

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

© Copyright 2024 Philipp Seiringer

Diese Arbeit wird unter den Bedingungen der Creative Commons Lizenz *Attribution-NonCommercial-NoDerivatives 4.0 International* (CC BY-NC-ND 4.0) veröffentlicht – siehe <https://creativecommons.org/licenses/by-nc-nd/4.0/>.

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

Erklärung	iv
Kurzfassung	viii
Abstract	ix
1 Einleitung	1
1.1 Motivation	1
1.2 Ziel	1
1.3 Aufbau	1
2 Grundlagen der Plugin Entwicklung	2
2.1 Entwicklungsumgebungen	2
2.1.1 Visual Studio Code	2
2.1.2 IntelliJ IDEA	2
2.2 Programmiersprachen	3
2.2.1 TypeScript	3
2.2.2 Java	4
2.3 Aufbau der Plugin API	5
2.3.1 Visual Studio Code	5
2.3.2 IntelliJ IDEA	6
2.4 Funktionalität der Plugin API	8
2.4.1 Visual Studio Code	8
2.4.2 IntelliJ IDEA	11
2.4.3 IntelliJ Flora Plugins	14
3 Anforderungen an den Prototyp	16
3.1 Aufbau	16
4 Entwicklung des Prototyps für Visual Studio Code	17
4.1 Design	17
4.2 Implementierung	17
4.2.1 Aufsetzen des Projektes	17
4.2.2 Entwicklung	17
4.3 Tests	17
4.4 Publishing	17

4.5	CI/CD	17
5	Entwicklung des Prototyps für IntelliJ	18
5.1	Design	18
5.2	Implementierung	18
5.2.1	Aufsetzen des Projektes	18
5.2.2	Entwicklung	18
5.3	Tests	18
5.4	Publishing	18
5.5	CI/CD	18
6	Bewertungskriterien	19
6.1	Popularität der Entwicklungsumgebung	20
6.1.1	Visual Studio Code	20
6.1.2	IntelliJ IDEA	20
6.2	Performance	20
6.2.1	Visual Studio Code	20
6.2.2	IntelliJ IDEA	20
6.3	Feature Umfang	20
6.3.1	Visual Studio Code	20
6.3.2	IntelliJ IDEA	20
6.4	Intuitivität der API	20
6.4.1	Visual Studio Code	20
6.4.2	IntelliJ IDEA	20
6.5	Dokumentation der API	20
6.5.1	Visual Studio Code	20
6.5.2	IntelliJ IDEA	20
6.6	Testbarkeit des Plugins	20
6.6.1	Visual Studio Code	20
6.6.2	IntelliJ IDEA	20
6.7	Möglichkeiten des Publishings	20
6.7.1	Visual Studio Code	20
6.7.2	IntelliJ IDEA	20
6.8	Installationsprozess des Plugins	20
6.8.1	Visual Studio Code	20
6.8.2	IntelliJ IDEA	20
7	Vergleich der Kriterien	21
7.1	Popularität der Entwicklungsumgebung	22
7.1.1	Visual Studio Code	22
7.1.2	IntelliJ IDEA	22
7.1.3	Vergleich	22
7.2	Performance	22
7.2.1	Visual Studio Code	22
7.2.2	IntelliJ IDEA	22
7.2.3	Vergleich	22
7.3	Feature Umfang	22

7.3.1	Visual Studio Code	22
7.3.2	IntelliJ IDEA	22
7.3.3	Vergleich	22
7.4	Intuitivität der API	22
7.4.1	Visual Studio Code	22
7.4.2	IntelliJ IDEA	22
7.4.3	Vergleich	22
7.5	Dokumentation der API	22
7.5.1	Visual Studio Code	22
7.5.2	IntelliJ IDEA	22
7.5.3	Vergleich	22
7.6	Testbarkeit des Plugins	22
7.6.1	Visual Studio Code	22
7.6.2	IntelliJ IDEA	22
7.6.3	Vergleich	22
7.7	Möglichkeiten des Publishings	22
7.7.1	Visual Studio Code	22
7.7.2	IntelliJ IDEA	22
7.7.3	Vergleich	22
7.8	Installationsprozess des Plugins	22
7.8.1	Visual Studio Code	22
7.8.2	IntelliJ IDEA	22
7.8.3	Vergleich	22
8	Conclusion	23
A	Technische Informationen	24
	Quellenverzeichnis	25
	Literatur	25
	Online-Quellen	25

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 [20] von Microsoft veröffentlicht. Die Idee hinter VS Code war einen möglichst einfachen Code Editor anzubieten, welcher nur die wichtigsten und besten Funktionen für EntwicklerInnen beinhaltet. 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 [22].

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 von 73,71% der Abstimmenden aktiv genutzt[17, 18].

Ein Grund für diesen Erfolg mag vermutlich die Möglichkeit zur Entwicklung und zum Anbieten von Plugins sein. Durch die direkte Einbindung des 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, oft 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 [9, 10] 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 [2].

