TECHNISCHE HOCHSCHULE LÜBECK

Bachelor-Thesis

Entwicklung eines maßgeschneiderten Tools zur syntaktischen und semantischen Analyse und zum automatisiertem Refactoring von Quellcode

vorgelegt von: Matthias Steffen

Fachbereich: Elektrotechnik und Informatik
Studiengang: Informatik/Softwaretechnik
Erstprüfer: Prof. Dr.-Ing. Uwe Krohn

Ausgabedatum: 12. November 2018
Abgabedatum: 10. Januar 2019

(Professor Dr. Andreas Hanemann)
Vorsitzender des Prüfungsausschusses

Aufgabenstellung

Der Umfang und die Komplexität moderner Softwareprojekte nehmen immer weiter zu. Deshalb ist es heutzutage nötig, dass die Softwareentwickler Unterstützung von automatisierten Werkzeugen bekommen.
Die bilettix GmbH entwickelt seit über 10 Jahren eine webbasierte Ticketing und Enterprise-Resource-Planning (ERP) Software speziell für den Kulturbereich. Im Verlauf der Jahre ist die Codebase so auf mehr als 250.000 Zeilen Code gewachsen, an denen viele Entwickler mitgewirkt haben. Durch den Einfluss der Zeit und der vielen unterschiedlichen Entwickler haben sich die verwendeten Design Patterns stetig weiterentwickelt. Dabei haben sich einige typische Probleme herausgestellt, welche nach der Entdeckung in einem Teil der Codebase behoben und anschließend vermieden wurden. Im Rest der Codebase sind diese Probleme jedoch weiterhin vorhanden. Aufgrund der Größe der Codebase und der begrenzten Anzahl an Entwicklern ist es zu aufwendig, diese Probleme manuell zu suchen und zu beheben. Deshalb wird ein Tool benötigt, welches den Code automatisch analysiert, die Probleme findet und diese automatisch behebt.
Im Rahmen der Bachelorarbeit sollen zuerst Anti-Pattern und Fehler im Quellcode gesucht und klassifiziert werden. Im Anschluss soll ein Prototyp zur automatischen Erkennung und Verbesserung dieser Anti-Pattern implementiert und evaluiert werden. Dabei ist es erforderlich, dass das Tool sowohl eine syntaktische als auch eine semantische Analyse des Quellcodes durchführen kann. Da das Tool automatisch arbeiten soll, ist es wünschenswert, dass eine ebenfalls automatische Prüfung der Korrektheit des Tools implementiert wird. Bei der Implementierung des Tools sollte darauf geachtet werden, dass softwaretechnische Methoden genutzt werden, um die Wartbarkeit und die Erweiterbarkeit des Tools zu ermöglichen. Außerdem muss das Tool für die Entwickler einfach zu bedienen sein, damit diese es in ihren Alltag integrieren können. Da alle Entwickler der bilettix GmbH Microsoft Visual Studio als Entwicklungsumgebung nutzen, soll das Tool als Erweiterung für Visual Studio verfügbar sein. Des Weiteren wäre eine Integration in den automatischen Buildprozess der Software wünschenswert.

TECHNISCHE
HOCHSCHULE
LÜBECK

Fachbereich
Elektrotechnik und Informatik
Dekanat - Prüfungsausschuss

Erklärung zur Bachelorarbeit

Ich versichere, dass ich die Arbeit selbständig, ohne fremde Hilfe, verfasst habe.

Bei der Abfassung der Arbeit sind nur die angegebenen Quellen benutzt worden. Wörtlich oder dem Sinne nach entnommene Stellen sind als solche gekennzeichnet.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, insbesondere dass die Arbeit Dritten zur Einsichtnahme vorgelegt oder Kopien der Arbeit zur Weitergabe an Dritte angefertigt werden.

Lübeck, den 10. Januar 2019

. . . . . .

(Unterschrift)

Zusammenfassung der Arbeit / Abstract of Thesis

Fachbereich: **Department:**	Elektrotechnik und Informatik
Studiengang: **University course:**	Informatik Softwaretechnik
Thema: **Subject:**	Entwicklung eines maßgeschneiderten Tools zur syntaktischen und semantischen Analyse und zum automatisiertem Refactoring von Quellcode Implementation of a custom tool for syntactic and semantic analysis and automated refactoring of source code.
Zusammenfassung:	Die vorliegende Arbeit untersucht die Implementierung eines automatisierten Tools zur Analyse und zum Refactoring von Quellcode. Es werden sowohl allgemeine Grundlagen des Compilerbaus betrachtet, als auch spezifische Grundlagen der verwendeten Technologien. Des Weiteren stehen die Details der Implementierung und der Entwicklungsprozess im Vordergrund.
Abstract:	The following work examines the implementation of a automated tool for analysing and refactoring source code. It will look into both the fundamentals of compiler design and the specific fundamentals of the technologies in use. Furthermore the work will focus on the details of implementation and the development process.
Verfasser: **Author:**	Matthias Steffen
Betreuender Professor: **Attending professor:**	Prof. Dr.-Ing. Uwe Krohn
WS / SS:	WS 2018/19

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Software ist heutzutage allgegenwärtig. Dies führt dazu, dass Softwareprojekte zunehmend umfangreicher werden. Auf Grund ständig neuer Anforderungen ist Softwareentwicklung ein kontinuierlicher Prozess. Eine Folge dieser kontinuierlichen Anpassung an neue Anforderungen ist, dass die entwickelte Software zunehmend komplexer wird. Dies stellt alle Softwareentwickler vor die Herausforderung, diese Komplexität zu beherrschen. Dabei ist die Unterstützung automatisierter Werkzeuge nicht mehr wegzudenken. Dazu enthält jede moderne Entwicklungsumgebung eine Vielzahl von Werkzeugen, angefangen bei Selbstverständlichkeiten wie der Syntaxhervorhebung.
Bisher war die Einstiegshürde bei der Entwicklung solcher Werkzeuge sehr hoch, da meistens jedes Werkzeug eine eigene Möglichkeit schaffen musste, um den Quellcode zu verarbeiten. Bereits existierende Compiler konnten für diese Aufgabe häufig nicht wiederverwendet werden, da diese als 'Black Box' funktionieren und keine Programmierschnittstellen bereitstellen. Dies machte es für kleine und mittelständische Unternehmen nahezu unmöglich, maßgeschneiderte Werkzeuge zur Unterstützung der Softwareentwickler zu erstellen.
Um dieses Problem für die .NET Softwareplattformen zu lösen, hat Microsoft im Jahr 2014 [1] das Projekt 'Roslyn' unter einer freien Lizenz veröffentlicht. Im Rahmen dieses Projekts wurden die Compiler für die Programmiersprachen Visual Basic und C# komplett neu implementiert. Der Fokus lag dabei auf der Erweiterbarkeit des Kompilierprozesses. Teil des Roslyn Projekts ist eine Sammlung von Programmierwerkzeugen (engl. *'Software Development Kit (SDK)'*) zum Erstellen von automatisierten Analysewerkzeugen und Werkzeugen zur automatisierten Transformation von Quellcode.
Diese Möglichkeiten sollen in Zusammenarbeit mit der bilettix GmbH dazu genutzt werden, domänenspezifische Probleme automatisiert zu erkennen und zu beheben.

[1] [Pil18] Erstes Commit des Roslyn Projekts auf GitHub

1.2 Zielsetzung

Ziel dieser Arbeit ist es, einen Prototyp für ein Werkzeug zur automatischen Erkennung und Behebung domänenspezifischer Probleme zu erstellen. Um den Prototyp zu implementieren, sollen die Programmierschnittstellen des Roslyn Projekts genutzt werden. Da es sich um einen Prototyp mit beschränktem Funktionsumfang handelt, ist es besonders wichtig, dass bei der Softwarearchitektur auf die Erweiterbarkeit geachtet wird. Ein weiterer wichtiger Aspekt für den Erfolg des Prototyp ist es, die Bedienung möglichst benutzerfreundlich zu gestalten, damit die Softwareentwickler der bilettix dieses Werkzeug reibungslos in ihren Alltag integrieren können.

1.3 bilettix GmbH

Die bilettix GmbH ist ein mittelständisches Softwareunternehmen, welches sich auf Softwarelösungen für den Kulturbereich spezialisiert hat. Das Produkt bilettix.net ermöglicht sowohl den Verkauf von Tickets, als auch die Verwaltung von Geschäftsressourcen (engl. *Enterprise-Resource-Planning (ERP)*). Um den Ticketkauf für den Endkunden so einfach wie möglich zu gestalten, bietet bilettix.net mehrere Möglichkeiten für den Vertrieb von Tickets. So können die Kulturbetriebe die Tickets selbst vertreiben, z.B. über den Verkauf an der Abendkasse. Zusätzlich dazu bietet bilettix.net die Möglichkeit, Tickets über ein Vertriebsnetz von Vorverkaufsstellen oder über einen Webshop zu verkaufen.
Eine weitere sehr wichtige Funktion von bilettix.net ist das integrierte Kundenbeziehungsmanagement (engl. *Customer-Relationship-Management (CRM)*). Dieses bietet unter anderem umfangreiche Möglichkeiten zur Kundenbindung. Zum Leistungsumfang von bilettix.net gehören außerdem eine integrierte Finanzbuchhaltung und eine große Auswahl an automatisch generierten kaufmännischen Berichten.
Die Software bilettix.net ist komplett webbasiert und wurde mit Hilfe des .NET-Frameworks implementiert. Diese Arbeit wird von nur einem Entwicklerteam geleistet. So haben sich im Verlauf der letzten zehn Jahre ca. 205.000 Zeilen Quellcode angesammelt. Mit solch einem großen Softwareprojekt geht ein großer Aufwand für die Wartung und Fehlerbehebung einher. Der Prototyp, welcher Ziel dieser Arbeit ist, soll dabei helfen, diesen Aufwand für die Entwickler der bilettix zu minimieren.

1.4 Aufbau dieser Arbeit

In Kapitel 2 werden die relevanten Grundlagen über die Funktionsweise und den Aufbau von Compilern behandelt. Anschließend wird in Kapitel 3 ein Überblick über die Roslyn Compiler Plattform gegeben. Dazu sind die Informationen aus Kapitel 2 unerlässlich, da sich die Struktur der Roslyn Plattform an den vorgestellten Grundlagen orientiert. Der Fokus dieses

Kapitels liegt auf den Möglichkeiten der Roslyn Plattform zur Analyse von Quellcode. Im 4. Kapitel werden die domänenspezifischen Probleme erläutert, welche der Prototyp erkennen und beheben soll. Zusätzlich werden mögliche Lösungsansätze diskutiert. Aus diesen Problemen und den zugehörigen Lösungsansätzen leiten sich die Anforderungen an den Prototyp ab. Kapitel 5 beschäftigt sich mit dem Entwicklungsprozess, den Rahmenbedingungen der Entwicklung und der Implementierung des Prototyp.
In Kapitel 6 werden die Eigenschaften der Implementierung mit den Anforderungen aus Kapitel 4 abgeglichen. Kapitel 7 liefert eine kurze Zusammenfassung dieser Arbeit und bietet einen Ausblick über die das zukünftige Potential des Implementierten Tools.
Auf der beiliegenden CD sind der Quellcode der Implementierung und eine elektronische Kopie dieser Arbeit zu finden.

2 Grundlagen

2.1 Struktur eines Compilers

Der folgende Abschnitt soll einen Überblick über die grundlegende Struktur eines Compilers geben. Dazu werden Konzepte, statt konkreter Techniken zur Implementierung betrachtet. Vertiefende Informationen und konkrete Techniken sind in Alfred V. Aho. *Compilers: Principles, techniques, & tools*[Aho07] zu finden.
Der Kompilierprozess wird in die *Analysephase* und die *Synthesephase* aufgeteilt. Während der Analysephase wird sichergestellt, dass das Programm sowohl syntaktisch, als auch semantisch mit der Spezifikation der Programmiersprache übereinstimmt. Im Rahmen der Synthesephase wird, mit Hilfe geeigneter Zwischenrepräsentationen, der Zielcode erzeugt. Diese beiden Phasen können jeweils noch weiter unterteilt werden, so dass jede Teilphase für eine spezielle Aufgabe verantwortlich ist. Die Abfolge dieser Teilphasen bildet eine Pipeline, bei der die Ausgabe einer Phase, die Eingabe der direkt folgenden Phase ist. Eine typische Compiler-Pipeline ist in Abbildung 2.1 zu sehen. Der Aufbau des folgenden Abschnitts orientiert sich am Aufbau dieser Pipeline und dabei wird jede Phase kurz erläutert. Eine so detaillierte Aufteilung ist für die Erläuterung der verschiedenen Probleme im Kompilierprozess sinnvoll. Bei der tatsächlichen Implementierung von Compiler kommt es jedoch häufig vor, dass mehrere Phasen zusammengefasst oder deren Aufgaben aufgeteilt werden. Trotzdem erfolgt auch in der Praxis eine Trennung zwischen dem *Frontend* und dem *Backend* des Compilers. Das Frontend umfasst dabei alle Analyseschritte und das Erstellen des Zwischencodes. [1] Die Aufgabe des Backends ist die Generierung des Zielcodes aus dem Zwischencode. Die lose Koppelung der beiden Komponenten, über den Zwischencode, ermöglicht es verschiedenen Implementierungen wiederzuverwenden und zu kombinieren. Als Beispiel aus der Praxis erwähnt Aho, dass der erste C++ Compiler nur ein vorangestelltes Frontend für den C Compiler war.[2] Dazu generierte der C++ Compiler aus dem C++ Code zunächst C Code, welcher anschließend einem C Compiler übergeben wurde, der daraus Maschinencode erstellte.

[1] [Aho07, Seite 41] 'Chapter 2. A Simple Syntax-Directed Translator'
[2] [Aho07, Seite 358] 'Chapter 6. Intermediate-Code Generation'

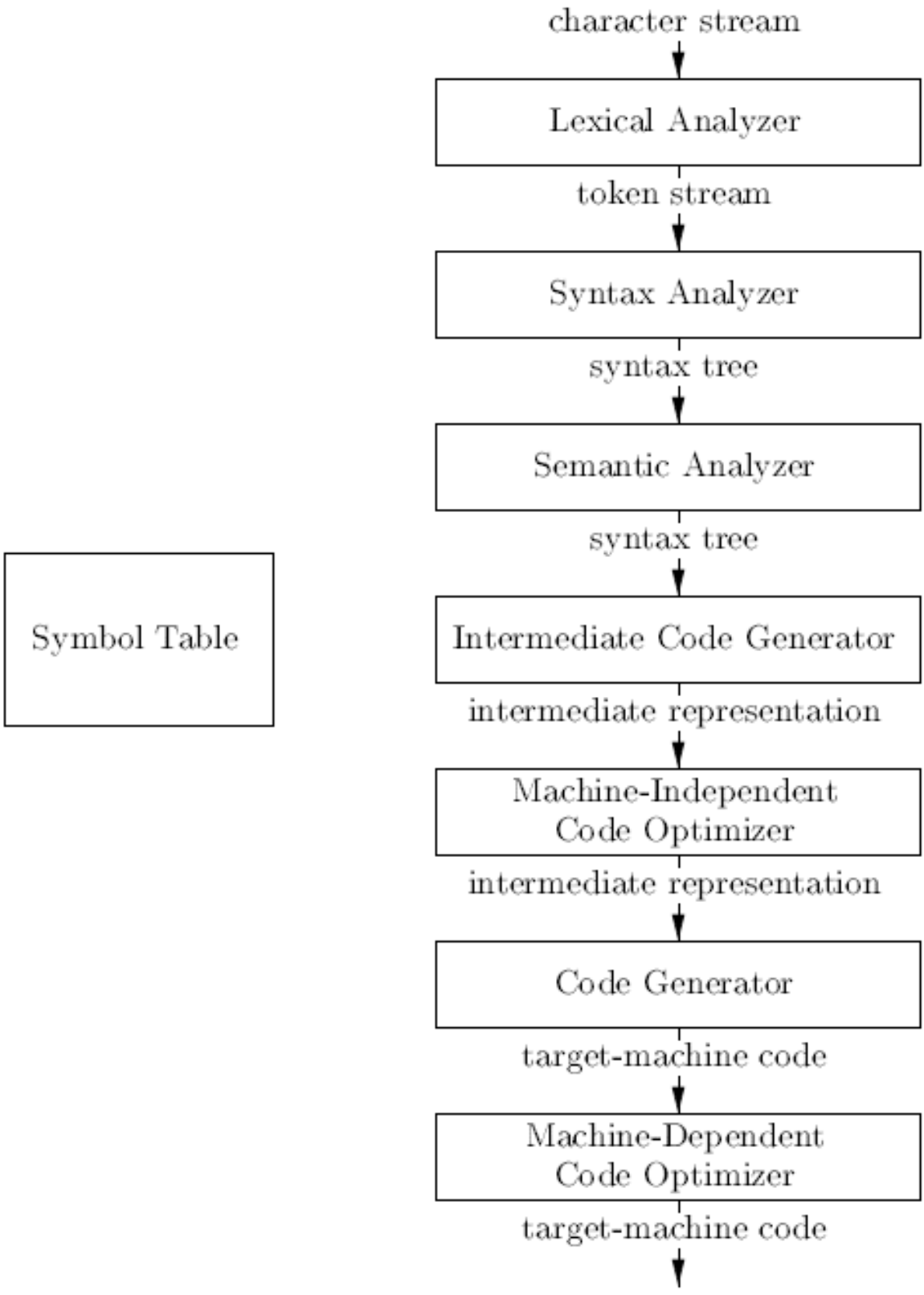

Abbildung 2.1: Typische Compiler-Pipeline mit ihren Teilphasen aus Alfred V. Aho. *Compilers: Principles, techniques, & tools*. 2nd ed. Boston: Pearson/Addison Wesley, 2007. ISBN: 0321486811

Wie in Abbildung 2.1 zu erkennen ist, bauen die unterschiedlichen Phasen direkt aufeinander auf. Der dabei entstehende Datenfluss überführt benötigte Daten von einer Phase in die darauf folgende. Zusätzlich wird eine *Symboltabelle* verwendet. Die Symboltabelle ist eine Datenstruktur, welche Informationen festhält, die im gesamten Verlauf des Kompilierprozesses von Bedeutung sind. So können z.B. Informationen, welche bei der syntaktischen Analyse gewonnen werden, später im Rahmen der semantischen Analyse genutzt werden. Da es moderne Programmiersprachen ermöglichen, Deklarationen in unterschiedlichen Gültigkeitsbereichen vorzunehmen, werden häufig mehrere Symboltabellen während des Kompilierprozesses erstellt.

2.1.1 Analysephase

Die Aufgabe der Analysephase ist es sicherzustellen, dass das zu übersetzende Programm mit der syntaktischen und semantischen Spezifikation der Programmiersprache übereinstimmt. Dazu müssen die einzelnen Zeichen der Quelldatei zu sinnvollen Einheiten zusammengefasst werden. Anschließend müssen die Syntax und die Semantik überprüft werden.

Lexikalische Analyse

Im Rahmen der lexikalischen Analyse werden die einzelnen Zeichen der Quelldatei zu sinnvollen Einheiten, den sogenannten *Lexemen*, zusammengefasst. Lexeme werden durch bestimmte Muster von Zeichen gebildet und sind der kleinste lexikalische Bestandteil, welcher im Rahmen der Analysephase genutzt wird.[3] Für die Schlüsselwörter einer Programmiersprache bilden die konkreten Wörter das Muster, welches ein Lexem definiert. Für Bezeichner sind komplexere Muster, wie zum Beispiel eine beliebige Abfolge alphanumerischer Zeichen möglich. Ein *Token* besteht aus zwei Teilen, dem *Namen* des Tokens und dessen *Wert*. Der Name des Tokens ist ein abstraktes Symbol, welches zur Bestimmung der Art des Tokens dient. Jedes Token muss einen Namen haben. Der Wert eines Tokens kann hingegen leer sein. Während der lexikalischen Analyse wird für jedes Lexem ein Token erstellt.[4] Zusätzlich dazu wird für jedes Token, das zu einem Bezeichner gehört, ein Eintrag in der Symboltabelle angelegt. Diesem werden im Verlauf der Analyse weitere Informationen hinzugefügt. Zusätzlich können Tokens noch *Attribute* haben, welche spezifische Informationen zu den Lexemen des Tokens enthalten. So kann ein Token vom Typ *NumberToken* sowohl zu einem Lexem mit dem Wert '23' als auch zu einem Lexem mit dem Wert '42' gehören. Der tatsächliche Wert des Lexems kann über ein Attribut des NumberTokens identifiziert werden.[5]
Die Komponente, welche die lexikalische Analyse durchführt, wird häufig als *Lexer* bezeichnet. In der Praxis werden Lexer häufig mit Hilfe geeigneter Tools, wie z.B. *lex* oder *flex*,

[3] [Aho07, Seite 5] 'Chapter 1. Introduction'
[4] [Aho07, Seite 6] 'Chapter 1. Introduction'
[5] [Aho07, Seite 112] 'Chapter 3. Lexical analysis'

erstellt. Für so erstellte Lexer ist es üblich, dass sie Zeichen, welche für die weitere Analyse unbedeutend sind, wie z.B. Leerzeichen oder Kommentare, direkt entfernen.

Syntaktische Analyse

Da es sich bei Programmiersprachen um formale Sprache handelt, wird ihre Syntax durch sogenannte *Formale Grammatiken* beschrieben. Eine Definition für solche Grammatiken wurde bereits 1956 von dem amerikanischen Linguisten Noam Chomsky aufgestellt.[6] Ein grundlegendes Verständnis für formale Grammatiken und Sprachen wird im Folgenden vorausgesetzt. Eine kompakte Übersicht dazu kann in Alfred V. Aho. *Compilers: Principles, techniques, & tools, 'Chapter 4.2 Context-Free Grammars'*[7] gefunden werden.
Da jedes syntaktisch korrekte Programm ein Wort der zugehörigen Programmiersprache ist, muss für jedes dieser Programme eine Ableitung existieren, welche das Programm, unter Anwendung der Produktionen, aus dem Startsymbol herleitet. Diese Ableitung kann in Form eines sogenannten *Parsebaums* dargestellt werden. In diesem Baum werden die Nichtterminale der Grammatik durch die inneren Knoten und die Terminale durch die Blätter abgebildet. Bei einer Produktion der Form $A \to BC$ bildet das Nichtterminal A die Wurzel des zugehörigen Teilbaums und die beiden Nichtterminale B und C die Kinder. In dem so entstehenden Baum sind alle Informationen der Quelldatei enthalten, auch solche die keine semantische Bedeutung haben. Deshalb wird der Parsebaum auch als *konkreter Syntaxbaum* bezeichnet. Im Gegensatz zum konkreten Syntaxbaum gibt es auch noch den *abstrakten Syntaxbaum (AST)*. In diesem sind alle Tokens, welche keine semantische Bedeutung haben, nicht mehr enthalten. Abstrakte Syntaxbäume werden häufig als eine Form von Zwischencode genutzt.

Semantische Analyse

Ziel der semantischen Analyse ist es, sicherzustellen, dass die Semantik des Programms mit der Spezifikation der Programmiersprache übereinstimmt. Dazu werden sowohl der (abstrakte) Syntaxbaum aus der syntaktischen Analyse, als auch die Informationen aus der Symboltabelle benötigt. Der Syntaxbaum allein reicht für die semantische Analyse nicht aus, da die Aussagekraft der darin enthaltenen Informationen begrenzt ist. So ist es z.B. nicht immer möglich alle Referenzen korrekt aufzulösen, da Namen in unterschiedlichen Gültigkeitsbereichen mehrfach verwendet werden können. Eine der wichtigsten Aufgaben, welche bei der semantischen Analyse anfallen, ist die Typprüfung. Ziel der Typprüfung ist es, sicherzustellen, dass alle verwendeten Operatoren nur auf kompatible Operanden angewandt werden.[8] In vielen modernen Programmiersprachen ist beispielsweise der '+' Operator für zwei Strings als deren Konkatenation definiert. Der '-' Operator ist hingegen für Strings in der Regel

[6] N. Chomsky. "Three models for the description of language". In: *IEEE Transactions on Information Theory* 2.3 (1956), pp. 113–124. ISSN: 0018-9448. DOI: 10.1109/TIT.1956.1056813
[7] [Aho07, Seite 197] 'Chapter 4.2 Context-Free Grammars'
[8] [Aho07, Seite 357] 'Chapter 6 Intermediate-Code Generation'

```
1 var foo = 1 + 1;
2 var bar = 1.0 + 1.0;
```

Abbildung 2.2: Type Inference bei der Deklaration von lokalen Variablen

undefiniert. Während der Typprüfung kommt es häufig vor, dass der Typ eines Ausdrucks, wie z.B. eines Methodenaufrufs, bestimmt werden muss. Dieser Vorgang wird Typableitung (engl. 'Type Inference') genannt. Dabei wird der Typ eines Ausdrucks aus dem Kontext seiner Verwendung bestimmt. Dies ermöglicht es, auch in statischen oder strikt typisierten Programmiersprachen, bei der Deklaration einer Variablen, das Schlüsselwort *var* statt des Typs der Variablen zu verwenden. Der Compiler leitet dabei den Typ der Variablen, aus dem zugewiesenen Ausdruck ab.
Der Pseudodcode in Abbildung 2.2 zeigt die Deklaration von zwei lokalen Variablen. Auch wenn es auf den ersten Blick nicht so aussieht, haben die beiden Variablen *foo* und *bar* unterschiedliche Typen. Die Variable *foo* ist vom Typ *int*, da es sich um das Ergebnis der Addition von zwei ganzen Zahlen handelt. Die Variable *bar* ist vom Typ *float*. Der Compiler erkennt dies daran, dass der '+' Operator auf zwei Werte vom Typ *float* angewandt wird und das Resultat, per Definition des Operators, ebenfalls vom Typ *float* sein muss. Die genaue Bestimmung des Typs eines Ausdrucks ist wichtig, da unterschiedliche Typen unterschiedlich viel Speicher benötigen. Dies muss bei der Codegenerierung berücksichtigt werden, da dabei Entscheidungen über das Speicherlayout getroffen werden.[9]
Eine weitere wichtige Aufgabe, welche der Compiler im Rahmen der semantischen Analyse verrichtet, ist es, implizite Typumwandlungen durchzuführen. Soll z.B. der '+' Operator auf einen Operanden vom Typ *int* und einen Operanden vom Typ *float* angewandt werden, so muss eine Umwandlung der ganzen Zahl in eine Fließkommazahl erfolgen, da der '+' Operator in der Regel nur für zwei Operanden des selben Typs definiert ist. Der Grund dafür ist, dass die meisten Formen des Zielcodes unterschiedliche Anweisungen für die Addition von ganzen Zahlen und Fließkommazahlen nutzen.

2.1.2 Synthesephase

In der Synthesephase wird der gewünschte Zielcode erzeugt. Dazu werden in der Regel mehrere Zwischenformen erstellt, welche zur plattformunabhängigen Optimierung des Codes genutzt werden. Der folgende Abschnitt ist bewusst kurz gehalten, da der Fokus dieser Arbeit auf der Analyse von Quellcode liegt.

[9] [Aho07, Seite 370] 'Chapter 6 Intermediate-Code Generation'

Zwischencodegenerierung

In dieser Teilphase wird aus der Eingabe, welche in der Regel der Parsebaum ist, eine oder mehrere geeignete Formen von Zwischencode erstellt. Der Zwischencode ist eine abstrakte maschinenunabhängige Repräsentation des Programms.[10] Der in Abschnitt 2.1.1 'Syntaktische Analyse' bereits erwähnte abstrakte Syntaxbaum ist eine häufig verwendete Form von Zwischencode. Der AST bietet den Vorteil, dass die hierarchische Struktur des Quellcodes auf eine einfache Weise abgebildet werden kann. Dabei bilden die inneren Knoten des Baums die Konstrukte der Programmiersprache ab.
Eine weitere, sehr verbreitete Form von Zwischencode ist der sogenannte Drei-Adress-Code. Dabei handelt es sich um eine Abfolge von Instruktionen der Form $a = b\ op\ c$, wobei es sich bei a, b und c um Identifier oder Konstanten und bei op um einen Operator handelt.[11] Der Name Drei-Adress-Code erklärt sich daraus, dass die Namen der Identifier deren Adressen im Zielcode widerspiegeln. Der größte Vorteil des Drei-Adress-Codes ist, dass er auf Grund seiner Einfachheit sehr gut in der folgenden Phase der Compiler-Pipeline optimiert werden kann.

Codeoptimierung

Im Rahmen der Codeoptimierung wird zunächst versucht, den maschinenunabhängigen Zwischencode zu verbessern. Zusätzlich dazu können aber auch Verbesserungen des Codes vorgenommen werden, welche nur für bestimmte Formen des Zielcodes möglich sind. Viele der Probleme, die während der Codeoptimierung auftreten, sind nicht effizient berechenbar, deshalb wird bei der Codeoptimierung mit Heuristiken gearbeitet, die eine optimale Lösung approximieren.[12] Die Optimierung wird zusätzlich dadurch erschwert, dass der Zielcode viele unterschiedliche Qualitäten haben kann, die je nach Anwendungsfall optimiert werden können. Für eingebettete Systeme, mit wenig Speicher, ist eine geringe Größe des Zielcodes sehr wichtig. Für moderne CPU Architekturen ist es hingegen oft sehr wichtig, dafür zu sorgen, dass Instruktionen im Zielcode möglichst oft parallel ausgeführt werden können. Bei portablen Geräten, wie beispielsweise Smartphones, ist die Energieeffizienz ein entschiedenes Kriterium. Zwischen diesen, oft gegensätzlichen Eigenschaften, muss bei der Codeoptimierung immer eine Abwägung getroffen werden, um einen möglichst guten Zielcode generieren zu können.
Eine Methode, die häufig zur Optimierung angewendet wird, ist das Entfernen von Code, welcher auf Grund des Kontrollflusses niemals ausgeführt werden kann. Das Entfernen dieses 'toten' Codes verringert die Größe des entstehenden Zielcodes. Eine einfache Methode um die Ausführung des Codes zu beschleunigen ist, das Ergebnis komplizierter arithmetischer

[10] [Aho07, Seite 9] 'Chapter 1 Introduction'
[11] [Aho07, Seite 558] 'Chapter 6. Intermediate-Code Generation'
[12] [Aho07, Seite 505] 'Chapter 8 Code Generation'

Operationen einmalig zu berechnen und dann zu speichern. Alle folgenden Vorkommen dieser Operationen können durch einfaches Kopieren des gespeicherten Ergebnisses ersetzt werden. Bei allen Optimierungen des Codes ist die wichtigste Anforderung jedoch, dass die Semantik des Programms nicht verändert wird. Weitere Konzepte und Techniken zur Codeoptimierung können in Alfred V. Aho. *Compilers: Principles, techniques, & tools*[Aho07] in *'Chapter 9 Machine-Indepent Optimization'* und *'Chapter 11 Optimizing for Parallelism and Locality'* gefunden werden.

Zielcodegenerierung

Der letzte Schritt in der Compiler-Pipeline ist die Generierung des Zielcodes. Dazu werden der Zwischencode aus der vorherigen Phase und die Informationen aus der Symboltabelle benötigt. Die wichtigste Anforderung bei der Generierung des Zielcodes ist es, die Semantik des Programms zu erhalten. Die verwendeten Techniken und die Komplexität der Zielcodegenerierung hängen von der Art des Zielcodes ab. Die häufigsten Arten von Zielcode sind statische Binaries, welche direkt ausgeführt werden können, und dynamische Binaries. Diese beiden Arten von Zielcode unterscheiden sich im Speicherlayout. Bei den statischen Binaries kommen fixe Speicheradressen zum Einsatz. Bei den dynamischen Binaries werden hingegen relative Adressen genutzt. Dies ermöglicht es, die dynamischen Binaries mit anderen statischen Binaries zusammen zu binden und den Code so wiederzuverwenden. Deshalb werden diese oft für Bibliotheken genutzt.
Eine weitere Art von Zielcode, die heutzutage sehr weit verbreitet ist, ist der *Bytecode*. Dieser wird nicht, wie die beiden anderen Arten, direkt auf einer CPU ausgeführt, sondern in einer virtuellen Maschine. Ein prominentes Beispiel für die Verwendung von Bytecode ist die Programmiersprache Java mit der zugehörigen Java Virtual Machine. Auch die Programmiersprache C#, welche für Implementierung zu dieser Arbeit genutzt wurde, arbeitet mit Bytecode. Die zugehörige Laufzeitumgebung nennt sich *Common Language Runtime (CLR)*. Die Verwendung dieser virtuellen Zielplattformen hat den Vorteil, dass der entstehende Code unabhängig von der verwendeten Prozessorarchitektur ist, solange eine Implementierung der virtuellen Maschine existiert.

3 Die Roslyn Compiler Plattform

Um das Problem des Compilers als 'Black Box' für das .NET Umfeld zu lösen, hat Microsoft im Jahr 2014[1] ein Projekt mit dem Codenamen *Roslyn* veröffentlicht, dessen Ziel es ist, eine Compiler-Plattform für Visual Basic (VB) und C# zu schaffen. Die Implementierung der Compiler-Plattform ist quelloffen und kann auf GitHub eingesehen werden. Dies bietet den Vorteil, dass Drittentwickler keine eigenen Compiler für ihre Analysetools entwickeln müssen. Dadurch sinkt die Wahrscheinlichkeit, dass bei der Implementierung von Drittentwicklern Fehler entstehen und die Adaption neuer Sprachfeatures kann schneller erfolgen. Die Roslyn Compiler-Plattform schafft eine Vielzahl von Möglichkeiten für Drittentwickler, da sie Schnittstellen für alle Phasen des Kompilierprozesses bereitstellt. Besonders interessant sind die Schnittstellen zur syntaktischen und semantischen Analyse von Quellcode, die es ermöglichen, domänenspezifische Analysetools zu implementieren. Doch die Fähigkeiten von Roslyn sind nicht auf die Analyse von Quellcode beschränkt. Die Plattform bietet Schnittstellen zur Modifikation von Quell- und Bytecode. Ein Teil dieser Fähigkeiten zur Analyse und zur Modifikation von Quellcode werden im Rahmen der Implementierung dieser Arbeit genutzt. Da sich die Implementierung nur auf die Programmiersprache C# bezieht, wird im Folgenden nicht auf die Möglichkeiten zur Verarbeitung von VB Quellcode eingegangen. Alle Konzepte und Schnittstellen sind für VB sehr ähnlich oder gar identisch.
Dieses Kapitel soll einen Überblick über die Funktionsweise von Roslyn und den entsprechenden Schnittstellen geben. Zuvor ist es jedoch von Vorteil, den Aufbau eines C#-Projekts kurz zu erläutern.

3.1 Aufbau eines C#Projekts

Ein C#-Projekt enthält immer eine Projektdatei mit der Endung *.csproj*, in der alle relevanten Metadaten für das Projekt enthalten sind. Bei der Projektdatei handelt es sich um eine XML-Datei, welche einem von Microsoft definierten Schema folgt. Sie enthält unter anderem Referenzen auf alle Abhängigkeiten des Projektes und die Zielplattform für die das Projekt kompiliert werden soll. Die Abhängigkeiten liegen in der Regel in Form von sogenannten *Assemblies* vor. Assemblies sind kompilierte Binaries für die .NET Laufzeitumgebung. Außerdem enthält die Projektdatei Referenzen auf alle Quell- und Konfigurationsdateien, die

[1] [Pil18] Erstes Commit zu Roslyn auf GitHub.

zum Projekt gehören. Damit definiert die Projektdatei alle Informationen, die benötigt werden, um eine ausführbare Datei zu erstellen.[2] Mehre Projekte können zu einer sogenannten *Solution* zusammengefasst werden. Dies ermöglicht eine bessere Modularisierung des Quellcodes, ohne den Überblick zu verlieren.

3.2 Roslyn Compiler Pipeline

Die Roslyn Compiler Pipeline besteht aus vier aufeinanderfolgenden Phasen, welche alle ein eigenes Objektmodell und eine eigene Programmierschnittstelle besitzen. Der Aufbau der Pipeline orientiert sich dabei am Aufbau einer klassischen Compiler Pipeline, wie sie in Abbildung 2.1 dargestellt ist. Einige der Phasen wurde jedoch zusammengefasst. Abbildung 3.1 zeigt die Struktur der Roslyn Compiler Pipeline und die zu jeder Phase zugehörige Programmierschnittstelle.

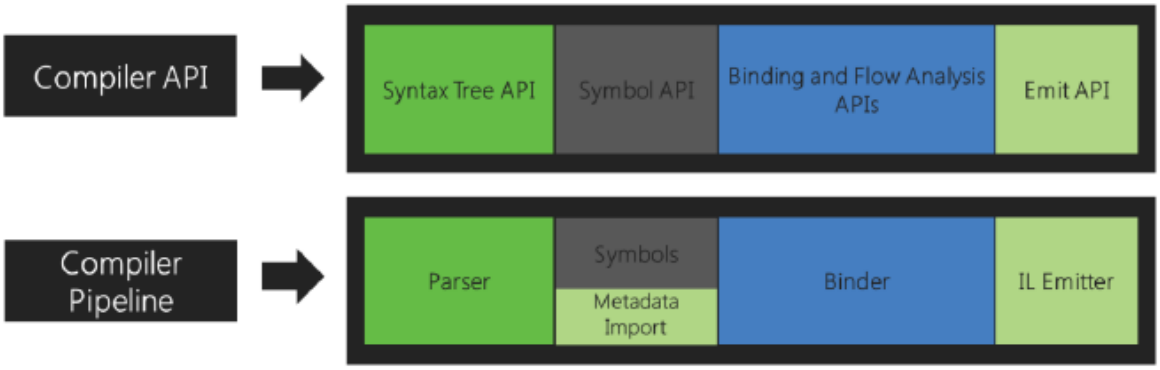

Abbildung 3.1: Roslyn Compiler Pipeline mit den zugehörigen Schnittstellen aus Karen Ng et al. *The Roslyn Project: Exposing the C# and VB compiler's code analysis.* 2012 (Seite 4)

Die erste Phase in der Roslyn Compiler Pipeline ist die *Parsing-Phase*. Diese Phase vereint einen Großteil der Aufgaben der lexikalischen und syntaktischen Analyse aus der klassischen Compiler Pipeline. Das Objektmodell dieser Phase sind die *Syntax Trees*, welche wie der Name bereits andeutet, eine direkte Implementation von Syntaxbäumen sind. Es handelt sich bei den von Roslyn genutzten Syntaxbäumen stets um konkrete Syntaxbäume, welche alle Informationen des zugehörigen Quellcodes enthalten. Der Zugriff auf die Syntaxbäume erfolgt über die zugehörige Programmierschnittstelle, welche in Abschnitt 3.5 genauer erläutert wird.
Die zweite Phase in der Roslyn Compiler Pipeline ist die *Deklarations-Phase*. Dabei werde alle Deklarationen aus dem Quellcode und den Metadaten der referenzierten Assemblies analysiert. Die Deklarations-Phase implementiert einen Teil der Aufgaben, welche in der

[2] [Cur12, Seite 32] 'MSBuild and the Project File'

klassischen Compiler Pipeline in der Phase der semantischen Analyse anfallen. Das in der Deklarations-Phase genutzte Objektmodell ist die *Symboltabelle*. In dieser wird für jedes relevante Symbol ein Eintrag angelegt, welcher im späteren Verlauf mit weiteren Informationen gefüllt wird. Die Möglichkeiten der zugehörigen *Symbol API* werden in Abschnitt 3.6 erläutert.
Als drittes folgt die *Bindungs-Phase*, welche ebenfalls hauptsächlich Aufgaben aus der Phase der semantischen Analyse implementiert. In dieser Phase werden die Einträge der Symboltabelle mit allen relevanten Informationen befüllt und den Bezeichnern werden die entsprechenden Symbole zugeordnet. Das zugehörige Objektmodell ist das sogenannte *Semantic Model*, welches Zugriff auf die semantischen Informationen einer Quellcodedatei bietet. Die Analysemöglichkeiten dieses semantischen Modells werden in Abschnitt 3.6 ausführlicher dargestellt.
Die letzte Phase in der Roslyn Compiler Pipeline ist die *Emittierungs-Phase*, diese erstellt mit Hilfe der Informationen aus den vorangegangenen Phasen den Bytecode des Programms. Diese Phase realisiert die Aufgaben der Zielcodegenerierung. Die zugehörige Programmierschnittstelle definiert dazu eine Menge von domänenspezifischen Modellen. Dazu zählen unter anderem die Datenstrukturen *Compilation* oder *MetadataReference*. Diese repräsentieren kompilierten Quellcode und dessen benötigte Referenzen. Darauf wird im Rahmen dieser Arbeit nicht näher eingegangen, da die Schnittstellen und Datenstrukturen dieser Phase keine große Bedeutung für die Implementierung besitzen.

3.3 Unveränderlichkeit der Roslyn-Datenstrukturen

Nahezu alle von Roslyn bereitgestellten Datenstrukturen sind unveränderlich (engl. *immutable*). Dies hat zur Folge, dass eine erstellte Instanz niemals verändert werden kann. Um eine Datenstruktur zu ändern, ist es deshalb nötig, aus der bereits bestehenden Instanz eine neue Instanz zu erstellen. Beim Erstellen der Instanz können alle benötigten Änderungen angewandt werden. Diese Einschränkung wurde bei der Entwicklung von Roslyn bewusst in Kauf genommen, da die Unveränderlichkeit der Datenstrukturen die Performance deutlich erhöht. Unveränderliche Datenstrukturen bieten auch ohne die Verwendung von Locks oder ähnlichen Konzepten zur Prozesssynchronisation, Threadsicherheit.[3] Dies verhindert, dass sich mehrere Prozesse bei nebenläufigen Zugriffen auf die Datenstrukturen blockieren müssen. Die Möglichkeit, aus verschiedenen Prozessen auf die Datenstrukturen zuzugreifen, ist besonders wichtig, wenn der Compiler Analyseaufgaben verrichtet, während der Nutzer den Quellcode in der IDE bearbeitet.

[3] [Kar+12, Seite 5] '3.1.1 Syntax Trees'

3.4 Abstraktionsebenen der Roslyn API

Roslyn bietet mit seiner Vielzahl von Schnittstellen unter anderem Zugriff auf die Compiler Pipeline, die Daten aus der C# Projekt-Datei und sogar auf die Entwicklungsumgebung in der Roslyn ausgeführt wird. Diese Schnittstellen sind auf drei verschiedenen Abstraktionsebenen gruppiert, welche im Folgenden vorgestellt werden.

3.4.1 Compiler Service

Die Schnittstellen des Compiler Services bieten Zugriff auf jeden Schritt in der Compiler Pipeline. So ist es beispielsweise möglich, eine Rückruffunktion zur Analyse von Quellcode zu definieren, welche der Compiler, beim Eintreten bestimmter Ereignisse, ausführt. Im Kontext der Analyse wird unter anderem Zugriff auf den konkreten Syntaxbaum ermöglicht. Des Weiteren bietet der Compiler Service Zugriff auf das *Semantic Model*. Dies ist ein Modell des gesamten Programms, welches der Compiler nach einem Durchlauf erstellt hat. Es enthält alle semantischen Informationen aus den Quelldateien und den referenzierten Assemblies. Die Schnittstellen des Compiler Services unterscheiden sich für C# und VB. Die für diese Arbeit interessanten Bestandteile des Compiler Service sind die *Diagnostic Analyzer* und die *Code Fix Provider*, da diese Schnittstellen für die Analyse und das automatisierte Refactoring von Quellcode bieten. Deshalb ist diesen beiden Themen jeweils ein eigener Abschnitt gewidmet.

3.4.2 Diagnostic Analyzer

Diagnostic Analyzer, im Folgenden der Einfachheit halber *Analyzer*, bieten die Möglichkeit, den Kompilierprozesses um benutzerdefinierte Analyseschritte zu erweitern. Dazu muss eine Klasse implementiert werden, welche von der Basisklasse *Microsoft.CodeAnalysis.Diagnostics.DiagnosticAnalyzer* erbt. Die Basisklasse gibt dabei die Implementierungen der zwei Methoden *SupportedDiagnostics* und *Initialize* vor. Erstere stellt lediglich ein Array bereit, in dem alle Regeln enthalten sind, welche der Analyzer untersucht. Letztere bekommt einen Anlysecontext übergeben, welcher die Möglichkeit bietet, eine Rückruffunktion für bestimmte Compilerereignisse zu registrieren. Es gibt eine Vielzahl von Compileraktionen, welche sich sowohl auf syntaktische als auch auf semantische Aktionen im Kopilierprozess beziehen können. Ein Beispiel für solch eine syntaktische Aktion ist die generische *RegisterSyntaxNodeAction*. Diese benötigt als Parameter noch eine bestimmte Syntaxart, auf die sich die Aktion beziehen soll. Wird beispielsweise die Syntaxart *LocalDeclarationStatement* übergeben, so wird die Rückruffunktion immer dann ausgeführt, wenn der Compiler die Deklaration einer lokalen Variablen im Syntaxbaum findet.
In der übergebenen Rückruffunktion kann domänenspezifische Analyselogik implementiert werden. Im Rahmen des Analysecontext besteht sowohl Zugriff auf den Syntaxbaum als auch auf das semantische Modell des Programms. Da Teile der Analyse sich auf die Syntax der

zu untersuchenden Sprache beziehen können, ist es notwendig, die Zielsprache in Form eines Attributes an der Klasse zu annotieren. Analyzer können sowohl spezifisch für C# oder VB, aber auch für beide Sprachen gleichzeitig geeignet sein. Wird im Rahmen der Analyse ein Problem festgestellt, kann dies dem Analysecontext als Instanz der Klasse *DiagnosticDescriptor* hinzugefügt werden. Diese Instanz enthält alle für den Compiler und den Nutzer wichtigen Informationen, wie z.B. den Schweregrad des Problems, welcher in vier Stufen unterteilt wird. Die Schweregrade sind, in der Priorität aufsteigend, *Hidden*, *Info*, *Warning* und *Error*. Eine weitere wichtige Eigenschaft ist die *DiagnosticId*, welche genutzt wird, um das Problem eindeutig zuzuordnen. Anhand der *DiagnosticId* werden dem Problem passende Lösungsvorschläge in Form von *Code Fixes* zugeordnet. Wird der Analyzer in einer Host-Umgebung, wie z.B. Visual Studio ausgeführt, so wird der problematische Quellcode direkt hervorgehoben, wie in Abbildung 3.2 zu sehen.

3.4.3 Code Fix Provider

Ein *Code Fix Provider*, im Folgenden der Einfachheit halber *Code Fix*, bietet Lösungsvorschläge für ein bestimmtes Problem. Dazu muss die Klasse des Code Fixes von der Klasse *Microsoft.CodeAnalysis.CodeFixProvider* erben. Die Basisklasse schreibt die Implementierung der beiden Methoden *FixableDiagnosticIds* und *RegisterCodeFixesAsync* vor. Erstere stellt lediglich ein Array mit den DiagnosticIds der Probleme bereit, die von diesem Code Fix behoben werden können. Auf diese Weise wird die Verbindung zwischen Analyzer und Code Fix ohne harte Abhängigkeiten hergestellt. Die Methode *RegisterCodeFixesAsync* ist deutlich interessanter, da diese die Logik zur Behebung des Problems enthält. Das Suffix *Async* deutet an, dass es sich um eine asynchrone Methode handelt. Dies ermöglicht es, den Code Fix im Hintergrund zu berechnen, während der Nutzer in der Entwicklungsumgebung arbeitet. Die Methode bekommt einen *CodeFixContext* als Argument übergeben. Dieser bietet, wie auch der Analysecontext des Analyzers, Zugriff auf den Syntaxbaum und auf das semantische Modell des Programms. Des Weiteren enthält der Kontext das diagnostizierte Problem und dessen Position im Quellcode. Mit Hilfe dieser Informationen kann das Programm so geändert werden, dass das Problem behoben wird. Häufig ist dazu die Änderung des Syntaxbaums nötig. Da dieser aber unveränderlich ist, muss unter Anwendung aller erwünschten Änderungen ein neuer Syntaxbaum aus dem bereits vorhandenen Syntaxbaum erstellt werden. Konkret werden einige dieser Möglichkeiten in Abschnitt 6.1 thematisiert.

Der Code Fix wird in der Regel in einer Host-Umgebung wie z.B. Visual Studio ausgeführt. Es bestehen jedoch keine Abhängigkeiten des Code Fixes zur Host-Umgebung. Abbildung 3.2 zeigt ein diagnostiziertes Problem mit vorhandenem Code Fix in Visual Studio. Der problematische Quellcode ist dabei grünlich Unterstrichen. Das Glühbirnen Symbol zeigt im Kontextmenü alle vorhandenen Code Fixes zu diesem Problem an. Für den ausgewählten

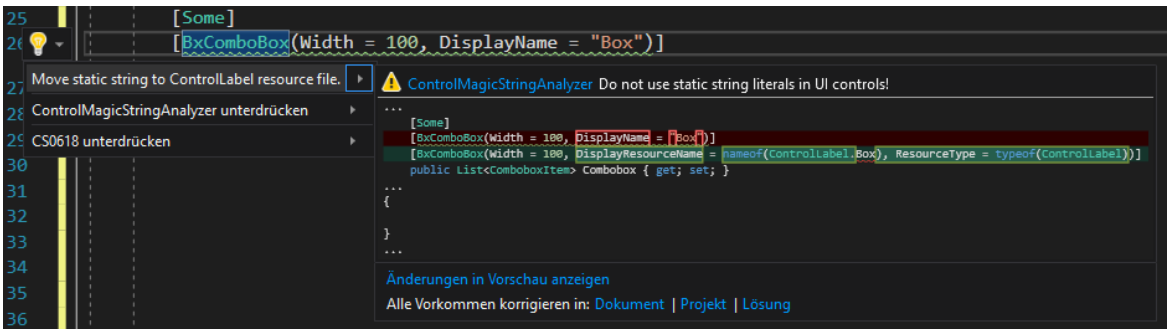

Abbildung 3.2: Screenshot: Problematischer Quellcode mit Vorschau der automatischen Lösung in Visual Studio

Code Fix wird eine Vorschau des transformierten Quellcodes angezeigt. Durch Ausführen des Code Fixes, wird diese eine Instanz des Problems behoben. Das Vorschaufenster bietet zusätzlich die Möglichkeit, mehrere Vorkommen des diagnostizierten Problems zu beheben. Dazu muss der Code Fix die Methode *GetFixAllProvider* implementieren. In dieser Methode muss die Logik zur Behebung mehrerer Vorkommen des Problems implementiert werden. Die einfachste Strategie ist eine Batch-Verarbeitung, bei der die einzelnen Vorkommen des Problems nacheinander behoben werden. Diese Strategie ist meistens ausreichend, weshalb bereits eine Standardimplementierung in den Schnittstellen der Roslyn Plattform enthalten ist. Beeinflussen sich mehrere Vorkommen des Problems jedoch gegenseitig, kann es nötig sein eine komplexere Strategie zur Behebung zu implementieren. Hier kommt eine große Stärke von Roslyn zum Vorscheinen. Sobald ein Analyzer und der entsprechende Code Fix implementiert und getestet sind, können tausende Probleme in einem großen Software Projekt mit einem einzigen Klick gelöst werden.

3.4.4 Workspace Service

Der Workspace Service ist eine weitere Abstraktionsebene der Roslyn Schnittstellen und ermöglicht das Arbeiten mit Workspaces. Ein *Workspace* repräsentiert den aktuellen Zustand einer kompletten Solution.[4] Also aller enthaltenen Projekte, deren Abhängigkeiten und Quelldateien. Dies ermöglicht den Analyzern und Code Fixes ein komplettes semantisches Modell des Programms zu untersuchen. So kann zum Beispiel bei einem Code Fix, welcher den Namen eines Typen verändert, sichergestellt werden, dass alle Vorkommen des Namens geändert werden, auch wenn der Typ in einem anderen Projekt der Solution verwendet wird.

Abbildung 3.3 zeigt die hierarchische Struktur eines Workspaces. Da Teile der Solution, wie z.B. einzelne Quelldateien, von außen geändert werden können, muss der Workspace ständig aktualisiert werden. Da die Datenstruktur des Workspaces ebenfalls unveränderlich ist, muss

[4] [Kar+12, Seite 20] '5.1.1 Workspace'

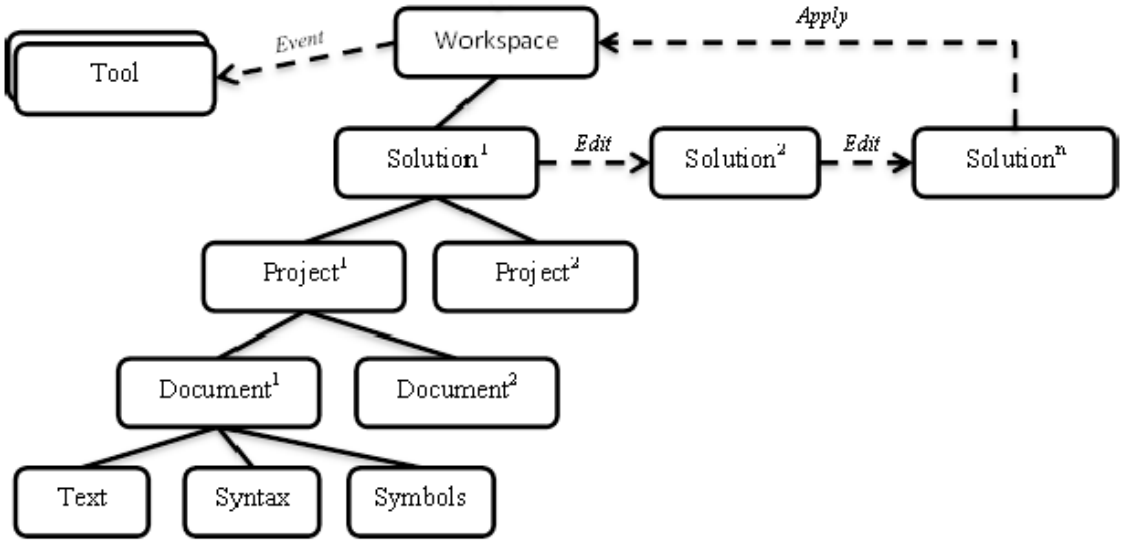

Abbildung 3.3: Aufbau eines Workspaces aus Karen Ng et al. *The Roslyn Project: Exposing the C# and VB compiler's code analysis.* 2012

bei jeder Änderung eine neue geänderte Instanz erstellt werden. Dies erledigt Roslyn jedoch automatisch. Eine Veränderung des Workspace ist eines der Compileereignisse, welche die Ausführung einer Analysefunktion veranlassen können.

3.4.5 Editor Service

Die dritte Abstraktionsebene der Roslyn Schnittstellen ist der sogenannte *Editor Service*, welcher die Schnittstelle zur Host-Umgebung ist. Da jede Host-Umgebung andere Schnittstellen bereitstellt, hat diese Schicht der Roslyn Schnittstellen Abhängigkeiten zu der jeweiligen Host-Umgebung. Der Editor Service der Roslyn Schnittstellen ist standardmäßig auf die Integration in Visual Studio ausgerichtet. Dies ermöglicht es Tools, welche die Roslyn Schnittstellen nutzen, ohne Mehraufwand direkt in Visual Studio in Form einer Erweiterung zu integrieren. So besteht z.B. Zugriff auf Editor Features wie die Autovervollständigung oder das integrierte Fehlerlog. Da alle Abhängigkeiten zu Visual Studio in der Schicht des Editor Services gekapselt sind und dessen Verwendung für die Analyse und Modifikation von Quellcode nicht notwendig ist, sind Roslyn Tools im Allgemeinen nicht auf Visual Studio angewiesen. Somit ist die Plattformunabhängigkeit der mit Roslyn erstellten Tools gewährleistet.

3.5 Syntaktische Analyse

Der folgende Abschnitt gibt einen Überblick darüber, wie die Schnittstellen der Roslyn Plattform zur syntaktischen Analyse von Quellcode genutzt werden können. Die dafür wichtigste

Datenstruktur ist die Klasse *SyntaxTree*, welche, wie der Name vermuten lässt, das Konstrukt des Syntaxbaums implementiert. Häufig werden für die syntaktische Analyse von Quellcode abstrakte Syntaxbäume verwendet, bei denen Informationen, welche für die Semantik des Programms irrelevant sind, außer Acht gelassen werden. Eine Instanz der Klasse SyntaxTree enthält hingegen alle Informationen der zugehörigen Quellcodedatei. Dies schließt z.B. auch Leerzeichen und Kommentare ein, welche für die Semantik des Programms nicht von Bedeutung sind. Somit handelt es sich bei SyntaxTrees um konkrete Syntaxbäume. Ist eine Quellcodedatei syntaktisch inkorrekt, so ist der zugehörige SyntaxTree fehlerhaft. Der Compiler erkennt diese Fehler und versucht sie zu behandeln. Fehlen beispielsweise Zeichen in der Quellcodedatei, so fügt der Compiler spezielle 'Fehlerknoten' in den SyntaxTree ein. Dies ermöglicht es, Syntaxfehler am Anfang der Compiler Pipeline zu erkennen, bevor weitere, unter Umständen deutlich rechenintensivere, Analysen durchgeführt werden. Da ein SyntaxTree die zugehörige Quelldatei repräsentiert, entspricht eine Änderung des SyntaxTrees einer Änderung des Quellcodes. Diese Möglichkeit kann genutzt werden, um mit Hilfe eines Code Fixes den Quellcode zu modifizieren. Wie nahezu alle Datenstrukturen der Roslyn Schnittstellen sind auch die SyntaxTrees unveränderlich.

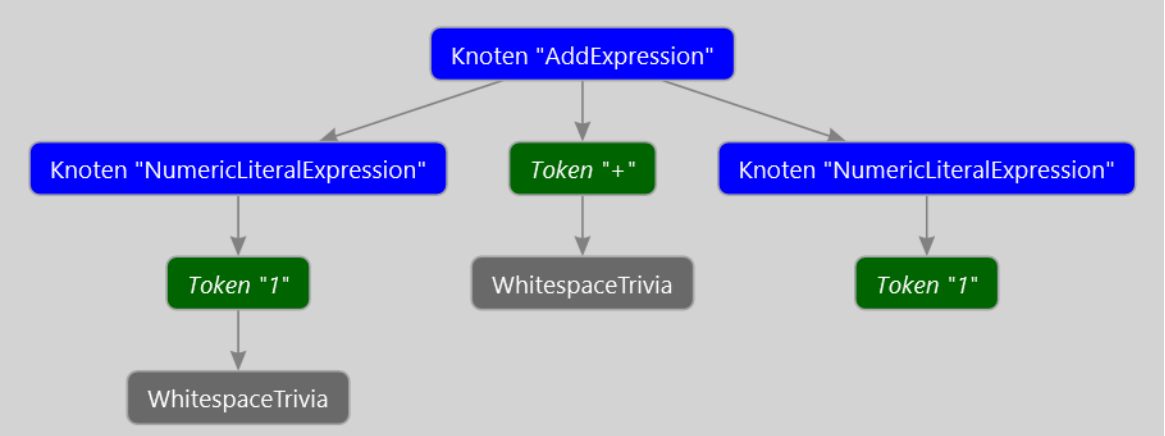

Abbildung 3.4: Visualisierung eines SyntaxTrees zum Ausdruck '1 + 1' mit seinen unterschiedlichen Knoten

SyntaxTrees sind simple Baumstrukturen, mit drei verschiedenen Arten von Knoten. Am interessantesten sind dabei die *SyntaxNodes*, welche die syntaktischen Konstrukte der Programmiersprache abbilden. SyntaxNodes repräsentieren deshalb auch die Nichtterminale der zugrundeliegenden Grammatik. SyntaxNodes sind niemals Blätter im Baum und haben immer weitere Kinder. Diese Kindknoten sind strikt typisiert und hängen vom Typ des jeweiligen SyntaxNodes ab. Wie in Abbildung 3.4 zu sehen ist, hat ein SyntaxNode vom Typ *BinaryExpressionSyntax* die drei Kinder *Left*, *OperatorToken* und *Right*.[5] Zusätzlich dazu

[5] [Kar+12, Seite 7] '3.1.2 Syntax Nodes'

hat jeder SyntaxNode noch eine *SyntaxKind* Eigenschaft, welche genauer spezifiziert, um welches Syntaxelemente es sich handelt. Ein SyntaxNode vom Typ BinaryExpressionSyntax kann verschiedene Werte für die SyntaxKind Eigenschaft enthalten, beispielsweise sind *AddExpression* oder *SubtractExpression* gültige Werte. Eine weitere Art von Knoten, welche ebenfalls in der Abbildung zu sehen sind, sind die *Tokens*. Die Klasse der Tokens implementiert das Konzept, welches in Kapitel 1 im Abschnitt 2.1.1 'Lexikalische Analyse' bereits vorgestellt wurde. Tokens repräsentieren die Terminale der zugrundeliegenden Grammatik und sind somit immer Blätter im Baum. Alle Tokens haben die Eigenschaft *Value*, welche den Wert des Tokens enthält. Der Wert dieser Eigenschaft hängt dabei von der Art des Tokens ab. So ist der Wert eines *Integer Value Tokens* beispielsweise immer eine ganze Zahl. Zusätzlich dazu enthält jedes Token noch die *ValueText* Eigenschaft, welche immer vom Typ String ist und die Textrepräsentation des Tokenwerts enthält. Die dritte Art von Elementen, die in einem SyntaxTree vorkommen kann, ist die sogenannte *Trivia*. Die Trivia beinhaltet alles was keine semantische Bedeutung für das Programm besitzt, wie z.B. Leerzeichen oder Kommentare. Die Trivia ist für die Analyse und Modifikation von Quellcode eher uninteressant. Alle Bestandteile des SyntaxTrees besitzen eine Eigenschaft namens *Span*, welche die zugehörige Position im Quellcode enthält. Ein Span enthält dazu die Position des ersten Zeichens und die Länge des Textabschnitts. Zusätzlich zur Span Eigenschaft gibt es noch die Eigenschaft *FullSpan*, welche zusätzlich zum Span noch eventuell vor- und nachgestellte Trivia enthält.

3.6 Semantische Analyse

Da die Möglichkeiten der syntaktischen Analyse begrenzt sind, bietet Roslyn eine große Menge an Schnittstellen zur semantischen Analyse. Zu analysierende C#-Projekte referenzieren häufig die Assemblies der verwendeten Libraries. Da eine Assembly bereits kompiliert ist, ist eine syntaktische Analyse unmöglich. Die Schnittstellen zur semantischen Analyse können die Assembly, auf Grund der beinhalteten Metadaten, trotzdem in das Modell des Programms integrieren.
Dies geschieht mit Hilfe der *Compilation* Datenstruktur. Sie enthält alle Informationen, die benötigt werden, um das Programm zu kompilieren. Dazu gehören unter anderem die Quellcodedateien, alle referenzierten Assemblies und die Compileroptionen.[6] Da eine Compilation all diese Informationen aggregiert, bietet sie einen guten Ausgangspunkt für eine umfangreiche semantische Analyse. Eine Compilation bietet unter anderem Zugriff auf alle deklarierten Typen und Member. Diese werden als Instanzen der Klasse *Symbol* repräsentiert. Es besteht sowohl Zugriff auf Symbole, welche im Quellcode des Projekts deklariert sind, als auch auf die Symbole welche aus den referenzierten Assemblies importiert werden. Diese Symbole re-

[6] [Kar+12, Seite 14] '4.1.1 Compilation'

präsentieren einzelne Sprachkonzepte. Es gibt für jedes Sprachkonzept eine eigene Klasse, welche von der Basisklasse Symbol erbt. So repräsentiert die Klasse *MethodSymbol* beispielsweise das Konzept der Programmiersprachen von Methoden. Da die Programmiersprachen C# und VB auf einer abstrakten Ebene einige gemeinsame Sprachkonzepte besitzen, kann es vorkommen, dass bei der Analyse der verschiedenen Sprachen die selben Symbole genutzt werden.[7] Dies kann bei der Generierung des Zielcodes dazu führen, dass das selbe Symbol, je nach Zielplattform, zu unterschiedlichem Bytecode übersetzt wird.
Eine weitere Datenstruktur die im Rahmen der semantischen Analyse von Bedeutung ist, ist das *SemanticModel*. Dieses enthält, wie eine Compilation, eine umfangreiche Menge an Informationen. Allerdings enthält das Semantic Model lediglich die semantischen Informationen einer einzelnen Quelldatei.[8] Dazu gehören unter anderem alle verwendeten Symbole.

[7] [Kar+12, Seite 14] '4.1.2 Symbols'
[8] [Kar+12, Seite 14] '4.1.3 Semantic Model'

4 Anforderungsanalyse und Konzept

Um ein strukturiertes und effizientes Arbeiten zu ermöglichen, wurde vor Beginn der Implementierung eine Anforderungsanalyse durchgeführt. Dabei werden funktionale und nichtfunktionale Anforderungen unterschieden. Da der zeitliche Rahmen der Implementierung begrenzt war, wurden alle Anforderungen priorisiert. Die drei verwendeten Prioritäten sind *muss*, *soll* und *kann*. Die Implementierung aller Anforderungen mit der Priorität *muss*, ist für die Funktionalität der Software unerlässlich, weshalb diese Anforderungen realisiert werden müssen. Die Anforderungen der Priorität *soll* sind vorteilhaft, aber nicht unerlässlich. Alle Anforderungen der Priorität *kann* sind sind zwar wünschenswert, aber für die Grundfunktion der Software nicht von Bedeutung und daher ist ihre Implementierung optional.

4.1 Funktionale Anforderungen

Ziel der Implementierung ist es, den Prototyp einer Software zu entwickeln, welche automatisch domänenspezifische Probleme findet und behebt. Dieses Ziel allein stellt jedoch keine einzige große funktionale Anforderung dar, da es zu monolithisch und zu unspezifisch ist. Deshalb wurden im Rahmen der Analyse der funktionalen Anforderungen verschiedene Problemklassen und zugehörige Lösungsansätze identifiziert. Das automatische Erkennen und Beheben eines jeden Problems, stellt eine Anforderung für sich dar.
Die domänenspezifischen Probleme mussten dazu erst manuell identifiziert werden. Dazu wurde ein Brainstorming mit den Entwicklern der bilettix GmbH veranstaltet, da sie täglich mit dem betroffenen Quellcode arbeiten und diesen daher am besten kennen. Außerdem sind die Entwickler der bilettix GmbH die späteren Nutzer der zu entwickelnden Software. Da das Entwicklerteam nur eine Größe von fünf Personen hat, war es möglich, alle Entwickler miteinzubeziehen. Zusätzlich zum Brainstorming wurden weitere Probleme in Zusammenarbeit mit dem Leiter des Entwicklungsteams identifiziert.
Die gefunden Probleme wurde in drei Klassen eingeteilt, welche im Folgenden erklärt werden. Zusätzlich zum Problem an sich werden noch der Grund des Problems und der zugehörige Lösungsansatz erklärt.

4.1.1 BX01 - Coding Style

BX01-01 If-Statements ohne Block

Die Grammatik der Programmiersprache C# erlaubt es, *if*-Statements ohne geschweifte Klammern für den darauf folgenden Block zu verwenden. In diesem Fall ist nur der Ausdruck in der direkt folgenden Zeile konditional. Dies birgt die Gefahr, dass bei einer späteren Erweiterung des Codes der Kontrollfluss ungewollt verändert wird. Soll eine weitere konditionale Instruktion hinzugefügt werden, muss darauf geachtet werden, dass durch die Verwendung von geschweiften Klammern, ein neuer Block erstellt wird. Geschieht dies nicht, wird die neu hinzugefügte Instruktion, unabhängig von der Bedingung im *if*-Statement, ausgeführt.
Dieses Problem mag auf den ersten Blick unwichtig und konstruiert erscheinen. Ein Beispiel aus der Praxis zeigt aber, dass dem nicht so ist. Im Jahr 2014 wurde ein Fehler in Apples Betriebssystemen gefunden, welcher dafür sorgte, dass im Rahmen eines SSL Handshakes eine Singnaturprüfung übersprungen wurde. Der Bug mit der Nummer *CVE-2014-1266* wurde auch als *goto-Fail* bekannt, da er durch eine nicht konditionale *goto* Anweisung verursacht wurde.[1]

```
1 if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
2         goto fail;
3 if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
4         goto fail;
5         goto fail;
6 if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
7         goto fail;
```

Abbildung 4.1: Der für den *goto-Fail* verantwortliche Code.[App14, sslKeyExchange.c]

Abbildung 4.1 zeigt einige wenige Zeilen aus dem betroffenen Code. Der Fehler ist in Zeile fünf zu finden. Dort wurde unabsichtlich eine zweite *goto* Anweisung eingefügt, welche unabhängig von der Bedingung des *if*-Statements ausgeführt wird. Der Fehler wurde sowohl bei der Entwicklung, als auch bei der Qualitätssicherung übersehen.
Damit dieser Fehler bei bilettix.net nicht auftreten kann, wurde entschieden, dass *if*-Statements, ohne Block für die konditionalen Anweisungen, ein Problem darstellen. Die Lösung des Problems ist trivial. Es ist ausreichend, ein Paar geschweifte Klammern um die konditionale Instruktion einzufügen.

Dazu muss ein Analyzer entwickelt werden, welcher die Entwickler auf das Problem hinweist, um es in der Zukunft zu vermeiden. Zusätzlich muss ein Code Fix implementiert werden,

[1] [NIS14, NVD - CVE-2014-1266]

Priorität	Anforderung	Beschreibung
Muss	BX01-01	Bei Verwendung eines *if*-Statements muss immer ein neuer Block folgen.

Tabelle 4.1: Anforderungen der Klasse BX01

der bereits bestehende Vorkommen des Problems automatisch behebt. Die zugehörige Anforderung ist in Tabelle 4.1 aufgelistet.

4.1.2 BX02 - Vermeiden von 'Magic Strings'

Die Internationalisierung von Benutzeroberflächen ist ein Problem, mit dem die meisten Softwareentwickler früher oder später in Berührung kommen. Deshalb bieten viele UI-Frameworks bereits fertige Lösungen für dieses Problem.
Das in bilettix.net verwendete UI-Framework bietet diese Möglichkeit jedoch nicht direkt. Die in der Benutzeroberfläche angezeigten Texte werden im C# Quellcode als String Literale konfiguriert. Dazu werden die einzelnen Properties, welche auf der Benutzeroberfläche angezeigt werden sollen, mit einem Attribut versehen. Dieses enthält alle Metadaten, welche das Framework benötigt, um automatisch entsprechende UI-Controls zu generieren.

Abbildung 4.2: Beispiel für die automatische Generierung von UI-Controls

In der Abbildung 4.2 ist eine Modellklasse mit vier Eigenschaften zu sehen. Diese sind mit entsprechenden Attributen versehen. Mit Hilfe dieser Metainformationen werden die zugehörigen UI-Controls generiert. Diese sind im Screenshot rechts zu sehen. So wird beispielsweise für die Eigenschaft vom Typ *DateTime* mit Hilfe des *BxDateField* Attributes ein Datumsfeld mit einem Kalender Widget generiert.
Der gezeigte Quellcode ist unter dem Aspekt der Erweiterbarkeit jedoch problematisch, da die anzuzeigenden Texte konstant zur Kompilierzeit festgelegt werden. Somit ist eine einfache Übersetzung nicht mehr möglich, da für unterschiedliche Sprache unterschiedliche Versionen der Software erstellt werden müssten.
Dieses Problem weist starke Ähnlichkeit zum *Magic Number* Anti-Pattern auf. Dabei wer-

den Zahlen, welche im Kontext einer bestimmten Berechnung eine besondere Bedeutung haben so verwendet, dass diese Bedeutung nicht mehr nachzuvollziehen ist. Abbildung 4.3 zeigt ein Beispiel für die Verwendung einer Magic Number. Bei der ersten Implementierung ist auf den ersten Blick nicht nachvollziehbar, was die Zahl 3.142 im Kontext der Funktion bedeutet. Um dieses Problem zu lösen, schlägt Martin Fowler vor, für den Wert der Magic Number eine benannte Konstante bereitzustellen und alle Vorkommen der Magic Number durch die Verwendung der Konstanten zu ersetzen. [2] Dies hat den Vorteil, dass für jeden Entwickler sofort erkennbar ist, worum es sich bei dem Wert handelt. Des Weiteren können auf diese Weise missverständliche Mehrdeutigkeiten vermieden werden, da dieselbe Zahl in unterschiedlichen Kontexten eine unterschiedliche Bedeutung haben kann. Ein weiterer entscheidender Vorteil ist, dass bei einer späteren Änderung des Werts, nur die Zuweisung der Konstante angepasst werden muss. Die zweite Implementierung in Abbildung 4.3 nutzt eine Konstante um den approximierten Wert für π zu definieren.

```
1 public static double calculateCircleArea(double radious){
2         return 3.142 * radius * radius;
3 }
```

```
1 public const double PI = 3.142;
2
3 public static double calculateCircleArea(double radious){
4         return PI * radius * radius;
5 }
```

Abbildung 4.3: Das Anti-Pattern der Magic Numbers

Dieser Lösungsansatz ist für das Problem der Internationalisierung jedoch leider nicht ausreichend, da es notwendig ist, den anzuzeigenden Text zur Laufzeit des Programms an die richtige Region anzupassen. Da dieses Problem bei nahezu jeder größeren Software auftritt, bietet Microsoft eine vorgefertigte Lösung im .NET-Framework. Diese besteht aus der Verwendung sogenannter *RESX*-Dateien. Dabei handelt es sich um spezielle XML-Dateien, welche ein Lookup definieren, bei dem der Schlüssel ein beliebiger Bezeichner und der Wert eine (lokalisierte) Ressource ist. Um eine Internationalisierung zu ermöglichen, kann für jede gewünschte Sprache eine eigene RESX-Datei angelegt werden, welche die entsprechenden Übersetzungen enthält. Aus diesen RESX-Dateien wird zur Kompilierzeit automatisch Quellcode erstellt, welcher dazu genutzt werden kann, die entsprechende Ressource zu einem Schlüssel auszulesen. Beim Start des Programms wird aus der Umgebung die Spracheinstellung ausgelesen, um bei jedem Lookup die passende Ressource zu bestimmen.

[2] [Fow99, Seite 166] 'Chapter 8. Organizing Data'

Priorität	Anforderung	Beschreibung
Muss	BX02-01	Nutzung von String Literalen für die *DisplayName* Poperty vermeiden
Muss	BX02-02	Nutzung von String Literalen für die *ToolTip* Poperty vermeiden
Muss	BX02-03	Nutzung von String Literalen für die *FieldLabel* Poperty vermeiden

Tabelle 4.2: Anforderungen der Klasse BX02

Diese Möglichkeit des .NET-Frameworks soll für die Internationalisierung der Benutzeroberfläche von bilettix.net genutzt werden. Da es bereits mehrere tausend Texte gibt, die in Form von String Literalen definiert sind, muss ein Code Fix entwickelt werden, welche die String Literale automatisch in die entsprechende RESX-Datei extrahiert. Anschließend muss das String Literal im Quellcode durch einen Zugriff auf die RESX-Datei ersetzt werden. Dabei muss darauf geachtet werden, dass Strings die bereits in der RESX-Datei enthalten sind, nicht doppelt hinzugefügt werden. Zusätzlich zum Code Fix muss ein Analyzer entwickelt werden, welcher dieses Problem erkennt und die Entwickler zukünftig warnt, sobald sie String Literale für die Benutzeroberfläche verwenden.
Im Rahmen des zu entwickelnden Prototyps soll zunächst die Verwendung von String Literalen bei den Eigenschaften von UI-Controls erkannt und behoben werden, welche am häufigsten genutzt werden. Diese können aus Tabelle 4.2 entnommen werden.

4.1.3 BX03 - Implementierungen aus der Basisklasse nutzen

Bei einer Enterprise Resource Planing (ERP) Anwendung wie bilettix.net sind sich viele der Benutzeroberflächen sehr ähnlich. Oft unterscheiden sie sich nur darin, welche Ressourcen dargestellt werden. Eine der Ansichten, die in bilettix.net am häufigsten verwendet wird, ist die Listenansicht für unterschiedliche Ressourcen wie z.B. Veranstaltungen oder Erlösarten. Zu jeder einzelnen Listenansicht existiert eine zugehörige Modellklasse. Alle diese Modellklassen erben von der gemeinsamen Basisklasse *ListViewDataModel*. Diese implementiert eine Vielzahl von Methoden, unter anderem zur Initialisierung der einzelnen Elemente der Benutzeroberfläche. Einige dieser Methoden wurden jedoch erst hinzugefügt, nachdem schon eine große Anzahl an Listenansichten fertig implementiert war. Dies hat zur Folge, dass ältere Listenansichten viele, nahezu identische, Codewiederholungen enthalten.

Codewiederholung gilt in allen Programmierparadigmen als großes Problem. Der Autor Robert Martin ging sogar soweit, Codeduplizierung als die Wurzel allen Übels in Software zu bezeichnen.[3] Aus dieser These leitet er das bekannte *DRY (Don't Repeat Yourself)* Prinzip

[3] [Mar09, Seite 128] Don't Repeat Yourself (DRY)

ab, dessen Ziel es ist, Codeduplizierung zu vermeiden. Die Verletzung dieses Prinzips gilt als eigenes Anti-Pattern und wird von Martin Fowler sogar als das 'Nummer eins' Anti-Pattern bezeichnet.[4] Um dieses Problem zu beheben, schlägt Fowler vor, Code, welcher an mehr als zwei Stellen dupliziert ist, in eine Methode zu extrahieren. Wird diese Methode in mehreren unterschiedlichen Klassen aufgerufen, bietet es sich an, die Methode in einer gemeinsamen Basisklasse zu implementieren. Alle anderen Klassen, die Zugriff auf die Methode benötigten, können die Basisklasse beerben und sich die Implementierung so teilen. Der größte Vorteil dieser Vorgehensweise ist, dass es so nur eine Methode gibt, welche gewartet werden muss. Des Weiteren wird die Erweiterbarkeit des Codes deutlich verbessert, da eine einzige Änderung in der Basisimplementierung genügt, um das Verhalten aller erbenden Klassen anzupassen.
Um diese Vorteile auch in bilettix.net zu nutzen, sollen alle Modellklassen für Listenansichten die Methoden der Basisklasse zur Initialisierung der Benutzeroberfläche nutzen. Da bereits sehr viele dieser Modellklassen existieren, wäre es ein erheblicher Aufwand, diese Anpassung manuell vorzunehmen und sicherzustellen, dass dabei nicht versehentlich neue Fehler eingeführt werden. Deshalb soll ein automatischer Code Fix erstellt werden, zu dem ein Analyzer gehört, welcher die Entwickler beim Erstellen neuer Modellklassen darauf hinweist, die Implementierungen der Basisklasse zu nutzen. Die einzelnen Vorkommen sind in der Tabelle 4.3 aufgelistet.

Priorität	**Anforderung**	**Basisimplementierung**	**Beschreibung**
Soll	BX03-01	AddTrainItem	fügt eine weitere Hierarchieebene in der Navigation hinzu
Soll	BX03-02	AddColumnDefinition	fügt eine Spalte mit allen relevanten Metadaten, wie z.B. Titel, zur Listenansicht hinzu
Soll	BX03-03	AddSidebarElement	fügt ein Element, wie z.B. eine Vorschau, in die Sidebar der Listenansicht hinzu

Tabelle 4.3: Anforderungen der Klasse BX03 zur Initialisierung der Benutzeroberfläche in Listenansichten

Eine ERP-Anwendung bietet nicht nur Listenansichten für die vielen unterschiedlichen Ressourcen, sondern häufig auch sogenannte *CRUD*-Operationen. CRUD ist eine Abkürzung und steht für *Create, Read, Update, Delete.* Diese Operationen müssen über die Benutzeroberfläche zugänglich gemacht werden. Die *Read*-Operation wird in bilettix.net häufig

[4] [Fow99, Seite 63] Duplicate Code

Priorität	Anforderung	Standardimplementierung	Beschreibung
Kann	BX03-04	StandardAddButton	Initialisiert einen Button zum Erstellen einer Ressource.
Kann	BX03-05	StandardEditButton	Initialisiert einen Button zum Bearbeiten einer Ressource.
Kann	BX03-06	StandardDeleteButton	Initialisiert einen Button zum Löschen einer Ressource.
Kann	BX03-07	StandardCopyButton	Initialisiert einen Button zum Kopieren einer Ressource.

Tabelle 4.4: Anforderungen der Klasse BX03 zur Initialisierung von Buttons

durch die bereits erwähnten Listenansichten implementiert. Für die anderen drei Operationen bietet die Benutzeroberfläche von bilettix.net jeweils einen Button. Die Icons der Buttons sind dabei in allen Ansichten gleich, um eine kohärente Bedienung der Software zu ermöglichen. Die *Delete*-Operation wird beispielsweise immer durch einen Button mit einem Papierkorb-Icon symbolisiert. Da sich weder das Aussehen noch die Funktion von verschiedenen Buttons des gleichen Typs unterscheiden, existiert für deren Initialisierung eine einzige Funktion, welche immer genutzt werden sollte. Der einzige Unterschied zwischen diesen Buttons ist die Ressource, auf welche sich die Operation bezieht. Deshalb wird bei der Initialisierung des Buttons der Typ der Ressource als Parameter übergeben.

Da die Verwendung der Buttons für die verschiedenen Operationen nicht nur auf Listenansichten beschränkt ist, kommt es sehr häufig vor, dass eine manuelle Initialisierung statt der Standardmethode genutzt wird. Analog zu den vorangegangenen Anforderungen in diesem Abschnitt soll für jede Buttonart ein Analyzer mit zugehörigem Code Fix erstellt werden. Die entsprechenden Anforderungen sind in der Tabelle 4.4 zu finden.

4.2 Nicht-Funktionale Anforderungen

Nicht-funktionale Anforderungen sind alle Qualitäten, welche beschreiben, wie die Software die Leistungen der funktionalen Anforderungen erbringt. Für den zu entwickelnden Prototyp sind Benutzbarkeit und Korrektheit die wichtigsten nicht-funktionalen Anforderungen.

NF-01 Erstellen eines NuGet-Pakets

Ein essenzieller Bestandteil eines jeden modernen Softwareökosystems ist ein Tool, welches das Erstellen und Verwenden von Bibliotheken vereinfacht. Für die .NET Umgebung wird dazu *NuGet* genutzt. NuGet ist ein Paketmanager, dessen Hauptaufgabe es ist, die Abhängigkeiten eins C#-Projekts korrekt aufzulösen. Dazu existiert für jede Abhängigkeit ein Datei

mit der Endung *.nupkg*.[5] Dies ist eine spezielle ZIP-Datei, welche alle benötigten Dateien für das Paket enthält. Dazu zählt unter anderem der bereits kompilierte Quellcode des Pakets. Zusätzlich zu dieser ausführbaren Datei sind auch alle benötigten statischen Dateien, wie z.B. Ressourcen-Dateien, im NuGet-Paket enthalten.
Um die Benutzung des zu entwickelnden Prototyp möglichst einfach zu gestalten, soll für diesen ein NuGet-Paket erstellt werden. Dies bietet den Vorteil, dass das Tool mit nur einem Klick für ein bereits bestehendes C# Projekt installiert werden kann. Des Weiteren ermöglicht die projektbezogene Installation, dass auch die Einstellungen des Tools für jedes Projekt angepasst werden können.

NF-02 Maßnahmen zur Sicherstellung der Korrektheit

Da die Code Fixes des zu entwickelnden Prototyps automatisch Quellcode verändern werden, ist es besonders wichtig, dafür zu sorgen, dass dabei keine neuen Fehler im Quellcode entstehen. Auch wenn es nicht möglich ist, eine absolute Korrektheit des Tools sicherzustellen, muss bei der Entwicklung darauf geachtet werden, dass geeignete Maßnahmen zur Überprüfung der Korrektheit ergriffen werden.

NF-03 Maßnahmen zur Sicherstellung der Erweiterbarkeit

Da es sich bei der Implementierung dieser Arbeit um einen ersten Prototyp handelt, muss dieser möglichst einfach zu erweitern sein. Der Prototyp diagnostiziert und behebt nur eine begrenzte Anzahl an Problemen. Es denkbar, dass in der Zukunft weitere domänenspezifische Probleme identifiziert werden, welche ebenfalls automatisch behoben werden sollen. Des Weiteren sollte es für andere Softwareentwickler möglichst einfach sein, dass Tool selbst zu erweitern.

[5] [Kar+18]'An introduction to NuGet'

Priorität	Anforderung	Kategorie	Beschreibung
Muss	NF-01	Benutzbarkeit	Bereitstellen eines NuGet Paketes
Muss	NF-02	Korrektheit	Maßnahmen zur Überprüfung der Korrektheit
Muss	NF-03	Erweiterbarkeit	Maßnahmen, um Erweiterbarkeit zu ermöglichen
Soll	NF-04	Benutzbarkeit	Bereitstellen einer Visual Studio Erweiterung

Tabelle 4.5: Übersicht der nicht-funktionalen Anforderungen

NF-04 Erstellen einer Visual Studio Erweiterung

Alle Entwickler der bilettix GmbH arbeiten mit der Entwicklungsumgebung Microsoft Visual Studio. Diese bietet extensive Möglichkeiten zur Erweiterung. Um die Benutzbarkeit des zu entwickelnden Prototyps möglichst einfach zu gestalten, sollen diese Möglichkeiten zur Erweiterung der Entwicklungsumgebung genutzt werden. Dazu soll das zu entwickelnde Tool als eigenständige Visual Studio Erweiterung bereitgestellt werden. Dies bietet den Vorteil, dass die Analyzer und Code Fixes auch unabhängig vom aktuellen C#-Projekt zu Verfügung stehen.

5 Implementierung

Dieses Kapitel behandelt die technischen Details und Herausforderungen der Implementierung. Dafür wurde eine Solution angelegt, welche aus drei C#-Projekten besteht.

- **BxAnalyzer** Dies ist das Hauptprojekt, es enthält alle Analyzer und Code Fixes. Außerdem enthält es zusätzliche Funktionalitäten, auf welche die Analyzer und Code Fixes angewiesen sind.
- **BxAnalyzer.Test** Dieses Projekt enthält die Unit Test für das Hauptprojekt.
- **BxAnalyzer.Vsix** Dieses Projekt enthält alle nötigen Details zum Erstellen der Visual Studio Erweiterung.

5.1 Rahmenbedingungen

Aus einigen der funktionalen und nicht-funktionalen Anforderungen gehen bestimmte technische Anforderungen an die Software hervor. Um das zu entwickelnde Tool als Erweiterung für Visual Studio verfügbar zu machen, ist es erforderlich, den *.NET-Standard* als Zielplattform zu wählen. Dies bringt einige Herausforderungen mit sich, da die Zielplattform von bilettix.net das .NET-Framework ist. Der .NET-Standard ist eine formale Spezifikation von .NET-Schnittstellen, welche in allen konkreten Implementierungen vorhanden sind.[1] Da das .NET-Framework den .NET-Standard implementiert, bietet es alle nötigen Schnittstellen. Bilettix.net nutzt jedoch einige exklusive Schnittstellen des .NET-Frameworks, deshalb mussten diese in einer minimalen Form für den Prototyp implementiert werden. Konkrete Beispiele dafür finden sich im Verlauf des folgenden Kapitels.
Der .NET-Standard bietet jedoch auch einige Vorteile. Der größte Vorteils ist, dass die entwickelte Software nicht an ein bestimmtes Betriebssystem gebunden ist. Dies ist möglich, da viele verschiedene Implementierungen des .NET-Standards existieren. Unter Linux und MacOs kann *.NET-Core* statt des .NET-Frameworks verwendet werden.

Außerdem erleichtert die vielseitige Buildkonfiguration des .NET-Standards das Erstellen eines NuGet Pakets (NF-01 Bereitstellen eines NuGet Pakets). Um automatisch bei jedem Buildvorgang ein NuGet Paket zu erstellen, genügen einige wenige Zeilen zur Konfiguration

[1] [Mai+18] .NET Standard Dokumentation

```
<ItemGroup>
  <None Include="$(OutputPath)\$(AssemblyName).dll"
   Pack="true" PackagePath="analyzers/dotnet/cs" Visible="false" />
  <None Update="tools\install.ps1"
   CopyToOutputDirectory="Always" Pack="true" PackagePath="" />
  <None Update="tools\uninstall.ps1"
   CopyToOutputDirectory="Always" Pack="true" PackagePath="" />
  <None Include="Util\bxAnalyzerSettings.json" Pack="true"
   PackagePath="content" Visible="false" />
</ItemGroup>
```

Abbildung 5.1: Konfiguration zum Erstellen eines NuGet Pakets

in der Projektdatei des C#-Projekts. In der Abbildung 5.1 sind alle Befehle gezeigt, die zum Erstellen eine NuGet Pakets benötigt werden. Dazu werden alle benötigten Dateien und deren Zielverzeichnis im späteren Paket definiert. Da es sich bei dem entwickelten Prototyp um einen C# Analyzer handelt, muss dieser im NuGet Paket im Verzeichnis *analyzers/dotnet/cs* platziert werden. Außerdem wird eine Vorlage für die benötigte Konfigurationsdatei im Pfad *content/* platziert. Diese wird während der Installation der Paketes im Wurzelverzeichnis des Projekts installiert. Der Ablauf des Installationsprozesses wird über ein Powershell Skript definiert, welches im sich *tools/* Verzeichnis des Pakets befindet. Dort befindet sich noch ein weiteres Powershell Skript für die Deinstallation des Pakets. Diese beiden Skripte werden automatisch bei der Erstellung eines neuen Roslyn Projekts erzeugt und können ohne Modifikationen genutzt werden, solange die Konventionen zur Paketstruktur eingehalten werden.

5.2 Strukturierung des Entwicklungsprozesses

Um für eine saubere Softwarearchitektur und die Korrektheit der Implementierung zu sorgen, wurde die testgetriebene Entwicklung (engl. Test Driven Development TDD) als Methodik gewählt. Diese Vorgehensweise sieht vor, zu erst einen Unit Test zu programmieren, bevor die spezifizierte Funktionalität implementiert wird. Dadurch wird sichergestellt, dass die Testabdeckung und somit auch die Korrektheit der Implementierung möglichst hoch ist. Außerdem muss dafür gesorgt werden, dass die Implementierung, unabhängig von der realen Laufzeitumgebung, in den Unit Tests getestet werden kann. Dies ist nur möglich, wenn alle Zugriffe auf die Umgebung in der Implementierung abstrahiert werden. Somit wird dieser Teilaspekt einer sauberen Softwarearchitektur, direkt durch die Strukturierung des Entwicklungsprozesses erzwungen.
Da die Analyzer und Code Fix auf die Roslyn Schnittstellen angewiesen sind und diese externe Abhängigkeiten hat, muss deren Verhalten in der Testumgebung nachgestellt werden. Dazu existiert bereits eine kleine Bibliothek mit Testcode, welche von Microsoft mit dem

Roslyn SDK ausgeliefert wird. Diese Bibliothek zum Testen von Analyzern und Code Fixes wurde verwendet, jedoch reichten die Möglichkeiten der Bibliothek nicht für dieses Projekt aus. Daher bestand der erste Schritt darin, die Test Bibliothek zu erweitern, um eine testgetriebene Entwicklung zu ermöglichen.
Wie in Kapitel 3 beschrieben, erstellt Roslyn ein Model des gesamten Programms inklusive aller Abhängigkeiten. Zwei entscheidende Abhängigkeiten eines jeden C# Projekts sind die *mscorlib.dll* und die *system.dll*. Diese beiden Assemblies stellen alle benötigten Abhängigkeiten zur Laufzeitumgebung und zum Beitriebsystem bereit. Im BxAnalyzer Projekt werden dazu die entsprechenden Versionen der Assemblies aus dem .NET-Standard genutzt. Wird nun Roslyn in der Testumgebung gestartet, nutzt Roslyn ebenfalls die Versionen des .NET-Standards. Dies stellt in diesem Fall jedoch ein Problem dar, da die bilettix.net Bibliotheken nur in Verwendung mit dem .NET-Framework funktionieren. Deshalb wurde die Test Bibliothek von Microsoft so erweitert, dass beim Erstellen des Roslyn Kontextes andere Versionen der beiden Assemblies genutzt werden können. Diese müssen als .dll-Datei auf dem Rechner, auf dem die Tests ausgeführt werden sollen, verfügbar sein. Vor Ausführung der Tests müssen die Dateipfade der beiden Dateien in der Klasse *Constants* hinterlegt werden.

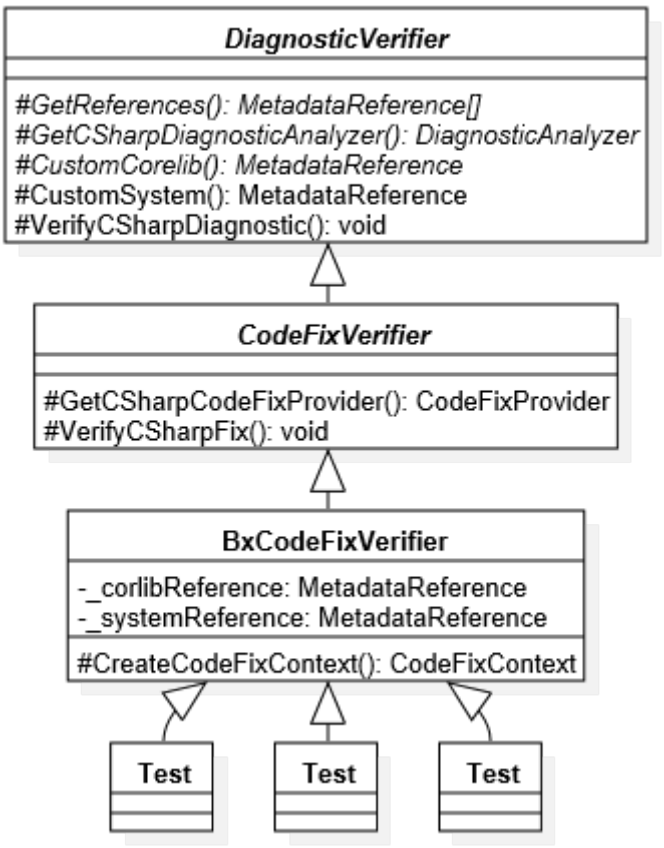

Abbildung 5.2: Vereinfachtes UML Klassendiagramm für die Klasse *BxCodeFixVerifier*

Diese Erweiterung zum Testen der Analyzer und der Code Fixes ist in der Klasse *BxCodeFixVerifier* zu finden. Wie in Abbildung 5.2 zu sehen erbt die Klasse *BxCodeFixVerifier* von der Klasse *CodeFixVerifier*, welche die von Microsoft implementierten Standardfunktionen zum Testen von Analyzern und Code Fixes bereitstellt. Die Klasse *BxCodeFixVerifier* enthält noch eine weitere entscheidende Erweiterung. Da Roslyn, wie bereits erläutert, zur Analyse Zugriff auf die Assemblies aller Abhängigkeiten benötigt und die entwickelten Analyzer bilettix.net spezifischen Quellcode untersuchen, muss in der Testumgebung Zugriff auf die zugehörigen Assemblies bestehen. Dazu wurden die Methoden zur Erstellung des Roslyn Kontext in der Klasse *BxCodeFixVerifier* so erweitert, dass alle *.dll* Dateien aus dem Ordner *./Assemblies* geladen und bereitgestellt werden.
Damit die einzelnen Test Klassen Zugriff darauf erhalten, erben sie alle von der Klasse *BxCodeFixVerifier*. Um zu testen, ob ein Analyzer die richtigen Diagnosen erstellt, wird der zu untersuchende Quelltext als String in der Testmethode definiert. Zusätzlich wird das erwartete Diagnoseresultat definiert. Dazu gehören Details, wie z.B. die Id des gefundenen Problems und die genaue Position im Quelltext. Dies dient als Parameter für die Methode *VerifyCSharpDiagnostic*, welche von der Basisklasse bereitgestellt wird. Diese Methode

initialisiert die Testumgebung, führt den Analyzer unter Realbedingungen aus und prüft, ob das Resultat des Analyzers mit der erwarteten Diagnose übereinstimmt. Analog dazu existiert eine weitere Methode *VerifyCSharpFix* zum Testen von Code Fixes. Diese wird ebenfalls von der Basisklasse bereitgestellt, jedoch erhält sie zwei Strings als Eingabe. Der erste der beiden Eingabewerte ist C# Quellcode, welcher analysiert wird. Wird bei dieser Analyse ein Problem diagnostiziert, wird der zu testende Code Fix darauf angewandt. Der zweite Parameter der Methode *VerifyCSharpFix* ist das erwartete Resultat nach Anwendung des Code Fixes. Diese beiden Strings werden nach der Ausführung des Code Fixes verglichen. Ein Problem dieser Testmethode ist, dass beim Vergleich der Strings auch Zeichen verglichen werden, welche für die Semantik des Programms irrelevant sind. Außerdem werden auch nicht sichtbare Zeichen verglichen. Dies kann dazu führen, dass zwei Strings die für einen Menschen gleich aussehen und das selbe Programm darstellen, einen Test trotzdem nicht bestehen. Dieses Problem kann jedoch sehr einfach vermieden werden, indem vor dem Vergleichen der Strings, alle Whitespaces entfernt werden.
Nachdem die Testumgebung vollständig eingerichtet wurde, konnte mit der Implementierung der Analyzer und Code Fixes begonnen werden.

5.3 Analyse- und Code Fix Strategien

Im folgenden Abschnitt werde die Algorithmen ausgewählter Analyzer und Code Fixes erläutert. Der Fokus liegt dabei auf der Interaktion mit den Roslyn Schnittstellen, den Herausforderungen bei der Implementierung und deren Lösungen. Da der Umfang der Implementierung zu groß ist, um alle Analyzer und Code Fixes zu betrachten, werden nur die ausgewählten interessanten Analyzer und Code Fixes behandelt.

5.3.1 Verwendung der Roslyn APIs zur Behebung des Problems BX0303

Als erstes soll die Verwendung der unterschiedlichen Programmierschnittstellen von Roslyn zu Analyse und Modifikation von Quelltext gezeigt werden. Dazu wird exemplarisch der Code Fix für das Problem BX0303 'Erstellen von Sidebar Elementen' betrachtet, da dieser noch relativ simpel ist, aber trotzdem mehrere interessante Aspekte der Roslyn Schnittstellen nutzt. Ziel dieses Fixes ist es, dass neue Sidebar Elemente in Listenansichten nur noch durch die Verwendung der Methode *'AddSidebar'* aus der Basisklasse erstellt werden.
Abbildung 5.3 zeigt ein vereinfachtes Beispiel des Problems. In den Zeilen 5-7 wird eine neue Instanz der Klasse *SidebarElement* erstellt und initialisiert. Dazu wird eine spezielle Syntax verwendet, welche es erlaubt, die Felder eines neu erstellten Objekts direkt mit passenden Werten zu initialisieren. In Zeile 7 wird dem Feld *Title* so der Wert *'Title'* zugewiesen. Das Initialisieren von Feldern auf diese Art ist, im Vergleich zu den Parametern des Konstruktors, optional. In Abbildung 5.3 wird dem Konstruktor der String *'ID'* übergeben, welcher zur

Identifikation des Sidebar Elements dient. Da jedes Sidebar Element eine ID benötigt, wird diese als Parameter im Konstruktor übergeben.

```
1  public class ListForSidebarTest : ListViewDataModel
2  {
3    public void Initialize()
4    {
5      Sidebar.Add(new SidebarElement("ID")
6      {
7        Title = "Title"
8      });
9    }
10 }
```

Abbildung 5.3: Hinzufügen eines Sidebar Elements ohne Verwendung der Methode aus der Basisklasse

Abbildung 5.4 zeigt einen vereinfachten Syntaxbaum, welche die Objekterstellung aus den Zeilen 5-7 aus Abbildung 5.3 repräsentiert. In diesem Syntaxbaum wurden die Trivia Knoten entfernt, da sie für die Funktion des Code Fixes irrelevant sind.

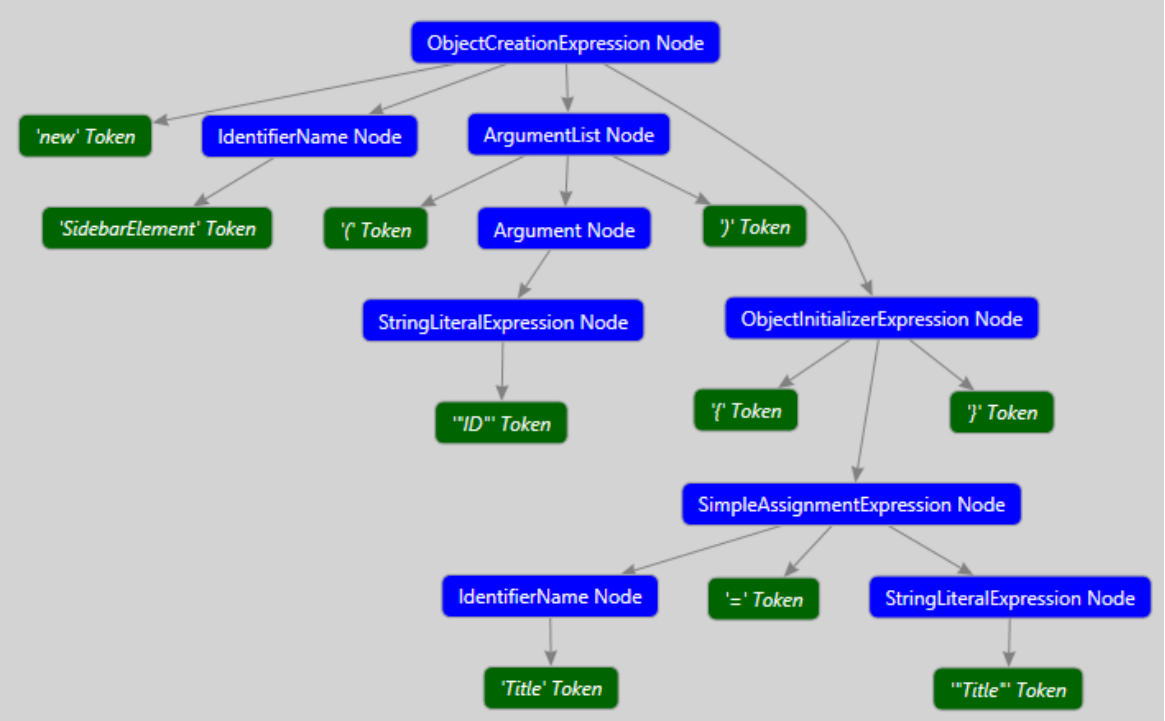

Abbildung 5.4: Syntaxbaum für die Instanziierung und Initialisierung eines neuen Sidebar Elements

Die Blätter des Baums sind Tokens, deren Abfolge den zugehörigen Quelltext ergibt. Die Knoten des Baums repräsentieren die unterschiedlichen syntaktischen Konstrukte der Programmiersprache. Um den Quellcode zu transformieren, muss eine Syntaxbaum für den Aufruf der Methode aus der Basisklasse erstellt werden. Dieser Methodenaufruf enthält die optionalen Zuweisungen aus der Objektinitialisierung als benannte Parameter. Damit diese Transformation erfolgen kann, müssen alle relevanten Informationen aus dem Syntaxbaum in Abbildung 5.4 extrahiert werden. Diese Informationen müssen anschließend korrekt in einen neuen Syntaxbaum eingefügt werden.

```
1  private ArgumentListSyntax GetOptionalArguments(
2  ObjectCreationExpressionSyntax objectCreationSyntax)
3  {
4    ArgumentListSyntax optionalArguments =
5      SyntaxFactory.ArgumentList();
6
7    foreach (string optionalArgumentName in _OptionalArgumentNames)
8    {
9      if (objectCreationSyntax.Initializer.ChildNodes()
10     .OfType<AssignmentExpressionSyntax>()
11     .SingleOrDefault(a =>
12       a.Left.GetText().ToString().Trim() == optionalArgumentName)
13     is AssignmentExpressionSyntax assignment)
14     {
15       string argumentName =
16       optionalArgumentName.Substring(0, 1).ToLower()
17         + optionalArgumentName.Substring(1);
18
19       optionalArguments = optionalArguments.AddArguments(
20        SyntaxFactory.Argument(assignment.Right)
21        .WithNameColon(SyntaxFactory.NameColon(argumentName)));
22     }
23   }
24
25   return optionalArguments;
26 }
```

Abbildung 5.5: Erstellen einer benannten Parameterliste für optionale Parameter aus den Parametern der Objektinitialisierung

Abbildung 5.5 zeigt Quellcode, welcher alle optionalen Parameter aus der Objektinitialisierung extrahiert und daraus eine neue Liste von Argumenten erstellt, welche dem Methodenaufruf übergeben werden kann. Als Parameter bekommt die abgebildete Methode *GetOptionalArguments* eine Instanz der Klasse *ObjectCreationExpressionSyntax*. Diese repräsentiert einen Syntaxbaum für die Instanziierung eines neuen Objekts. Die Abbildung 5.4 zeigt einen solchen Syntaxbaum. In der Bedingung des *if*-Ausdrucks in den Zeilen 9-13 des Quelltextes aus Abbildung 5.5 wird auf den Teilbaum der Objektinitialisierung zugegrif-

fen. Es werden alle Kindknoten des Typs *AssignmentExpressionSyntax* herausgefiltert. Diese repräsentierten die Zuweisungen von Werten zu den Feldern des neu erstellten Objekts. Zuletzt wird geprüft, ob eine Zuweisung für einen bestimmten Parameter mit dem gegebenen Namen existiert. Ist dies der Fall, so wird für diesen Parameter ein neues Element zur Liste der optionalen Argumente hinzugefügt. Der entsprechende Quellcode ist in den Zeilen 19-21 zu finden. Dazu wird mit Hilfe der Methoden der Klasse *SyntaxFactory* ein neuer Teilsyntaxbaum erstellt. Dieser enthält die Syntax für einen optionalen Parameter und den zugehörigen Wert aus der ursprünglichen Objektinitialisierung. In der Programmiersprache C# folgen benannte Parameter der Form: *Name: Wert*. Der Name des Parameters wird dem zu erstellenden Syntaxbaum in Form eines *NameColon* Knotens hinzugefügt.
Ein weiterer nennenswerter Aspekt, beim Erstellen der Liste der Argumente, ist in Zeile 19 zu finden. Dort kommt die in Abschnitt 3.3 beschriebene Unveränderbarkeit der Datenstrukturen zum Vorschein. Nachdem hinzufügen eines neuen Arguments zur Liste, muss eine erneute Zuweisung erfolgen, da das neue Argument zu einer neuen Liste hinzugefügt wird. Würde diese Zuweisung nicht erfolgen, wäre die Liste der Argumente nach dem Durchlaufen der Schleife immer leer. Der Grund dafür ist, dass die von der Variablen *optionalArguments* referenzierte Instanz der Klasse *ArgumentListSyntax* nicht verändert wurde.

```
1 public class ListForSidebarTest : ListViewDataModel
2 {
3   public void Initialize()
4   {
5     AddSidebar("ID", title: "Title");
6   }
7 }
```

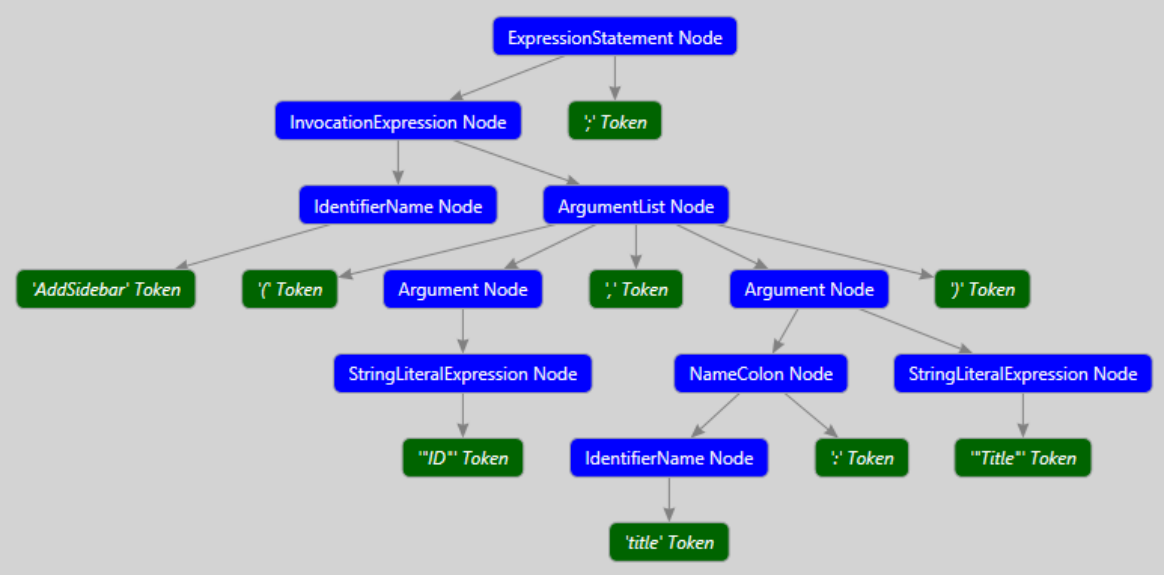

Abbildung 5.6: Syntaxbaum für die Instanziierung und Initialisierung eines neuen Sidebar Elements

Die so erstellte Liste von optionalen Argumenten wird von der Methode zurückgegeben und im aufrufenden Kontext dazu verwendet, den Syntaxbaum für den Aufruf der Methode aus der Basisklasse zu konstruieren. Nachdem der neue Syntaxbaum konstruiert wurde, ersetzt der Code Fix den ursprünglichen Syntaxbaum mit dem neu konstruierten. Abbildung 5.6 zeigt den neuen Quellcode und den Syntaxbaum für den Aufruf der Methode aus der Basisklasse. Die ID des Sidebar Elements ist dabei der erste Parameter. Da es sich bei der ID um einen obligatorischen Parameter handelt, ist dieser nicht benannt. Der optionale Titel, welcher ursprünglich durch die Objektinitialisierung zugewiesen wurde, ist jetzt ein benannter Parameter im Methodenaufruf.

5.3.2 Analyzer aus der Diagnoseklasse BX02

Das zugrundeliegende Problem für alle Analyzer aus dieser Diagnoseklasse ist die Vermeidung von String Literalen beim Erstellen von UI-Controls. Um diese Probleme automatisch im Quellcode zu identifizieren, muss als erstes ein Kriterium definiert werden, um zu erken-

nen, ob ein Attribut dazu verwendet wird, ein UI-Control zu erstellen. Solch ein Kriterium zu finden war einfach, da alle Attribute, welche zur Generierung von UI-Controls genutzt werden, von der gemeinsamen Basisklasse *BaseInputAttribute* erben. Da Roslyn extensive Möglichkeiten zu semantischen Analyse und zur Typprüfung bietet, ist es kein Problem, im Rahmen der Analyse die Vererbungshierarchie einer Klasse zu überprüfen. Der zugehörige Quellcode lässt sich in der Klasse *SymbolExtension* finden.
Um die Analyse zu starten, muss zunächst ein geeigneter Einstiegspunkt definiert werden. Dazu muss wie in Abschnitt 3.4.2 'Diagnostic Analyzer' bei der Initialisierung des Analyzers eine Callback Funktion im Analysekontext registriert werden. Als Einstiegspunkt für die Callback Funktion dieses Analyzers bieten sich alle Aktionen auf Syntax Knoten an, welche bei der Deklaration einer Eigenschaft involviert sind. Der entsprechende Quellcode ist in Abbildung 5.7 zu sehen.

```
1  public override void Initialize(AnalysisContext context)
2  {
3    context.RegisterSyntaxNodeAction(
4      AnalyzePropertyDeclaration,
5      SyntaxKind.PropertyDeclaration);
6  }
7
8  private void AnalyzePropertyDeclaration
9  (SyntaxNodeAnalysisContext context)
10 {
11   //Algorithmus für die Analyse
12 }
```

Abbildung 5.7: Quellcode zum Registrieren der Rückruffunktion zur Analyse aus der Klasse *ControlArgumentAnalyzer*

Im Teilsyntaxbaum für die Deklaration der Eigenschaft kann die eigentliche Analyse durchgeführt werden, da er alle benötigten Informationen enthält. Der Algorithmus für die Analyse der Deklaration ist eher simpel. Der entsprechende Programmablaufplan ist in Abbildung 5.8 dargestellt.
Bei der Analyse wird zu erst überprüft, ob das Attribut von der Klasse *BaseInputAttribute* erbt. Ist dies nicht der Fall, ist die Eigenschaft für die Generierung der UI-Controls irrelevant und die Analyse kann beendet werden. Ist das Attribut jedoch von Interesse, so wird überprüft, ob es einen Parameter enthält, welcher Text in der Benutzeroberfläche definiert. Ein Beispiel für solch einen Parameter ist *ToolTip*. Ist der Parameter in der Liste enthalten, so wird überprüft, ob dem Parameter ein String Literal zugewiesen wird. Ist dies der Fall erstellt der Analyzer eine entsprechende Diagnose.

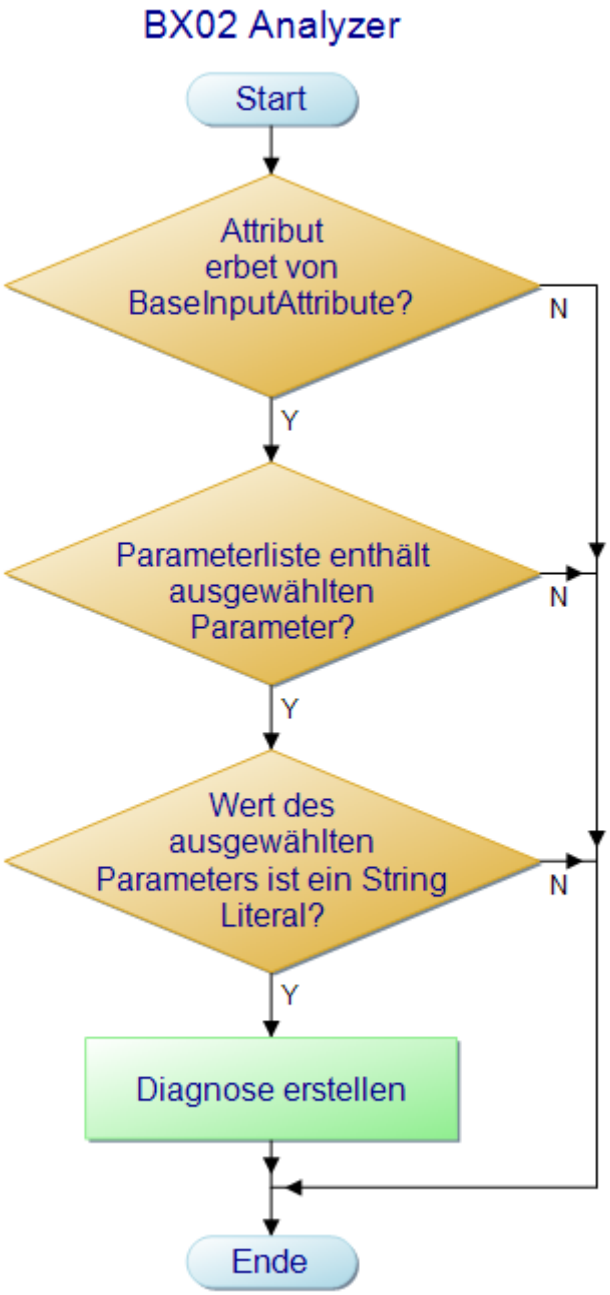

Abbildung 5.8: Vereinfachtes UML Klassendiagramm für die Klasse *BxCodeFixVerifier*

Da sich die Struktur des Syntaxbaums für die Analyse unterschiedlicher Parameter nicht verändert, wurde beim Entwurf des Algorithmus darauf geachtet, dieses so generische wie möglich zu gestalten. Diese Entscheidung zielt darauf ab, den Algorithmus einmalig zu implementieren und dann für verschiedene Parameter dieselbe Implementierung zu nutzen. Der

zu untersuchende Parameter wird im Programmablaufplan als *ausgewählter Parameter* bezeichnet, sein Wert ist nicht hart in den Algorithmus kodiert. Dies bietet den Vorteil, dass mit einer einzigen Implementierung beliebig viele unterschiedliche Parameter untersucht werden können, so lange die Struktur sich sich nicht verändert. So kann dieser Analyzer später einfach erweitert werden.

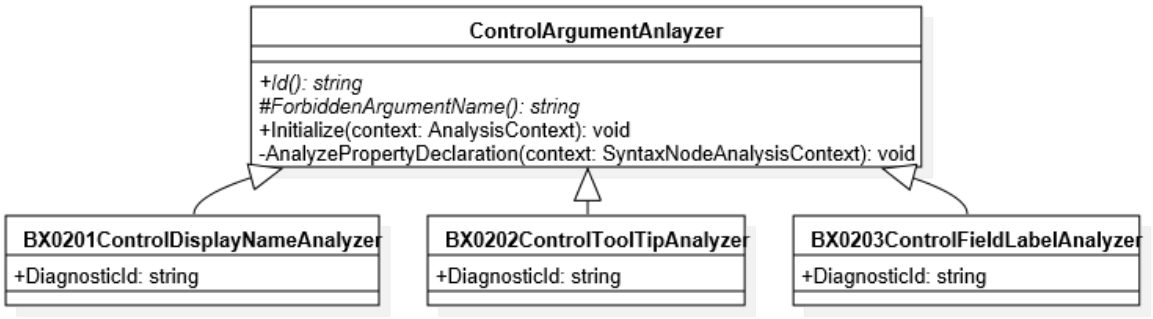

Abbildung 5.9: Vereinfachtes UML Klassendiagramm für die Vererbungshierarchie der Analyzer BX0201-0203

Diese Möglichkeit wurde zur Implementierung der Anforderungen BX0201-0203 genutzt. Die abstrakte Basisklasse *ControlArgumentAnalyzer* enthält die Implementierung des Analysealgorithmus und eine abstrakte Getter-Methode, welche den Namen des zu untersuchenden Parameters liefert. Wie in Abbildung 5.9 zu sehen ist, erben die konkreten Klassen, zur Analyse eines ausgewählten Parameters, von der abstrakten Basisklasse und müssen nur noch die Getter-Methode für den Parameternamen implementieren. Des Weiteren besitzt jede der Klassen ein Attribut mit der eindeutigen Diagnose Id. Diese wird von Roslyn dazu genutzt, um der erstellten Diagnose einen entsprechenden Code Fix zuzuordnen.

5.3.3 Code Fixes aus der Diagnoseklasse BX02

Um das Problem der String Literale in der Benutzeroberfläche zu beheben, wurde bereits im Kapitel 4 ein Lösungsvorschlag vorgestellt. Dieser sieht vor, für jedes String Literal einen Eintrag in einer speziellen Ressourcen Datei zu erstellen und das Literal im Quellcode durch eine Referenz auf die Ressourcen Datei zu ersetzen. Dieser Ansatz wurde genutzt, da bei der Weiterentwicklung von bilettix.net bereits damit begonnen wurden, String Literale auf diese Weise aus dem Quellcode zu eliminieren. Dazu wurde, wie bereits erläutert, eine .NET-Framework Schnittstelle genutzt, welche Zugriff auf RESX-Dateien bietet.
Bei der Entwicklung der Code Fixes konnte diese Schnittstelle leider nicht genutzt werden, da die Code Fixes dem .NET-Standard folgen und dieser, mit seinem kleineren Funktionsumfang, keine Schnittstelle für die Interaktion mit RESX-Dateien bietet. Deshalb wurde die .NET-Standard Schnittstelle zur Interaktion mit XML-Dateien genutzt, um die benötigten

Funktionalitäten zu implementieren. Dabei wurde darauf geachtet, alle Abhängigkeiten in geeignete Interfaces zu extrahieren. Objektinstanzen, welche diese Interfaces implementieren, werden den Code Fixes als Parameter übergeben. Dieses Einbringen der Abhängigkeiten (engl. *Dependency Injection*), ermöglicht es, die Code Fixes für Erweiterungen offen zu halten, da Abhängigkeiten zu konkreten Klassen vermieden werden. Des Weiteren ermöglicht diese Architektur es, die Abhängigkeiten der Code Fixes bei der Durchführung von Test durch Attrappen (engl. *Mocks*) zu ersetzen, welche eine korrekte Implementierung vortäuschen. Somit kann die Logik der Code Fixes unabhängig von den Schnittstellen zur Umgebung getestet werden. Die konkreten Implementierungen können in separaten Unit Tests getestet werden.

Zugriff auf RESX-Dateien

Um den Code Fixes einen lose gekoppelten Zugriff auf Ressourcen Dateien zu ermöglichen, wurde das Interface *IResourceManager* deklariert. Die wichtigsten Methoden, welche das Interface vorschreibt, dienen dazu zu prüfen, ob eine Ressource bereits existiert und dazu eine Ressource hinzuzufügen. Die Verwendung eines Interfaces an dieser Stelle ermöglicht es, in der Zukunft weitere Klassen für den Zugriff auf andere Ressourcen Formate hinzuzufügen. In Abbildung 5.10 ist zu sehen, dass die Klasse *ControlArgumentResourceFix* nur von dem Interface *IResourceManager*, nicht aber von der konkreten Realisierung, abhängig ist. Das Interface wird von der Klasse *ResxManager* implementiert, welche Zugriff auf RESX-Dateien bietet. Um diese Aufgabe zu erfüllen, benötigt die Klasse *ResxManager* Zugriff auf XML-Dateien. Diese Beziehung wird im UML Klassendiagramm in Abbildung 5.10 durch die Komposition zwischen der Klasse *ResxManager* und dem Interface *IXmlDocumentLoader* repräsentiert.

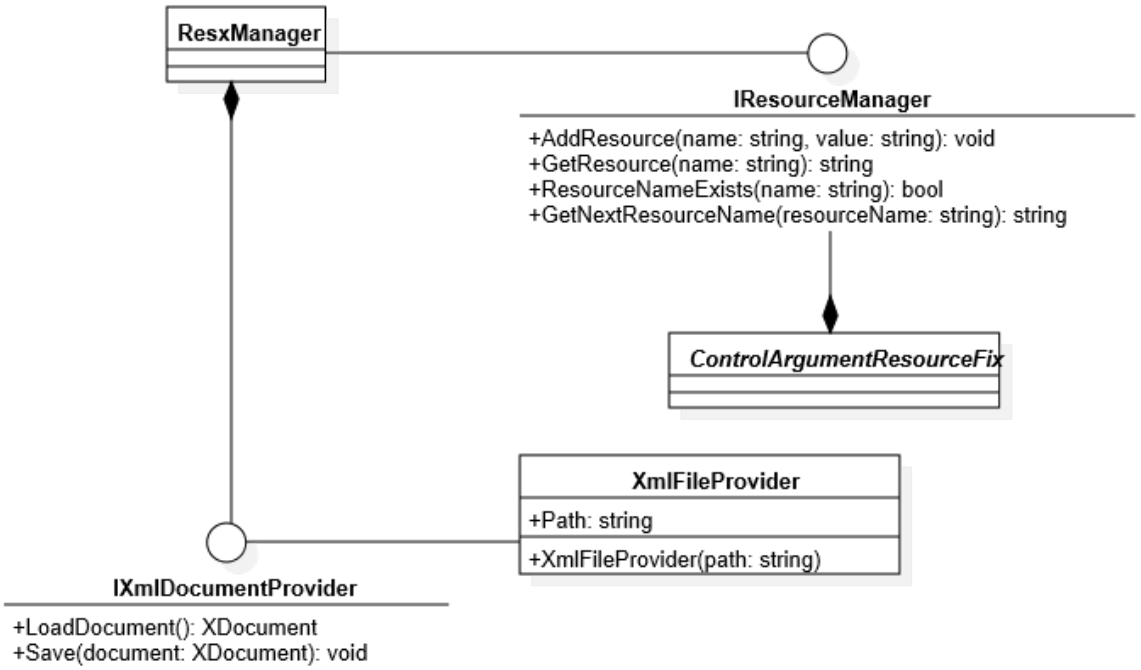

Abbildung 5.10: UML Klassendiagramm der für den Ressourcenzugriff zuständigen Klassen.

Durch die Verwendung eines Interfaces für den Zugriff auf XML-Dateien, besitzt die Klasse *ResxManager* keine Abhängigkeiten zu den XML-Schnittstellen des .NET-Standards. Dies bietet den Vorteil, dass bei einer zukünftigen Änderung dieser Schnittstellen, nur die Klasse angepasst werden muss, welche den konkreten Zugriff auf XML-Dateien bereitstellt. Die Klasse *ResxManager* ist von eventuellen Änderungen nicht betroffen.
Das Interface *IXmlDocumentLoader* wird von der Klasse *XmlDocumentLoader* implementiert, welche eine Wrapper-Klassen um die XML-Schnittstellen des .NET-Standards ist.
Um auf eine XML-Datei zuzugreifen, wird der entsprechende Dateipfad benötigt. Da dieser auf jedem System unterschiedlich sein kann, musste eine Möglichkeit geschaffen werden, diese Einstellung zu konfigurieren. Dazu wurde die Klasse *BxAnalyzerSettings* erstellt. Diese enthält für jede Einstellung eine Eigenschaft. Zum jetzigen Zeitpunkt existiert nur die Eigenschaft *ControlLabelResourcePath*, welche den Pfad zur entsprechenden Ressourcen Datei enthält. Eine Instanz der Klasse *BxAnalyzerSettings* kann aus einer JSON-Datei deserialisiert werden. Dies ermöglicht es, die Einstellungen zu persistieren. Da der .NET-Standard keine Schnittstelle zur Interkation mit JSON-Dateien bietet, wurde die Bibliothek *LightJson*[2] genutzt. Diese ist auf GitHub unter der MIT-Lizenz veröffentlicht.
Um Abhängigkeiten auf Schnittstellen für den Zugriff auf das Dateisystem zu vermeiden, wurde für den Zugriff auf die RESX-Dateien die Schnittstellen des Roslyn Workspace Services genutzt. Dieser bietet während der Analyse Zugriff auf alle Dateien, die zum untersuchten

[2]https://github.com/MarcosLopezC/LightJson

C#-Projekt gehören. Um auf diese Weise Zugriff auf eine RESX-Datei zu erhalten, muss diese zum C#-Projekt hinzugefügt und ihr relativer Dateipfad zum Projektverzeichnis in der Datei *bxAnalyzerSettings.json* gespeichert werden. Eine Vorlage dieser Datei wird bei der Installation des NuGet-Pakets automatisch angelegt. Da es bei der Installation von Paketen leider nicht möglich ist, diese Datei zum C#-Projekt hinzuzufügen, muss dies einmalig manuell erledigt werden. Dazu muss die Projektdatei um einen Eintrag vom Typ *'AdditionalFiles'* ergänzt werden, welcher den Dateipfad zu der entsprechenden Datei enthält. Bei der Ausführung der entsprechenden Code Fixes, wird diese Datei dazu verwendet, eine Instanz der Klasse *BxAnalyzerSettings* zu erstellen.

Code Preview Action

Eine weitere Herausforderung bei der Implementierung der Code Fixes, welche auf Grund des Zugriffs auf die RESX-Dateien entstand, war die Implementierung der Vorschau für die Code Fixes. Wie bereits in Kapitel 3 in der Abbildung 3.2 gezeigt, erstellt Visual Studio eine Vorschau des transformierten Quellcodes, bevor die *Code Action* eines Code Fixes ausgeführt wird. Bei der Roslyn Standardimplementierung der abstrakten Klasse *CodeAction* wird dazu die Aktion ausgeführt und ihr Ergebnis im Vorschaufenster angezeigt. Dies ist für die in diesem Abschnitt beschriebenen Code Fixes jedoch nicht praktikabel, da auf diese Weise beim Erstellen der Vorschau bereits eine neue Ressource zur Ressourcen Datei hinzugefügt werden würde. Entschiede sich der Nutzer nach dem Erstellen der Vorschau, den Code Fix nicht anzuwenden, so würde trotzdem ein neuer Eintrag in der Ressourcen Datei erstellt werden.
Um dieses Problem zu lösen, wurde die Klasse *PreviewCodeAction* implementiert. Diese erbt von der abstrakten Klasse *CodeAction*. Im Vergleich zur Standardimplementierung bietet die Klasse *PreviewCodeAction* den Vorteil, dass bei der Instanziierung zwei unterschiedliche Funktionen zum Erstellen der Vorschau und zum Ausführen der eigentlichen Operation übergeben werden können. Um diesen Vorteil zu nutzen, wurden in der Klasse *ControlArgumentResourceFix* zwei Methoden implementiert, welche dieselbe syntaktische Transformation durchführen. Diese beiden Methoden unterscheiden sich darin, dass eine der beiden Methoden eine Ressource zur Ressourcen Datei hinzufügt und die andere nur die syntaktische Transformation ausführt. Die Implementierung der letzteren Methode wurde zusätzlich dazu genutzt, den Code Fix so zu erweitern, dass bereits existierende Ressourcen genutzt werden können.

6 Evaluation

Das folgende Kapitel soll einen kurzen Überblick darüber geben, in welchen Umfang die der entwickelte Prototyp die Anforderungen aus Kapitel 4 erfüllt.

6.1 Funktionale Anforderungen

Die Evaluation der funktionalen Anforderungen gestaltet sich kurz, da alle Anforderungen erfüllt werden konnten. Zu jeder Anforderung existieren sowohl ein Analyzer, als auch ein Code Fix. Es ist ein wenig überraschend, dass in so kurzer Zeit alle Anforderungen erfüllt werden konnten. Der Grund dafür ist, dass sich die Verwendung der von Roslyn gebotenen Möglichkeiten, nach einer kurzen Einarbeitungszeit, einfacher darstellte, als ursprünglich gedacht.

6.2 Nicht-Funktionale Anforderungen

Die Evaluation der nicht-funktionalen Anforderungen ist etwas ausführlicher, da es sich bei diesen um Qualitätseigenschaften handelt, welche häufig einen gewissen Spielraum haben.

NF-01 Erstellen eines NuGet-Pakets

Diese Anforderung wurde erfüllt, wenn auch mit einigen Einschränkungen. Das Erstellen und installieren des NuGet-Pakets funktioniert, aus der Benutzerperspektive, problemlos. Allerdings gab es im Rahmen der Entwicklung das Problem, dass zur Laufzeit das Laden von Abhängigkeiten nicht möglich war, falls diese durch den NuGet-Paketmanager installiert wurden. Die Abhängigkeiten wurden sowohl korrekt erkannt, als auch installiert, konnten aber zur Laufzeit aus unerfindlichen Gründen nicht geladen werden. Dies führte dazu, dass das Programm, ohne eine aussagekräftige Fehlermeldung, abstürzte, sobald Programmcode aus einer verwendeten Bibliotheken ausgeführt werden sollte.
Da der entwickelte Prototyp nur eine Abhängigkeit besitzt, konnte diese Problem gelöst werden, indem der Quellcode der Bibliothek *LightJSON* direkt zum C#-Projekt hinzugefügt wurde. Dadurch wird der Quellcode der Bibliothek beim Kompilieren des Projekts ein Teil des Programms. Dieser Lösungsansatz hat leider einen entscheidenden Nachteil. Er ist nur praktikabel, falls der Quellcode der verwendeten Bibliothek quelloffen ist. Dies ist bei der

verwendeten Bibliothek 'LightJSON'[1] der Fall, da diese unter der MIT-Lizenz veröffentlicht ist.

NF-02 Maßnahmen zur Sicherstellung der Korrektheit

Die erste Maßnahme zur Sicherstellung der Korrektheit des Prototyps war, die Herangehensweise der testgetriebenen Entwicklung wie in Abschnitt 5.2 beschrieben. So wurden während der Entwicklung über 50 einzelne Testfälle erstellt, welche unterschiedliche Funktionalitäten des Prototyps testen. Diese 50 Testfälle erreichen eine Testabdeckung von ca. 80%. Einige Methoden des Prototyps konnten, auf Grund externer Umstände, leider nicht getestet werden. Bei anderen Methoden ist es schlicht nicht sinnvoll diese zu testen. So werden z.B. Getter- und Setter-Methoden für Eigenschaften automatisch vom Compiler generiert, weshalb diese beim Testen außer Acht gelassen werden können.
Der Nutzen dieser umfangreichen Tests konnte bei der ersten Anwendung des Tools direkt festgestellt werden. Initial wurde mit Hilfe des Prototyps der gesamte Quellcode von bilettix.net nach Problemen der Kategorie BX02 (String Literale im Quellcode der Benutzeroberfläche) gesucht. Das Tool fand ca. 2'200 Probleme, welche fast alle automatische behoben werden konnten. Bei ca. 40 Problemen musste manuell nachgebessert werden, da die Benennung einiger Vorhandener Ressourcen nicht dem Schema entsprach, welches das Tool erwartet. Dies entspricht einer Fehlerquote von unter 2%.
In Anbetracht der hohen Testabdeckung und der geringen Fehlerquote gilt diese Anforderung als erfüllt.

NF-03 Maßnahmen zur Sicherstellung der Erweiterbarkeit

Bei der Implementierung des Prototyps wurde darauf geachtet, harte Abhängigkeiten auf die Umgebung zu vermeiden. So wurde der Zugriff auf Ressourcen- und XML-Dateien, wie in Abschnitt 5.3.3, mit Hilfe von Interfaces abstrahiert. Dies erleichtert eine spätere Erweiterung der Implementierung, da für die Nutzung anderer Formate für Ressourcen-Dateien, lediglich neue Klassen hinzugefügt werden müssen, welche die bereits vorhandenen Interfaces realisieren.
Zusätzlich dazu wurde bei der Implementierung der Analyzer und Code Fixes darauf geachtet, die verwendeten Algorithmen möglichst generisch zu gestalteten und diese in einer Basisklasse zu kapseln. Dieses Vorgehen ermöglicht es, sehr einfach weitere Analyzer und Code Fixes für ähnliche Probleme zu erstellen.
Unter Berücksichtigung dieser Aspekte gilt die Anforderung als erfüllt.

[1] https://github.com/MarcosLopezC/LightJson

NF-04 Erstellen einer Visual Studio Erweiterung

Um eine eigenständige Erweiterung für Visual Studio zu erstellen, wurde der Solution ein weiteres C#-Projekt 'BxAnalyzer.Vsix' hinzugefügt. Dieses besitzt eine Abhängigkeit auf das BxAnalyzer Projekt. Es enthält alle benötigten Konfigurationen, um die Visual Studio Erweiterung zu erstellen. Zu dieser Erweiterung gehört ein Installationsprogramm, welches es erlaubt, die Erweiterung mit wenigen Klicks zu installieren.
Diese Anforderung ist somit erfüllt.

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

Im Rahmen dieser Arbeit sollte untersucht werden, ob es möglich ist ein automatisiertes Werkzeug, zur Erkennung und Behebung von domänenspezifischen Problemen in Quellcode, zu implementieren. Da der zu untersuchende Quellcode in der Programmiersprache C# vorliegt, wurde für die Realisierung des Prototyps die Roslyn Compiler Plattform zu Hilfe genommen.
Um verstehen zu können, wie die Roslyn Compiler Plattform funktioniert, wurden in Kapitel 2 die Struktur und Funktionsweise eines Compilers im Allgemeinen beschrieben.
In Kapitel 3 wurde der Aufbau der Roslyn Compiler Plattform beschrieben. Zusätzlich dazu würde einige Besonderheiten der Implementierung der Roslyn Plattform erklärt. Diese Informationen sind wichtig, um die von Roslyn gebotenen Möglichkeiten zur Analyse und Modifikation von Quellcode zu verstehen.
Bevor domänenspezifische Probleme automatisch erkannt un behoben werden können, mussten dieser einmalig manuell identifiziert werden. Kapitel 4 liefert eine Übersicht zu allen identifizierten Problemen. Zu jedem Problem gehört eine kurze Begründung und ein möglicher Lösungsvorschlag. Das Beheben eines jeden Problems stellt jeweils eine eigne funktionale Anforderung dar. Zusätzlich dazu werden in Kapitel 4 alle nicht-funktionalen Anforderungen aufgeführt.
In Kapitel 5 werden zunächst die Rahmenbedienungen der Implementierung und die Strukturierung des Entwicklungsprozesses beschrieben. Anschließend werden interessante Aspekte einiger ausgewählter Analyzer und Code Fixes beschrieben. Der Fokus liegt dabei auf der Verwendung der Schnittstellen der Roslyn Plattform und der Softwarearchitektur.
Abschließend wird in Kapitel 6 eine Evaluation der entstandenen Implementierungen vorgenommen. Dazu wurden die in Kapitel 4 erstellten Anforderungen mit der Funktionalität des implementierten Prototyps abgeglichen.

7.2 Ausblick

Diese Arbeit hat gezeigt, dass es heutzutage mit verhältnismäßig geringen Aufwand möglich ist, maßgeschneiderte Werkzeuge zur Unterstützung der Softwareentwickler im Entwicklungsprozess zu erstellen. Diese Möglichkeit soll in Zukunft von den Entwicklern der bilettix GmbH genutzt werden. Dazu wurden nach Abschluss der Implementierung des Prototyps bereits weitere Probleme im Quellcode identifiziert. Der Prototyp soll in naher Zukunft so erweitert werde, dass diese Probleme ebenfalls automatisch erkannt und behoben werden können.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Aho07] Alfred V. Aho. *Compilers: Principles, techniques, & tools*. 2nd ed. Boston: Pearson/Addison Wesley, 2007. ISBN: 0321486811.

[App14] Apple. *Apple sslKeyExchange.c - goto fail*. 2014. URL: opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c (visited on 11/01/2018).

[Cho56] N. Chomsky. "Three models for the description of language". In: *IEEE Transactions on Information Theory* 2.3 (1956), pp. 113–124. ISSN: 0018-9448. DOI: 10.1109/TIT.1956.1056813.

[Cur12] Benjamin Curry. "Deploying Web Applications in Enterprise Scenarios". In: (2012). URL: https://docs.microsoft.com/en-us/aspnet/web-forms/overview/deployment/deploying-web-applications-in-enterprise-scenarios/deploying-web-applications-in-enterprise-scenarios.

[Fow99] Martin Fowler. *Refactoring: Improving the design of existing code / Martin Fowler*. The Addison-Wesley object technology series. Reading, MA: Addison-Wesley, 1999. ISBN: 0201485672.

[Kar+12] Karen Ng et al. *The Roslyn Project: Exposing the C# and VB compiler's code analysis*. 2012.

[Kar+18] Karan Nandwani et al. *NuGet Documentation*. 21/11/2018. URL: https://docs.microsoft.com/en-us/nuget/what-is-nuget (visited on 11/26/2018).

[Mai+18] Maira Wenzel et al. *.NET-Standard Documentation*. 29/11/2018. URL: https://docs.microsoft.com/de-de/dotnet/standard/net-standard (visited on).

[Mar09] Robert C. Martin. *Clean code: A handbook of agile software craftsmanship / Robert C. Martin ... [et al.]* Indianapolis, Ind.: Prentice Hall and London : Pearson Education [distributor], 2009. ISBN: 0132350882.

[NIS14] NIST. *NVD - CVE-2014-1266*. 2014. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-1266 (visited on 11/01/2018).

[Pil18] Kevin Pilch. *Roslyn.Says('Hello World');* Sept. 2018. URL: https://github.com/dotnet/roslyn/commit/3611ed35610793e814c8aa25715aa582ec08a8b6.