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 unter anderem Aspekte wie die Einfachheit und Flexibilität in der Entwicklung, der Umfang an angebotener Funktionalität und die Möglichkeit die Nutzerinteraktion und somit die User Experience zu steuern 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 [69] 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 Entwicklungsumgebungen (engl. 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 automatische Codevervollständigung namens „IntelliSense“ unterstützte. Weiters war Visual Studio Code das erste Produkt der Visual Studio Familie, welches plattformübergreifen für Windows, Linux und macOS angeboten wurde [71].

Aus den Stack Overflow Developer Surveys[65, 66] 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) die 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[65, 66].

Ein Grund für diesen Erfolg mag vermutlich die Möglichkeit zur Entwicklung und zum Anbieten von Extensions 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 Extensions entwickelt, verbessert und betreut. Durch solche, oft von der Community erstellte, Extensions kann VS Code auch eine enorme Anzahl von Programmiersprachen unterstützen.

2.1.2 IntelliJ IDEA

IntelliJ IDEA wurde erstmals im Januar 2001 [17] von dem Unternehmen JetBrains veröffentlicht. Im Gegensatz zu VS Code handelt es sich bei IntelliJ um eine vollausgestattetes Integrate Development Environment (IDE), welche speziell auf die Entwicklung von Programmen in den Programmiersprachen Java, Kotlin und Groovy ausgelegt ist.

IntelliJ IDEA wird in einer freien, Open Source „Community Edition“ [56] sowie in einer kommerziellen Form „IntelliJ IDEA Ultimate“ angeboten [4].

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 und WebStorm für JavaScript. 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, auch für mehrere IDEs dieser Plattform veröffentlicht werden kann [54].

Im Gegensatz zu Visual Studio Code ist IntelliJ ein 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 [18]. Für Visual Studio Code sind es hingegen inzwischen über 51.000 Extensions [70].

2.2 Programmiersprachen

2.2.1 TypeScript

Die Programmiersprache TypeScript wurde erstmalig am 1. Oktober 2012 [11] von Microsoft in Form eines Open-Source-Projekts veröffentlicht. Entworfen wurde sie von Anders Hejlsberg, der zuvor auch die Programmiersprache C# entworfen hatte.

Die grundsätzliche Idee der Sprache ist, eine typsichere, kompilierte, und somit bessere Version von JavaScript zu sein. JavaScript ist aufgrund des Erfolgszugs des World Wide Web zu einer sehr wichtigen Sprache geworden und war auch schon 2012 aus den TOP-Listen für Programmiersprachen nicht mehr wegzudenken [63, 66, 68]. Webseiten setzen heute sehr stark auf JavaScript, um durch interaktive Elemente die User Experience zu verbessern oder um neue Funktionalität anbieten zu können. Durch Node.js kann JavaScript nicht nur im Browser verwendet werden, sondern es können auch Desktop-, Server- oder mobile-Anwendungen in JavaScript entwickelt werden [8]. Durch diesen großen Umfang an Möglichkeiten, die JavaScript dadurch bietet, werden auch immer größere Projekte damit entwickelt. Und hier kommen die großen Schwächen von JavaScript zu tragen. Je größer die Projekte werden und je mehr EntwicklerInnen an einem Projekt mitarbeiten, desto mehr Fehler entstehen aufgrund der fehlenden Typsicherheit und der fehlenden statischen Überprüfungen (wie zum Beispiel bei einem Compiler). Beide Schwachstellen versucht TypeScript auszubessern.

TypeScript Quellcode wird mithilfe des TypeScript Compilers *tsc* in JavaScript Dateien transpiliert. Dadurch kann auf die Popularität und Verbreitung von JavaScript aufgebaut werden. TypeScript ist überall dort verwendbar, wo JavaScript ausführbar

ist. Weiters ist TypeScript eine echte Übermenge von JavaScript. Es gilt also: „Any valid .js file can be renamed .ts and be compiled with other TypeScript file." [6].

Jedoch bietet TypeScript eine Menge von Vorteilen gegenüber Javascript.

- 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 Werkzeug zur statischen Programmanalyse, wie JSLint. Dieser Kompilierschritt kann natürlich auch in eine CI/CD Pipeline eingebunden werden, um auch bei Änderungen in einem Repository Informationen über die Gültigkeit des Quellcodes zu erhalten.
- Durch die statische Typisierung können Programmierfehler bezüglich der Verwendung von Variablen vermieden werden. Auch die Unterstützung durch verschiedene IDEs, zum Beispiel mittels IntelliSense kann durch 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 Eigenschaften mit Zugriffsmodifikatoren (*private*/*public*) versehen sind.
- TypeScript unterstützt Vererbung, Schnittstellen und generische Programmierung.
- In TypeScript können bereits bestehende JavaScript-Bibliotheken verwendet werden. Weiters ist es möglich, durch zusätzliche Dateien Typinformationen zu JavaScript-Bibliotheken zu liefern.

2.2.2 Java

Die Entwicklung der Programmiersprache Java begann im Jahr 1991. Java wurde von James Gosling, Mike Sheridan und Patrick Naughton entworfen [10]. Java wurde erstmals im Jahr 1995 von Sun Microsystems veröffentlicht. Im Januar 2010 wurde Sun Microsystems 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 EntwicklerInnen einen leichten Umstieg auf die neue Sprache Java zu ermöglichen. Allerdings versuchte Java die teils sehr komplexen (wenn auch effektiven) Sprachfeatures von C++ etwas zu vereinfachen. Java sollte eine einfach, objektorientierte und robuste Sprache werden. Die Funktionalität die Java zu dem großen Erfolg verhalf, den sie später hatte, war das Prinzip

> "write once, run anywhere"

(WORA) [9].

Im Gegensatz zu den zuvor gängigen Programmiersprachen muss Java nämlich nicht für bestimmte Hardwarearchitekturen kompiliert werden. Java-Programme werden in eine Zwischensprache, den 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 der Java Runtime Environment (JRE) mitgeliefert wird. Ein kompiliertes Java-Programm kann also auf allen Geräten ausgeführt werden, auf denen eine passende 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 unterbunden werden.

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

2.3 Aufbau der Plugin API

Um die Bezeichnungen für IntelliJ Plugins und VS Code Extensions zu vereinheitlichen, wird in den folgenden Abschnitten und Kapiteln *Plugin* als Überbegriff verwendet, der auch VS Code Extensions einschließt.

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 Datei *package.json*. In dieser werden statische Einstellungen vorgenommen und Metainformationen über das Plugin bekannt gegeben. So kann hier unter anderem Name, Beschreibung, Herausgeber und Lizenzvereinbarungen eingestellt werden. Weiters definiert das Manifest eine sogenannte JavaScript- oder TypeScript-Datei *main* und dazu passende *Activation Events* und *Contribution Points* [80, 81].

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 bei Bedarf (on demand) zu Laden. Allerdings muss die Aktivierung erfolgen bevor die erste Funktionalität des Plugins erwartet wird. Typische Aktivierungsereignisse sind *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 [72, 80].

Contribution Points legen fest, welche Funktionalität das Plugin anbietet und somit auch, welche zusätzlichen UI-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 Plugin-definiertem Content, Keyboard Shortcuts, Unterstützung für neue Sprache und vieles mehr auszustatten [76].

Visual Studio Code API

Die eigentliche VS Code API kann in jeder TypeScript-Datei (zum Beispiel auch in der Datei *main*) genutzt werden. Hierfür wird das Modul *vscode* importiert. Dieses beinhaltet eine vollständige Definition der angebotenen Schnittstelle, auf welche zugegriffen werden kann [80, 96].

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

Über diese API kann zum Beispiel festgelegt werden, durch welchen Code die zuvor definierten Contribution Points implementiert werden sollen. Der Plugin Code wird in VS 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 [15, 62].

Ablauf

Wie in Abbildung 2.1 zu sehen ist, analysiert VS Code 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 VS Code aufgerufen und können so beliebigen Code ausführen.

2.3.2 IntelliJ IDEA

Der Aufbau der Plugin-Architektur wirkt bei IntelliJ im ersten Moment gleich wie bei VS Code. Es gibt auch hier ein *Plugin Configuration File*, sowie ein Modul mit API Schnittstellen [39]. Der große Unterschied liegt allerdings in der Funktionsweise und der Interaktion mit den Plugins sowie der Art, wie der auszuführende Code angegeben wird.

Plugin Configuration File

Die Konfiguration eines Plugins liegt in der Datei *plugin.xml* und beinhaltet, äquivalent zum Extension Manifest in VS Code, alle für das Plugin notwendigen Meta-Informationen. So können auch hier Werte wie der Name, eine Beschreibung und die aktuelle Versionsnummer angegeben werden. Für die Funktionen, die das Plugin mitbringt, gibt es Actions, Extension Points und Listener. Hier ist anzumerken, dass es sich

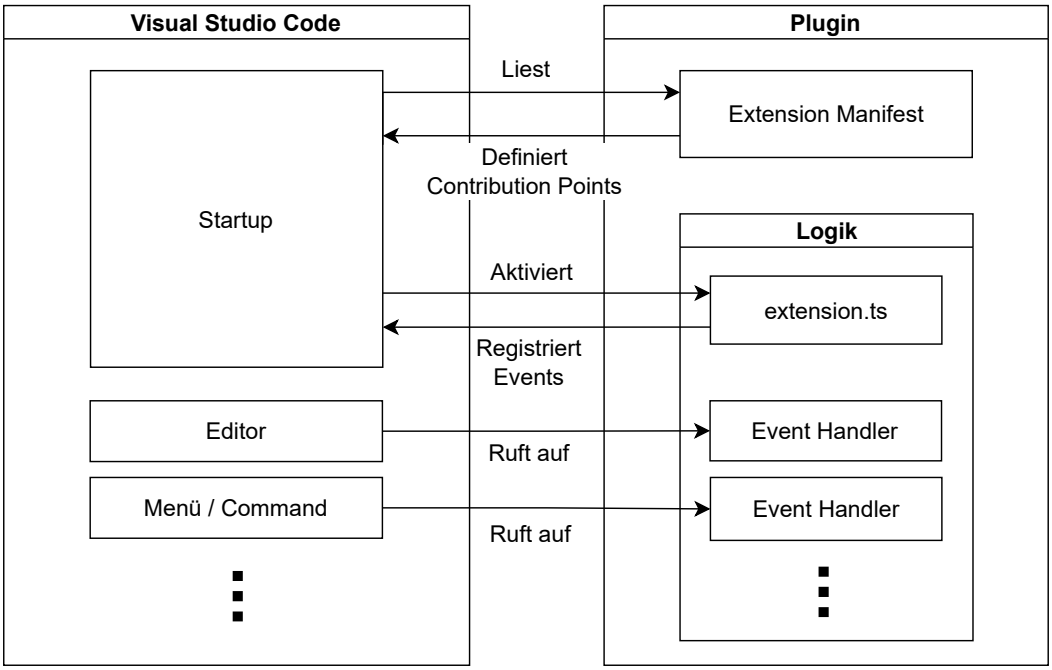


Abbildung 2.1: Übersicht über den Ablauf eines VS Code Plugins.

sowohl bei den Extension Points als auch den Listenern immer 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 dieser durch die Zuordnung der auszuführenden Klassen sowieso durch die Konfigurationsdatei festgelegt wird [37]. Eine Besonderheit an IntelliJ ist, dass Plugins auch eigene Extension Points definieren können, um weiteren Plugins das Erweitern des ursprünglichen Plugins zu ermöglichen [27].

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. 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 [23].

Ablauf

Wie in Abbildung 2.2 zu sehen ist, analysiert IntelliJ 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.

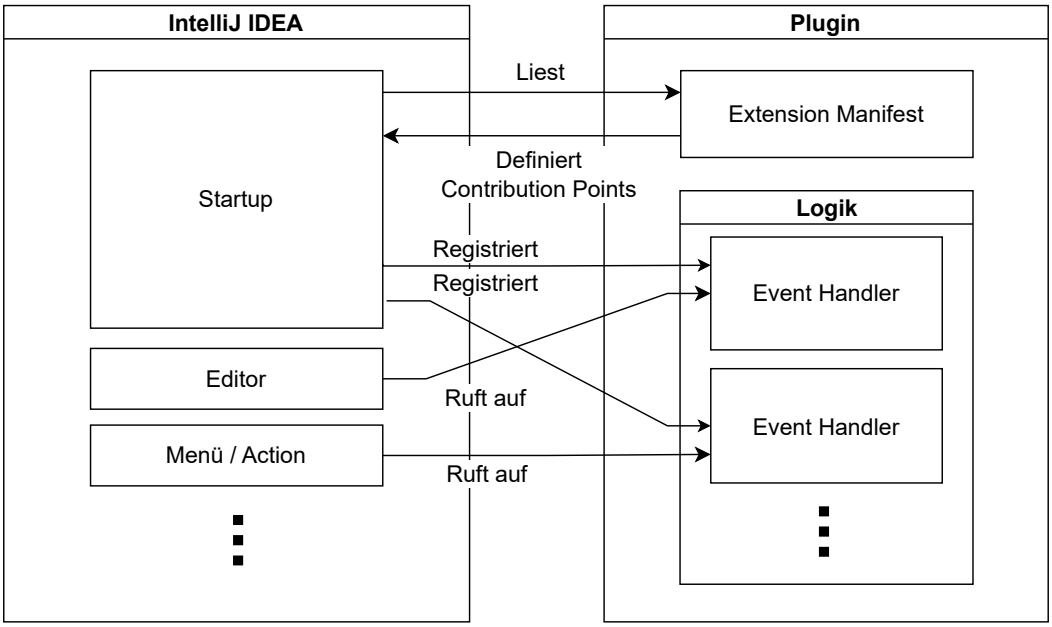


Abbildung 2.2: Übersicht über den Ablauf eines IntelliJ Plugins.

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, 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 BenutzerInnen 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 und eine Kurzbezeichnung bestimmt werden. Weiters ist es möglich eine Bedingung festzulegen, die beeinflusst, wann der Command verwendbar ist und wann nicht [77].

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

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 Objekt retournieren, welches *Disposable* implementiert. Dieses muss der API bekannt gegeben werden. Die API kümmert sich dann auch um das Deaktivieren des Commands, falls zum Beispiel die Erweiterung deaktiviert werden sollte [73].

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

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 in der Benutzeroberfläche eingehängt werden. Gängig sind hierfür die Titelleiste des Editors, verschiedene Kontext-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 [79].

Spracherweiterungen

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

Highlighting Das Syntax Highlighting wird in VS Code durch eine TextMate-Grammatik erledigt [67]. Diese Grammatik wird dabei nicht nur für das Highlighting genutzt, sondern sie ist auch für das „Tokenization“ zuständig. Durch eine Codeanalyse anhand der gegebenen Grammatik wird der Text in kleine zusammengehörige Abschnitte (sogenannte Tokens) unterteilt. Diese Tokens werden zusätzlich noch klassifiziert, sodass zum Beispiel zwischen Kommentaren, Zeichenkettenliteralen, regulären Ausdrücken 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 Namenskonventionen von TextMate halten. Scopes mit diesem Namensschema werden auch von vielen Themes unterstützt werden [93]. Um die Kategorisierung von bestimmten Tokens noch genauer zu erledigen, gibt es in VS Code auch „Semantic Highlighting“. Es kann in der API ein *DocumentSemanticTokensProvider* registriert werden, welcher den Code analysiert und zusätzliche (zum Beispiel kontextabhängige) Informationen über die Tokens bereitstellt [91].

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 Ex-

perience 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 Zeilenumbruch kann eine Regel erstellt werden [83]. Programmatisch ist das Erweitern von Language Features etwas komplexer, allerdings steigt auch die Menge an Möglichkeiten. In der API können verschiedene Provider registriert werden, durch die Features wie „Go to Definition“, „IntelliSense“ Codevervollständigung oder diagnostische Fehleranalysen und entsprechende Verbesserungsvorschläge ermöglicht werden [86]. 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 dass er mehrfach implementiert werden muss. [3].

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 [92].

Benutzereingaben

Für die Eingabe von Daten bietet VS Code die Quick Pick API, die File Picker API und den Configuration Contribution Point [74].

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 [90]. Möchte man mehrere solcher Dialoge als Abfolge hintereinander anzeigen, so muss dies selber programmiert werden. Hierfür gibt es ein Beispiel namens *quickinput-sample* [89] im Repository *vscode-extension-sample* [58].

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 zum Beispiel beeinflussen, ob Ordner und/oder Dateien gewählt werden dürfen, ob mehrere Elemente selektiert werden dürfen oder ob nach Dateinamen gefiltert werden soll [82].

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*-Eigenschaften 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 Datei „settings.json“ verwiesen. Für die Validierung der Einstellungen können Validierungsproperties von JSON Schema verwendet werden. Es ist also zum Beispiel möglich, ein maximum/minimum, einen regulären Ausdruck oder ein enum Array mit erlaubten Werten anzugeben. Zusätzlich ist es zu jeder Einstellung möglich, einen Titel und eine Beschreibung anzugeben, wobei es sogar Beschreibungen gibt, welche Markdown-Formattierungen enthalten dürfen [78].

Ausgaben und Anzeigen

Um den BenutzerInnen auch Feedback über die Ausführung des Plugin-Codes zu geben, ist in VS Code für drei allgemeine Anwendungsfälle vorgesorgt [74].

Um den BenutzerInnen 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 Fehlermeldung dargestellt werden kann [84]. 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 den BenutzerInnen dann in einer einfachen Textausgabe präsentiert [85]. In vielen Fällen reicht es schon als Feedback eine einfache Fortschrittsanzeige anzuzeigen. So kann den BenutzerInnen klar gemacht werden, dass das Plugin immer noch arbeitet und noch kein Fehler aufgetreten ist. Für diesen Anwendungsfall kann die Progress API genutzt werden [87].

Eine etwas komplexere Anzeige bieten Views, die die sogenannte Workbench erweitern. Mit der Tree View API kann eine einfache 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 [95]. Die Webview API bietet im Gegensatz dazu mehr Optionen. Diese kann in einer 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 [97].

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 BenutzerInnen zum

Beispiel über Menüeinträge angestoßen werden kann. Eine Action ist dabei eine einfache Java-Klasse, welche von der Klasse *AnAction* abgeleitet wird. Dabei muss die Methode *actionPerformed* überschrieben werden. Diese enthält den Code, der von der Action ausgeführt wird. Optional sollte auch die *update*-Methode überschrieben werden, durch welche bestimmt wird, wann die Action aktiviert oder versteckt ist [20, 21].

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 [22].

```
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>
```

Services

IntelliJ erlaubt es, Plugin Services zu definieren, welche dann auf drei Ebenen in jeweils zwei Varianten implementiert werden können [47]. 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 [2] anbietet. Und es gibt *project-level* und *module-level* Services, bei welchen für jedes geöffnetet 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 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. Vollwertige 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 vollwertigen Services funktioniert auch die Registrierung von Listenern 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 [27, 34].

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 [41]. Immer wenn eine Datei mit Quellcode wird von IntelliJ geparsed wird, wird zugleich auch ein Objekt der Klasse *PsiFile* generiert. Dieses *PsiFile*-Objekt enthält dann eine Baumstruktur von Objekten der Klasse *PsiElement*, welche der Struktur des Quellcodes entspricht [42, 43]. 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 Anweisungen, Code-Blöcken, Methoden, Klassen u.s.w. gebündelt. Diese Struktur kann mithilfe des PsiViewers sehr gut analysiert werden [44]. Der Vorteil dieser *PsiFile*-Objekte 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 Syntaxregeln 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 [7, 14]. So gibt es zum Beispiel Attribute wie private, inner oder upper, welche die Struktur des PSI Baumes beinflussen. Mithilfe des Grammar Kit Plugins kann aus einer solchen Grammatik automatisch ein Parser und die entsprechenden PSI-Elemente generiert werden [30]. Zusätzlich zum Parser wird auch ein Lexer benötigt. Dabei wird empfohlen, mithilfe von JFlex eine Datei mit Regeln automatisch in einen Lexer übersetzen zu lassen [5, 31, 57].

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 [25, 29].

User Interface Komponenten

Auch in IntelliJ gibt es einige vorgefertigte UI-Komponenten. Diese dienen vor allem dazu, die User Experience über verschiedene Plugins hinweg möglichst einheitlich und im Stile der IntelliJ Platform zu halten. Häufig verwendete Komponenten sind hierbei Dialoge, Popups, Notifications und Tool Windows. Für die Programmierung und Darstellung dieser Komponenten setzt IntelliJ auf das Java Swing Framework [52]. Dafür wurden auch bereits existierende Elemente des Swing Frameworks zusätzlich überarbeitet und verbessert. So wurde zum Beispiel die Swing *JList* durch die JetBrains *JBList* oder der *JTree* durch den *Tree* ersetzt [33, 35].

Dialoge erlauben die Anzeige einer beliebigen Java-Swing-Komponente. Diese kann im einfachsten Fall ein Label mit einer Frage sein, es können allerdings auch kom-

plexere 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 [26].

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 [40].

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 häufig für Fehlermeldungen oder Warnungen eingesetzt, welche die Projektstruktur oder die Projekteinstellungen betreffen. So werden sie zum Beispiel von IntelliJ angewendet, wenn innerhalb eines Projekts kein passendes Java Development Kit (JDK) 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“ geschalten werden, damit sie nicht mehr verschwinden können [36].

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 [51].

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 [54]. 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

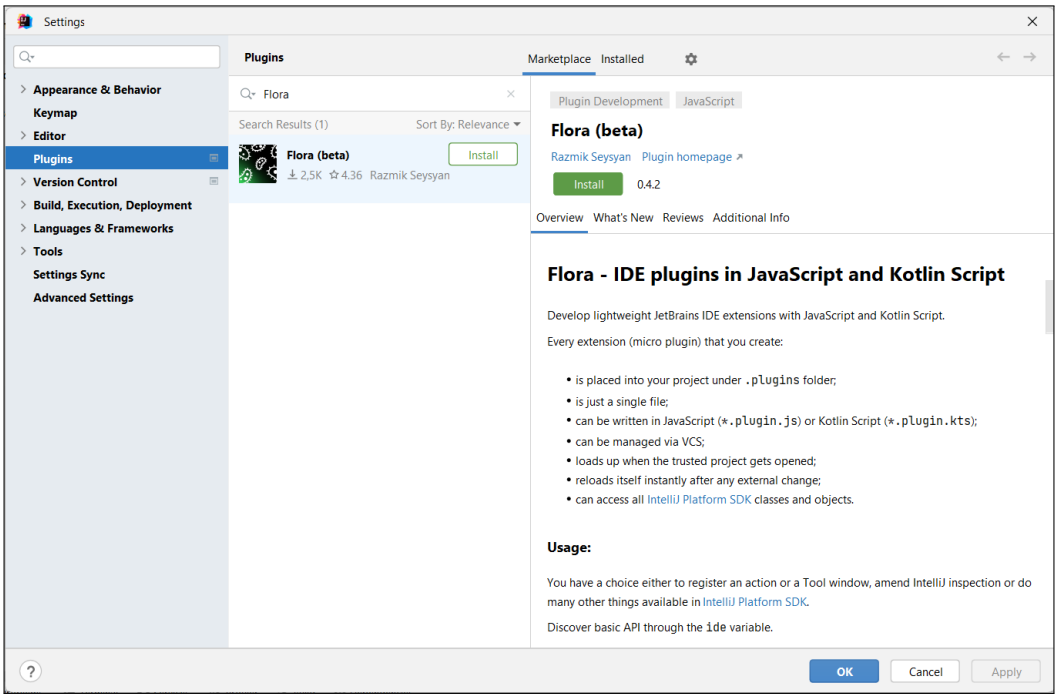


Abbildung 2.3: Flora Plugin im IntelliJ Plugin Marketplace.

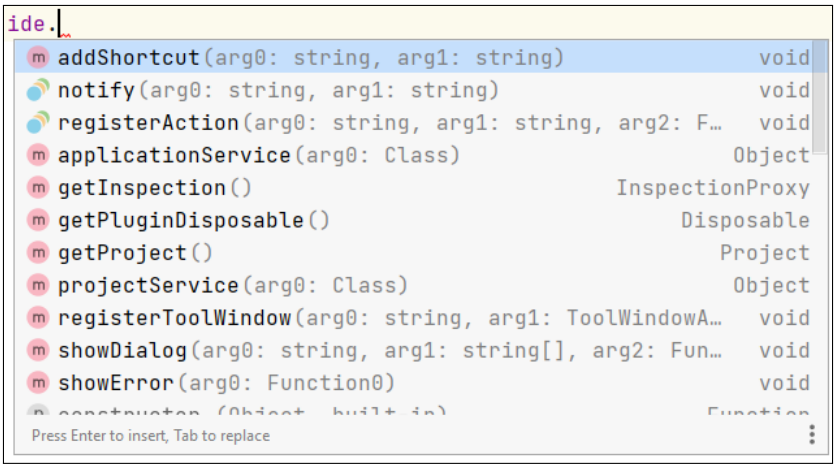


Abbildung 2.4: Übersicht über die API des Flora Plugins.

„micro plugin“-Dateien, die JavaScript oder Kotlin-Quellcode enthalten. Diese müssen sich in einem Ordner namens *.plugins* befinden und auf *.plugin.js* oder *.plugin.kts* enden [64]. Innerhalb dieser 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 werden kann.

Kapitel 3

Anforderungen an den Prototyp

Für den Vergleich der beiden Plugin APIs soll in VS Code und in IntelliJ ein möglichst funktionsgleicher Prototyp implementiert werden.

Durch diese Prototypen können die Features der beiden APIs demonstriert und bewertet werden. Weiters bietet die Entwicklung der beiden Prototypen eine praxisnahe Methode, um sich auch mit dem Arbeitsablauf der beiden Plattformen, speziell auch dem Veröffentlichen der Plugins, vertraut zu machen.

3.1 Aufbau

Um diese Aufgaben möglichst gut zu erfüllen, wurde das sogenannte „RecentChangesPlugin“ entworfen. Es handelt sich dabei um ein Plugin, welches Text- oder Codeänderungen im Editor mitliest und sich merkt. Dadurch wird die spätere Wiederholung von gleichen oder ähnlichen Änderungen erleichtert. Als Änderung wird hierbei das einfache Abändern oder das Ersetzen eines Wortes durch ein anderes verstanden. Das Mitschreiben von komplexeren Veränderungen, zum Beispiel, wenn in mehreren Zeilen gleichzeitig Änderungen vorgenommen werden, ist nicht Ziel des Prototyps.