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 [12].

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 7.500 Plugins die in die IDE integriert werden können [11]. Für Visual Studio Code sind es hingegen inzwischen über 51.000 Plugins [21].

2.2 Programmiersprachen

2.2.1 TypeScript

Die TypeScript Programmiersprache wurde erstmalig am 1. Oktober 2012 [7] 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 [15, 18, 19]. 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 transpiert. Dadurch kann auf die Popularität und Verbreitung von JavaScript aufgebaut werden und TypeScript ist überall dort verwendbar, wo JavaScript ausführbar

ist. Weiters ist TypeScript ein Superset von JavaScript. Es gilt also: „Any valid .js file can be renamed .ts and be compiled with other TypeScript file.“ [3].

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 zusätzlichen 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 James Gosling, Mike Sheridan und Patrick Naughton designed [6]. 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 [1], 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 [4]. Im Gegensatz zu den zuvor gängigen Programmiersprachen muss Java nämlich nicht 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 welches 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 allokalieren 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 separaten „extension host process“. Dadurch kann verhindert werden, dass Plugins die Performance und die Interaktivität von VS Code negativ beeinflussen [8, 14].

Ablauf

Visual Studio Code analysiert zuerst das Extension Manifest des Plugins. Je nachdem welche Activation Points definiert sind, wird zu einem bestimmten Zeitpunkt 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 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 Listnern, 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,

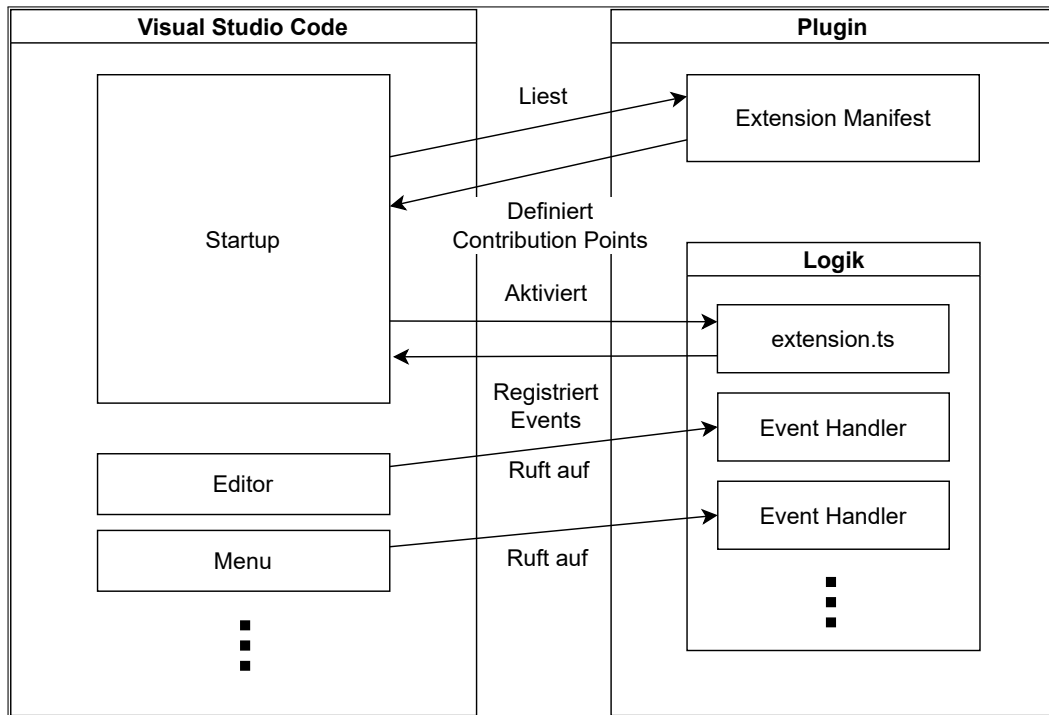


Abbildung 2.1: Übersicht über den Ablauf eines VS Code Plugins.

dass 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 Listnern implementiert werden können. Die Implementierung eines Plugins kann in den Sprachen Java und Kotlin erledigt werden. Da die Plugin API allerdings auf Java basiert, können nicht alle Sprachfeatures von Kotlin problemlos genutzt werden.

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.

