Studio Code Marketplace*. URL: https://marketplace.visualstudio.com/search?target=VSCode&category=All%20categories&sortBy=Installs (besucht am 06. 10. 2023) (siehe S. 3).

[19] *Visual Studio Code Preview. Archived from the original on 9 October 2015*. URL: https://web.archive.org/web/20151009211114/http://blogs.msdn.com/b/vscode/archive/2015/04/29/announcing-visual-studio-code-preview.aspx (besucht am 06. 10. 2023) (siehe S. 2).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —