Einsatz intelligenter Werkzeuge zur Softwareentwicklung

Jonathan Schwab[1], Felix Wochele[2], Jonas Weis[3]

Abstract: Zahlreiche moderne Werkzeuge zur Unterstützung der Softwareentwicklung sind mittlerweile nutzbar. Diese Arbeit schafft einen Überblick über verschiedene Einsatzbereiche. Behandelt werden die Bereiche Codevervollständigung, Codegenerierung, Codeanalyse, Refactoring, Dokumentation und Kollaboration. Neben der Vorstellung grundlegender Konzepte und Umsetzungen, steht dabei die Analyse des Nutzens und der möglichen Risiken oder Grenzen im Vordergrund. Außerdem gibt die Vorstellung einer oder mehrerer Werkzeuge je Kategorie die Möglichkeit zum Transfer in die Praxis. Neben der Funktionalität wurde bei der Auswahl vor allem auf die Unterstützung möglichst vieler Programmiersprachen geachtet.

Keywords: Künstliche Intelligenz; Softwareentwicklung; Intelligente Werkzeuge; Codevervollständigung; Refactoring; Codegeneration; Codeanalyse; Dokumentation; Kollaboration

1 Einleitung

Das Thema beschäftigt sich mit verschiedenen intelligenten Werkzeugen zur Unterstützung bei der Softwareentwicklung. Viele Prozesse der Softwareentwicklung enthalten repetitive, triviale oder inferentielle Vorgänge. Der Einsatz von intelligenten Werkzeugen soll hier Abhilfe schaffen. Konkrete Vorteile können Zeitersparnis, eine Steigerung der Codequalität oder eine bessere Nachverfolgbarkeit sein. Dies steigert die Wirtschaftlichkeit von Softwareprojekten und schafft dem Entwickler mehr Zeit für komplexe Arbeit. Um die Werkzeuge differenziert zu bewerten ist es notwendig die zugrunde liegenden Konzepte zu verstehen. Somit lassen sich neben den Vorteilen auch vorhandene Grenzen und Risiken erkennen.

Ziel dieser Arbeit ist das Schaffen eines Überblickes über intelligente Werkzeuge zur Erleichterung der Softwareentwicklung. Dabei sollen Konzepte aus den Bereichen Codevervollständigung, Codegenerierung, Codeanalyse, Refactoring, Dokumentation und Kollaboration genauer betrachtet werden. Wesentlicher Bestandteil ist das Herausstellen von Nutzen und Risiken beim Einsatz derartiger Werkzeuge. Zu jedem Konzept wird eine Marktrecherche durchgeführt, welche besonders geeignete Werkzeuge vorstellen soll. Anhand dessen wird eine Handlungsempfehlung gegeben, welche beim Bearbeiten zukünftiger Projekte zur Unterstützung herangezogen werden kann.

[1] DHBW Stuttgart - Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland, i20030@hb.dhbw-stuttgart.de
[2] DHBW Stuttgart - Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland, i20036@hb.dhbw-stuttgart.de
[3] DHBW Stuttgart - Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland, i20035@hb.dhbw-stuttgart.de

Mittlerweile wird die Auswahl an intelligenten Werkzeugen zur Softwareentwicklung immer größer. Neben herkömmlichen Verfahren ermöglicht vor allem künstliche Intelligenz einen großen Fortschritt in diesem Bereich. Aufgrund der Breite dieses Sektors kann nicht jedes intelligente Werkzeug vorgestellt werden. Daher folgt eine Fokussierung auf die bereits aufgezählten Konzepte, da diese sowohl im Unternehmens-, als auch privaten Bereich Potential besitzen. Beispielsweise wären Werkzeuge zum Testen von Software eine weitere Möglichkeit gewesen. Diese sehen wir allerdings aufgrund der Vorlesung *Software Engineering I* als ausreichend bekannt an. Das Thema *Requirements Engineering* bietet auch eine Vielzahl an Werkzeugen an. Es ist allerdings derart umfangreich, dass es wenige Seiten nicht ordnungsgemäß darstellen könnten. Nur eines von vielen Unterthemen wäre das Testmanagement. Außerdem werden derartige Werkzeuge in der Regel vom Unternehmen festgelegt und variieren damit stark. Sie finden im privaten Bereich teilweise kaum Einsatz. Daher werden sie in der folgenden Arbeit nicht näher betrachtet. Der Fokus liegt stattdessen auf Verfahren, welche auch im privaten Bereich, beispielsweise in Freizeitprojekten, Anwendung finden. Informationen zum Thema *Requirements Engineering* können aber beispielsweise unter [Ch13] gefunden werden.

2 Codevervollständigung

Sind in einem Projekt die Anforderungen definiert und von Hand erste Modelle und Architekturen entworfen, müssen diese implementiert werden. Dieses Kapitel behandelt die automatische und intelligente Codevervollständigung in Integrierten Entwicklungsumgebungen, also die Unterstützung während diesem Vorgang.

2.1 Allgemeine Konzepte

Automatische Codevervollständigung ist heutzutage in nahezu allen IDEs vorhanden. Es ist eines der meistgenutztesten Features von Softwareentwicklern [GKF06]. Anhand verschiedener Ansätze werden dabei wahrscheinliche Vervollständigungen vorgeschlagen. Diese bestehen aus verschiedenen Dingen und variieren je nach Sprache. Übliche Vorschläge sind aber beispielsweise im Kontext verfügbare *Methoden, Events, Klassen, Interfaces, Strukturen, Enums, Schlüsselwörter, lokale Variablen, Parameter oder Attribute, Dateien, Farben und Datentypen*. Durch verschiedene Methoden wird die Wahrscheinlichkeit dieser berechnet und in der Regel anhand einer Textbox zur Auswahl gestellt. Alternativ wird die wahrscheinlichste Option direkt ausgegraut angezeigt und kann beispielsweise durch die Tab-Taste akzeptiert werden. Abbildung 1 zeigt diese Optionen anhand von Visual Studio auf.

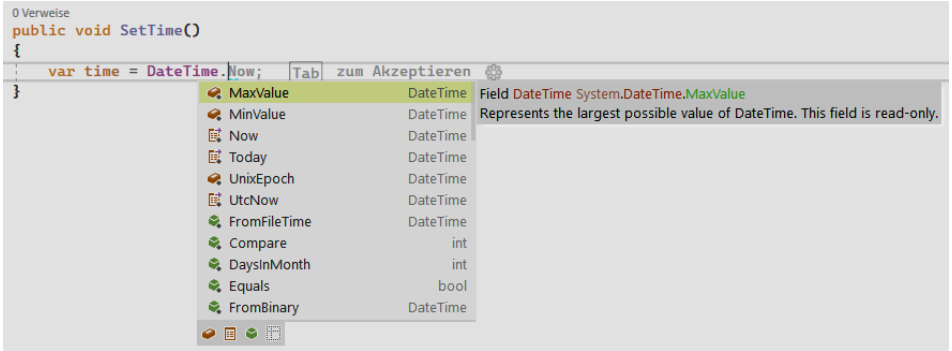

Abb. 1: Code Vervollständigung in Visual Studio 2022

Je mehr Zeichen bereits geschrieben wurden, desto genauer können die Vorschläge werden. Dies liegt daran, dass durch die schon geschriebenen Zeichen gewisse Optionen auszuschließen sind. Die Eingabe wird also ständig geparst und es ergeben sich mögliche Ergebnisse mit dem gleichen Präfix. Das Parsen kann beispielsweise durch reguläre Ausdrücke oder das Aufbauen von abstrakten Syntaxbäumen umgesetzt werden. Hierbei wird von der lexikalischen Analyse gesprochen. Ein anderer klassischer Ansatz ist die semantische Analyse. Diese verwendet ein Grammatikmodell der Sprache um anhand definierter Regeln nur Methoden anzuzeigen, welche auch legal sind. Dabei werden beispielsweise private Methoden herausgefiltert, wenn auf diese kein Zugriff vorhanden ist [MCB15]. In der Regel werden beide Analysearten verwendet.

Die Bestimmung der besten Ergebnisse ist dann durch verschiedene Implementierungen umsetzbar. Einfachste Lösung ist die Sortierung nach alphabetischer Reihenfolge oder dem Zeitpunkt der letzten Modifikation einer Methode. Auch denkbar ist die Priorisierung nach Lokalität, also beispielsweise zuerst die aktuelle Klasse, dann das Projekt und zuletzt Imports [RL08].

Naheliegend ist auch die Sortierung nach der Anzahl der bisherigen Verwendungen eines Vorschlages. Diese Arten zählen zu den statistischen. Auch diese erzielen heutzutage bessere Ergebnisse, da öffentliche Repositories die notwendigen Daten zur Verfügung stellen. Anhand dieser können die Auftrittswahrscheinlichkeiten verschiedener Elemente ausgelesen werden um somit bessere Vorschläge zu erzielen. Verwendung findet dabei beispielsweise das N-Gram-Modell. Dieses zählt die Vorkommen verschiedener Kombinationen aus N Elementen [Ro00]. Problematisch ist, dass diese Modelle nur Abhängigkeiten zwischen wenigen Elementen feststellen. Außerdem funktioniert es nur, wenn Statistiken zu allen Element-Kombinationen vorhanden sind.

Moderne Systeme nutzen immer häufiger einen anderen Ansatz: Künstliche Intelligenz. Im Gegensatz zur statistischen Analyse werden nicht nur Statistiken gesammelt sondern anhand von Daten wiederkehrende Muster erkannt. Diese schaffen es besser umfangreichen

Kontext miteinzubeziehen und auf in den Trainingsdaten nicht vorhandene Elemente zu reagieren. Zur genauen Umsetzung gibt es verschiedenste Optionen. In [Sc19b] werden einige der verwendbaren rekurrenten neuronalen Netze gegenüber gestellt. Erwähnenswert ist außerdem der Algorithmus Best-Matching-Neighbour, welcher sich als sehr effizient herausgestellt hat [BMM09].

2.2 Nutzen und Risiken

Die Nutzung von Codevervollständigung bringt zahlreiche Vorteile mit sich. Der größte ist die Zeitersparnis. Diese unterscheidet sich je nach Effektivität des Tools. Statt dem Ausschreiben ganzer Methodenaufrufe reicht oftmals der erste Buchstabe. Selbst wenn der gesuchte Vorschlag erst an dritter Stelle ist, muss nur zweimal die Pfeiltaste zur Auswahl betätigt werden. Im schlimmsten Fall müssen bereits mehrere Buchstaben vorhanden sein, damit die gesuchte Empfehlung kommt. Trotzdem findet eine enorme Zeiteinsparung statt. Gleiches gilt auch für die anderen Elemente, wie Variablen und Co. Einige Tools sprechen von ca. 40% weniger Tastaturanschlägen [Ki22]. Außerdem werden durch die Vorschläge Tipp- und Logikfehler vermieden. Syntaktisch oder semantisch nicht korrekte Eingaben rufen keine Vorschläge hervor. Wird stattdessen die Eingabe durch Vorschläge vervollständigt, kann von syntaktischer und semantischer Korrektheit ausgegangen werden. Ein weiterer Vorteil ist, dass kein Detailwissen mehr benötigt wird. Sobald in objektorientierten Sprachen beispielsweise ein Punkt hinter das Objekt gesetzt wurde, werden dessen zugreifbare Methoden und Felder angezeigt. Dabei wird in der Regel bereits gefiltert, ob eine Zuweisung oder nur ein Aufruf stattfindet. Findet ersteres statt, werden nur Methoden und Felder mit Rückgabewert angezeigt. Besonders bei Nutzung fremder Bibliotheken ist dies von großem Vorteil. Somit können selbst schlecht dokumentierte Anwendungen und Bibliotheken verwendet werden, sofern die Namen verständlich gewählt wurden. Durch die Nutzung von Inline-Dokumentation sind die jeweiligen Methoden oftmals sogar direkt in der IDE beschrieben. Neben dem Vermeiden von Logik-Fehlern helfen diese Tools bereits beim Formulieren der Logik. Beispielsweise wird bei Zuweisung eines Wertes an eine Variable automatisch erkannt, wenn der Datentyp nicht übereinstimmt. Daher folgt ein Vorschlag zum Casting der Eingabe. Bei verschiedenen Collection-Arten werden dagegen beispielsweise auf diese anwendbare Strukturen wie `foreach` vorgeschlagen. Das Gerüst dieser kann automatisch vervollständigt werden. Ein weiteres Beispiel ist die Rückgabe in einer Methode. Wird das passende Keyword geschrieben, kommen beispielsweise Vorschläge aus lokalen Variablen mit dem passenden Rückgabetyp. Die Beispiele sind zahlreich.

Diese Vorteile müssen allerdings mit Vorsicht genutzt werden. Die automatische Vervollständigung verleitet beispielsweise dazu, Vorschläge blind anzunehmen und einfach davon auszugehen, dass die dahinterliegende Funktionalität passend ist. Besonders bei fremden Bibliotheken wäre es dagegen sinnvoll, dies zu überprüfen oder die technische Dokumentation zu studieren. Gegebenenfalls hat eine verwendete Methode besonders

schlechte Performance, ist bereits veraltet oder nicht Thread-safe, dies wird aber für die Anwendung benötigt. Hier muss zwingend vorsichtig vorgegangen werden.

2.3 Marktanalyse

Es gibt verschiedene Tools zur Nutzung mit mehreren Sprachen. Bezüglich Microsoft und Visual Studio wird **IntelliSense** bzw. **IntelliCode** verwendet. Dieses unterstützt JavaScript, TypeScript, JSON, HTML, CSS, SCSS, C++, C#, J#, Visual Basic, XML, XSLT, SQL. Bei Nutzung der IDE Visual Studio Code ermöglichen Erweiterungen außerdem dutzende weitere Sprachen. Es basiert auf tausenden Repositories, deren Qualität durch eine Mindestanzahl an Sternen sichergestellt werden soll.

Ähnlich viele Sprachen in verschiedenen Editoren unterstützt die alleinstehende Anwendung **Kite**. Diese wurden anhand von über 25 Millionen Dateien trainiert und soll die Tastenanschläge um ca. 40% reduzieren [Ki22].

Besonders erwähnenswert ist allerdings das Projekt **Tabnine**. Es handelt sich um eine moderne, sehr umfangreiche Erweiterung. Bezüglich der Funktion Codevervollständigung ist diese kostenlos. Unterstützt werden 20 verschiedene IDEs und folgende Sprachen: Angular, C, C++, C#, CSS, Dart, Go, Haskell, HTML, Java, Javascript, Kotlin, Matlab, NodeJS, ObjectiveC, Perl, PHP, Pyhton, React, Ruby, Rust, Sass, Scala, Swift und Typescript. Erkennbar ist das definierte Ziel, nicht auf spezielle Sprachen beschränkt zu sein. Besonderheit ist außerdem die Möglichkeit eigene Modelle zu trainieren indem ausgewählte Repositories verbunden werden. Dies ermöglicht die optimalen Vorschläge bezogen auf spezielle Projekte. So kann beispielsweise der Programmierstil innerhalb eines Unternehmens besser miteinbezogen werden. Dazu muss allerdings die kostenpflichtige Version (12€ pro User je Monat) in Verwendung sein.

Abb. 2: Code Vervollständigung mit TabNine in VS Code

Abbildung 2 zeigt die Vervollständigung durch TabNine in Visual Studio Code. Erkennbar ist, dass mehr als nur ein Methodenname vervollständigt wird. Anhand des Befehls `app.listen (port, ...)` schlägt die Software einen passenden Text für das Logging unter Einbezug der Parameter vor. Als Grundlage von *Tabnine* gilt das Modell GPT-2. Die verwendeten Daten stammen von GitHub-Repositories, für welche Qualität sichergestellt wurde. Neue Daten finden regelmäßig Einsatz im Training, um auch neuste Entwicklungen miteinzubeziehen.

3 Codegenerierung

In bestimmten Fällen ist es mittlerweile nicht mehr notwendig Quellcode von Hand zu schreiben. Stattdessen kann spezifiziert werden, was der Programmcode tun soll und aus dieser Information automatisch Code generiert werden. Dieser Anwendungsfall kann zwar nicht so weitläufig eingesetzt werden wie die Codevervollständigung, spart stellenweise aber noch mehr Arbeit ein.

3.1 Allgemeine Konzepte

Programmiersprachen dienen dazu Datenstrukturen und Algorithmen formal aber vom Menschen lesbar auszudrücken. Hierbei unterscheidet man zwischen High-Level- und Low-Level Programmiersprachen (Abbildung 3). Low-Level Programmiersprachen befinden sich auf einer niedrigen, High-Level Programmiersprachen auf einer höheren Abstraktionsebene. Ziel dieser Abstraktion ist es, die Lesbarkeit zu erhöhen und die Komplexität

zu reduzieren. Hierfür bieten High-Level Programmiersprachen beispielsweise Kontrollstrukturen und Datentypen an. Bei der Codegenerierung wird diese Idee aufgegriffen und erweitert. Im Allgemeinen geht es um die Frage, ob es nicht einfachere und kompaktere Darstellungsweisen für Problemlösungen gibt, die anschließend in Programmcode einer tieferen Abstraktionsebene umgewandelt werden können.

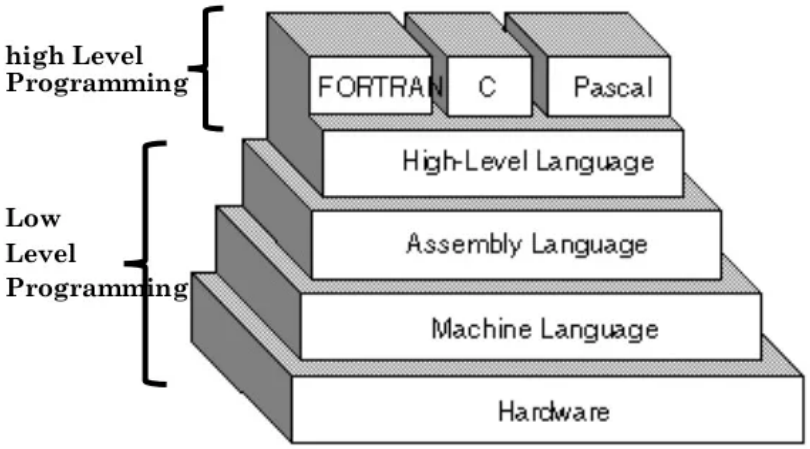

Abb. 3: Abstraktionsebenen von Programmiersprachen [Th21]

Ein **Compiler** ist wohl das bekannteste Werkzeug zur Codegenerierung. Dieser wandelt Quellcode einer höheren Programmiersprache in Befehle einer niedrigeren Sprache um. Ein Compiler für die Programmiersprache Java besteht aus verschiedenen Bestandteilen: Ein Scanner liest Lexeme und generiert Tokens. Der Parser generiert eine abstrakte Syntax. Im Semantik-Check wird eine getypte abstrakte Syntax generiert. Zuletzt generiert ein Code-Generator Bytecode für die JVM. Compiler sind ein fester Bestandteil von nahezu jedem Softwareentwicklungsprozess und werden aus diesem Grund der Vollständigkeit halber hier angemerkt, jedoch nicht näher behandelt.

Modellgetriebene Softwareentwicklung (MDSD) *ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.* [St07, p. 11]

Die formalen Modelle werden meist im zugeschnittenen Modellierungssprachen wie DLS oder UML spezifiziert. Formal bedeutet in diesem Zusammenhang, dass das Modell

einen bestimmten Aspekt der Software vollständig beschreibt. Dies ist essentiell für die anschließend automatische Codeerzeugung. Ein wichtiger Begriff in diesem Kontext ist das Round-Trip Engineering. Dies beschreibt die bidirektionale Synchronisation zwischen Modellen und Quellcode. So können die Vorteile der Modellentwicklung (Übersichtlichkeit und Dokumentation) mit den der Quellcodeprogrammierung (Präzision und Spezifikation) kombiniert werden.

Die Vorteile einer modellgetriebenen Softwareentwicklung sind [St07, p. 13-16]:

(a) **Abstraktion:** Die Modelle sind einfacher und allgemeiner als der zu generierende Quellcode.

(b) **Einheitliche Architektur:** Die Softwareerzeugung aus den Modellen erfolgt nach streng formalen Vorschriften unter Berücksichtigung eines vorgegebenen Rahmens.

(c) **Softwarequalität** Das Projekt wird durch den modellgetriebenen Entwicklungsprozess in eine einheitliche, testbare und dokumentierte Architektur gegossen. Selbstverständlich ist dies dennoch keine Garantie für gute Softwarequalität.

(d) **Entwicklungsgeschwindigkeit:** Durch eine höhere Softwarequalität ist das Projekt besser Wartbar und Komponenten sind besser Austauschbar, was langfristig die Entwicklungsgeschwindigkeit erhöht. Zudem gibt es immer eine Designdokumentation, was die Übersichtlichkeit erhöht.

(e) **Interoperabilität und Plattformunabhängigkeit:** Durch die modellgetriebene Entwicklung soll eine Plattform- und Frameworkunabhängigkeit durch Standardisierung erreicht werden. Die Modelle sollen streng vom generierten Programmcode entkoppelt und somit von diesem unabhängig sein. Dies ist jedoch meist nur in der Theorie möglich und kann beispielsweise beim Round-Trip Engineering nicht gewährleistet werden.

Die Nachteile der modellgetriebenen Entwicklung sind:

(a) **Hoher Initialisierungsaufwand:** Der Initialisierungsaufwand ist sehr hoch und zahlt sich entsprechend nur bei großen Projekten aus.

(b) **Hoher Einarbeitungsaufwand:** Da die modellgetriebene Entwicklung nicht so verbreitet ist wie die Verwendung herkömmlicher Programmiersprachen, jedoch eine hohe Komplexität aufweist, ist ein hoher Einarbeitungsaufwand notwendig.

Werkzeuge für die modellgetriebene Entwicklung fangen bei einfachen UML-Codegeneratoren (z.B. Visual Paradigm) an und hören bei Komplettlösungen für bestimmte Domänen (z.B. ASCET für eingebettete Automobilsoftware) auf [St07].

Durch AI-basierte Werkzeuge gibt es die Möglichkeit zur **Quellcodeerzeugung aus natürlicher Sprache**. Dies stellt sich vor allem für stark spezialisierte Bereiche als nützlich

heraus. Ein Beispiel hierfür ist die Generierung von SQL Statements oder einfachen Programmieranweisungen aus natürlicher Sprache.