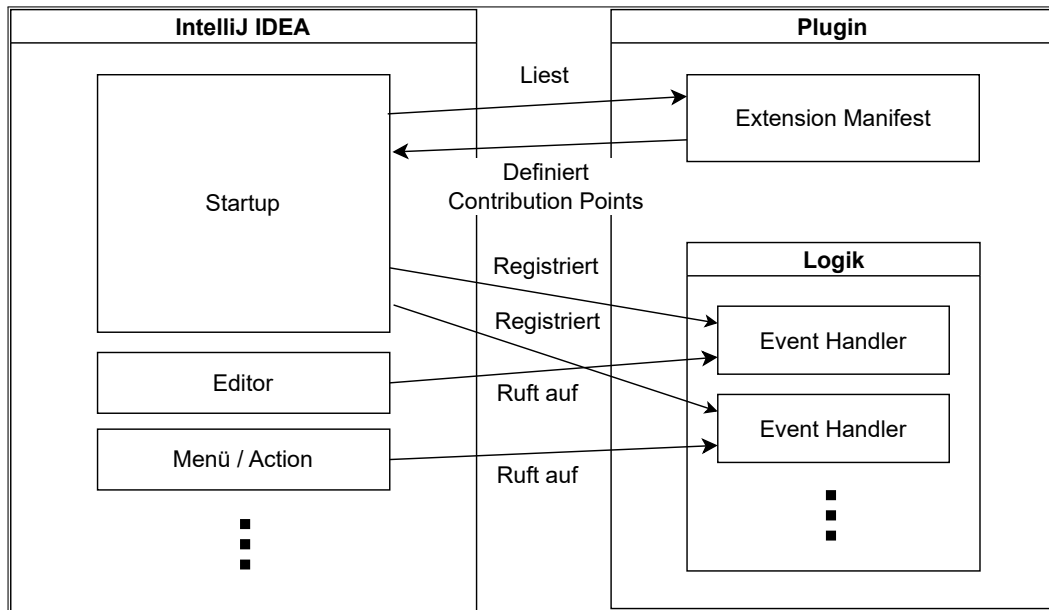


Abbildung 2.2: Übersicht über den Ablauf eines IntelliJ Plugins.

2.4 Funktionalität der Plugin API

2.4.1 Visual Studio Code

Die VS Code API erlaubt es VS Code durch Commands, Code Completion und Spracherweiterung, Themes, Custom Editor oder Notebooks, Views, Source Control, Debugger, Tests und vielem mehr zu erweitern. Um dies zu ermöglichen werden unter anderem auch Einstellungen, Datenspeicherung und verschiedene Arten der Ein und Ausgabe von Daten bereitgestellt. In den folgenden Abschnitten werden die wichtigsten Elemente genauer vorgestellt.

Commands und Menüs

Commands ermöglichen es dem Plugin bestimmten Code sozusagen „auf Befehl“ auszuführen. So können häufig wiederkehrende Aufgaben der User ganz einfach automatisiert werden. Um einen Command anzulegen, muss dieser im Extension Manifest definiert werden. Dabei muss das Plugin mindestens eine eindeutige Bezeichnung und zur Darstellung verwendeten Titel angeben. Optional können auch eine Kategorie, ein Icon, eine Kurzbezeichnung und eine Bedingung, zu der der Command verwendbar wird, bestimmt werden.

```

1  "commands": [
2    {
3      "command": "vscodeplugindemo.helloWorld",
4      "title": "Hello World",
5    }
6  ]
7

```

Welcher Code dann ausgeführt wird, muss über die API festgelegt werden. So wird meist in der activate Funktion mithilfe von „registerCommand“ oder „registerTextEditorCommand“ ein Callback festgelegt welches aufgerufen werden soll. Wichtig ist hier, dass die Register Funktionen ein Disposable Objekt retournieren welches der API bekannt gegeben werden muss. Die API kümmert sich dann auch um das Deaktivieren des Commands, falls zum Beispiel die Erweiterung deaktiviert werden sollte.

```
1 context.subscriptions.push(vscode.commands.registerCommand('
  vscodeplugindemo.helloWorld', () => {
2     vscode.window.showInformationMessage('Hello World from VsCodePluginDemo!');
3   }));
4
```

Um einen Command aufzurufen können die Nutzer direkt nach dem Command suchen (Tastenkombination Strg+Shift+P). Komfortabler ist es allerdings den Nutzern direkt einen passenden Menüeintrag oder ein Keybinding bereitzustellen. Menüeinträge können dabei an verschiedenen Stellen im IDE eingehängt werden. Gängig sind hierfür die Titelleiste des Editors, verschiedene Kontext (Rechtsklick) Menüs, der Dialog zum Anlegen einer neuen Datei, die Titelleiste einer bestimmten View oder ein neues Submenü in der Menüleiste. Sowohl Menüs als auch Keybindings können im Manifest registriert werden.

Spracherweiterungen

Ein wichtiger Teil der VS Code Plugin API sind Language Extensions. Visual Studio Code unterscheidet bei diesen Funktionen nach Highlighting, Language Features und Snippets.