3.1.1 Beispiel

Zur besseren Demonstration der Funktionalität wurde folgendes Beispiel konstruiert, welches in Abbildung 3.1 und in Abbildung Abbildung 3.2 zu sehen ist:

Eine Person entwickelt gerade ein Programm und erstellt in der Datei *Main.java* einige statische Variablen. Nun bemerkt die Person, dass es sich bei den Variablen *someValue2* und *someValue4* eigentlich um Werte vom Typ *double* handelt. Sie macht daher eine erste Änderung und passt den Datentyp von *someValue2* an. Wie in Abbildung 3.1 zu erkennen ist, bemerkt das Plugin die Änderung und schreibt diese sofort mit.

Die Person möchte nun dieselbe Änderung auch noch einmal für die Variable *someValue4* wiederholen. Sie stellt sich daher mit dem Cursor an die Position im Text, an der das *int* steht. Durch das Drücken einer Tastenkombination wird nun die zuvor erkannte Änderung wieder angewandt und das *int* an der aktuellen Position wird durch ein *double* ersetzt. Dieser Schritt ist in Abbildung 3.2 zu sehen.

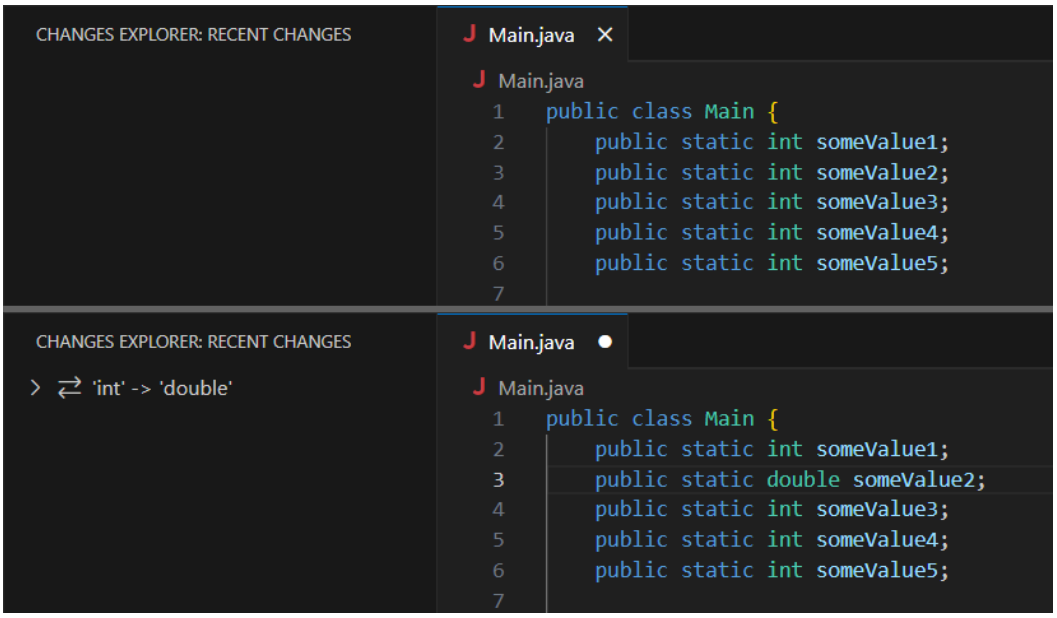


Abbildung 3.1: Mitschreiben einer Änderung durch das *RecentChangesPlugin*.

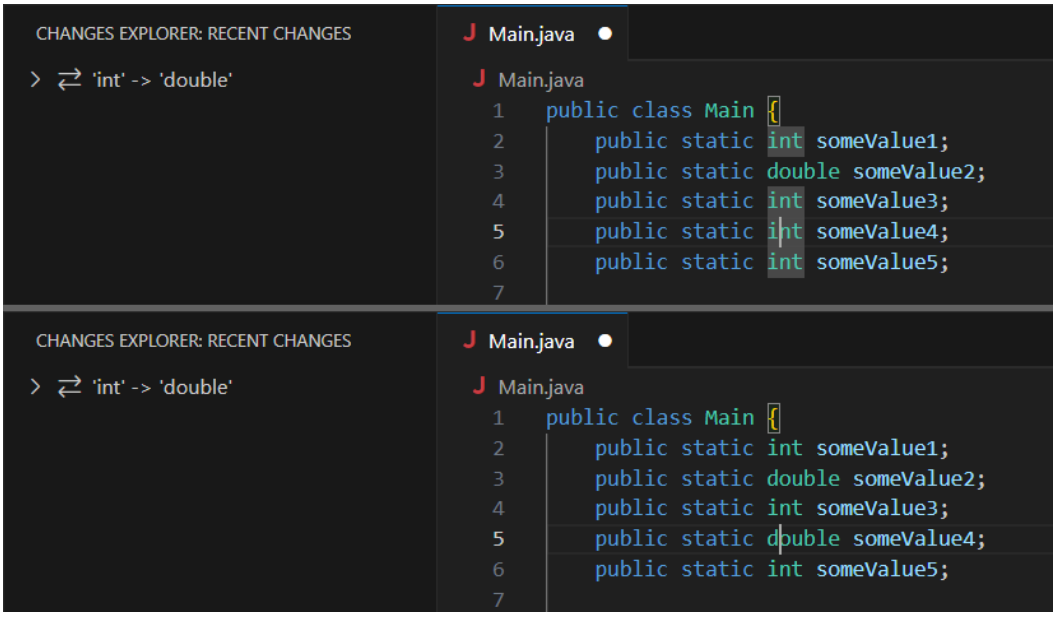


Abbildung 3.2: Erneutes Anwenden einer Änderung durch das *RecentChangesPlugin*.

3.1.2 Funktionen

Für das Plugin wurden folgende Funktionen vorgesehen:

Bemerken von Textänderungen Der Prototyp soll Änderungen in Text- oder Codedateien automatisch erkennen, um sich diese zu merken. Um solche Änderungen zu erkennen, muss auch erkannt werden, wann eine Änderung vollständig abge-

schlossen ist. Hierfür soll ein Debounce-Effekt genutzt werden. Dabei wird nach jedem Tastendruck eine gewisse Zeit abgewartet, ob noch ein weiterer Tastendruck passiert. Erst wenn diese kurze Zeit ohne weitere Tastatureingaben verstreicht, gilt die Änderung als abgeschlossen. Die erkannten Änderungen müssen analysiert und zwischengespeichert werden.

Anwenden von Änderungen auf Befehl Der Prototyp soll zuvor erkannte Änderungen auf Befehl an der aktuellen Position im Editor anwenden können. Dafür muss zuerst die aktuelle Textcursor-Position analysiert werden, um das Wort zu finden, an dem sich der Textcursor befindet. Danach muss in den zwischengespeicherten Änderungen ein passender Eintrag gefunden werden, der auch auf die aktuelle Position angewendet werden kann. Dabei sollen neuere Änderungen gegenüber älteren bevorzugt werden.

Anwenden von Änderungen durch Tastenkombination Der Vorgang des Anwenden von Änderungen soll auch durch eine Tastenkombination auslösbar sein.

Vergessen von alten Änderungen Um erkannte Änderungen nicht ewig im Zwischenspeicher zu halten, soll nur eine bestimmte Menge gleichzeitig gespeichert werden können. Wird diese Menge überschritten, so soll die älteste Änderung aus dem Zwischenspeicher entfernt und somit „vergessen“ werden.

Automatische Codevervollständigung Anhand der kürzlichen Änderungen sollen auch Vorschläge in der Codevervollständigung angezeigt werden. Da eine Änderung ja immer aus einem entfernten Wort und einem ersetzenden Wort besteht, liegt es hier nahe, für die Codevervollständigung nur die ersetzenden Worte vorzuschlagen. Die entfernten Worte werden also ignoriert.

Anzeige der Änderungen Damit die NutzerInnen einen Überblick über die kürzlichen Änderungen haben, sollen alle Änderungen, die sich im Zwischenspeicher befinden, über eine View angezeigt werden können.

Einstellungen Es soll möglich sein, Einstellungen des Plugins festzulegen, welche auch beim Schließen der Entwicklungsumgebunge persistent bleiben.

Es soll gespeichert werden:

- wie viele Änderungen gleichzeitig im Zwischenspeicher gehalten werden und
- welche Debounce-Zeit für das Erkennen von Änderungen verwendet wird.

Tests Es soll eine kleine, demonstrative Menge von Unit- und Integrationstests für den Plugin Code geschrieben werden.

Kapitel 4

Entwicklung des Prototpys für Visual Studio Code

4.1 Design

Um das in Kapitel 3 beschriebene Plugin in VS Code zu entwickeln, werden die Komponenten verwendet, die in Abbildung 4.1 abgebildet sind.

Das Herzstück des Plugins sind der *SimpleChangeHandler* und der *RecentChangeStorage*. Der *SimpleChangeHandler* hat die Aufgabe, alle Veränderungen im geöffneten Dokument zu analysieren. Falls es sich um eine verarbeitbare Veränderung handelt, wird

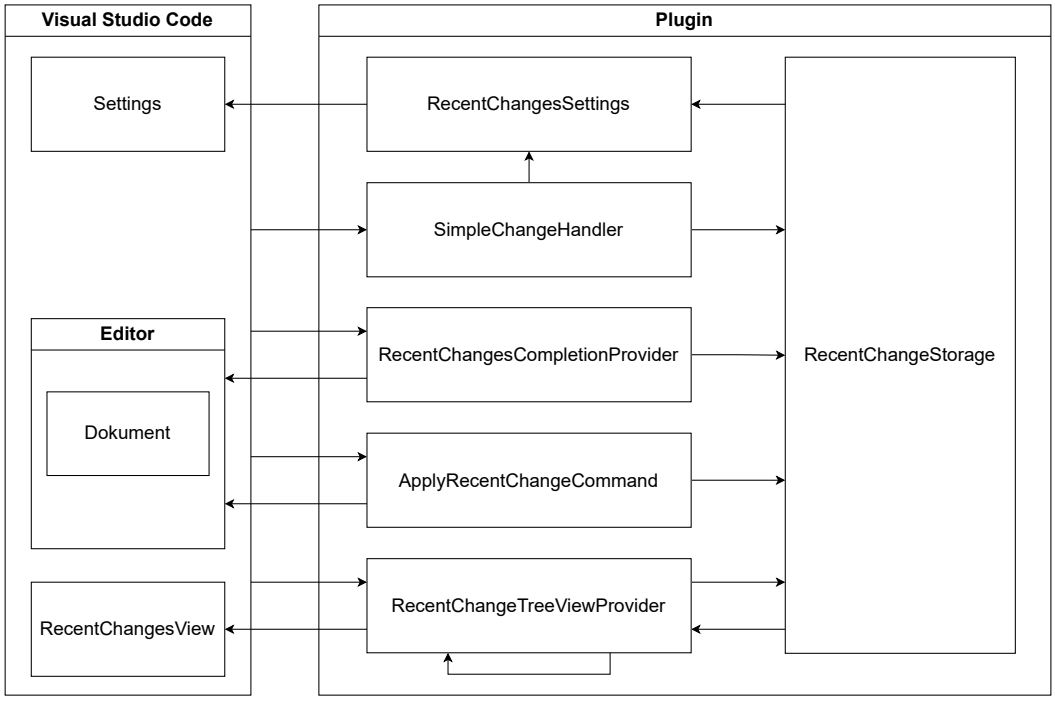


Abbildung 4.1: Vereinfachte Design-Übersicht des Plugins in VS Code.

diese im *RecentChangeStorage* gespeichert. Der *RecentChangeStorage* bietet Methoden, um die gespeicherten Änderungen abzufragen und kümmert sich auch selber um das Löschen von veralteten Änderungen.

Die Klasse *RecentChangesSettings* bietet Zugriffsmethoden an, um die von VS Code angebotenen Einstellungen abzufragen. Über sie werden die Einstellungen für die *QueueSize* (Anzahl von Veränderungen) und die *DebounceTime* (Debounce Zeit für das Erkennen von Änderungen) zugänglich gemacht.

Der *RecentChangesCompletionProvider* implementiert die VS-Code-Schnittstelle für Codevervollständigung. Er analysiert dabei das zu vervollständigende Wort im Editor und fragt daraufhin den *RecentChangeStorage* nach einer passenden Änderung ab.

Der *ApplyRecentChangeCommand* ist ein ausführbarer Command. Er liest das aktuelle Wort aus dem Editor aus, sucht im *RecentChangeStorage* nach einer passenden Änderung und ersetzt das Wort im Editor, falls er fündig wird. Wird keine passende Änderung gefunden, so wird eine entsprechende Nachricht angezeigt. Durch die Konfiguration des Plugins, kann der Command auch über eine Tastenkombination aktiviert werden.

Der *RecentChangesTreeViewProvider* implementiert eine Schnittstelle, um VS Code ein TreeView-Objekt bereitzustellen. Ein solches TreeView wird hier genutzt um die aktuell gespeicherten Änderungen für die BenutzerInnen anzuzeigen. Die Daten für dieses TreeView erhält er vom *RecentChangeStorage*. Damit die TreeView auf dem neuesten Stand gehalten wird, wird der *RecentChangeStorage* mithilfe eines Observer-Patterns [2] beobachtet und die TreeView bei Veränderungen aktualisiert.

4.2 Implementierung

4.2.1 Aufsetzen des Projektes

Um mit der Entwicklung eines VS Code Plugins zu starten, muss zuerst *Node.js* [61] auf dem Gerät installiert sein. Mithilfe von Node.js können das Werkzeug *Yeoman* [99] und ein dazugehöriger Generator mit dem Befehl

```
npm install -g yo generator-code
```

installiert werden. *Yeoman* kann daraufhin ein Plugin-Projekt automatisch über den Befehl

```
yo code
```

erstellen. Während dieses Erstellungsprozesses können verschiedene Einstellungen wie der Name und die Art der Extension festgelegt werden. Generiert wird dann, je nach vorgenommenen Einstellungen, eine Ordnerstruktur, die die wichtigsten Elemente eines Plugins beinhaltet. Diese Ordnerstruktur ist in Abbildung 4.2 zu sehen. So wird mit den Standardeinstellungen ein Projekt angelegt, welches bereits ein Extension Manifest, eine Datei *extension.ts* mit dem Aktivierungs-Code und einige leere Testfälle enthält [98].

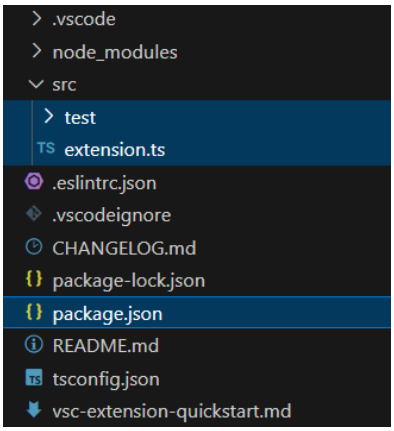


Abbildung 4.2: Durch *Yeoman* generierte Ordnerstruktur.

4.2.2 Entwicklung

Klasse RecentChangeStorage

Die Klasse *RecentChangeStorage*, die in Abbildung 4.3 abgebildet ist, wird von fast allen anderen Komponenten benötigt. Sie speichert ihre Daten in einer sogenannten *EvictingQueue*. Hierfür gibt es in TypeScript keine mitgelieferte Implementierung, allerdings kann diese sehr einfach über die Methode *.shift()* eines Arrays nachgebaut werden. So werden Elemente am Ende der Schlange automatisch wieder entfernt, sobald sich die Schlange füllt. In der gespeicherten Warteschlange werden Objekte der Klasse *SimpleDiff* gespeichert. Diese bilden jeweils eine Änderung ab, indem sie den entfernten und den ersetzenden Text speichern. Der Zugriff auf die *EvictingQueue*, also das Hinzufügen, das Auslesen und das Verändern der Anzahl von gespeicherten Änderungen, wird durch verschiedene Methoden bereitgestellt. Zusätzlich ist relevant, dass *RecentChangeStorage* von der Schnittstelle *EventTarget* ableitet. Durch solche Events kann ein Observer Pattern implementiert werden. Das *storageChanged*-Event wird dabei immer ausgelöst, wenn sich der Inhalt der Warteschlange verändert. Initialisiert wird der *RecentChangeStorage* zu Beginn der Methode *activate*, in der Datei *extension.ts*. So kann das erstellte Objekt allen anderen Komponenten über den Konstruktor übergeben werden.

Klasse SimpleChangeHandler

Die Klasse *SimpleChangeHandler*, deren Aufbau auch in Abbildung 4.4 ersichtlich ist, enthält mehrere Listener-Methoden, die das Öffnen von Dateien, das Bearbeiten von Dateien und das Wechseln des aktiven Editors beobachten. Diese Listener werden in der Methode *activate* registriert. Durch diese Listener kann der aktuelle Zustand der geöffneten Datei, sowie ihr Zustand zu Beginn einer Änderung, ständig beobachtet werden. Während die NutzerInnen die Datei bearbeiten, wird immer wieder ein Timer gestartet. Erst wenn der Timer, ohne einer Eingabe in der Zwischenzeit, abläuft wird eine Änderung registriert. Auf diese Weise wird ein Debounce-Effekt erzeugt. Sobald eine Änderung registriert wurde, werden der Ausgangszustand und der neue Zustand der Datei verglichen. Hierfür wird Googles *diff-match-patch*-Algorithmus [12] verwen-

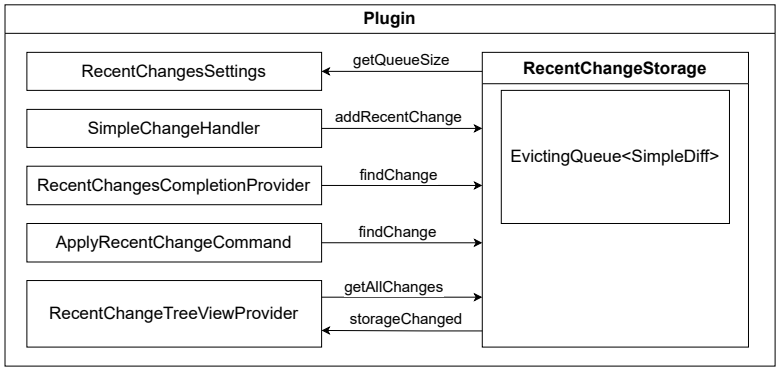


Abbildung 4.3: Detaillierte Darstellung des *RecentChangeStorage*.

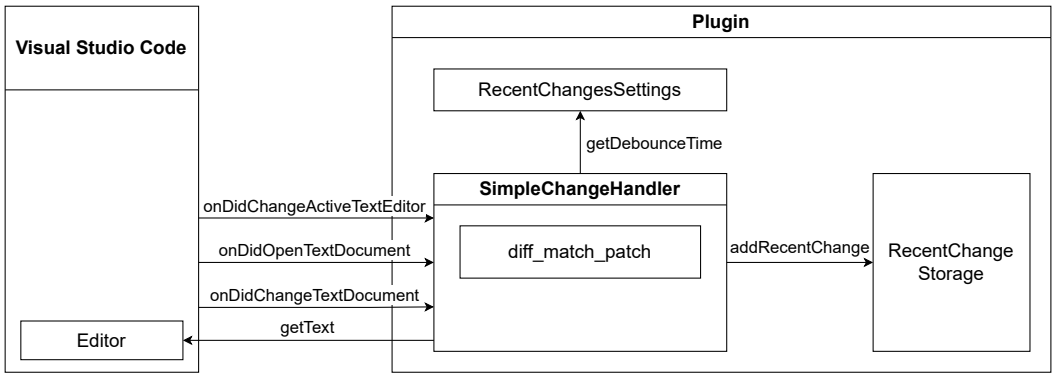


Abbildung 4.4: Detaillierte Darstellung des *SimpleChangeHandler*.

det, der auch komplexe Veränderungen in Texten erkennen kann. Immer wenn durch den Algorithmus eine einfache Veränderung gefunden wurde, wird diese in Form eines *SimpleDiff*-Objekts in den *RecentChangeStorage* eingefügt.

Klasse ApplyRecentChangeCommand

Die Klasse *ApplyRecentChangeCommand*, welche in Abbildung 4.5 dargestellt ist, enthält eine einfache Methode, die bei Aktivierung des Commands aufgerufen wird. Sie prüft zuerst einfache Bedingungen, zum Beispiel, ob ein Editor aktiv ist. Danach selektiert sie Mithilfe der Methode *getWordRangeAtPosition*, welche von der VS Code API bereitgestellt wird, das zu ersetzende Wort. Für dieses Wort wird der *RecentChangeStorage* nach einer passenden Änderung durchsucht. Wird diese gefunden, so werden die zu ersetzenden Positionen berechnet und dann im Editor ersetzt. Um den Command auch verwendbar zu machen, müssen zusätzlich ein Command und ein Keybinding in der Datei *package.json* registriert werden. Die Zuordnung der auszuführenden Methode erfolgt in der Methode *activate* durch Registrierung.

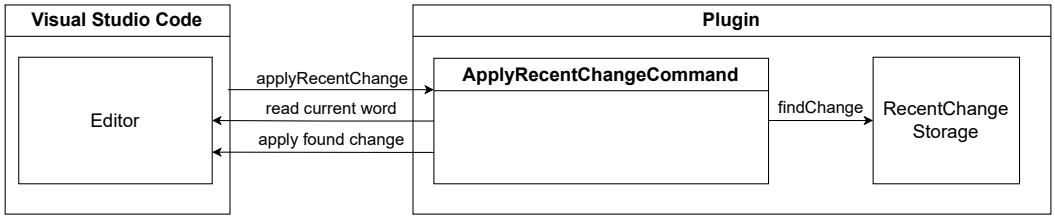


Abbildung 4.5: Detaillierte Darstellung des *ApplyRecentChangeCommand*.

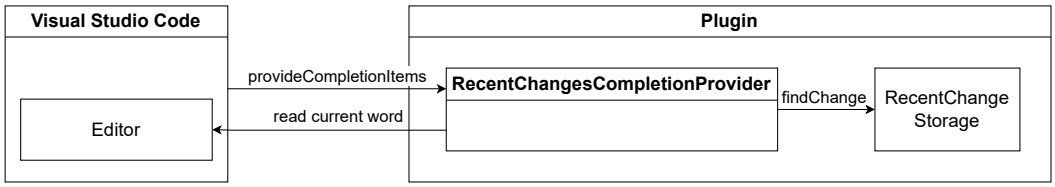


Abbildung 4.6: Detaillierte Darstellung des *RecentChangesCompletionProvider*.

Klasse RecentChangesCompletionProvider

Die Logik der Klasse *RecentChangesCompletionProvider*, die in Abbildung 4.6 dargestellt wird, ist ähnlich zu der des *ApplyRecentChangeCommand*, da im Grunde nur für die aktuelle Position im Editor nach einer passenden Änderung gesucht wird. Allerdings muss die Änderung hier nicht sofort angewendet werden, sondern nur, in Form von *CompletionItem*-Objekten, retourniert werden. Das Registrieren des Providers wird durch Aufruf der Methode *activateRecentChangesCompletionProvider* bei der Aktivierung des Plugins erledigt. Hier muss bei der Registrierung auch ein Pattern für die Dateien angegeben werden, auf die der Provider anwendbar ist.

Klasse RecentChangeTreeViewProvider

Das TreeView, für die Darstellung der gespeicherten Änderungen, wird von der Klasse *RecentChangeTreeViewProvider* bereitgestellt, die in Abbildung 4.7 abgebildet ist. Um das TreeView bereitzustellen, muss die Schnittstelle *TreeDataProvider*<*T*> implementiert werden. Diese enthält eine Methode, um alle Kinder eines *T*-Elements zu erhalten und eine Methode um ein *T*-Element in ein *TreeItem* umzuwandeln. Um die zweite Methode etwas zu vereinfachen, kann eine Klasse *SimpleDiffTreeItem* direkt von *TreeItem* abgeleitet werden. Jedes TreeItem definiert bestimmte Werte wie zum Beispiel eine Beschriftung (*label*), ein Symbol (*iconPath*) oder einen Hinweis (*tooltip*). Die *SimpleDiffTreeItem*-Objekte enthalten zusätzliche Metainformationen über ihre Position im Baum. Um die Baumstruktur neu zu laden, kann das Event *_onDidChangeTreeData* aktiviert werden. Dies muss immer dann passieren, wenn vom *RecentChangeStorage* das *storageChanged*-Event gefeuert wird. Um die TreeView auch anzuzeigen, müssen in der Datei *package.json* sowohl ein ViewContainer als auch eine dazugehörige View registriert werden. In der Methode *activate* wird die Provider-Klasse mittels des Aufrufes von *vscode.window.createTreeView* dem registrierten View zugeordnet.

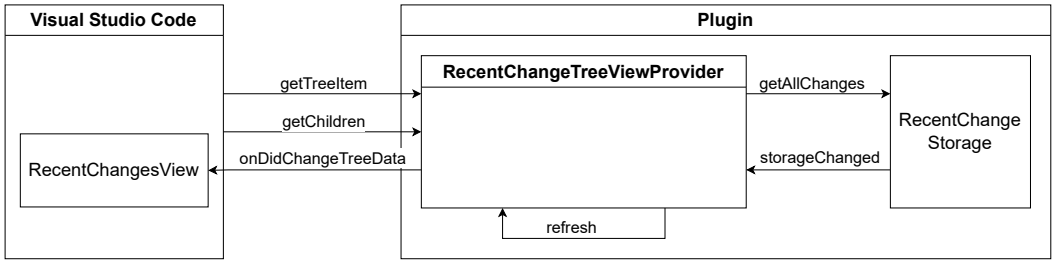


Abbildung 4.7: Detaillierte Darstellung des *RecentChangeTreeViewProvider*.

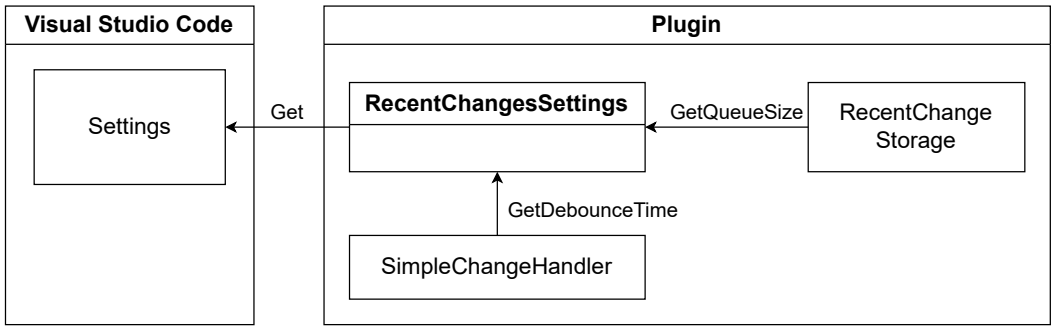


Abbildung 4.8: Detaillierte Darstellung der Komponente *RecentChangesSettings*.

Klasse RecentChangesSettings

Die Einstellungen der Extension werden hauptsächlich in der Datei *package.json* festgelegt. Hier werden die Namen, Typen, Beschreibungen und Standardwerte der einzelnen Einstellungen bestimmt. Die Klasse *RecentChangesSettings* enthält nur statische Methoden, welche das Auslesen der Einstellungen übernehmen und von überall im Code aufgerufen werden können. Diese Klasse ist auch in der Abbildung 4.8 ersichtlich.

4.3 Tests

Beim Erstellen einer neuen Extension wird im Ordner *src* auch automatisch ein Ordner *test* angelegt. In den generierten Dateien befindet sich eine Konfiguration für das Testframework Mocha [60], die bereits ausführbare Tests enthält. Natürlich kann Mocha auch durch andere Test-Frameworks ersetzt werden. Die Testfälle nutzen eine Schnittstelle aus dem Paket *@vscode/test-electron* [59]. Diese führt die Tests innerhalb einer speziellen Instanz von VS Code namens *Extension Development Host* aus. Auf diese Weise kann auch während der Tests auf das Modul *vscode* zugegriffen werden. Dadurch ist es nicht nur möglich, einfache Unit-Tests zu entwickeln, sondern es können auch größer angelegte Integrationstests geprüft werden, die eine Interaktion mit VS Code benötigen [94].

4.4 Publishing

Das Veröffentlichen einer VS Code Extension funktioniert über das Komandozeilenwerkzeug *vsce*. Dieses kann mithilfe des Befehls

```
npm install -g @vscode/vsce
```

installiert werden. Das fertige Plugin kann daraufhin mit

```
vsce package
```

verpackt und mit

```
vsce publish
```

in den VS Code Marketplace hochgeladen werden [88].

Beim Hochladen ist zu beachten, dass ein *personal access token* und ein *publisher* Name benötigt werden. Um den Token zu erhalten, muss man zuerst in Azure DevOps eine neue Organisation anlegen (sofern man nicht bereits Teil einer Organisation ist). In den Benutzereinstellungen kann ein neuer Token erstellt werden. Dabei müssen auch die Berechtigungen für den Token festgelegt werden. Bei dem Abschnitt für *Marketplace* muss hier die Berechtigung *Manage* gesetzt sein. Ein Publisher kann erstellt werden, indem man sich im Marketplace anmeldet und die Seite https://marketplace.visualstudio.com/manage aufruft. Hier gibt es einen Dialog zum Erstellen eines neuen Publishers. Dieser Publisher benötigt mindestens eine eindeutige ID und einen eindeutigen Namen, welche später im Marketplace angezeigt werden. Der Publisher Name muss weiters noch in der Datei *package.json* eingetragen werden.

Das Verwalten von bereits hochgeladenen Extensions kann entweder über das Werkzeug *vsce* oder über die grafische Benutzeroberfläche im Marketplace erledigt werden.

4.5 CI/CD

Die Einbindung von automatisiertem Testen und Veröffentlichen einer VS Code Extension ist unkompliziert, da es hierzu ausführliche Dokumentation gibt. Die Dokumentation bietet weiters vorgefertigte Beispiele für Azure Pipelines, GitHub Actions, GitLab CI und Travis CI [75].

Im Zuge der Entwicklung wurde mit GitHub Actions gearbeitet. Der vorgeschlagene Workflow testet die Anwendung dabei unter MacOS, Ubuntu und Windows. Beim automatischen Deployment muss beachtet werden, dass in der Datei *package.json* im Abschnitt *„scripts“* die zusätzliche Zeile

```
"deploy": "vsce publish"
```

nötig ist. Weiters muss beachtet werden, dass auch der zuvor erstellte *personal access token* benötigt wird (wie beschrieben in Abschnitt 4.4). Dieser muss im Workflow als Umgebungsvariable gesetzt werden und wird, im Falle von GitHub Actions, am besten als verschlüsseltes *secret* gespeichert.

Kapitel 5

Entwicklung des Prototpys für IntelliJ

5.1 Design

Das beschriebene Plugin setzt sich in IntelliJ durch die Komponenten zusammen, die in Abbildung 5.1 abgebildet sind.

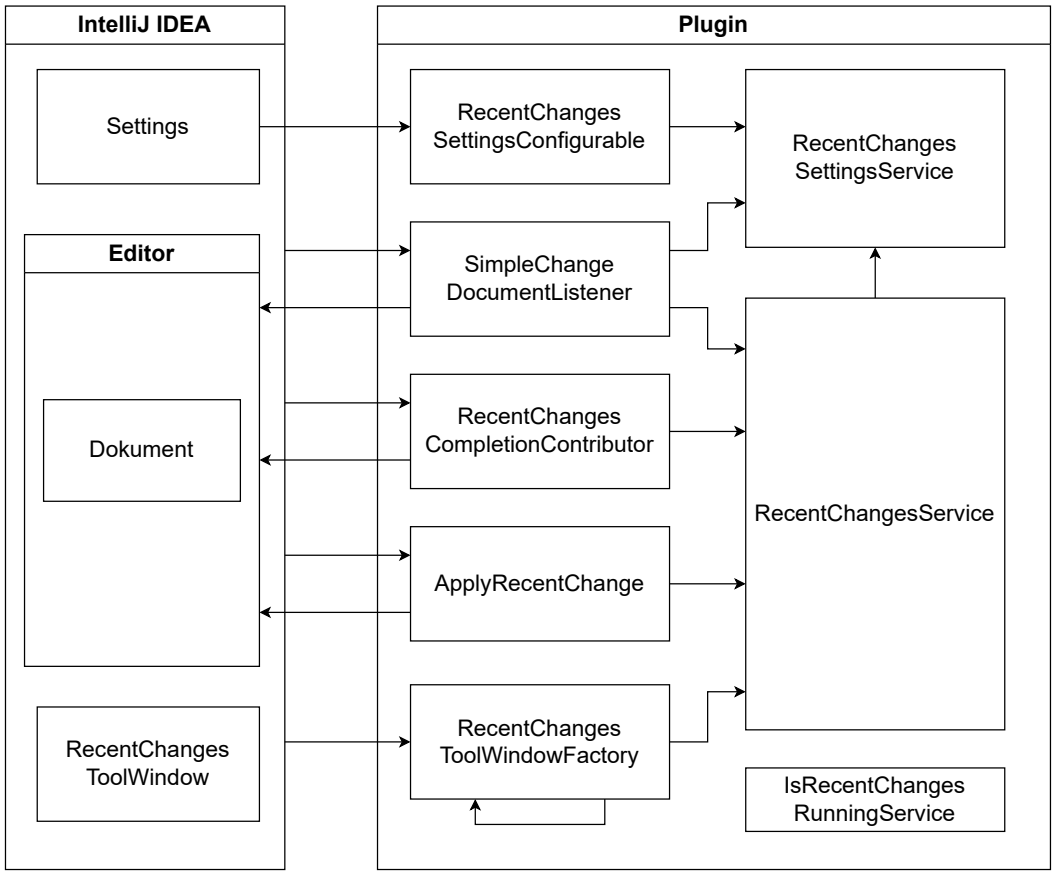


Abbildung 5.1: Vereinfachte Design-Übersicht des Plugins in IntelliJ.

Da die beiden Plugins im Aufbau sehr ähnlich sind, sind auch die Aufgaben der Einzelnen Komponenten beinahe deckungsgleich. Die Unterschiede liegen eher in den Details.

Der *SimpleChangeDocumentListener* übernimmt hier die Aufgabe des Beobachten des geöffneten Dokuments auf Veränderungen. Wird eine Änderung festgestellt, so wird diese an den *RecentChangesService* zur Speicherung übergeben. Im Gegensatz zum *RecentChangeStorage* ist dieser allerdings als expliziter Service deklariert, der von IntelliJ selbst verwaltet wird.

Die Komponenten für Einstellungen teilen sich auf *RecentChangesSettingsConfigurable* und *RecentChangesSettingsService* auf. Die Klasse *RecentChangesSettingsConfigurable* kümmert sich dabei um die Darstellung und die Interaktivität der Einstellungen in der Benutzerschnittstelle. Die Klasse *RecentChangesSettingsService* wird wieder als Service angeboten und kümmert sich um das Speichern, Auslesen und Persistieren der Einstellungen.

Die Codevervollständigung wird im IntelliJ Plugin durch den *RecentChangesCompletionContributor* durchgeführt. Dieser sucht im *RecentChangesService* nach Änderungen, die auf das ausgewählte Wort passen und gibt diese als Vorschläge zurück.

Für das Einsetzen der Änderungen im Editor wird die Klasse *ApplyRecentChange* verwendet, die als Action registriert ist. Bei der Action ist wie bereits im VS Code Plugin eine Tastenkombination zum schnelleren Aufrufen hinterlegt. Sollte keine passende Änderung gefunden werden, so wird direkt im Editor ein Warnhinweis mit einer entsprechenden Meldung angezeigt.

Die Klasse *RecentChangesToolWindowFactory* ist für die Darstellung einer UI-Komponente zuständig, die dem TreeView aus dem VS Code Plugin ähnelt. Sie liest dafür den Zustand des *RecentChangesService* aus und baut eine entsprechende Baumstruktur auf. Um die Ansicht auch zum richtigen Zeitpunkt aktualisieren zu können, wird der Service mithilfe eines Observer-Patterns [2] beobachtet.

Die Komponente *IsRecentChangesRunningService* existiert für den Fall, das zwei Instanzen von IntelliJ gleichzeitig auf einem Gerät gestartet werden. Beim Start der Anwendung müssen nämlich einige Initialisierungen vorgenommen werden, die nur einmalig durchgeführt werden dürfen. Gegen den *IsRecentChangesRunningService* kann auf diese Weise geprüft werden, ob die Initialisierung noch nötig ist oder bereits gemacht wurde.

5.2 Implementierung

5.2.1 Aufsetzen des Projektes

Das Erstellen eines neuen Plugin-Projektes kann in IntelliJ über den „New Project Wizard“ erledigt werden [24]. Dafür muss einfach im Programm IntelliJ IDEA das Menü *File -> New -> Project...* gewählt werden. Hier kann in der Liste der Projekt-Vorlagen auch ein Eintrag für ein *IDE Plugin*-Projekt gefunden werden. Für diesen Projekttyp kann dann unter anderem ein Name für das Projekt gewählt werden. Dieser Name wird initial auch als Name des Plugins verwendet.

Die durch IntelliJ generierte Ordnerstruktur ist (ausschnittsweise) in Abbildung 5.2 dargestellt. Relevant ist hier vor allem die Datei *plugin.xml*, die das entsprechend vorbe-

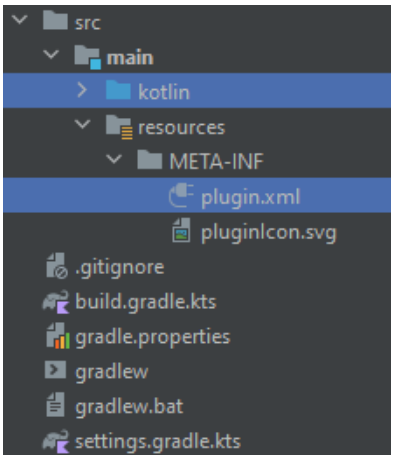


Abbildung 5.2: Ausschnitt der durch *IntelliJ IDEA* generierten Ordnerstruktur.

reitete Plugin Manifest beinhaltet. Der Ordner *kotlin* ist als Verzeichnis für den Plugin-Code vorgesehen. Für die Implementierung des *RecentChangesPlugin* wurde dieser allerdings durch einen Ordner *java* ersetzt. Ein Ordner für Tests wird nicht automatisch generiert und muss manuell hinzugefügt werden.

5.2.2 Entwicklung

RecentChangesService

Die Klasse *RecentChangesService* (dargestellt in Abbildung 5.3) wird in der Form eines Services auf Applikationsebene implementiert. Dies wird über das Attribut *@Service(Service.Level.APP)* festgelegt. Auf diese Weise kann eine Instanz der Klasse in der gesamten Anwendung bereitgestellt werden. Die statische Methode *getInstance* abstrahiert dabei die Aufrufe der IntelliJ API, die nötig sind, um eine Referenz auf diese Instanz zu erhalten. Genau wie im VS Code Plugin, werden die Änderungen in einer *EvictingQueue* von *SimpleDiff*-Objekten gespeichert. Allerdings muss diese Warteschlange hier nicht selbst implementiert werden, da es in der Bibliothek *Guava* von Google bereits eine Implementierung gibt [13]. Diese Bibliothek kann in der Datei *build.gradle.kts* eingebunden werden. Die verschiedenen Methoden der Klasse *RecentChangesService* erlauben das Einfügen sowie das Auslesen von *SimpleDiff*-Objekten aus der darunterliegenden Datenstruktur. Über die Methoden *addChangeListener*, *removeChangeListener* und *notifyListeners* wird ein Observer-Pattern abgebildet. Observer, welche die eigens erstellte Schnittstelle *RecentDiffsChangedListener* implementieren, können sich also beim *RecentChangesService* anmelden. Sie werden dann bei Änderungen an den Daten über die Methode *notifyChanged* notifiziert. Da es sich bei der Klasse *RecentChangesService* aufgrund der Implementierung als IntelliJ Service um eine Singleton-Klasse [2] handelt, wird zusätzlich eine Methode *reset* benötigt, um beim Testen die Unabhängigkeit der verschiedenen Unit-Tests zu erhalten. Da für die einzelnen Testfälle keine neuen Instanzen erzeugt werden können, muss die Klasse also vor jedem Test zurückgesetzt werden.

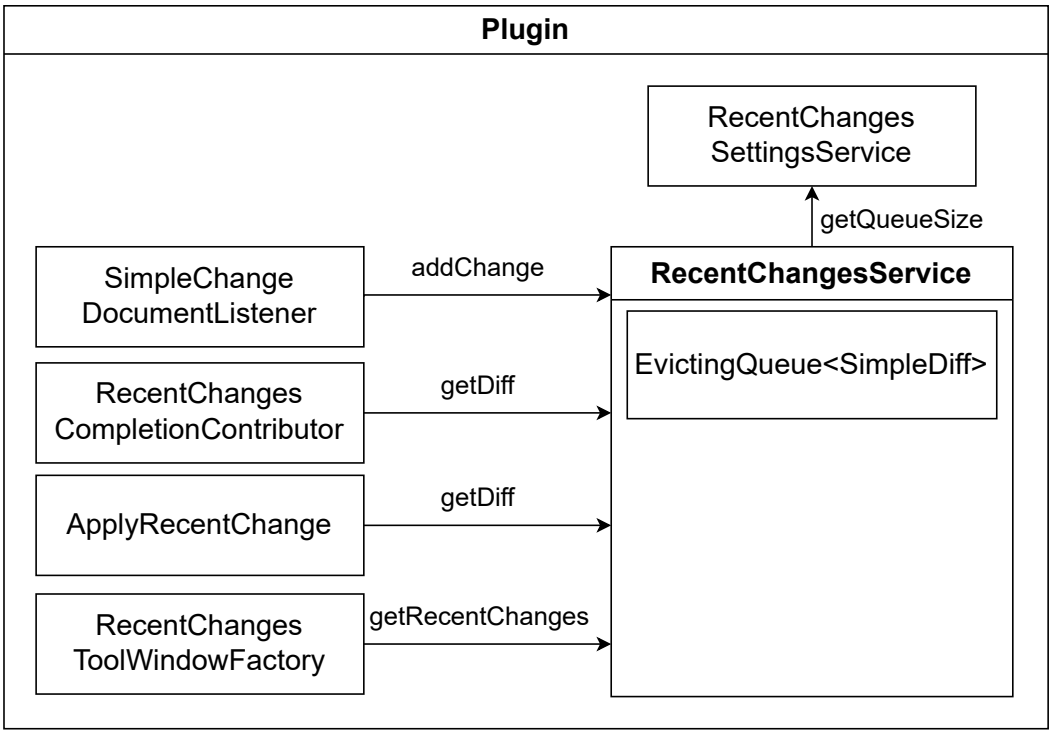


Abbildung 5.3: Detaillierte Darstellung des *RecentChangesService*.

SimpleChangeDocumentListener

Der detaillierte Aufbau der Komponente *SimpleChangeDocumentListener* kann in Abbildung 5.4 betrachtet werden. Eine Instanz der Klasse wird beim ersten Start von IntelliJ so registriert, dass er über die Änderungen in *allen* geöffneten Dokumenten informiert wird. Hierfür muss die Schnittstelle *DocumentListener* implementiert werden. Diese definiert unter anderem die Methodensignaturen *beforeDocumentChange* und *documentChanged*. Durch das Zusammenspiel dieser beiden Methoden wird der gewünschte Debounce-Effekt erzeugt. Zu Beginn einer Änderung wird in der Methode *beforeDocumentChange* der aktuelle Text aus der unveränderten Datei ausgelesen. In der Methode *documentChanged* wird dann ein Timer gestartet (oder neu gestartet, falls er bereits laufen sollte). Erst nach Ablauf des Timers (ohne eine weitere Eingabe) wird eine Änderung als abgeschlossen erkannt. Sobald dies geschieht, wird der Algorithmus *diff-match-patch* verwendet, um die Änderung zu analysieren. Wird daraufhin festgestellt, dass es sich um eine einfache Änderung handelt, so wird diese in den Speicher des *RecentChangesService* eingefügt. Zu beachten ist in dieser Klasse weiters dass ein Aufruf der privaten Methode *getOriginalTextFromDocument* nötig ist. Die Klasse *Document* hätte zwar eigentlich eine Methode *getText*, dieser Text befindet sich aber möglicherweise in einem Zwischenzustand, in dem IntelliJ spezielle Zeichenketten einsetzt, um die Anzeige der Codevervollständigung zu erleichtern [16, 55]. Um den originalen Dateiinhalt (und nicht eine modifizierte Kopie) zu erhalten, müssen also einige Umwege gegangen werden. Das Problem wurde gelöst, indem diese speziellen Zeichenketten einfach aus dem Text her-

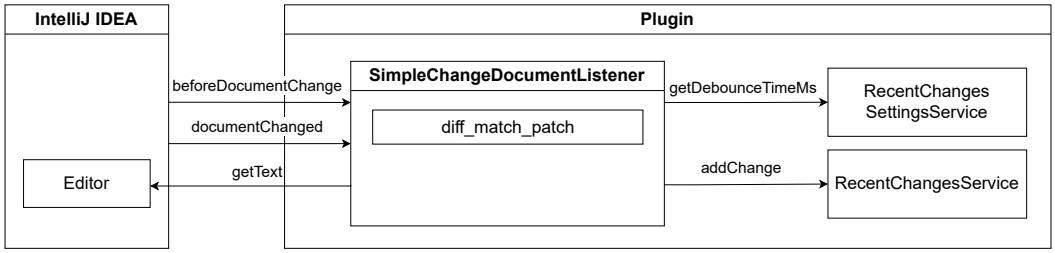


Abbildung 5.4: Detaillierte Darstellung des *SimpleChangeDocumentListener*.

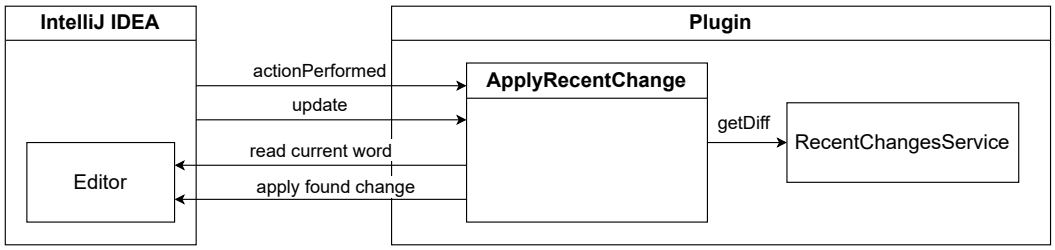


Abbildung 5.5: Detaillierte Darstellung der Action *ApplyRecentChange*.

ausgefiltert und somit ignoriert werden.

ApplyRecentChange

Bei *ApplyRecentChange* handelt es sich um eine von *AnAction* abgeleitete Klasse. In Abbildung 5.5 sind die Interaktionen der Klasse dargestellt. Die Methoden *actionPerformed* und *update* werden von IntelliJ aufgerufen. In beiden Methoden muss zuerst der Inhalt der Datei an der aktuellen Position gelesen werden, um herauszufinden, ob das Anwenden der Action momentan möglich ist. Sollte das Anwenden nicht möglich sein, wird in der Methode *update* mithilfe des übergebenen *AnActionEvent*-Objekt die Sichtbarkeit der Action (in den registrierten Menüs des User Interface) gesetzt. In der Methode *actionPerformed* wird in einem solchen Fall eine Fehlermeldung mithilfe des IntelliJ Services *HintManager* angezeigt. Sollte ein Ersetzen möglich sein, so wird mittels der statischen Methode *WriteCommandAction.runWriteCommandAction* ein Schreibbefehl abgesetzt, in welchem der zu ersetzende Inhalt des Dokuments ausgetauscht wird. Dies ist nur über einen solchen Schreibbefehl möglich, da ansonsten Synchronisationsprobleme auftreten könnten [28, 46]. Im Manifest des Plugins wird die Action mit der Tastenkombination *alt + R* registriert, um auch Aufrufbar zu sein.

RecentChangesCompletionContributor

Die Klasse *RecentChangesCompletionContributor*, die in Abbildung 5.6 dargestellt wird, leitet von *CompletionContributor* ab. Im Konstruktor muss die Methode *extend* aufgerufen werden, der eine Instanz des eigentlichen *CompletionProvider* übergeben wird. Dieser *CompletionProvider* überschreibt wiederum die Methode *addCompletions*, welche die eigentliche Arbeit erledigt. Genau wie bei der Action *ApplyRecentChange*, wird

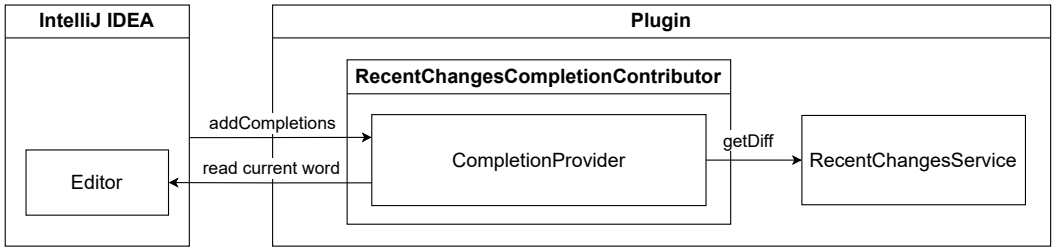


Abbildung 5.6: Detaillierte Darstellung des *RecentChangesCompletionContributor*.

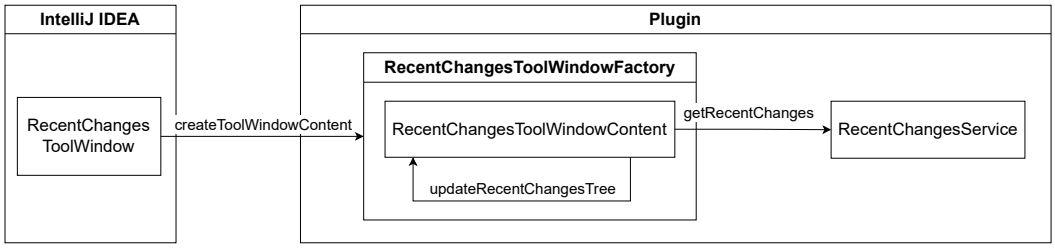


Abbildung 5.7: Detaillierte Darstellung der *RecentChangesToolWindowFactory*.

der Inhalt der Datei an der aktuellen Position ausgelesen. Das gefundene Wort wird im *RecentChangesService* nachgeschlagen. Falls eine passende Änderung gefunden wird, wird diese über den Parameter *resultSet* zu den Vorschlägen für die Codevervollständigung hinzugefügt. Im Manifest wird der Contributor für die Sprache *any* registriert, um sprachunabhängig zu funktionieren.

RecentChangesToolWindowFactory

Die Klasse *RecentChangesToolWindowFactory* ist in Abbildung 5.7 dargestellt. Sie implementiert die Schnittstelle *ToolWindowFactory* und überschreibt die Methode *createToolWindowContent*. Zur besseren Abgrenzung der Aufgaben wird in dieser zuerst eine neue Instanz der Klasse *RecentChangesToolWindowContent* erstellt. Diese innere Klasse verwaltet den Inhalt und das Aussehen des Fensters. Sie lädt die nötigen Daten aus dem *RecentChangesService* und meldet sich bei dem Service auch als Listener an, um gegebenenfalls den Inhalt zu aktualisieren. Die Darstellung des Fensterinhalts geschieht über ein *JPanel* aus Java Swing und einem *Tree* aus der UI-Bibliothek von IntelliJ. Dieser Inhalt wird durch die Methode *getContentPanel* von außen zugänglich gemacht. Zum Schluss setzt die Methode *createToolWindowContent* den eigentlichen Inhalt des Fensters durch den Aufruf von *toolWindow.getContentManager().addContent* auf dem übergebenen Parameter. Im Manifest wird *RecentChangesToolWindowFactory* als Factory-Klasse für ein ToolWindow namens *Recent Changes* registriert.