Eine populäre Möglichkeit hierfür ist das AI-Modell OpenAI Codex. Dieses stellt auch die Basis für das Werkzeug GitHub Copilot dar, welches gemeinsam von OpenAI und GitHub entwickelt wurde [Op21]. Dieses Modell kann über die API oder dem Browser-Playground von OpenAI genutzt werden. In einem Test wurde ein Ratespiel in JavaScript mittels dem OpenAI Codex Model generiert (Quellcode 1). Aus dem in den Kommentaren angegebene Text in natürlicher Sprache wurde jeweils der darunter stehende Quellcode generiert.

```
1
2  /* Generiere eine zufaellige Zahl zwischen 0 und 10 */
3  var randomNumber = Math.floor(Math.random() * 10);
4
5  /* Frage den Nutzer nach einer Texteingabe */
6  var userInput = prompt('Gib eine Zahl zwischen 0 und 10 ein');
7
8  /* konvertiere die Nutzereingabe in eine Zahl */
9  var userNumber = parseInt(userInput);
10
11 /* wenn die generierte Zufallszahl gleich der userNumber ist "gewonnen", sonst "
       verloren" ausgeben */
12 if (randomNumber === userNumber) {
13   alert('gewonnen');
14 } else {
15   alert('verloren');
16 }
```

Quellcode 1: Code aus natürlicher Sprache

Auch die Generierung von SQL Statements liefert ein korrektes Ergebnis (Quellcode 2).

```
1  ### Postgres SQL tables, with their properties:
2  #
3  # Mitarbeiter(mid, vorname, nachname)
4  # Teilnahme(mid, pid)
5  # Projekt(pid, bezeichnung)
6  #
7  ### Liste die Mitarbeiternamen der Mitarbeiter des Projektes "Test" auf
8  --------------------------------------------------------------------------
9  SELECT m.vorname, m.nachname
10 FROM Mitarbeiter m
11 JOIN Teilnahme t ON m.mid=t.mid
12 JOIN Projekt p ON t.pid=p.pid
13 WHERE p.bezeichnung='Test'
```

Quellcode 2: SQL Statement aus natürlicher Sprache

Selbst etwas komplexere Aufgaben, wie API-Aufrufe und Deserialisierung sind möglich. Hierzu musste jedoch mit der textuellen Eingabe etwas experimentiert werden (Quellcode 3) [Op22].

```
//API aufrufen: https://jsonplaceholder.typicode.com/todos/1
//Ergebnisse Deserialisieren & Ausgeben
//Quellcode in C#:
------------------------------------------------------------------------
using System;
using System.IO;
using System.Net;
using System.Runtime.Serialization.Json;

namespace ConsoleApp1
{
  class Program
  {
    static void Main(string[] args)
    {
      string url = @"https://jsonplaceholder.typicode.com/todos/1";
      WebRequest req = WebRequest.Create(url);
      req.Method = "GET";
      WebResponse resp = req.GetResponse();
      var json = new DataContractJsonSerializer(typeof(Todo));
      var todo = (Todo)json.ReadObject(resp.GetResponseStream());
      Console.WriteLine(todo);
    }
  }

  [DataContract]
  public class Todo
  {
    [DataMember]
    public bool completed { get; set; }
    [DataMember]
    public int userId { get; set; }
    [DataMember]
    public int id { get; set; }
    [DataMember]
    public string title { get; set; }

    public override string ToString()
    {
      return $"{completed} {id} {title}";
    }
  }
}
*/
```

Quellcode 3: API Aufruf und Deserialisierung aus natürlicher Sprache

Nüchtern betrachtet bieten solche Modelle dem Entwickler jedoch kaum eine Zeitersparnis und die Ergebnisse müssten zudem stets manuell auf Korrektheit überprüft werden. Als unterstützendes Werkzeug können solche Modelle jedoch nützlich sein. Wenn der Entwickler beispielsweise nicht mit einer Programmiersprache vertraut ist, können Befehle (wie z.B. die Ein- und Ausgabe) mittels natürlicher Sprache umschrieben werden. Für die Erstellung komplexer Software sind solche Generierungsverfahren jedoch ungeeignet, da die Umschreibung mit natürlicher Sprache meist zu unscharf und inkonsistent ist [Ch21].

Weitere Möglichkeiten, die jedoch nur gewisse Spezialgebiete der Softwareentwicklung abdecken sind folgende [Do08]: Annotationen, Präprozessoranweisungen, O/R-Mapper, Interface-Definition-Languages oder Codeerzeugung durch grafischen Oberflächendesigner.

3.2 Vorteile und Grenzen

Codegenerierung kann in der Softwareentwicklung unterschiedlich eingesetzt werden und bestimmte Vorgänge erleichtern (wie bereits in den Unterabschnitten beschrieben). Während die modellgetriebene Softwareentwicklung ein gesamtes Paradigma darstellt, sind Werkzeuge wie OpenAI Codex nur für bestimmte Randgebiete sinnvoll. Unabhängig vom verwendeten Codegenerator gilt weiterhin das bekannte Garbage In, Garbage Out Prinzip der Informatik. Darüber hinaus muss für jeden Anwendungsfall abgeschätzt werden, ob das verwendete Werkzeug einen Zeit, Qualitäts oder Produktivitätsgewinn darstellt oder eher hinderlich ist. Zu erwähnen ist außerdem, dass diese Entwicklung noch lange nicht abgeschlossen ist. Es bleibt abzuwarten was bessere Modelle und immer mehr Beispieldatensätze in Zukunft ermöglichen werden.

4 Analyse

Wurden Teile eines Programms programmiert, ist es wichtig dessen Funktion und Leistung zu überprüfen. Neben der Identifikation von Problemen geht es dabei vor allem um Optimierungen. Dieser Abschnitt beschäftigt sich mit automatisierten Verfahren zur Analyse von Software. Zunächst werden die allgemeinen Konzepte statische und dynamische Analyse erklärt. Darauffolgend werden die Vorteile und Grenzen dieser Verfahren erläutert. Abschließend wird ein Werkzeug zur Analyse von Software vorgestellt.

4.1 Allgemeine Konzepte

Die **statische Quellcodeanalyse** ist ein Verfahren, welches Quellcode unabhängig von der Kompilierung oder Ausführung nach verschiedenen Kriterien auswertet. Dies soll Fehler, Inkonsistenzen und Unsicherheiten im Programmcode aufdecken. Statisch bedeutet in diesem Kontext, dass der Code nicht ausgeführt, jedoch unter anderem auch semantisch ausgewertet wird. Verwendete Verfahren sind [Fo08, p. 12]:

(a) **Taint Analyse**: Analyse zur Verhinderung von böswilligen Benutzereingaben, die Code auf dem Hostcomputer ausführen (z.B. SQL Injection)

(b) **Datenfluss Analyse**: Analyse, welche Daten zwischen Programmteilen ausgetauscht werden und welche Abhänigkeiten daraus entstehen.

(c) **Kontrollfluss Analyse**: Analyse zur Evaluation und Integritätsprüfung des Programmablaufes mit dem Ziel der Feststellung von Anomalien (z.B. Endlosschleifen)

(d) **Lexikalische Analyse**: Syntaktische Prüfung des Quellcodes und Generierung von Tokens

Historisch ist das Programmierwerkzeug Lint der Pionier der statischen Codeanalyse. Ursprünglich wurde Lint für die Programmiersprache C entwickelt. Nach und nach wurden Abwandlungen für andere Programmiersprachen, darunter beispielsweise JavaScript, TypeScript und Python, entwickelt. Die Entwicklung von Lint lief synergetisch zu der Entwicklung moderner Compiler [Da88, p. 2]. Ursprünglich sind Lint Programme so gedacht, dass diese vom Benutzer explizit ausgeführt werden müssen und anschließend den Quellcode analysieren. In modernen IDE sind Lint-ähnliche Verfahren meist automatisch integriert. Diese analysieren kontinuierlich den Quellcode und zeigen mögliche Probleme direkt an (Abbildung 4). Zudem werden solche Analysewerkzeuge als zusätzliche Plugins für viele IDEs angeboten [Da88][Wi22a].

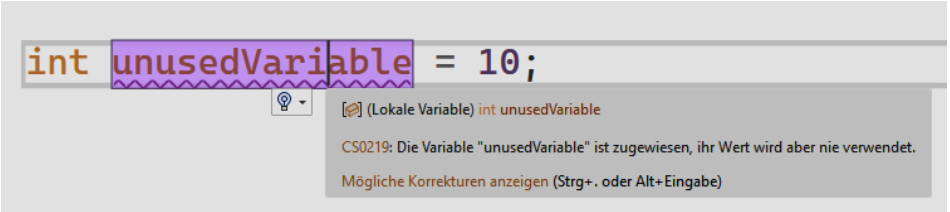

Abb. 4: Statische Codeanalyse in Visual Studio

Die Möglichkeiten der statischen Codeanalyse lassen sich in die Kategorien *Stylechecking*, *Semantische Analyse* und *Weiteres* unterteilen.

Bei der Softwareentwicklung wird häufig im voraus ein Programmierstil, auch oft Coding Conventions genannt, festgelegt. Dieser baut auf den üblichen Vorgehensweisen des verwendeten Programmierparadigma auf. Die meisten Programmiersprachen bieten ein Katalog von solchen Codierungsrichtlinien an. So existiert beispielsweise für die objektorientierte Sprache Java ein Dokument mit einer Empfehlung für einen zu verwendenden Programmierstil [Or97].

Für Projekte ist es gegebenenfalls sinnvoll die Codierungsrichtlinien der verwendeten Programmiersprache anzupassen und zu erweitern. Beispiele für Codierungsrichtlinien sind die Anwendung von Entwurfsmustern, die Festlegung der Namenskonventionen, der Umfang der Dokumentation oder die Gestaltung von Funktionsaufrufen.

Ein **Style Checker** ist ein Werkzeug, welches die Einhaltung des definierten Programmierstils überprüft. Dies ermöglicht es beispielsweise Verstöße gegen Benennungskonventionen oder einer Überschreitung der maximal zulässigen Zeilenlänge von Code zu erkennen. Style Checker bilden jedoch nur eine Teilmenge davon ab, was mit statischer Codeanalyse möglich ist, da meistens keine semantische Quellcodeanalyse vorgenommen wird. So kann beispielsweise nicht erklärt werden, dass in Quellcode 4 eine Endlosschleife produziert wurde [gm12][Wi22d].

```
1  while (1<10)
2  {
3  }
```

Quellcode 4: Endlosschleife

In der **semantischen Analyse** wird der Quellcode auf inhaltliche Konsistenz und mögliche Fehler geprüft. So können beispielsweise folgende Probleme erkannt werden: Memory Leaks, Pufferüberläufe, Divisionen durch null, Array Zugriffe außerhalb der Grenzen, Endlosschleifen, mögliche Nullverweise, nicht verwendete Variablen und Methoden, Anti-Patterns, mögliche Code Injections, Substitutionen und Generalisierungen oder toter Quellcode. Diese Art der Analyse ist besonders wichtig, da sie folgenschwere Fehler aufdecken soll.

Wird ein objektorientiertes Programmierparadigma verwendet besteht eine **weitere Möglichkeit** in der Bestrebung durch statische Quellcodeanalysen das Design zu bewerten. Dies ist natürlich nicht vollumfänglich möglich, jedoch gibt es bestimmte statisch ermittelbare Metriken, die auf ein gutes oder schlechtes Design schließen lassen. Diese lassen sich aus den Prinzipien für ein agiles, objektorientiertes Design ableiten [Ma02]. Folgende Metricken sind hierfür denkbar und teilweise auch schon in statischen Analysewerkzeugen implementiert [Fo08][Da88][Jeb][Wi22c]:

(a) Lines of Code (Single Responsiblility)
  - Innerhalb einer Komponente, Komponente oder Methode

(b) Ringabhänigkeiten

(c) Coderedundanzen (Don't repeat yourself, Abstraction)

(d) Große Vererbungsstrukturen (Composition over inheritance)

(e) Abhänigkeiten auf Implementierungen statt auf Interfaces (Dependency Inversion, Open-Closed)

(f) Größe der Interfaces (Interface Seggregation)

(g) Fehlende Implementierungen für Basismethoden (Liskov Subsitution)

Bei der **dynamischen Analyse** liegt der Fokus auf Bereichen, die nicht von der statischen Quellcodeanalyse abgedeckt werden können. Dies sind primär Fehler in der Anwendungslogik, fehlende Sicherheitsvalidierung und Performance.

Wesentlich wird die dynamische Analyse von sogenannten Profiling Werkzeugen abgewickelt. Diese können verschiedene Messwerte zu der entwickelten Software ermitteln. Dazu gehört die Anzahl der Funktionsaufrufe und Funktionsdurchläufe, die Speicherauslastung, nicht freigegebenen Speicherbereiche, nebenläufige Prozesse und Deadlocks. Durch statistische Analysen kann herausgefunden werden, welche Programmteile lohnenswert optimierbar sind, um eine bessere Leistungs- und Speicherperformance zu erhalten. Auch Fehler in der Anwendungslogik, die sich in Deadlocks oder Memoryleaks manifestieren, können aufgedeckt werden [Pa19][Wi22b][No22][Sc19a].

4.2 Vorteile und Grenzen

Durch Automatisierte Analyseverfahren können tausende Zeilen Quellcode in kürzester Zeit mit einer sehr hohen Präzision durchsucht und ausgewertet werden. Mögliche Problemstellen werden unmittelbar angezeigt und können meist auch zur Quelle zurückverfolgt werden (z.B. nicht validierte Parameter). Über Konfigurationen können die Codeconventions festgelegt werden und Antipatterns definiert werden, die anschließend bei der statischen Codeanalyse überprüft werden. So kann auch in großen Projekten eine Einheitlichkeit und Stabilität im Quellcode erreicht und somit die Softwarequalität erhöht werden. Häufig auftretende Probleme oder Muster, die zu Schwierigkeiten im Entwicklungsprozess führen können, sind in den Analysewerkzeugen bereits vorkonfiguriert. Intellegente Analysewerkzeuge bieten zudem die Möglichkeit einer automatischen Problembehebung. Hier wird der Quellcode automatisch refaktorisiert (Abschnitt 5) oder abgeändert.

Auch wenn Analysewerkzeuge eine große Hilfe für Softwareentwickler darstellen, ist es wichtig die Grenzen dieser zu kennen. Analysewerkzeuge sind nur so gut, wie sie programmiert und konfiguriert wurden. Sind bestimmte Antipatterns oder Konstrukte für unsauberen Code nicht in der Konfiguration und Programmierung berücksichtigt, werden diese auch nicht erkannt. Zudem lassen sich auch technisch nicht alle Problemfelder ermitteln. So können Logikfehler in komplexen Geschäftsprozessen häufig nicht identifiziert werden, da das Analysewerkzeug kein Verständnis hierfür besitzt. Auch bei Framework-spezifischen Anwendungsfeldern können Probleme auftreten, da die Analysewerkzeuge auf die Framework Version abgestimmt sein müssen. Werden spezielle Techniken verwendet, wie z.B. Dependency Injection, die bei der Implementierung des Werkzeuges nicht berücksichtigt sind, kann es ebenfalls zu Schwierigkeiten kommen. So kann es in diesem Fall sein, dass Abhängigkeiten nicht mehr nachvollzogen werden können, weil keine klassische Referenzerzeugung und Übergabe mehr stattfindet. Sprachen die ständig weiter entwickelt werden, erzwingen ebenfalls eine kontinuierliche Weiterentwicklung der zugehörigen Analysewerkzeuge [Fo08][Wi22c][Sc19a].

4.3 Beispiel Werkzeug

Als eine Suite von Beispielwerkzeugen werden die Produkte des Softwareherstellers JetBrains gewählt. Die Werkzeuge sind entweder in die IDEs, die von Jetbrains angeboten werden integriert, oder als seperates Plugin für Visual Studio oder den Standalonebetrieb verfügbar. Angeboten werden:

(a) ReSharper (Statische Codeanalyse, Refactoring)

(b) DotTrace (Dynamische Analyse, Leistungsprofiling)

(c) DotMemory (Dynamische Analyse, Memoryprofiling)

(d) DotCover (Statische und dynamische Analyse, Testabdeckung, Unittesting)

Jetbrains bietet elf IDEs für verschiedene Programmiersprachen an, die alle auf einem gemeinsamen Framework basieren [Je20]. Somit werden die Sprachen Objective C, Swift, PHP, Python, C#, Visual Basic, Go, Rust, Ruby, Kotlin, JavaScript und Java unterstützt [Jea].

5 Refactoring

Nach einer Analyse stehen in der Regel Änderungen im Code an. *Refactoring*, oder zu Deutsch *Refaktorierung*, beschreibt die nachträgliche Veränderung von Quellcode in Software. Konkret wird unter Beibehaltung des Verhaltens eines Programms eine verbesserte Struktur erzielt. Es ist also abgegrenzt von der Entwicklung neuer Funktionalitäten. Dies sollte vor allem bei großen Projekten ständiger Bestandteil sein. Schlechter Code erzeugt sonst immer größere Probleme und senkt somit auf Dauer die Produktivität [Ma09, S. 28]. Stattdessen sollte sauberer Code angestrebt werden. Dies bezieht sich sowohl auf Softwarearchitektur, als auch generelle Lesbarkeit von Code. Clean Code wird zum einen durch die im vorherigen Kapitel (Abschnitt 4) genannten Kriterien, oder auch die ISO-Normen, auf höherer Ebene analysiert. Eine Sicht auf niedrigerer Ebene bietet Grady Booch, Autor von *Object-Oriented Analysis and Design with Application*:

> *Sauberer Code ist einfach und direkt. Sauberer Code liest sich wie wohlgeschriebene Prosa. Sauberer Code verdunkelt niemals die Absicht des Designers, sondern ist voller griffiger (engl. crisp) Abstraktionen und geradliniger Kontrollstrukturen.* [Ma09, S. 34]

Durchgehend sauberer Code ist zum Beispiel aufgrund von sich verändernden Anforderungen schwer zu erreichen. Daher ist Refactoring ein derart wichtiger Prozess. Da Refactoring aber auch bedeutet, dass in dieser Zeit keine neuen Funktionen entwickelt werden, ist die Balance sehr wichtig.

5.1 Betroffene Stellen

Betroffen sind zum Einen die Ergebnisse von Analysen, welche Probleme aufgezeigt haben. Diese wurde im vorherigen Kapitel (siehe Abschnitt 4 erläutert). Weitere unsaubere Stellen im Code werden auch "Code-Smell"genannt [Fo00, S. 67]. Martin Fowler kategorisiert diese in 22 Arten der Probleme [Fo00, S. 67 - S. 82]. Diese können teilweise in der Analyse erkannt werden, erfordern aber oftmals auch manuelle Überprüfung. Im Folgenden sind die wichtigsten zusammengefasst und erklärt:

(i) **Redundanter Code** verringert die Änderbarkeit, sorgt für doppelte Entwicklung und großen Overhead.

(ii) **Große Methoden, Parameterlisten oder Klassen** sind schwer verständlich und weniger Wiederverwendbar. Weitere mögliche Probleme sind schlechte Testbarkeit und Verletzung des Single Responsibility Prinzips.

(iii) **Hinzufügen von nicht zugehörigen Funktionalitäten zu einer Klasse** verletzt das *Single Repsonsibilty*-Prinzip und verschlechtert somit die Wartbarkeit. Gleiches gilt für das Gegenteil, also die Aufteilung einer Funktionalität auf verschiedene Klassen. Jede Klasse sollte also eine Verantwortlichkeit haben.

(iv) **Unübersichtliche Gruppen aus zusammengehörigen Parametern anstelle der Nutzung eines Objektes** schaden der Lesbarkeit und sorgen für geringere Wartbarkeit. Änderungen müssen immer an allen Stellen, statt nur in der definierten Klasse erfolgen.

(v) **Verschachtelte Switch-Befehle** behindern die Lesbarkeit. Je mehr Ebenen im Code sind, desto unübersichtlicher wird es.

(vi) **Nachrichtenketten** schaden der Performance durch zu viele Zwischenaufrufe.

(vii) **Klassen ohne eigene, nicht-triviale Verantwortlichkeit** schaden der Wartbarkeit und deuten auf schlechte Architektur hin.

(viii) **Unangebrachte Abhängigkeiten von Details statt Schnittstellen** sorgen für schlechte Austauschbarkeit. Gleiches gilt für funktional gleiche Klassen, welche unterschiedliche Schnittstellen anstelle einer gemeinsamen besitzen.

(ix) **Übermäßige Kommentare** implizieren eine Unverständlichkeit des Codes und schaden der Lesbarkeit.

(x) **Nichtssagende Namen** sorgen für schlechte Lesbarkeit und Problemen bei späteren Änderungen.

Es könnten viele weitere genannt werden, genauere Details bietet auch Robert Martin in seinem Buch *Clean Code* [Ma09].

5.2 Allgemeine Konzepte zur Lösung

In seinem Buch *Refactoring* stellt Martin Fowler einen großen Katalog von möglichen Refactorings auf [Fo00, S. 99 - 387]. Im Folgenden findet eine Betrachtung einiger ausgewählter Refactorings statt, welche automatisch von Software umgesetzt werden können. Viele weitere sind manuell durchführbar, aber nicht Teil dieser Arbeit.

(a) **Extrahieren oder Verschieben von Methoden und Klassen**: Ist eine Methode zu lange (siehe i), kann ein Teil von ihr ausgewählt und in eine eigene Methode extrahiert werden. Die Software erkennt Rückgabewert und Parameter automatisch, nur der Methodenname muss gewählt werden. Problematisch zur Auslagerung können dabei die lokalen Variablen sein. Diese müssen oftmals von Hand angepasst werden, zum Beispiel durch die Aufteilung in kleinere Variablen. Klassen dagegen werden oftmals in einer anderen Klassendatei definiert. Software ermöglicht das Verschieben in eine eigene Datei, was der Übersichtlichkeit dient.

(b) **Vereinfachen von Ausdrücken und Statements**: Aufgrund vorheriger Analysen wurden mögliche Vereinfachungen im Code erkannt (Unterabschnitt 4.3). Die Software macht daraufhin Vorschläge zur Umwandlung. Dies kann beispielsweise die Invertierung einer If-Abfrage sein, welche für höhere Übersichtlichkeit sorgt. Oder auch die Umwandlung einer Verkettung von If-Abfragen in ein Switch-Statement. Ein anderes Beispiel wäre die Vereinfachung eines bedingten Ausdruckes, da die gleiche Logik in kompakterer Form erreicht werden kann.

(c) **Umbenennung von Bezeichnern**: Aussagekräftige Bezeichner sind besonders wichtig (siehe x), da Quellcode sich selbst erklären soll. Software ermöglicht das Umbenennen eines Bezeichners an all seinen Vorkommen. Dies erspart Arbeit und die Gefahr, nicht alle Verwendungen zu finden. Oftmals weißt ein Stylechecker (Abbildung 4.1) auf problematische Bezeichner hin und bietet Gegenvorschläge.

(d) **Migration von Datentypen**: Wird einer Variable oder einem Feld ein falscher Typ zugewiesen, muss dieser migriert bzw. geparst oder der Datentyp des Felder verändert werden. Software kann diese Lösungen automatisch vornehmen.

(e) **Einführung von Feldern oder Variablen**: Oftmals findet eine hohe Verschachtlung statt, was ein Zeichen für Code-Smell ist (siehe v). In der Regel ist eine Extraktion in erklärende Variablen sinnvoll. Dies kann durch Software erledigt werden, nur die Auswahl des Namens muss stattfinden.

(f) **Parameter ergänzen, entfernen oder überladen**: Wird eine Methode aufgerufen mit anderen Parametertypen, als aktuell definiert, gibt es verschiedene Möglichkeiten. Entweder wird die Parameterliste verändert oder es muss eine Überladung der Methode mit passenden Parametern erzeugt werden. Dies kann eine Software automatisch übernehmen.

(g) **Schnittstelle implementieren**: Implementiert eine Klasse ein Interface, muss es alle dessen Methoden überschreiben. Software ermöglicht das Einfügen aller Methoden des Interfaces mit leerem Methodenkörper.

(h) **Attribute kapseln**: Ein Kernprinzip der objektorientierten Programmierung ist die Datenkapselung über Getter und Setter. Diese können ausgehend von einem Feld automatisch erzeugt werden durch Software.

(i) **Maximale Abstraktion verwenden**: Sofern es möglich ist, sollte immer maximale Abstraktion und minimales Detail verwendet werden (siehe viii). Wird daher in der Analyse erkannt, dass eine höhere Abstraktion verwendbar ist, kann der Objekttyp automatisch ausgetauscht werden.

(j) **Verwenden und Austausch von Modifzierern**: Verschiedene Modifizierer für Variablen können Vorteile bieten, so beispielsweise wenn ein String als Konstante definiert wird. Wurde in der Analyse erkannt, dass dies möglich ist, kann der Modifizierer durch die Software automatisch eingefügt werden. Gleiches gilt bezüglich Sichtbarkeit. Ist eine höhere Kapselung möglich, also beispielsweise protected statt public, kann die Änderung automatisch vollzogen werden.

(k) **Formatierung**: Neben Architektur und Benennungen ist vor allem die Formatierung des Codes ausschlaggebend für Übersichtlichkeit. Software bietet hier die Möglichkeit, diese nach einem definierten Schema vorzunehmen. Es handelt sich um ein sogenanntes *Beautify* des Codes.

(l) **Entfernung von totem Code**: Wird in der Analyse Code erkannt, der nie erreicht wird, kann dieser durch Software automatisch entfernt werden. Somit wird toter Code verhindert.

(m) **Nutzung von besseren Sprachfeatures**: Viele Programmiersprachen entwickeln sich ständig weiter. Neue Features sind dem Programmierer aber oftmals unbekannt oder ihr Nutzen nicht verständlich. Wird bei der Analyse eine Stelle erkannt, welche besser durch ein neues Sprachfeature ersetzbar ist, kann Software dies automatisch umwandeln. Gleiches gilt für veraltete Features, welche verwendet werden. Beispielsweise wäre dies die Nutzung eines Switch anstelle von If-Else.

Die vorgestellten Konzepte können je nach Refactoring-Tool um weitere Fähigkeiten ergänzt sein.

5.3 Nutzen und Risiken

Der hauptsächlichen Vor- und Nachteile von Refactoring allgemein wurde zu Beginn des Kapitels bereits erläutert. Im Folgenden soll dies in Bezug auf den Einsatz von Software zum Refactoring analysiert werden.

Hauptsächlicher Vorteil ist die Einsparung von trivialer und redundanter Arbeit. Alle durch die Software automatisch erledigten Aufgaben, welche nach der Analyse anstehen, könnten auch manuell umgesetzt werden. Dazu wäre aber ein Vielfaches an Arbeit notwendig. Der Entwickler spart sich stattdessen viele Klicks und Tastenanschläge (siehe a-m), sowie Recherche und Denkarbeit zur genauen Umsetzung einer Aktion (siehe b, i, j, m). Außerdem sorgt die Umsetzung nach klaren Regeln dafür, dass weniger Fehler geschehen. Die manuelle Umwandlung birgt immer das Risiko von Denk- oder Schreibfehlern, sowie Unvollständigkeit (siehe beispielsweise a-c). Die Software garantiert, dass alle betroffenen Stellen identifiziert und bearbeitet werden. Durch die einfache Umsetzung und höhere Sicherheit ist die Motivation zum Refactoring höher. Aus den Umwandlungen resultierende Vorteile können performanterer (siehe j, m) und verständlicherer Code (siehe a-m) sein.

Refactoring-Software bringt jedoch auch Gefahren mit sich. Entwickler haben die Gefahr, sich zu sehr auf derartige Tools zu verlassen und viele Dinge nur noch mit ihrer Hilfe umsetzen zu können. Wenn beispielsweise verschiedene Datentypen unterschiedliche Performance erreichen und dies relevant für die Anwendung ist, wäre es falsch keine eigene Recherche durchzuführen und sich auf die Software zu verlassen. Diese zieht Performance gegebenenfalls einfach nicht in Betracht (beispielsweise bezogen auf b, d, i, m). Ein weiterer Punkt ist, dass Code nur aufgrund von noch fehlenden Implementierungen zum Zeitpunkt der Analyse Gründe zum refaktorisieren hat, welche später hinfällig sind (beispielsweise durch i, j, l). Die Einfachheit des Refactoring verleitet hierbei ggf. zu verfrühten Aktionen. Dies bezieht sich beispielsweise auf die öffentliche Sichtbarkeit von Methoden einer Klasse, welche aktuell aber nur privat verwendet werden. Später haben diese jedoch eine öffentliche Verwendung. Derartige Software muss also vorsichtig genutzt werden, da großflächige Änderungen schnell durchgeführt sind.

Aktuelle Studien legen nahe, dass im Jahr 2021 die meisten Refactorings noch manuell und ohne Tools durchgeführt werden [EM21]. Grund dafür ist unter anderem fehlendes Vertrauen. Oftmals wird Code an vielen Stellen verändert und zum Nachvollziehen dieser Änderungen müssen Tools wie GitHub verwendet werden. Auch fehlt das Hintergrundwissen, wie genau eine Änderung vollzogen wird. Nicht zuletzt benötigen die Tools zumindest für komplexere Operationen auch eine Einarbeitungszeit. Es ist möglich, dass sich anfangs die Zeiteinsparung kaum lohnt, da der Weg zum Refactoring ohne genaue Kenntnisse zu lange ist [EM21].

5.4 Marktanalyse

ReSharper ist ein kostenpflichtiges Tool von JetBrains. Die Preise variieren je nach Paket, liegen aber etwa bei 350€ je Nutzer pro Jahr. Mit Laufzeit oder dem Preis größerer Pakete verringert sich der Preis für einzelne Tools. Es kann in C#, Visual Basics, XAML, ASP.NET, ASP.NET MVC, Python, JavaScript, TypeScript, Ruby und Rails, CSS, HTML, XML, Java, JSON, PHP und C++ verwendet werden. In einigen Sprachen ist *ReSharper* direkt in der IDE inkludiert (beispielsweise PyCharm oder InelliJ), bei den restlichen Sprachen handelt

es sich um eine Erweiterung für Visual Studio. Die konkreten Features je Sprache sind auf der offiziellen Website zu finden [Re22c]. ReSharper bietet alle vorgestellten Konzepte zur Refaktorierung an, sowie viele weitere. Besonders mehr als 1200 sogenannte *Quick-Fixes* an analysierten Fehlern kann *ReShaper* durchführen. Dazu gehörigen beispielsweise folgende zuvor nicht aufgelistete:

(i) Hinzufügen eines `return`-Statements oder Umwandeln des Rückgabetypes in `void`, sofern dieses fehlt.

(ii) Passendes *escape* von sensiblen Zeichen im String, beispielsweise Backslash in einem Dateipfad.

(iii) Korrektur oder Import nicht aufgelöster Symbole, beispielsweise durch ein zugehöriges Paketsystem.

(iv) Konvertierung von Interfaces in abstrakte Klassen und umgekehrt.

(v) Ersetzen eines klassischen Konstruktors durch das Pattern der Factory Methode.

(vi) Benutzerdefinierte Aktionen, welche bei Erkennung von definierten Mustern vorgeschlagen werden.

(vii) Und viele mehr, siehe Dokumentation [Re22a].

Abbildung 5 zeigt einiger der Konzepte (siehe c, h, j, m) am Beispiel C#-Code in Visual Studio auf. Eine genaue Auflistung aller Features ist auf der Website zu finden [Re22b].

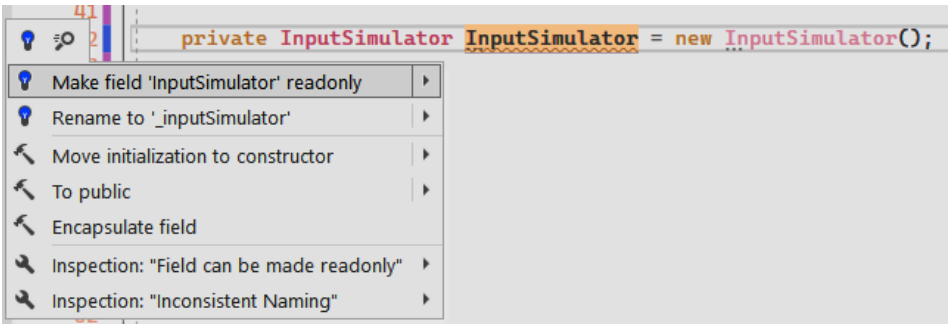

Abb. 5: Refactoring Vorschläge von ReSharper

Alternative Tools für eine derart große Anzahl an unterschiedlichen Sprachen und Funktionen sind aktuell keine auf dem Markt. Kostenlose Alternativen wie **CodeRush** beschränken sich zumeist auf wenige Sprachen [De22]. Allerdings bieten viele IDEs einige dieser Funktionen out of the box an und viele Entwickler arbeiten generell nur mit einer Programmiersprache. Daher können andere, für die jeweilige Programmiersprache geeignete, Tools verwendet werden.

6 Dokumentation

Um ein Projekt mit entwickeltem Code langfristig weiterentwickeln und warten zu können, ist die Dokumentation von diesem notwendig. Als *Softwaredokumentation* ist sämtliche Dokumentation einer Software im Entwicklungsprozess und darüber hinaus zu verstehen. Über den gesamten Entwicklungsprozess werden unterschiedlichste Dokumentationen erstellt. Sie dienen dazu, die Funktionalität, den Nutzen und die Entwicklung der Software für die verschiedenen Stakeholder nachvollziehbar zu machen. Dabei ist auch die Anzahl der Entwickler und die Größe des Projekts zu vernachlässigen. Eine gute Softwaredokumentation ist auch bei nur einem Entwickler essenziell und gewinnt bei größeren Kollaborationen nur noch mehr an Wichtigkeit. Die Softwaredokumentation muss alle wichtigen Fragen der mitwirkenden beantworten können und Aufschluss über die Software geben.

> *Documentation is highly valued, but often overlooked.*
>
> —opensourcesurvey

Eine großangelegte Umfrage von Open Source[4] zeigt auf, dass eines der Hauptprobleme bei der Entwicklung freier Software eine unvollständige oder verwirrende Dokumentation ist. Vielen Entwicklern ist die Wichtigkeit der Dokumentation nicht bewusst. Doch Schlechte Dokumentation kostet Geld [Gi17]!

Die Dokumentation geht mit einer erfolgreichen Kollaboration in einem Entwicklerteam einher. Hierbei sind die verschiedenen Entwickler auf Informationen über Schnittstellen, Funktionen einzelner Module und den Änderungsverlauf im Projekt angewiesen. Für die automatische Dokumentation solcher Informationen stehen dem Entwickler verschiedene intelligente Werkzeuge zur Verfügung. Diese helfen bei der Erstellung der Dokumentationen und verbessern den Entwicklungsprozess. Neben der automatischen Generierung wird auch die Qualität und Konsistenz der Dokumente eingehalten und verbessert.

Entscheidend ist die Auswahl der Hilfsmittel. Es existiert eine Vielzahl an Hilfsmitteln für verschiedene Arten der Softwaredokumentationen. Bei der Auswahl der Hilfsmittel müssen verschiedene Kriterien, die der Entwicklungsprozess mit sich bringt, berücksichtigt werden.

6.1 Allgemeine Konzepte

In dem Prozess der Softwareentwicklung gibt es viele verschiedene Arten von Dokumentationen. Dabei sind nicht immer alle Arten von Dokumentationen notwendig. Dokumentationen können in verschiedene Bereiche und unterschiedliche Zielgruppen unterteilt werden. Je nach Vorgehen im Entwicklungsprozess und dem Vorhandensein der Zielgruppen, müssen die notwendigen Dokumentationen individuell festgelegt werden. Trotz einer Vielzahl

[4] Softwaregruppierung zur Entwicklung freier Software. (Vgl. https://opensource.com/)

an unterschiedlichen Dokumentationsmöglichkeiten lassen sich zwei Hauptkategorien herausarbeiten.

Bei der **Projektdokumentation** wird der Fokus auf das Vorgehen, die Methodik, und die Werkzeuge gelegt. Sie ist für die verschiedenen Stakeholder und soll Aufschluss über den Verlauf des Projektes geben.

Die **Systemdokumentation** hingegen beschreibt das Produkt. Genauer beschreibt sie, aus was das Produkt besteht und wie es funktioniert und vorgeht. Diese Arte der Dokumentation ist primär für den Entwickler. Gerade bei großen Teams oder späterer Weiterentwicklung ist diese Art der Dokumentation sehr wichtig.

Das Feld der **Projektdokumentation** bezieht sich wie beschrieben, auf den Verlauf und die Planung des Entwicklungsprozesses. Hierbei sind die Abläufe projektabhängig und werden Kunden-/Produktspezifisch angepasst. Intelligente Werkzeuge können aufgrund der Individualität von Projekten nur begrenzt eingesetzt werden. Die unterstützenden Werkzeuge, die hierbei verwendet werden, sind meist auf die allgemeinen Hilfsmittel einer Projektplanung zurückzuführen. Anders verhält sich dies jedoch im Bereich der **Systemdokumentation**. Dort unterstützen sie bei der Dokumentation des eigentlichen Umsetzungsprozess und bei der Beschreibung des Produkts.

Ein nützliches Werkzeug zur Dokumentation des Aufbaus einer Anwendung ist das **automatische Generieren von UML-Diagrammen**. UML-Diagramme sind ein wichtiger Bestandteil der Anwendungsdokumentation. Sie dienen in der Softwareentwicklung als ein Standard zur Visualisierung des Systementwurfs [Am04]. Die UML-Diagramme begleiten den ganzen Entwicklungsprozess der Software. Sie sind sowohl bei der Planung ein wichtiger Bestandteil, als auch bei der dynamischen Umsetzung der Software. Bei der Umsetzung kann der Aufbau in den verschiedenen Stadien visualisiert und so nachverfolgt werden. Die automatische Generierung der Diagramme an sich, ist schon ein wichtiges intelligentes Werkzeug für den Entwickler. Jedoch bieten UML-Diagramme noch mehr Möglichkeiten den Entwicklungsprozess zu erleichtern. So wie ein Diagramm aus dem Qulltext automatisch generiert werden kann, so ist auch Quelltext aus einem Diagramm generierbar. Das bietet dem Entwickler während der Umsetzung die Möglichkeit, Klassen und andere Typen grafisch zu erstellen, zu ändern und zu löschen [Te22].

Inline-Kommentare sind lesbare Kommentare innerhalb des Quelltextes. Sie werden dem Quelltext hinzugefügt, um ihn verständlicher und nachvollziehbarer zu machen. Dies ist wichtig, wenn mehrere Entwickler gemeinsam an einer Anwendung arbeiten. Es kann viele verschiedene Gründe geben, einen Quelltext-Abschnitt oder eine Zeile mit Kommentaren zu versehen. Folgend sind Gründe für Inline-Kommentare aufgelistet.

(a) **Planen und Überprüfen:** Es kann vor dem Beginn der eigentlichen Programmierung anhand von Kommentaren die Umsetzung geplant werden. So kann an den konkreten Stellen, wie bspw. leeren Methoden, ein Kommentar mit der Beschreibung hinterlassen

werden. Anhand dieser Beschreibung übernimmt ein Entwickler dann die konkrete Implementierung (siehe Quellcode 5).

(b) **Beschreibung des Quelltextes:** Wie im vorherigen Paragraphen beschrieben, ist es wichtig einen schnellen Überblick über den Quelltext zu erhalten. Gerade wenn mehrere Entwickler an dem selben Projekt arbeiten, ist eine Beschreibung am Beginn einer Passage und relevanten Stellen (Methoden, Klassen, ...) sehr hilfreich.

(c) **Beschreibung des Algorithmus:** Noch wichtiger sind Kommentare bei unübersichtlichen Algorithmen. Auch wenn Quellcode eigentlich selbsterklärend sein soll, ist dies bei komplexen Algorithmen oftmals nicht machbar. Meist ist nicht auf den ersten Blick ersichtlich, was ein Algorithmus bewirkt. Daher kann eine vorhergehende Beschreibung und Kommentierung ausgewählter Stellen sinnvoll sein.

(d) **Verwendung von Ressourcen:** Wird eine externe Ressource verwendet, ist es hilfreich Informationen über den Speicherort und die verwendete Version der Ressource zu hinterlassen. Auch die offiziellen Namen sind bei der Verwendung von Abkürzungen im Quelltext als Kommentare hilfreich.

(e) **Debugging:** Beim Debugging ist es hilfreich, an bestimmten Stellen Markierungen zu setzen, um den Quelltext während des Debuggings übersichtlich zu halten.

Die Realisierung von Kommentare ist in unterschiedlichen Programmiersprachen unterschiedlich gestaltet. Folgend ist eine Übersicht über einige der verwendeten Token zum Realisieren von Kommentaren dargestellt.

Symbol	Programmiersprache
REM	BASIC, Batch files
::	cmd.exe
#	Cobra, Perl, Python, Ruby, Make, Windows PowerShell, PHP
%	TeX, Prolog, MATLAB
//	C (C99), C++, C#, D, F#, Go, Java, JavaScript, Kotlin

Tab. 1: Tokens zu Beginn eines Kommentars in unterschiedlichen Programmiersprachen

Ein wichtiges Werkzeug, welches Inline-Kommentare ermöglicht und viele Entwicklungsumgebungen unterstützt, ist das verwenden von Tags. Hierbei handelt es sich um Begriffe, die einem Kommentar vorangestellt werden, sodass dieser kategorisierbar ist. Dieses Werkzeug erleichtert das Auffinden von bestimmten Stellen im Quelltext immens. Dabei gibt es verschiedene Tags für unterschiedliche Kategorien. Als Beispiele soll hier *TODO* genannt werden (Quellcode 5). Dieser Tag kann an Stellen in einem Kommentar erwähnt werden, an welchen es noch etwas zu implementieren gilt (siehe Quellcode 5). Es ist nun dem Entwickler die Möglichkeit gegeben, sich alle Kommentare mit diesem Tag auflisten zu lassen. Dadurch hat er eine schnelle Übersicht, an welchen stellen Änderungen notwendig sind und kann automatisch zu diesen navigieren. Durch den Inhalt des Kommentars, ist auch auf einen Blick zu erkennen, was zu tun ist. Neben dem genannten *TODO* Schlüsselwort gibt

es noch weitere typische Tags, die folgend aufgelistet sind [Ji16]: *BUG, DEBUG, FIXME, TODO, UNDONE*.

```
1  //Konstruktor der Klasse
2  public Program(int a, int b){
3
4    //TODO: addiere a + b
5
6  }
```

Quellcode 5: Beispiel von Inline-Kommentaren

Eine weitere Möglichkeit welche Inline-Kommentare bieten, ist die automatische Generierung einer Dokumentation des Quelltextes. Durch Einhaltung einer bestimmten Syntax, kann im Nachgang eine vollständige Dokumentation mit Beschreibungen der Klassen, Methoden und Parameter erstellt werden. Diese wird in der Regel als HTML-Dokument erzeugt und kann somit einfach freigegeben werden. Dies kann mit der UML-Diagrammerzeugung verglichen werden, wobei es sich hier nur um textuelle Beschreibungen handelt. Es stehen mehr die einzelnen Aufgaben der Klassen und Methoden im Vordergrund, als deren Zusammenhang.

Als Syntax der Kommentare können unterschiedliche Formate vorgegeben sein. Je nach verwendetem Generator kann es auch eine individuelle Syntax und Semantik des Generators sein. Ein bekanntes Format was auch genutzt wird, ist das XML-Format. Mit diesen verschiedenen Standards lassen sich Beschreibungen zu Elementen und Kommentaren erstellen, mit denen anschließend automatisch eine Dokumentation generiert werden kann.

6.2 Marktanalyse

Bezüglich der **Erzeugung von UML-Diagrammen** sind viele verschiedene Werkzeuge auf dem Markt vertreten. In den meisten kommerziell genutzten Entwicklungsumgebungen ist ein UML-Generator standardmäßig vorhanden. Dies unterscheidet sich aber je nach Entwicklungsumgebung. In den bekannten Entwicklungsumgebungen Visual Studio von Microsoft (Abbildung 6) und Intelij von JetBrains sind UML-Generatoren vorhanden. Bei Visual Studio ist der UML-Generator auch in der kostenlosen Community Version enthalten.

Zudem unterstützt nur Visual Studio das Erstellen und Bearbeiten von Quelltext mit Hilfe der UML-Diagramme. Dies hat in eigens durchgeführten Versuchen sehr gut funktioniert. Es konnten Anwendungsstrukturen mit Hilfe der UML-Diagramme erstellt werden. Auch das nachträgliche Umbenennen von Klassen und Methoden hat hervorragend funktioniert [Te22][Je22].

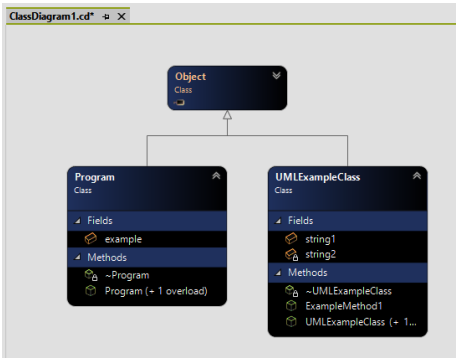

Abb. 6: Ein von Visual Studio generiertes UML-Diagramm

Ein Werkzeug, welches unabhängig von der Programmierumgebung arbeitet, ist Doxygen. Es handelt sich um das Standard Werkzeug für die automatische Dokumentationsgenerierung. Einige der unterstützten Programmiersprachen sind C++, C, Objective-C, C#, PHP, Java und Python [Do22].

Zur **Inline-Dokumentation** kann dagegen beispielhaft JavaDoc verwendet werden. Dieses ist in Intelij standardmäßig im Einsatz und verfügbar (siehe Quellcode 6).

```
1  /**
2  * Beschreibung der Methode
3  * @param integer1 Beschreibung Parameter 1
4  * @param integer2 Beschreibung Parameter 2
5  * @return Beschreibung des zu retunierenden Wert
6  */
7  public static String test(int integer1, int integer2){
8
9    return "foo";
10 }
```

Quellcode 6: Beispiel von Inline-Kommentaren

Die in Quellcode 6 verwendeten Kommentare erzeugen Abbildung 7.

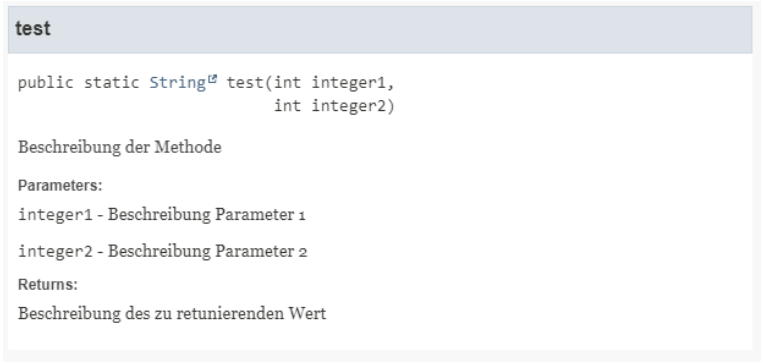

Abb. 7: Automatisch generierte Übersicht einer Java-Methode Quellcode 6

Dennoch gibt es auch in diesem Feld weiter Anbieter, die eine Dokumentationserzeugung aus Inline-Kommentaren erlauben. Um den erwähnte XML-Standard aufzugreifen, ist hier Visual Studio zu erwähnen. In Visual Studio ist ein Bordmittel integriert, mit welchem sich die XML-Kommentare in eine HTML-Dokumentation wandeln lassen. Dazu gibt es verschiedene Schlüsselwörter, welche zur Beschreibung verwendet werden. So wird zum Beispiel das Schlüsselwort *<summary>* zum Definieren einer Beschreibung eines bestimmten Elements verwendet werden. Des Weiteren können Parameter mit Kommentaren und Anmerkungen versehen werden. Dies geschieht mit dem Schlüsselwort *<param name="nameDesParameters»Beschreibung des Parameters</param>*.

Beim Verwenden der zwei Werkzeuge ist schon zu erkennen, dass sich das Prinzip nur leicht unterscheidet. Die Syntax und Semantik, mit welchen die Kommentare anzugeben sind, unterscheiden sich zwar im Format, jedoch ähneln sich die erzeugten Dokumentationen im Stil und Aufbau sehr.

Doxygen, wie schon erwähnt (Unterabschnitt 6.2), begrenzt sich nicht nur auf die UML-Diagrammerzeugung, sondern bietet auch in diesem Feld eine Lösung. Es kombiniert die Felder und gibt in der erzeugten Dokumentation sowohl die UML-Diagramme, als auch die einzeln definierten Beschreibungen an. Hierbei setzt Doxygen auf eigens definierte Befehle. Es gibt Befehle der Form *\Befehl* welche die Art des Kommentars beschreiben. Aber auch die erzeugte Dokumentation von Doxygen ähnelt den erwähnten Werkzeugen sehr.

Eine Alternative, die für viele Programmiersprachen verwendet werden kann, ist Natural Docs. Natural Docs unterstützt 21 Programmiersprachen und ist folglich an keine Programmierumgebung gebunden. Dabei bindet sich Natural Docs wie Doxygen an keinen Stadard, sondern definiert seine eigenen Befehle zum Kommentieren.

Als ein bekanntes Beispiel einer automatisch generierten Dokumentation, bietet Oracle eine

Dokumentation von Java[5]. Dabei sind die einzelne Methoden, Klassen und Schnittstellen von den verschiedenen Java-Versionen dokumentiert.

6.3 Nutzen und Risiken

Durch Werkzeuge zur Dokumentation werden Schritte des Prozesses automatisiert. Somit wird dieser weniger Fehleranfällig und einfacher anzuwenden. Durch die nun einfache Dokumentation sind Entwickler eher geneigt, ihren Code tatsächlich zu dokumentieren. Somit wird beispielsweise bei der Verwendung von dokumentierten Methoden klar, was die jeweilige Methode im Detail macht und worauf ggf. geachtet werden muss. Dies ist besonders bei großen Projekten wichtig, in welchen aufgrund getrennter Entwicklung regelmäßig fremde Methoden verwendet werden. So wird direkt bei den Vervollständigungsvorschlägen (siehe Abschnitt 2) die Inline-Dokumentation angezeigt. Außerdem müssen Diagramme nicht weiterhin von Hand erzeugt werden, sondern sind einfach exportierbar, sowie importierbar.

Trotz aller Vorteile muss jedoch immer ein Mittelmaß gefunden werden. Gute Dokumentation ersetzt nicht *Clean Code*, also beispielsweise die Verwendung von sinnvollen Namen. Außerdem muss darauf geachtet werden, dass Dokumentation auch bei Änderungen im Code aktuell gehalten wird. Ansonsten unterscheidet sich die Beschreibung von der tatsächlichen Funktionalität. Nicht zuletzt sollten Kommentare kurz und verständlich anstelle von lang und unübersichtlich sein. Ansonsten wird unnötig viel Zeit darauf verwendet und es fehlt eine einfache, schnelle Verständlichkeit. Eine übertriebene Nutzung der Kommentarfunktion kann den Quellcode sogar weniger leserlich machen.

Bezüglich von Diagrammerzeugung muss außerdem erwähnt werden, dass gewisse Besonderheiten ggf. nicht automatisch in ein Diagramm übersetzt und somit manuell nachgebessert werden müssen. Erwähnenswert sind außerdem die möglicherweise anfallenden Kosten. Dies betrifft auch Werkzeuge zur Diagrammerzeugung aus Quelltexten wie in der Marktanalyse ersichtlich ist (Unterabschnitt 6.2).

7 Kollaboration

Ein weiteres wichtiges intelligentes Werkzeug, welches sich im Bereich der Dokumentation und Kollaboration befindet, ist die Versionsverwaltung. Diese ermöglicht während des gesamten Projektes die Zusammenarbeit mehrerer Personen, Versionierung, als auch Dokumentation der Änderungen. Große Projekte sind undenkbar ohne Versionsverwaltung.

[5] Vgl.: https://docs.oracle.com/javase/7/docs/api/overview-summary.html

7.1 Allgemeines Konzept

Die Versionsverwaltung ist gerade bei Kollaborationen mehrerer Entwickler essenziell. Sie wird zum Erfassen von Änderungen an Dateien und Dokumenten verwendet. Eine Versionsverwaltung bietet einige Vorteile. Darunter zählen Transparenz, Übersicht, Nachverfolgbarkeit, Zusammenarbeit mehrerer Entwickler und Identifikation. All dies führt zu einer Effizienzsteigerung. Dabei muss es sich bei zu verwaltenden Daten nicht, wie weitläufig bekannt, nur um Quelltexte handeln. Eine Versionsverwaltung kann auch für Text- oder Tabellendokumente verwendet werden. Im Folgenden wird zwar überwiegend auf die Verwaltung von Quelltexten eingegangen, dennoch soll erwähnt werden, dass eine Versionsverwaltung in den verschiedensten Projektbereichen eingesetzt werden kann. Dies gibt den Entwicklern die Möglichkeit, Änderungen nachzuverfolgen und gegebenenfalls rückgängig zu machen. Des Weiteren lassen sich vielseitige Statistiken über die getätigten Änderungen erstellen. Weitere Hauptaufgaben einer Versionsverwaltung sind Protokollierung, Wiederherstellung, Archivierung, Koordinierung und das gleichzeitige entwickeln mehrere Zweige. Für die genannten Aufgaben verfügt jede Versionsverwaltung über fünf Basisaktionen, welche diese Aufgaben ermöglichen.

(a) **Add**: Fügt Dateien zur Versionsverwaltung hinzu.

(b) **Remove**: Entfernt Dateien aus der Versionsverwaltung.

(c) **Commit**: Veröffentlicht vorgenommene Änderungen.

(d) **Revert**: Setzt die aktuelle Version auf den letzten veröffentlichten Stand zurück.

(e) **Branch**: Erzeugt einen neuen Zweig aus einem bestehend Zweig.

(f) **Merge**: Fügt zwei Zweige zusammen und passt Differenzen an.

Darüber hinaus gibt es noch viele weitere Möglichkeiten, die ein Versionsverwaltungssystem besitzen kann. Die genannten Möglichkeiten sind die Grundlagen, die Versionsverwaltungssysteme beherrschen [DA20]. Die Möglichkeiten die dem Entwickler dadurch gegeben werden, sind im Entwicklungsprozess sehr wichtig und werden in den meisten Projekten eingesetzt. Auf Quelltexte bezogen, bietet die zentrale Lagerung der Quelltext-Dateien noch weitere Vorteile.

Wie in einem Verwaltungssystem üblich, können verschiedene Rollen vergeben werden. Diese ermöglichen die Vergabe von Freigaben und somit eine Rechteverwaltung. Dadurch ist die Möglichkeit gegeben, Änderungen erst nach einer Absprache und/oder Korrektur freizugeben. Eine weitere Hauptaufgabe, welche Versionsverwaltungssysteme ermöglichen ist die Integration spezieller Abläufe bei der Aktualisierung der Version. So können festgelegte Buildvorgänge, Sicherungen oder verschiedenen Tests der Anwendung durchgeführt werden.

7.2 Martkanalyse

Auf dem Markt gibt es unzählige Versionsverwaltungssysteme mit verschiedenen Schwerpunkten für verschiedensten Betriebssysteme. Darunter sind zum Beispiel *Git*, *Subversion (SVN)* und *Concurrent Version System (CVS)*. Das wohl bekannteste Versionsverwaltungssystem ist Git. Git unterscheidet sich insofern von anderen, als das es ein verteiltes System ist. Das bedeutet, dass jede Kopie des Verzeichnisses die gesamten Metadaten inklusive des Änderungsverlaufs enthält und so ein eigenständiges und abgekapseltes Verzeichnis bildet. Git ist kostenlos nutzbar und ein OpenSource-Projekt. Gleiches gilt für Subversion, welches wie git eine große Bekanntheit genießt und von der Apache Software Fundation entwickelt wird. Das Ziel von Subversion ist es, der Nachfolger des in der Vergangenheit sehr bekannten Concurrent Version System zu sein.

Mit 94 Millionen Nutzern ist GitHub[6] ein sehr bekanntes, auf Git basierendes, Versionsverwaltungssystem im Internet. An diesem Beispiel ist sehr gut zu erkennen, welche weiteren Möglichkeiten eine Versionsverwaltung bietet. Neben Integration der oben genannten Standard-Werkzeugen einer Versionsverwaltung, bietet Github noch viele Erweiterungen rund um die Versionsverwaltung. So können Teams mit Dashboards und verschiedenen Statistiken zum Projekt arbeiten. Auch können verschiedenste Automationen eingepflegt werden, welche den Entwicklungsprozess erleichtern. Bei größerem Interesse kann bei einer Recherche das Stichwort CI/CD oder DevOps unterstützen.

7.3 Nutzen und Risiken

Durch die dadurch VCS entstehenden Möglichkeiten ist eine große Effektivitätssteigerung in Projekten möglich. Sie vereinfachen die Zusammenarbeit in großen Teams und Kollaborationen. Die Werkzeuge sind in den unterschiedlichsten Ausprägungen in vielen Firmen vertreten und werden von vielen Entwicklern schon als Standard angesehen. Zudem sorgen die Möglichkeiten zur Nutzung verschiedener Branches, das Mergen und die Dokumentation der Änderungen generell für eine höhere Sicherheit des Codes. Fehler können schnell identifiziert werden, eine lauffähige Version ist immer vorhanden und es kann unabhängig von anderen entwickelt werden.

Ein Risiko, welches die Werkzeuge dennoch bieten ist, dass sie über ihre Maße angewandt werden. Auch bei der Versionsverwaltung ist dies ein Thema. Viele Anbieter bieten über die Standardmöglichkeiten einer Versionsverwaltung hinaus Werkzeuge an. Diese sind jedoch nicht immer erforderlich. Ein weiterer Punkt ist, dass die meisten Werkzeuge für den kommerziellen Einsatz in Unternehmen kostenpflichtig sind. Die genannten Werkzeuge werden meist auf Unternehmensinterne Daten angewandt. Deshalb ist auch der Datenschutz ein sehr wichtiger Punkt, welcher nicht vernachlässigt werden sollte. Hierfür bieten die

[6] https://github.com/

genannten Anbieter zwar Lösungen, wie zum Beispiel die Werkzeuge Unternehmensintern zu hosten, dennoch sollte der Datenschutz nicht vernachlässigt werden.

8 Schluss

Abschließend soll in einer kurzen Zusammenfassung die Arbeit zusammengefasst und die Erkenntnisse zentral gesammelt werden. Danach folgt mit den durch die Arbeit erlangten Erkenntnisse ein Ausblick in die Zukunft von intelligenten Werkzeugen zur Softwareentwicklung.

In der vorliegenden Arbeit wurden wichtige intelligente Werkzeuge zur Softwareentwicklung aufgezeigt. Dazu wurden die theoretischen Grundlagen der einzelnen Werkzeuge genauer aufgezeigt und die in einer Marktanalyse ausgewählte Werkzeuge, welche in der Praxis Anwendung finden, näher analysiert. Abschließend wurden der Nutzen und die möglichen Risiken der einzelnen Werkzeuge evaluiert. Die ausgewählten Werkzeuge, welche vorgestellt wurden, erleichtern den Entwicklungsprozess in essenzieller Weise und gestalten die Arbeit effizienter. Genauer wurden intelligente Werkzeuge zur Codevervollständigung, Codegenerierung, Analyse, Refactoring, Dokumentation und Kollaboration vorgestellt. Bei der Marktanalyse ist aufgefallen, dass es eine Vielzahl von Anbietern intelligenter Werkzeuge gibt. Zwar gibt es teilweise sprachübergreifende Werkzeuge, dies ist allerdings nicht immer der Fall. Manchmal kann daher eine gesonderte Analyse bezüglich verwendetem Technologie-Stack sinnvoller sein. Des Weiteren ist ersichtlich, dass bestimmte Entwicklungsumgebungen die genannten Werkzeuge bereits implementieren. Daher muss in diesen Fällen nicht auf eine externe Lösung gesetzt werden. Dies spiegelt aber auch die Akzeptanz und Wichtigkeit dieser Werkzeuge bei der Entwicklung wieder. Die Analysen zeigen, dass jedes der vorgestellten intelligenten Werkzeuge den Entwicklungsprozess effizienter gestaltet. Die genannten Gefahren dabei sind meist, dass sich der Entwickler zu sehr auf die Werkzeuge verlässt und diese nicht hinterfragt. Daher sind die Werkzeuge aufgrund ihrer Effizienzsteigerung zu empfehlen, jedoch sollten Entwickler ihnen nicht blind vertrauen und dennoch die Ausgabe selbst kontrollieren.

Wie aus der Arbeit hervorgeht, sind viele intelligente Werkzeuge jetzt schon in der Softwareentwicklung etabliert. Auch an den Nutzerzahlen der einzelnen Werkzeuge, die in den Marktanalysen genannt wurden, lässt sich dies aufzeigen. Dennoch gibt es auch andere Meinungen. Gerade wie sich in der Marktanalyse zum Thema Refactoring (Unterabschnitt 5.4) herausgestellt hat, ist das Vertrauen zu gewissen Werkzeugen noch nicht vollständig hergestellt und bedarf noch gewisser Erfahrung. Dennoch betrifft dies nicht alle Werkzeuge. Das Themengebiet der vollständig auf künstliche Intelligenz basierten Werkzeuge bietet in Zukunft noch viel Entwicklungspotenzial. Wie zum Beispiel das Werkzeug der Codegenerierung Abschnitt 2 zeigt, bietet es immense Möglichkeiten den Entwicklungsprozess zu erleichtern oder gar zu automatisieren. Von einer vollständigen Automatisierung ist hier aber in absehbarer Zeit nicht zu reden. Dennoch entwickelt sich das allgemeine Feld der künstlichen Intelligenz rasant und die Prognosen sind hierzu nur bestätigend [St22]. Deshalb

ist davon auszugehen, dass auch die intelligente Werkzeuge immer mehr von dieser Technik profitieren und dadurch weiterentwickelt werden. Wie die Arbeit des Weiteren aufzeigt ist das Themengebiet hoch aktuell und sehr wichtig. So werden auch in Zukunft noch einige interessante Neuerungen und Weiterentwicklungen in diesem Bereich stattfinden.

Literatur

[Am04] Ambler, S. W.: The object primer: Agile Model-driven development with UML 2.0 / Scott W. Ambler. Cambridge University Press, Cambridge, 2004, ISBN: 0521540186.

[BMM09] Bruch, M.; Monperrus, M.; Mezini, M.: Learning from Examples to Improve Code Completion Systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. Association for Computing Machinery, New York, NY, USA, S. 213–222, 2009, ISBN: 9781605580012.

[Ch13] Chemuturi, M.: Requirements Engineering and Management for Software Development Projects. Springer Verlag, 2013, ISBN: 978-1-4614-5376-5.

[Ch21] Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillet, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; Zaremba, W.: Evaluating Large Language Models Trained on Code. CoRR abs/2107.03374/, 2021, arXiv: 2107.03374, URL: https://arxiv.org/abs/2107.03374.

[DA20] Davis; Anglin, Hrsg.: Modern Programming Made Easy. Apress, [Place of publication not identified], 2020, ISBN: 978-1-4842-5568-1.

[Da88] Darwin, I. F.: Checking C Programs with Lint -. Ö'Reilly Media, Inc.", Sebastopol, 1988, ISBN: 978-0-937-17530-9.

[De22] DevExpress: CodeRush for Visual Studio, 2022, URL: https://www.devexpress.com/Products/CodeRush/, Stand: 05. 11. 2022.

[Do08] Dollard, K.: Code Generation in Microsoft .NET. Apress, New York, 2008, ISBN: 978-1-430-20705-4.

[Do22] Doxygen: Doxygen: Generate documentation from source code, 2022, URL: %5Curl%7Bhttps://www.doxygen.nl/index.html%7D, Stand: 15. 11. 2022.

[EM21] Eilertsen, A. M.; Murphy, G. C.: The Usability (or Not) of Refactoring Tools. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). S. 237–248, 2021.

[Fo00] Fowler Martin mit Beck, K.; Brant, J.; Opdyke, W.; Roberts, D.: Refactoring: Wie Sie das Design vorhandener Software verbessern. Addison-Wesley, 2000, ISBN: 3827316308.

[Fo08] Foundation, O.: Code Review Guide Book v. 2.0. 2008, ISBN: 978-1-304-58673-5.

[Gi17] GitHub, Inc.: Open Source Survey, 2017, URL: %5Curl%7Bhttps://opensourcesurvey.org/2017/%7D, Stand: 02. 11. 2022.

[GKF06] How are java software developers using the eclipse ide?, IEEE, 2006, S. 76–83.

[gm12] gmatht: Static Analysis vs Style Checkers, 2012, URL: https://lwn.net/Articles/483972/.

[Jea] Jetbrains: All Products, URL: https://www.jetbrains.com/all/.

[Jeb] Jetbrains: Static Code Analysis, URL: https://www.jetbrains.com/de-de/teamcity/ci-cd-guide/concepts/static-code-analysis/.

[Je20] Jetbrains: Why Does Jetbrains Separate Their Products Into Multiple IDEs, 2020, URL: https://intellij-support.jetbrains.com/hc/en-us/community/posts/360006942459-why-does-Jetbrains-separate-their-products-into-multiple-IDEs.

[Je22] JetBrains s.r.o.: UML class diagrams, 2022, URL: %5Curl%7Bhttps://www.jetbrains.com/help/idea/class-diagram.html%7D, Stand: 08. 11. 2022.

[Ji16] Jill Reinauer: Using the Task List, 2016, URL: %5Curl%7Bhttps://learn.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2015/ide/using-the-task-list?view=vs-2015&redirectedfrom=MSDN#tokenscomments%7D, Stand: 03. 11. 2022.

[Ki22] with Kite, C. F.: Kite - Free AI Coding Assistant and Code Auto-Complete Plugin, 2022, URL: %5Curl%7Bhttps://www.kite.com/%7D, Stand: 12. 11. 2022.

[Ma02] Martin, R. C.: Agile software development, principles, patterns, and practices. Pearson, Upper Saddle River, NJ, 2002.

[Ma09] Martin Robert C. mit Feathers, M. C.
bibinitperiod E. R.: Clean Code. Mitp-Verlag, 2009, ISBN: 9783826696381.

[MCB15] Marasoiu, M.; Church, L.; Blackwell, A.: An empirical investigation of code completion usage by professional software developers. In: Proceedings of PPIG 2015. S. 71–82, 2015.

[No22] Nolle, T.: Statische und dynamische Code Analyse zusammen einsetzen, 2022, URL: https://www.computerweekly.com/de/tipp/Statische-und-dynamische-Code-Analyse-zusammen-einsetzen.

[Op21] OpenAI: OpenAI Codex, 2021, URL: https://openai.com/blog/openai-codex/.

[Op22] OpenAI: OpenAI Codex Sandbox, 2022, URL: https://beta.openai.com/codex-javascript-sandbox.

[Or97] Oracle: Code Conventions, 1997, URL: https://www.oracle.com/technetwork/java/codeconventions-150003.pdf.

[Pa19] Papenbrock, T.: Data Profiling – Effiziente Entdeckung Struktureller Abhängigkeiten. In (Grust, T.; Naumann, F.; Böhm, A.; Lehner, W.; Härder, T.; Rahm, E.; Heuer, A.; Klettke, M.; Meyer, H., Hrsg.): BTW 2019. Gesellschaft für Informatik, Bonn, S. 467–476, 2019.

[Re22a] ReShaper, J.: Quick-Fixes, 2022, URL: https://www.jetbrains.com/de-de/resharper/features/quick_fixes.html, Stand: 15. 11. 2022.

[Re22b] ReShaper, J.: Refactorings, 2022, URL: https://www.jetbrains.com/help/resharper/Refactorings__Index.html, Stand: 05. 11. 2022.

[Re22c] ReShaper, J.: ReSharper features in different languages, 2022, URL: https://www.jetbrains.com/help/resharper/Introduction__Feature_Map.html, Stand: 02. 11. 2022.

[RL08] How Program History Can Improve Code Completion, IEEE, 2008, S. 317–326.

[Ro00] Rosenfeld, R.: Two decades of statistical language modeling: where do we go from here? Proceedings of the IEEE 88/8, S. 1270–1278, 2000.

[Sc19a] Schmidt, M.: Statische und dynamische Codeanalyse in einem kontinuierlichen Testprozess, 2019, URL: https://www.embedded-software-engineering.de/statische-und-dynamische-codeanalyse-in-einem-kontinuierlichen-testprozess-a-846419/.

[Sc19b] van Scharrenburg, E.: Code Completion with Recurrent Neural Networks. In. 2019.

[St07] Stahl, T.; Völter, M.; Efftinge, S.; Haase, A.: Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management. dpunkt, Heidelberg, 2007, ISBN: 978-3-89864-448-8.

[St22] Statista: Artificial Intelligence (AI) market size/revenue comparisons 2018-2030, Juni 2022, URL: https://www.statista.com/statistics/941835/artificial-intelligence-market-size-revenue-comparisons/.

[Te22] Terry G. Lee: Design and view classes and types with Class Designer, 2022, URL: %5Curl%7Bhttps://learn.microsoft.com/en-us/visualstudio/ide/class-designer/designing-and-viewing-classes-and-types?view=vs-2022%7D, Stand: 08. 11. 2022.

[Th21] TheCodeBytes, 2021, URL: https://thecodebytes.com/wp-content/uploads/2021/12/low-level-programming-languages.jpg.