Highlighting Das einfache Syntax Highlighting wird in VS Code durch eine TextMate Grammatik erledigt [5]. Diese Grammatik wird dabei nicht nur für das Highlighting genutzt, sondern sie ist auch für die „Tokenization“ zuständig. Durch eine Code-Analyse anhand der gegebenen Grammatik wird also der Text in kleine zusammengehörige Abschnitte (sogenannte Tokens) unterteilt. Diese Tokens werden dann zusätzlich noch klassifiziert, sodass zum Beispiel zwischen Kommentaren, Strings, RegExen und Code unterschieden werden kann. Die Grammatik wird hierfür in einer einfachen .json Datei im Plugin Projektordner abgelegt und dann per Manifest unter dem Contribution Point „grammar“ eingebunden. Mithilfe solcher Grammatiken können auch bereits bestehende Grammatiken erweitert werden. Bei der Auswahl von Scopes, die durch die Grammatik definiert werden, sollte man sich an die Naming Conventions von TextMate halten, da diese vordefinierten Scopes von vielen Themes unterstützt werden. Um die Kategorisierung von bestimmten Tokens noch genauer zu erledigen, gibt es auch „Semantic Highlighting“ in VS Code. Es kann in der API programmatisch ein „DocumentSemanticTokenProvider“ registriert werden, welcher den Code analysiert und zusätzliche (und zum Beispiel kontextabhängige) Informationen über die Tokens bereitstellt.

Language Features Auch hier gibt es statisch definierte und programmatische Language Features. Statisch kann unter dem Contribution Point „languages“ eine Reihe von Informationen angegeben werden, die es VS Code erlauben die User Experience stark zu verbessern. So kann man unter anderem Festlegen, mit welchen

Zeichen Kommentare eingeleitet werden, oder welche Klammern es gibt, damit VS Code das Zuklappen erlauben kann. Zusammengehörige Paare von Zeichen (also Klammern, Anführungszeichen, u.s.w.) können automatisch geschlossen werden. Und sogar für das automatische Einrücken der nächsten Zeile bei einem Zeilenbruch kann eine Regel erstellt werden. Programmatisch ist das Erweitern von Language Features etwas komplexer, allerdings steigt natürlich auch die Menge an Möglichkeiten. In der API können verschiedene Provider registriert werden, durch die Features wie „Go to Definition“, „IntelliSense“ code completion oder diagnostische Fehleranalysen und entsprechende Verbesserungsvorschläge ermöglicht werden. Grundsätzlich ist es auch möglich für einzelne Features einen Provider zu registrieren, allerdings empfiehlt es sich bei der Einbindung einer neuen Sprache einen „Language Server“ und das Language Server Protocol zu nutzen. Diese Lösung bringt nicht nur Performanceverbesserungen im Editor, sondern der Language Server kann auch für andere Editoren wiederverwendet werden, ohne ihn komplett neu implementieren zu müssen.

Snippets Snippets sind eine sehr einfache Form der Spracherweiterung. Es wird unter dem Contribution Point „snippets“ einfach eine Datei mit den Vorlagen angegeben. Eine Vorlage enthält dabei immer ein Kürzel für welches das Snippet vorgeschlagen werden soll, den zu ersetzenden Text und optional eine Beschreibung. Dabei können im Text auch Platzhalter genutzt werden an die der Cursor beim Einsetzen des Snippets springt.

Benutzereingaben

Für die Eingabe von Daten bietet Visual Studio Code die Quick Pick API, die File Picker API und den Configuration Contribution Point.

Quick Pick API gibt dem Plugin eine Möglichkeit den Nutzern ein einfaches Eingabefenster anzuzeigen. Dabei kann ein Fenster mit bereits vorgegebenen Auswahlmöglichkeiten durch den Aufruf von „showQuickPick“ oder „createQuickPick“ erstellt werden. Alternativ kann man mit den Funktionen „showInputBox“ oder „createInputBox“ die Nutzer auch selber einen Text eingeben lassen. Die show Funktionen bieten dabei immer eine einfache vorgefertigte Implementierung an. Falls diese Option nicht ausreichend ist, kann mit den create Funktionen auch eine komplexere Implementierung angegeben werden. Weiters kann auch eine Validierung des Inputs vorgenommen werden. Möchte man mehrere solcher Dialoge als Abfolge hintereinander anzeigen, so muss dies leider selber programmiert werden. Hierfür gibt es aber ein Beispiel im quickinput-sample im vscode-extension-sample repository [13].

File Picker API erlaubt das Auswählen von Ordnern oder Dateien aus dem Dateisystem des Betriebssystems mit der Funktion „showOpenDialog“. Dabei können Optionen angegeben werden die beeinflussen ob Ordner und/oder Dateien gewählt werden dürfen, ob mehrere Elemente selektiert werden dürfen, ob nach Dateinamen gefiltert werden soll und so weiter.

Configuration Contribution Point ermöglichen das Festlegen von Einstellungen die von den Nutzern eingegeben und vom Plugin ausgelesen werden können. Hier können Einstellungen vom Typ number, string und boolean definiert werden, welche

dann direkt im User Interface der VS Code Einstellungen bearbeitet werden können. Einfache object und array Properties können auch im UI dargestellt werden, allerdings dürfen diese keine verschachtelten Objekte oder Arrays enthalten. Ansonsten wird in den Einstellungen nur auf die manuell zu bearbeitende „settings.json“ Datei verwiesen. Für die Validierung der Einstellungen können Validierungsproperties von JSON Schema verwendet werden. Es ist also zum Beispiel möglich ein maximum/minimum, ein Regular Expression pattern oder ein enum Array mit erlaubten Werten angegeben werden. Zusätzlich ist es zu jeder Einstellung möglich einen Titel und eine Beschreibung anzugeben, wobei es sogar Beschreibungen gibt, welche Markdownformattierungen enthalten dürfen.

Ausgaben und Anzeigen

Um den Usern auch Feedback über die Ausführung des Plugin Codes zu geben, ist in VS Code für drei allgemeine Anwendungsfälle vorgesorgt.

Um den Usern eine kurze Rückmeldung zu geben können am besten Notifications genutzt werden. Diese zeigen eine Kurze Nachricht an, welche im Stil einer Information, einer Warnung oder einer Error Meldung dargestellt werden kann. Um einen längeren Fluss von Ausgaben (wie zum Beispiel Log-Nachrichten des Plugins) anzuzeigen, können Output Channels genutzt werden. An diese können Textzeilen nach und nach angehängt werden und sie werden dem User dann in einem terminalartigen Fenster präsentiert. In vielen Fällen reicht es schon als Feedback einen einfachen Ladebalken anzuzeigen. So kann dem User klar gemacht werden, dass das Plugin immer noch arbeitet und noch kein Fehler aufgetreten ist. Für diesen Anwendungsfall kann die Progress API genutzt werden.

Eine etwas komplexere Anzeige bieten Views, die die sogenannte Workbench erweitern. Mit der Tree View API kann eine einfach Baumstruktur, ähnlich der Dateiübersicht in der Explorer View, dargestellt werden. Für diese Implementierung muss ein TreeDataProvider erstellt werden, welcher die Baumstruktur und den Inhalt vorgibt. Die Webview API bietet im Gegensatz dazu sehr viel mehr Optionen. Diese kann in einem View eine Art „iframe“ anzeigen, in welchem dann HTML Inhalte dargestellt werden können. Dabei kann auch JavaScript und CSS Code eingebunden werden, es können Nachrichten vom Plugin an die Webview und zurück geschickt werden, es können Kontextmenüs in die View eingebunden werden und der Zustand der View kann persistiert werden.

2.4.2 IntelliJ IDEA

Das IntelliJ Platform SDK enthält einen sehr großen Umfang von Features und Extension Points die durch ein Plugin erweitert werden können. Einige wichtige Teile der API werden in den folgenden Abschnitten genauer beschrieben.

Actions und Menüs

Actions in IntelliJ funktionieren fast ident zu den Commands aus VS Code. Es handelt sich um einen vom Plugin definierten Code-Block, welcher von den Usern zum Beispiel über Menüeinträge angestoßen werden kann. Eine Action ist dabei eine einfache Java

Klasse, welche von „AnAction“ abgeleitet wird. Dabei muss die Methode „actionPerformed“ überschrieben werden. Diese enthält den Code der von der Action ausgeführt wird. Optional kann (und sollte) auch die „update“ Methode überschrieben werden, durch welche bestimmt wird wann die Action aktiviert oder versteckt ist.

Im Plugin Configuration File wird festgelegt wo und wie die programmierten Actions angezeigt werden. Dabei wird im Abschnitt „actions“ ein „action“ Element erstellt. Dieses hat für gewöhnlich eine eindeutige ID, eine Klasse mit der Code-Implementierung und einen Text, welcher zur Anzeige verwendet wird. Zusätzlich können eine Beschreibung und ein Icon festgelegt werden. Es können Gruppenzuordnungen bestimmt werden, die bestimmen wo und wie die Action angezeigt wird. Es können Keyboard Shortcuts bestimmt werden. Und es kann mit „override-text“ ein alternativer Text angegeben werden, der nur an bestimmten Orten angezeigt wird.

```
1    <actions>
2        <action id="my.simple.DemoAction"
3              class="my.simple.DemoAction"
4              text="Demo Action">
5            <add-to-group group-id="ToolsMenu" anchor="first"/>
6        </action>
7    </actions>
8
```

Services

IntelliJ erlaubt es einem Plugin Services zu definieren, welche dann auf drei Ebenen in jeweils zwei Varianten implementiert werden können. Instanzen solcher Services können dann an beliebigen Stellen im Plugin Code verwendet werden.

Die Ebene auf der ein Service erstellt wird, bestimmt wie viele Instanzen dieses Services existieren können. Dabei gibt es das application-level, welches den Service als globales Singleton anbietet. Und project-level und module-level Services, bei welchen für jedes geöffnete Projekt bzw. Modul je eine Instanz des Services besteht. Allerdings wird empfohlen aus Effizienzgründen keine Services auf Modul-Level zu erstellen. In Bezug auf die Varianten gibt es Light Services und normale Services.

Die Light Services sind dabei einfache Klassen, welche mit der Annotation „@Service“ versehen sind. Light Services sind sehr effizient, allerdings gibt es einige Einschränkungen. So können beispielsweise keine anderen Services per Dependency Injektion injiziert werden. Gewöhnliche Services haben diese Einschränkungen nicht. Bei ihnen wird ein beliebiges Interface und eine dazugehörige Implementierung definiert. Diese werden daraufhin im Plugin Configuration File unter den Extension Points „applicationService“ oder „projectService“ registriert. Die Project-Level Services erhalten dabei sowohl als Light Service, als auch als normaler Service, eine Referenz auf das aktuelle Projekt.

Listeners und Extension Points

Ganz ähnlich zu den (nicht Light) Services funktioniert auch die Registrierung von Listnern und Extension Points. Allerdings gibt es hier bereits vorgefertigte Interfaces die implementiert werden müssen. Wurde die Implementierung dann in der Project Configuration registriert, wird sie zu den entsprechenden Zeitpunkten aufgerufen.

Bei Extension Points gibt es die weitere Besonderheit, dass auch eigene Extension Points deklariert und im Plugin Code aufgerufen werden können. Diese werden dann für andere Plugins zur Erweiterung angeboten.

PSI und Spracherweiterungen

PSI steht für Program Structure Interface und ist die Grundlage jeder Sprachunterstützung in IntelliJ. Eine Datei mit Quellcode wird von IntelliJ immer zu einer PSI File geparsed. Dieses PSI File enthält dann eine Baumstruktur von PSI Elements, welche der Struktur des Quellcodes entspricht. Die Blattknoten eines solchen Baumes entsprechen dabei für gewöhnlich einzelnen Tokens und Identifiern, die nicht mehr zerlegt werden können. Nach oben hin werden diese dann zu Statements, Code-Blöcken, Methoden, Klassen u.s.w. gebündelt. Der Vorteil dieser PSI Files ist, dass sie von IntelliJ und auch von Plugins jederzeit verwendet werden können. Somit können unter anderem verschiedene Sprachfeatures sehr effizient implementiert werden.

Um die IntelliJ Plattform mit einer neuen Sprache zu erweitern, muss zuerst die entsprechende Sprache und ihre Grammatik definiert werden. Die Grammatik hat dabei nicht nur die Aufgabe die Regeln einer neuen Sprache festzulegen. Durch sie wird auch die Struktur der Sprache festgestellt, was später für den Aufbau der PSI Elemente wichtig ist. Hierfür wird eine Art der Backus Naur Form (BNF) verwendet, welche mit zusätzlichen Informationen gespickt wird. So gibt es zum Beispiel Attribute wie `private`, `inner` oder `upper`, welche die Struktur des PSI Baumes beeinflussen. Mithilfe des Grammar Kit Plugins kann aus einer solchen BNF automatisch ein Parser und die entsprechenden PSI Elemente generiert werden. Zusätzlich zum Parser wird auch ein eigener Lexer benötigt. Dabei wird empfohlen mithilfe von JFlex eine Datei mit Regeln automatisch in einen Lexer übersetzen zu lassen.

Sobald Parser und Lexer registriert sind steht das Grundgerüst der Spracherweiterung. Danach können verschiedene Features für die Sprache unabhängig voneinander implementiert werden. In den meisten Fällen ist dabei ein bestimmtes Interface zu implementieren und der entsprechende Contributor oder Provider im Configuration File zu registrieren. So können Auto Completion, Folding, Go To Symbol, References und Refactoring sowie viele weitere Funktionalitäten unterstützt werden.

User Interface Komponenten

Auch in IntelliJ gibt es einige vorgefertigte UI Komponenten. Das dient vor allem dazu, die User Experience über verschiedene Plugins hinweg möglichst einheitlich und im Stile der IntelliJ Plattform zu halten. Häufig verwendete Komponenten sind hierbei Dialoge, Popups, Notifications und Tool Windows. Für die Programmierung und Darstellung dieser Komponenten setzt IntelliJ stark auf das Java Swing Framework. Dafür wurden auch bereits existierende Elemente des Swing Frameworks zusätzlich überarbeitet und verbessert. So wurde zum Beispiel die Swing JList durch die JetBrains JList oder der JTree durch den Tree ersetzt.

Dialoge erlauben die Anzeige einer beliebigen Java Swing Komponente. Diese kann im einfachsten Fall ein Label mit einer Frage sein, es können allerdings auch komplexere Dialoge mit mehreren Eingabefeldern erstellt werden. Wobei sogar eine Validierung der Eingabefeldern eingebunden werden kann. Zusätzlich werden im

Dialog automatisch weitere Buttons für das Abschließen des Dialogs eingebunden. Diese Buttons sind per default OK und Cancel, allerdings können auch diese manuell neu konfiguriert werden. Die Implementierung eines solchen Dialogs kann durch Ableiten der „DialogWrapper“ Klasse geschafft werden.

Popups sind eine sehr einfache Form von Dialogen. Der Unterschied zu Dialogen ist, dass Popups keine zusätzlichen Buttons einbinden, sondern automatisch geschlossen werden, sobald der Fokus verloren geht. Um einige vorgefertigte Varianten zu nutzen, können die Methoden der JBPopupFactory aufgerufen werden. So können einfache Popups zur Auswahl aus einer Liste oder zur Bestätigung einer Frage erstellt werden. Natürlich ist es auch wieder möglich benutzerdefinierte Popups mit eigenen Swing Komponenten zu erstellen. Dabei sollte allerdings im Hinterkopf gehalten werden, dass diese nicht zu komplex werden sollten.

Notifications sind zum Anzeigen von kurzen Informationen gedacht, für die ein vollständiger Dialog oder ein Popup zu störend wäre. Dabei gibt es die Editor Hints, welche eine Art Sprechblase an der aktuellen Stelle des Editors anzeigen. Diese können vor allem für kurze Fehlermeldungen von Editor-bezogenen Actions genutzt werden. Editor Banner werden am besten für etwas gravierendere Fehlermeldungen eingesetzt. So werden sie zum Beispiel von IntelliJ angewendet, wenn innerhalb eines Projekts kein passendes JKD definiert ist. Für weitere Fehlermeldungen können am besten sogenannte „Balloons“ genutzt werden. Diese Meldungen tauchen für gewöhnlich in kleinen Blasen am rechten Rand des IntelliJ Fensters auf, und verschwinden dann nach zehn Sekunden automatisch. Sie können allerdings auch auf Sticky geschaltet werden, damit sie nicht mehr verschwinden können.

Tool Windows sind Fenster die in der Werkzeugleiste von IntelliJ eingehangen werden können. Am einfachsten können diese erstellt werden, indem eine Klasse von „ToolWindowFactory“ abgeleitet wird. Diese Klasse kann dann in der Plugin Configuration als ToolWindow eingebunden werden. Jedes ToolWindow kann mehrere „Contents“ besitzen, welche im Grunde beliebige JPanel Elemente anzeigen können. Solche ToolWindows eignen sich am besten um komplexere UI in das Plugin einzubinden.

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 gewünschte Funktionalität wirklich ein vollwertiges Plugin benötigt. Häufig kommt es nämlich vor, dass nur bestimmte kleine Tasks innerhalb des IDEs automatisiert werden sollen [12]. 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 [16]. Innerhalb dieser Plugin Dateien kann über die Variable „ide“ auf die angebotene Schnittstelle zugegriffen werden. Diese erlaubt es unter anderem Actions, Keyboard Shortcuts, Services

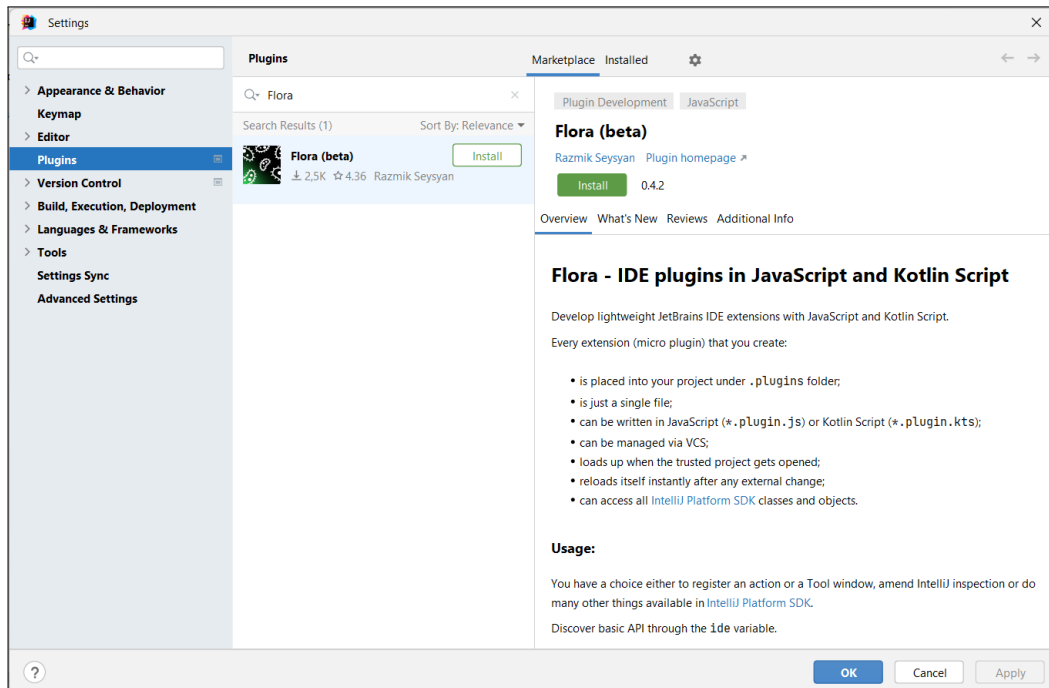


Abbildung 2.3: Flora Plugin im IntelliJ Plugin Marketplace.

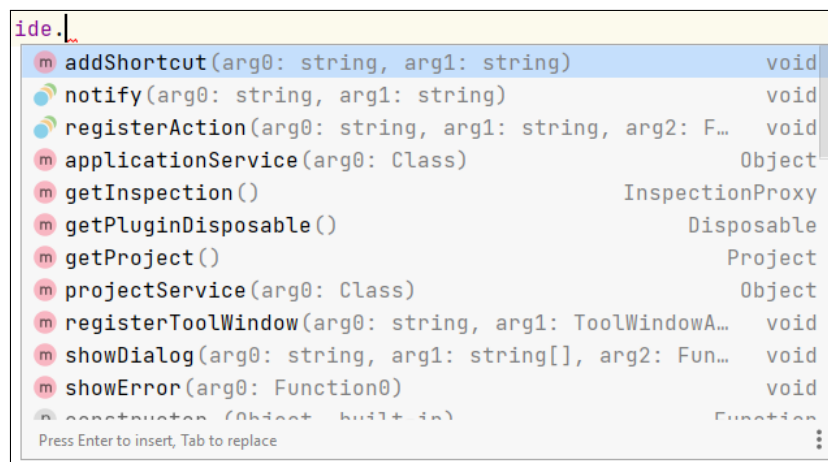


Abbildung 2.4: Übersicht über die API des Flora Plugins.

und ToolWindows zu erstellen.

Flora Plugins bieten sich vor allem dann an, wenn eine projektspezifische Aufgabe automatisiert werden soll. Hier sind vor Allem die Leichtigkeit 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 werden kann.

Kapitel 3

Anforderungen an den Prototyp

3.1 Aufbau

Kapitel 4

Entwicklung des Prototyps 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 Prototyps 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] Ken Arnold. *The Java programming language*. eng. 1. print.. The Java series. 1996 (siehe S. 4).
- [2] Ted Hagos. *Beginning IntelliJ IDEA : Integrated Development Environment for Java Programming*. eng. 1st ed. 2022.. 2022 (siehe S. 3).
- [3] Dan Maharry. *TypeScript Revealed*. eng. Berkeley, CA: Apress (siehe S. 4).
- [4] Kishori Sharan. *Beginning Java 17 fundamentals : : object-oriented programming in Java 17*. eng. Third edition.. 2022 (siehe S. 4).
- [5] *TextMate Language Grammar*. URL: https://macromates.com/manual/en/language_grammars (besucht am 10.10.2023) (siehe S. 9).
- [6] Doug Winnie. *Essential Java for AP CompSci: From Programming to Computer Science*. eng. Berkeley, CA: Apress L. P, 2021 (siehe S. 4).

Online-Quellen

- [7] „TypeScript“ on CodePlex. Archived from the original on 3 April 2015. URL: <http://web.archive.org/web/20150403224440/https://typescript.codeplex.com/releases/view/95554> (besucht am 06.10.2023) (siehe S. 3).
- [8] 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).
- [9] *IntelliJ IDEA Wikipedia*. URL: https://en.wikipedia.org/wiki/IntelliJ_IDEA (besucht am 06.10.2023) (siehe S. 2).
- [10] *IntelliJ IDEA*. Archived from the original on 28 January 2001. URL: <http://web.archive.org/web/20010128152900/http://www.intelij.com:80/idea/features.jsp> (besucht am 06.10.2023) (siehe S. 2).
- [11] *IntelliJ Marketplace*. URL: <https://plugins.jetbrains.com/> (besucht am 06.10.2023) (siehe S. 3).
- [12] *IntelliJ Platform SDK Documentation*. URL: <https://plugins.jetbrains.com/docs/intellij/welcome.html> (besucht am 06.10.2023) (siehe S. 3, 14).

- [13] Microsoft. *Visual Studio Code Extension Samples*. URL: <https://github.com/microsoft/vscode-extension-samples> (besucht am 06.10.2023) (siehe S. 10).
- [14] *Our Approach to Extensibility*. URL: <https://vscode-docs.readthedocs.io/en/stable/extensions/our-approach/> (besucht am 08.10.2023) (siehe S. 6).
- [15] *PYPL PopularitY of Programming Language index*. URL: <https://pypl.github.io/PYPL.html> (besucht am 06.10.2023) (siehe S. 3).
- [16] Razmik Seysyan. *Flora (beta) Plugin on JetBrains Marketplace*. URL: <https://plugins.jetbrains.com/plugin/17669-flora-beta-> (besucht am 06.10.2023) (siehe S. 14).
- [17] *Stack Overflow Developer Survey 2023*. URL: <https://survey.stackoverflow.co/2023/> (besucht am 06.10.2023) (siehe S. 2).
- [18] *Stack Overflow Insights - Survey Data*. URL: <https://insights.stackoverflow.com/survey> (besucht am 06.10.2023) (siehe S. 2, 3).
- [19] *TIOBE Index*. URL: <https://www.tiobe.com/tiobe-index/> (besucht am 06.10.2023) (siehe S. 3).
- [20] *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).
- [21] *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).
- [22] *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! —