Einstellungen

Die für die Einstellungen nötigen Klassen sind in Abbildung 5.8 zu sehen. *RecentChangesSettingsConfigurable* ist die Hauptschnittstelle zu IntelliJ, die auch im Manifest regis-

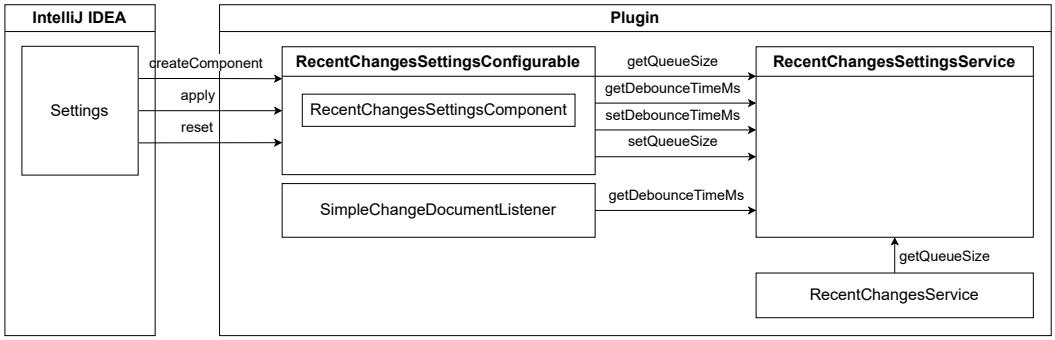


Abbildung 5.8: Detaillierte Darstellung der *Settings* Komponenten.

triert ist. Sie implementiert das Interface *Configurable* und bietet verschiedene Methoden an, die für die Darstellung und Interaktivität in der Benutzerschnittstelle nötig sind. Durch die Methode *apply* sollen die aktuell gewählten Einstellungen persistiert werden, durch die Methode *reset* die zuvor gespeicherten Einstellungen wieder angezeigt werden. Durch die Methode *isModified* wird überprüft, ob die angezeigten Einstellungen von den persistierten Einstellungen abweichen. Ist dies nicht der Fall, wird der Knopf zum Speichern der Einstellungen automatisch ausgegraut. Die Methode *createComponent* stellt den Inhalt der Einstellungsseite in Form eines *JComponent* bereit. Um gute Aufgabenteilung zu ermöglichen, ist diese Darstellung an die Klasse *RecentChangesSettingsComponent* ausgelagert. Sie beinhaltet Felder zum Eingeben der Warteschlangengröße und der Debouncezeit. Über Getter- und Setter-Methoden können diese Werte von *RecentChangesSettingsConfigurable* beliebig verwaltet werden. Um das Eingeben von ungültigen Werten zu vermeiden, werden zusätzlich Filter auf die Eingabefelder angewandt. Der *RecentChangesSettingsService* ist die Komponente die für das Persistieren der Einstellungen verantwortlich ist. Hierfür wird die Schnittstelle *PersistentStateComponent* implementiert und das Attribut *@State* angegeben. Auf diese Weise ruft IntelliJ zu den passenden Zeitpunkten automatisch die Methoden *getState* und *loadState* auf und serialisiert den Zustand in einer *.xml*-Datei. Zusätzlich ist die Klasse im Manifest als Service registriert, und bietet passende Methoden zum Auslesen und Setzen sowie ein Observer-Pattern zum Beobachten der Einstellungen an.

5.3 Tests

Für Tests eines IntelliJ-Plugins werden von JetBrains die Frameworks *JUnit*, *TestNG* und *Cucumber* empfohlen [50]. Alle Tests für das IntelliJ Plugin laufen innerhalb einer sogenannten *headless Umgebung* [48]. Das bedeutet, es wird eine echte Instanz des IntelliJ IDEA zum Testen verwendet. Dieses wird allerdings ohne einer Benutzerschnittstelle gestartet und wird daher nicht angezeigt. Als Hauptschnittstelle für die Tests bietet IntelliJ die Klassen *BasePlatformTestCase* und *HeavyPlatformTestCase* [32]. Bei Test-Klassen, die von *HeavyPlatformTestCase* ableiten, handelt es sich um *Heavy*-Tests. Diese erstellen in der Testumgebung für jeden Testfall ein neues (temporäres) Projekt, in welchem das Plugin arbeiten kann. Da das Erstellen solcher Projekte allerdings sehr aufwändig ist, führt das Einsetzen solcher Tests zu längeren Ausführungszeiten. Bei

Test-Klassen, die von *BasePlatformTestCase* ableiten, handelt es sich um *Light*-Tests. Diese sind darauf optimiert, möglichst effizient abzulaufen und versuchen, wenn möglich, das erstellte Projekt von vorherigen Tests weiter zu verwenden. Dadurch wird zwar an Geschwindigkeit gewonnen, allerdings muss speziell darauf geachtet werden, dass keine Abhängigkeiten zwischen den Testfällen entstehen. Sollten diese beiden Klassen zu einschränkend sein, kann auch die Klasse *IdeaTestFixtureFactory* verwendet werden [50]. Mit dieser müssen allerdings einige *setup*-Methoden manuell aufgerufen werden und man hat einen höheren Konfigurationsaufwand. Innerhalb der Testfälle kann beliebig mit der IntelliJ API interagiert werden. Es gibt sogar zusätzliche Methoden wie *copyFileToProject*, *type*, *performEditorAction* [49, 53]. Diese erleichtern das Arbeiten mit der headless Umgebung und erlauben es zum Beispiel, Tastatureingaben zu simulieren. Wenn Dateien in das Testprojekt geladen werden, kann spezielles Markup verwendet werden. So kann beispielsweise mit *<caret>* die Position markiert werden, an der sich der Cursor befinden soll.

5.4 Publishing

Um ein Plugin in IntelliJ zu veröffentlichen, gibt es mehrere Voraussetzungen [38]. Man benötigt einen Account im JetBrains Marketplace und ein Zertifikat zum Signieren des Plugins. Der Account kann über die Seite https://plugins.jetbrains.com/ erstellt werden. Das Generieren eines Zertifikats und den dafür benötigten Schlüsseln geht mithilfe des Programms *openssl* und ist in der IntelliJ Dokumentation beschrieben.

Im ersten Schritt muss das Plugin gebaut werden. Dies ist über den Gradle Task *buildPlugin* möglich, welcher das Plugin baut, und das Ergebnis im Ordner *build/distributions* als *.zip*-Datei speichert.

Um das gebaute Plugin zu signieren, können die im Projekt bereits vordefinierten Gradle Tasks genutzt werden. Hierfür muss allerdings die Datei *build.gradle.kts* angepasst werden. Im Abschnit *signPlugin* müssen das zuvor generierte Zertifikat, der dazugehörige Privatschlüssel und das Passwort, mit welchem der Schüssel erstellt wurde, angegeben werden. Um diese sensiblen Daten nicht unabsichtlich zu veröffentlichen, empfiehlt es sich hier auf Umgebungsvariablen des Systems zurückzugreifen, welche natürlich entsprechend gesetzt werden müssen.

```
signPlugin {
    certificateChain.set(System.getenv("IJ_PluginSign_CertChain"))
    privateKey.set(System.getenv("IJ_PluginSign_PK"))
    password.set(System.getenv("IJ_PluginSign_Pass"))
}
```

Durch Ausführung des Task *signPlugin* kann das Plugin signiert werden.

Da es sich bei dem Zertifikat und dem Privatschlüssel um mehrzeilige Werte handelt, kann es beim Speichern in Umgebungsvariablen zu Schwierigkeiten führen. Um hier etwaige Fehler zu vermeiden, müssen diese mehrzeiligen Werte als *base64* kodiert werden. Der *signPlugin*-Task erkennt diese Kodierung und dekodiert sie automatisch.

Nach dem Signieren kann das Plugin veröffentlicht werden [45]. Das Hochladen des Plugins in den JetBrains Marketplace muss beim ersten Mal manuell gemacht werden. Hierfür meldet man sich auf der Webseite an, klickt am oberen rechten Rand auf seinen Benutzernamen und daraufhin auf *Upload plugin*. Im erscheinenden Dialog gibt man da-

nach die nötigen Daten ein und lädt die zuvor generierte und signierte *.zip*-Datei hoch. Das hochgeladene Plugin kann dann auf der Seite des eigenen Profils verwaltet und mit zusätzlichen Informationen, wie zum Beispiel Screenshots, ausgeschmückt werden. Bevor das Plugin öffentlich im Marketplace angeboten wird, wird es zusätzlich von JetBrains Mitarbeitern geprüft. Um diese Prüfung zu bestehen, sollte bereits vor dem Hochladen auf verschiedene Anforderungen geachtet werden, die in der Dokumentation beschrieben sind. So muss zum Beispiel beachtet werden, was als Logo verwendet wird, welcher Titel gewählt wurde und wie die Beschreibung formatiert ist. Diese Richtlinien können unter https://plugins.jetbrains.com/docs/marketplace/plugin-overview-page.html nachgeschlagen werden.

Nach dem ersten manuellen Upload, ist auch möglich ein Plugin automatisiert in den Marketplace hochzuladen. Hierfür kann der Gradle Task *publishPlugin* in Kombination mit einem *Access Token* für den JetBrains Marketplace genutzt werden. Ein solcher *Access Token* kann auf der Marketplace Webseite unter https://plugins.jetbrains.com/author/me/tokens generiert werden. Der Token muss in der Datei *build.gradle.kts* gesetzt werden.

```
publishPlugin {
    token.set(System.getenv("IJ_PluginSign_PublishToken"))
}
```

Auch dieser Token wird am besten als Umgebungsvariable festgelegt.

5.5 CI/CD

Für das automatisierte Testen und Veröffentlichen eines Plugins gibt es einen vorgefertigten GitHub Actions Workflow für Plugin-Projekte, die mit Gradle arbeiten [19]. Für andere CI/CD Pipelines gibt es keine Dokumentation.

Die Pipeline muss im Grunde verschiedene vordefinierte Gradle Tasks ausführen. Das Ausführen der Tests ist beispielsweise durch den Befehl

```
./gradlew check
```

möglich. Das Veröffentlichen des Plugins kann durch

```
./gradlew publishPlugin
```

angestoßen werden. Natürlich muss hier beachtet werden, dass die zuvor definierten Umgebungsvariablen auch im GitHub Runner, am besten in Form von *secrets*, gesetzt sein müssen.

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

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] *Design patterns : elements of reusable object-oriented software*. eng. 32. print.. Addison-Wesley professional computing series. 2005 (siehe S. 12, 21, 28, 29).

[3] Nadeeshaan Gunasinghe. *Language server protocol and implementation : : supporting language-smart editing and programming tools*. eng. 2022 (siehe S. 10).

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

[5] Gerwin Klein. „Jflex users manual“. *Available on-line at www. jflex. de. Accessed August* (2010) (siehe S. 13).

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

[7] Daniel D McCracken und Edwin D Reilly. „Backus-naur form (bnf)“. In: *Encyclopedia of Computer Science*. 2003, S. 129–131 (siehe S. 13).

[8] Nathan Rozentals. *Mastering TypeScript* (siehe S. 3).

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

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

Online-Quellen

[11] *„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).

[12] Google. *Google Diff Match Patch Algorithm GitHub Repository*. URL: https://github.com/google/diff-match-patch (besucht am 06. 10. 2023) (siehe S. 22).

[13] Google. *Guava GitHub Repository*. URL: https://github.com/google/guava (besucht am 25. 11. 2023) (siehe S. 29).

[14] *Grammar-Kit GitHub Repository*. URL: https://github.com/JetBrains/Grammar-Kit (besucht am 31. 10. 2023) (siehe S. 13).

[15] 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).

[16] *IntelliJ Community Edition GitHub Repository - CompletionUtilCore*. URL: https://github.com/JetBrains/intellij-community/blob/23bb68de04cfd849d615dac6ceaa2738e5bd431d/platform/core-api/src/com/intellij/codeInsight/completion/CompletionUtilCore.java#L11 (besucht am 25. 11. 2023) (siehe S. 30).

[17] *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).

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

[19] *IntelliJ Platform Plugin Template GitHub Repository - Build Workflow*. URL: https://github.com/JetBrains/intellij-platform-plugin-template/blob/main/.github/workflows/build.yml (besucht am 25. 11. 2023) (siehe S. 35).

[20] *IntelliJ Platform SDK - Action System*. URL: https://plugins.jetbrains.com/docs/intellij/basic-action-system.html (besucht am 01. 11. 2023) (siehe S. 12).

[21] *IntelliJ Platform SDK - Actions*. URL: https://plugins.jetbrains.com/docs/intellij/plugin-actions.html (besucht am 01. 11. 2023) (siehe S. 12).

[22] *IntelliJ Platform SDK - Configuration - Actions*. URL: https://plugins.jetbrains.com/docs/intellij/plugin-configuration-file.html#idea-plugin__actions__action (besucht am 01. 11. 2023) (siehe S. 12).

[23] *IntelliJ Platform SDK - Configuring Kotlin Support*. URL: https://plugins.jetbrains.com/docs/intellij/using-kotlin.html (besucht am 01. 11. 2023) (siehe S. 7).

[24] *IntelliJ Platform SDK - Creating a Plugin Gradle Project*. URL: https://plugins.jetbrains.com/docs/intellij/creating-plugin-project.html (besucht am 25. 11. 2023) (siehe S. 28).

[25] *IntelliJ Platform SDK - Custom Language Support*. URL: https://plugins.jetbrains.com/docs/intellij/custom-language-support.html (besucht am 01. 11. 2023) (siehe S. 13).

[26] *IntelliJ Platform SDK - Dialogs*. URL: https://plugins.jetbrains.com/docs/intellij/dialog-wrapper.html (besucht am 01. 11. 2023) (siehe S. 14).

[27] *IntelliJ Platform SDK - Extension Points*. URL: https://plugins.jetbrains.com/docs/intellij/plugin-extension-points.html (besucht am 01. 11. 2023) (siehe S. 7, 12).

[28] *IntelliJ Platform SDK - General Threading Rules*. URL: https://plugins.jetbrains.com/docs/intellij/general-threading-rules.html (besucht am 25. 11. 2023) (siehe S. 31).

[29] *IntelliJ Platform SDK - Language Support Tutorial*. URL: https://plugins.jetbrains.com/docs/intellij/custom-language-support-tutorial.html (besucht am 01. 11. 2023) (siehe S. 13).

[30] *IntelliJ Platform SDK - Language Support Tutorial - Grammar and Parser*. URL: https://plugins.jetbrains.com/docs/intellij/grammar-and-parser.html (besucht am 01. 11. 2023) (siehe S. 13).

[31] *IntelliJ Platform SDK - Language Support Tutorial - Lexer and Parser Definition*. URL: https://plugins.jetbrains.com/docs/intellij/lexer-and-parser-definition.html (besucht am 01. 11. 2023) (siehe S. 13).

[32] *IntelliJ Platform SDK - Light and Heavy Tests*. URL: https://plugins.jetbrains.com/docs/intellij/light-and-heavy-tests.html (besucht am 25. 11. 2023) (siehe S. 33).

[33] *IntelliJ Platform SDK - List and Tree Controls*. URL: https://plugins.jetbrains.com/docs/intellij/lists-and-trees.html (besucht am 01. 11. 2023) (siehe S. 13).

[34] *IntelliJ Platform SDK - Listeners*. URL: https://plugins.jetbrains.com/docs/intellij/plugin-listeners.html (besucht am 01. 11. 2023) (siehe S. 12).

[35] *IntelliJ Platform SDK - Miscellaneous Swing Components*. URL: https://plugins.jetbrains.com/docs/intellij/misc-swing-components.html (besucht am 01. 11. 2023) (siehe S. 13).

[36] *IntelliJ Platform SDK - Notifications*. URL: https://plugins.jetbrains.com/docs/intellij/notifications.html (besucht am 01. 11. 2023) (siehe S. 14).

[37] *IntelliJ Platform SDK - Plugin Configuration File*. URL: https://plugins.jetbrains.com/docs/intellij/plugin-configuration-file.html (besucht am 01. 11. 2023) (siehe S. 7).

[38] *IntelliJ Platform SDK - Plugin Signing*. URL: https://plugins.jetbrains.com/docs/intellij/plugin-signing.html (besucht am 25. 11. 2023) (siehe S. 34).

[39] *IntelliJ Platform SDK - Plugin Structure*. URL: https://plugins.jetbrains.com/docs/intellij/plugin-structure.html (besucht am 01. 11. 2023) (siehe S. 6).

[40] *IntelliJ Platform SDK - Popups*. URL: https://plugins.jetbrains.com/docs/intellij/popups.html (besucht am 01. 11. 2023) (siehe S. 14).

[41] *IntelliJ Platform SDK - Program Structure Interface (PSI)*. URL: https://plugins.jetbrains.com/docs/intellij/psi.html (besucht am 01. 11. 2023) (siehe S. 13).

[42] *IntelliJ Platform SDK - PSI Elements*. URL: https://plugins.jetbrains.com/docs/intellij/psi-elements.html (besucht am 01. 11. 2023) (siehe S. 13).

[43] *IntelliJ Platform SDK - PSI Files*. URL: https://plugins.jetbrains.com/docs/intellij/psi-files.html (besucht am 01. 11. 2023) (siehe S. 13).

[44] *IntelliJ Platform SDK - PsiViewer*. URL: https://plugins.jetbrains.com/docs/intellij/explore-api.html#31-use-internal-mode-and-psiviewer (besucht am 01. 11. 2023) (siehe S. 13).

[45] *IntelliJ Platform SDK - Publishing a Plugin*. URL: https://plugins.jetbrains.com/docs/intellij/publishing-plugin.html (besucht am 25. 11. 2023) (siehe S. 34).

[46] *IntelliJ Platform SDK - Safely Replacing Selected Text in the Document*. URL: https://plugins.jetbrains.com/docs/intellij/working-with-text.html#safely-replacing-selected-text-in-the-document (besucht am 25. 11. 2023) (siehe S. 31).

[47] *IntelliJ Platform SDK - Services*. URL: https://plugins.jetbrains.com/docs/intellij/plugin-services.html (besucht am 01. 11. 2023) (siehe S. 12).

[48] *IntelliJ Platform SDK - Test Overview*. URL: https://plugins.jetbrains.com/docs/intellij/testing-plugins.html (besucht am 25. 11. 2023) (siehe S. 33).

[49] *IntelliJ Platform SDK - Test Project and Testdata Directories*. URL: https://plugins.jetbrains.com/docs/intellij/test-project-and-testdata-directories.html (besucht am 25. 11. 2023) (siehe S. 34).

[50] *IntelliJ Platform SDK - Tests and Fixtures*. URL: https://plugins.jetbrains.com/docs/intellij/tests-and-fixtures.html (besucht am 25. 11. 2023) (siehe S. 33, 34).

[51] *IntelliJ Platform SDK - Tool Windows*. URL: https://plugins.jetbrains.com/docs/intellij/tool-windows.html (besucht am 01. 11. 2023) (siehe S. 14).

[52] *IntelliJ Platform SDK - User Interface Components*. URL: https://plugins.jetbrains.com/docs/intellij/user-interface-components.html (besucht am 01. 11. 2023) (siehe S. 13).

[53] *IntelliJ Platform SDK - Writing Tests*. URL: https://plugins.jetbrains.com/docs/intellij/writing-tests.html (besucht am 25. 11. 2023) (siehe S. 34).

[54] *IntelliJ Platform SDK Documentation*. URL: https://plugins.jetbrains.com/docs/intellij/welcome.html (besucht am 06. 10. 2023) (siehe S. 3, 14).

[55] *IntelliJ Support Forum - The dreaded „IntelliJIdeaRulezzz“ string*. URL: https://intellij-support.jetbrains.com/hc/en-us/community/posts/206752355-The-dreaded-IntellijIdeaRulezzz-string (besucht am 25. 11. 2023) (siehe S. 30).

[56] JetBrains. *IntelliJ Community Edition GitHub Repository*. URL: https://github.com/JetBrains/intellij-community/tree/master (besucht am 06. 10. 2023) (siehe S. 3).

[57] *JFlex Homepage*. URL: https://jflex.de/ (besucht am 31. 10. 2023) (siehe S. 13).

[58] Microsoft. *Visual Studio Code Extension Samples*. URL: https://github.com/microsoft/vscode-extension-samples (besucht am 06. 10. 2023) (siehe S. 10).

[59] Microsoft. *VS Code test-electron GitHub Repository*. URL: https://github.com/Microsoft/vscode-test (besucht am 06. 10. 2023) (siehe S. 25).

[60] *Mocha*. URL: https://mochajs.org/ (besucht am 05. 11. 2023) (siehe S. 25).

[61] *Node.js*. URL: https://nodejs.org/en (besucht am 05. 11. 2023) (siehe S. 21).

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

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

[64] Razmik Seysyan. *Flora (beta) Plugin on JetBrains Marketplace*. URL: https://plugins.jetbrains.com/plugin/17669-flora-beta- (besucht am 06. 10. 2023) (siehe S. 15).

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

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

[67] *TextMate Language Grammar*. URL: https://macromates.com/manual/en/language_grammars (besucht am 10. 10. 2023) (siehe S. 9).

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

[69] *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).

[70] *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).

[71] *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).

[72] *VS Code Extension API - Activation Events*. URL: https://code.visualstudio.com/api/references/activation-events (besucht am 01. 11. 2023) (siehe S. 5).

[73] *VS Code Extension API - Commands*. URL: https://code.visualstudio.com/api/extension-guides/command (besucht am 01. 11. 2023) (siehe S. 9).

[74] *VS Code Extension API - Common Capabilities*. URL: https://code.visualstudio.com/api/extension-capabilities/common-capabilities (besucht am 01. 11. 2023) (siehe S. 10, 11).

[75] *VS Code Extension API - Continuous Integration*. URL: https://code.visualstudio.com/api/working-with-extensions/continuous-integration (besucht am 05. 11. 2023) (siehe S. 26).

[76] *VS Code Extension API - Contribution Points*. URL: https://code.visualstudio.com/api/references/contribution-points (besucht am 01. 11. 2023) (siehe S. 6).

[77] *VS Code Extension API - Contribution Points - Commands*. URL: https://code.visualstudio.com/api/references/contribution-points#contributes.commands (besucht am 01. 11. 2023) (siehe S. 8).

[78] *VS Code Extension API - Contribution Points - Configuration*. URL: https://code.visualstudio.com/api/references/contribution-points#contributes.configuration (besucht am 01. 11. 2023) (siehe S. 11).

[79] *VS Code Extension API - Contribution Points - Menus*. URL: https://code.visualstudio.com/api/references/contribution-points#contributes.menus (besucht am 01. 11. 2023) (siehe S. 9).

[80] *VS Code Extension API - Extension Anatomy*. URL: https://code.visualstudio.com/api/get-started/extension-anatomy (besucht am 01. 11. 2023) (siehe S. 5, 6).

[81] *VS Code Extension API - Extension Manifest*. URL: https://code.visualstudio.com/api/references/extension-manifest (besucht am 01. 11. 2023) (siehe S. 5).

[82] *VS Code Extension API - FilePicker*. URL: https://code.visualstudio.com/api/references/vscode-api#window.showOpenDialog (besucht am 01. 11. 2023) (siehe S. 10).

[83] *VS Code Extension API - Language Configuration Guide*. URL: https://code.visualstudio.com/api/language-extensions/language-configuration-guide (besucht am 01. 11. 2023) (siehe S. 10).

[84] *VS Code Extension API - Notifications*. URL: https://code.visualstudio.com/api/references/vscode-api#window.showInformationMessage (besucht am 01. 11. 2023) (siehe S. 11).

[85] *VS Code Extension API - OutputChannel*. URL: https://code.visualstudio.com/api/references/vscode-api#OutputChannel (besucht am 01. 11. 2023) (siehe S. 11).

[86] *VS Code Extension API - Programmatic Language Features*. URL: https://code.visualstudio.com/api/language-extensions/programmatic-language-features (besucht am 01. 11. 2023) (siehe S. 10).

[87] *VS Code Extension API - Progress*. URL: https://code.visualstudio.com/api/references/vscode-api#Progress (besucht am 01. 11. 2023) (siehe S. 11).

[88] *VS Code Extension API - Publishing Extension*. URL: https://code.visualstudio.com/api/working-with-extensions/publishing-extension (besucht am 05. 11. 2023) (siehe S. 26).

[89] *VS Code Extension API - QuickInput Sample*. URL: https://github.com/microsoft/vscode-extension-samples/tree/main/quickinput-sample (besucht am 01. 11. 2023) (siehe S. 10).

[90] *VS Code Extension API - QuickPick*. URL: https://code.visualstudio.com/api/references/vscode-api#QuickPick (besucht am 01. 11. 2023) (siehe S. 10).

[91] *VS Code Extension API - Semantic Highlight Guide*. URL: https://code.visualstudio.com/api/language-extensions/semantic-highlight-guide (besucht am 01. 11. 2023) (siehe S. 9).

[92] *VS Code Extension API - Snippet Guide*. URL: https://code.visualstudio.com/api/language-extensions/snippet-guide (besucht am 01. 11. 2023) (siehe S. 10).

[93] *VS Code Extension API - Syntax Highlight Guide*. URL: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide (besucht am 01. 11. 2023) (siehe S. 9).

[94] *VS Code Extension API - Testing Extensions*. URL: https://code.visualstudio.com/api/working-with-extensions/testing-extension (besucht am 05. 11. 2023) (siehe S. 25).

[95] *VS Code Extension API - Tree View API*. URL: https://code.visualstudio.com/api/extension-guides/tree-view (besucht am 01. 11. 2023) (siehe S. 11).

[96] *VS Code Extension API - VS Code API*. URL: https://code.visualstudio.com/api/references/vscode-api (besucht am 01. 11. 2023) (siehe S. 6).

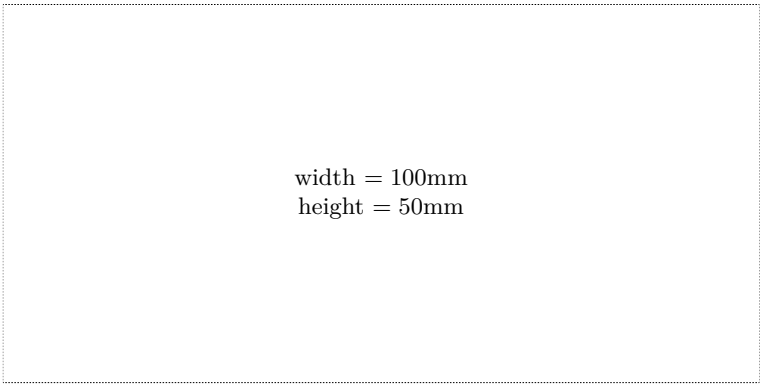
[97] *VS Code Extension API - Webview API*. URL: https://code.visualstudio.com/api/extension-guides/webview (besucht am 01. 11. 2023) (siehe S. 11).

[98] *VS Code Extension API - Your First Extension*. URL: https://code.visualstudio.com/api/get-started/your-first-extension (besucht am 05. 11. 2023) (siehe S. 21).

[99] *Yeoman*. URL: https://yeoman.io/ (besucht am 05. 11. 2023) (siehe S. 21).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —