Analysis of Legacy Code

31. Juli . 2014

Christian Wenzlick

Fachhochschule Lübeck
Studiengang Medieninformatik
Mönkhofer Weg 239
23562 Lübeck

Deutschland

Erklärung zur Bachelorarbeit

Ich versichere die Bachelorarbeit selbst ohne Hilfe angefertigt zu haben.

Nur die angegebenen Quellen wurden bei der Anfertigung dieser Arbeit benutzt. Wörtliche und sinngemäße Zitate sind als solche gekennzeichnet.

Ich stimme der Veröffentlichung dieser Arbeit und der Weitergabe an Dritte zur Korrektur sowie der Erstellung von Kopien für Dritte zu.

Sankt Wolfgang 31.07.2014

Inhalt

Abbildungsverzeichnis

Legacy Code

Würde man eine Umfrage unter Software Entwicklern durchführen, welcher Aspekt im Bereich Software Entwicklung ihnen am meisten ein Dorn im Auge ist, so dürfte vermutlich das Gebiet Legacy Code ein Kandidat für einen der Spitzenplätze sein. Stellenweise kann man zu dem Eindruck kommen der Bereich Legacy Code sei von einer ominösen Aura umgeben und schwebt wie ein unheilvoller Schatten über der Schulter des Entwicklers.

Dabei ist Legacy Code ein ganz konkretes und relativ weit verbreitetes Phänomen. Man kann Legacy Code lesen, man kann ihn analysieren und man kann ihn verbessern, warum also arbeiten viele Entwickler so ungern mit Legacy Code. Um darauf eine Antwort zu finden, muss man sich tiefer mit der Beschaffenheit und gewissen Eigenschaften von Legacy Code beschäftigen. Zuvor aber sollte man sich klar machen, was Legacy Code überhaupt ist und wie sich die Begrifflichkeiten darum entwickelt haben.

Geschichtliche Entwicklung des Begriffs Legacy

Der Begriff Legacy Code geht auf den früher gebräuchlicheren Begriff des Legacy Systems zurück, was sich im deutschen wohl gut mit dem Begriff des Altsystems übersetzen liese. Ursprünglich stammt der Begriff nicht aus der Software- oder Computerwelt, sondern wurde generell für veraltete Systeme, wie etwa für politische oder gesellschaftliche Systeme verwendet. Wann der Begriff zuerst im Bereich der Computertechnik verwendet wurde ist heutzutage nur noch schwer zu rekonstruieren, es dürfte aber wohl auf das Ende der 70er Jahre des vergangenen Jahrhunderts zurückgehen. Eine der frühen Erwähnungen von Legacy Programmen dürfte wahrscheinlich aus dem Tagungsband Proceedings of the 1978 Army Numerical and Computers Analysis Conference[Aro78] stammen. Im Buch New Directions in Project Management von Paul C. Tinnirello aus dem Jahre 2001 bekommt man auch einen guten Einblick, wie weit verbreitet das Phänomen Legacy System bereits damals war, wenn man im Kapitel „Managing Legacy Assets“ liest, dass „recent estimates show […] the current installed base of mission-critical legacy applications would cost $3 trillion to replace“ [Tin01][1].

Seit der Anfangszeit hat der Begriff des Legacy Systems allerdings eine deutliche Veränderung durchlaufen. Früher wurde der Begriff Legacy System in der Regel für Hardware oder fertige Softwareprodukte benutzt, die entweder veraltet waren oder im Fall von Software beispielsweise über keine Dokumentation oder keinen Sourcecode mehr verfügten. Es wurden damit also in der Regel Black Box Systeme bezeichnet, die durchaus noch funktionierten und ihren Dienst taten, aber es konnte unter Umständen niemand mehr genau sagen wie sie funktionierten oder es war schwierig an Ersatzteile zu gelangen. Solange diese Systeme also noch fehlerfrei funktionierten war es zwar möglicherweise eine unschöne Situation aber prinzipiell noch kein Problem. Viele dieser Legacy Systeme wurden damals – und zu einem großen Teil auch heute noch – teilweise über Jahre hinweg ohne größere Probleme als Legacy System betrieben. Die Probleme und damit auch Kosten treten dann in der Regel erst auf, wenn es zu Fehler in den Systemen kommt, oder wenn die Systeme an neue Begebenheiten angepasst werden sollen.

[1] [Tin01] Kapitel 37 – Managing Legacy Assets. Seite 403ff

Eines der bekannteren Beispiele für dieses Phänomen dürfte wohl das NASA Space Shuttle sein, dass bis zu seiner Außerdienststellung im Jahr 2011 zu sehr großen Teilen auf und mit Technik aus den 1970er Jahren funktionierte. Das führte sogar soweit, dass die NASA Ingenieure sich nach alternativen Bezugsquellen für ihre alte Hardware wie Floppy Drives und 8086 Prozessoren umsehen mussten. Ein Beispiel für die Beschaffung der 8086 Prozessoren wird im NASA Report „Astronautics and Aeronautics: A Chronology, 2001-2005“ veröffentlicht von der NASA History Division erwähnt, wo es heißt „For example, NASA had purchased outdated medical equipment to acquire the increasingly scarce Intel 8086 computer chip” [Ive10][2].

Im Laufe der Zeit hat sich diese Verwendung des Begriffs Legacy auch immer mehr auf das Innenleben der Software ausgebreitet und so den Begriff Legacy Code geprägt. Aber auch in diesem Bereich wurde der Begriff initial für veraltete Technik und veralteten Code benutzt. So wird beispielsweise alter Sourcecode in Programmiersprachen wie Cobol, Pascal oder Visual Basic vor der .NET Ära gerne als Legacy Code bezeichnet. Diese Zuordnung ist teilweise allerdings auch kontextabhängig, unter Windows Plattformen wird zum Beispiel auch gerne C-Code als Legacy Code geführt, da es hier inzwischen neuere und verbreiterte Varianten wie C#.NET gibt, während unter Linux und Unix C-Code nach wie vor zumindest einer der vorherrschenden Standards, wenn nicht gar der Standard selbst ist. Ganz ähnlich wurde auch häufig Code als Legacy Code bezeichnet, wenn er, als veraltet betrachtete Techniken und Bibliotheken benutzte oder für veraltete Umgebungen optimiert war, wie beispielsweise Webseiten die für Internet Explorer 6 optimiert sind oder Programme die Windows 3.1 voraussetzen, wie es unter anderem bei einigen Point of Sale Systemen üblich ist.

Diese ganzen Bedeutungen waren lange Zeit das was man unter Legacy Systemen verstand und auch heute findet man noch viele Menschen aus dem IT-Umfeld, die Legacy Systeme so definieren würden. Aber das ist nicht die Bedeutung von Legacy, die für diese Arbeit relevant sein soll. Hierfür soll eine relativ neue Definition von Legacy Code betrachtet werden, die erst in den letzten Jahren Verbreitung fand. Viele der im folgenden angesprochenen Techniken und Ideen lassen sich mit Sicherheit auch auf Legacy Systeme nach einer der alten Definitionen anwenden und übertragen, aber dies soll hier nicht im Vordergrund stehen.

Code ohne Test

Die für diese Arbeit wichtige Definition von Legacy Code wurde unter anderem von Michael C. Feathers in seinem Buch „Effektives Arbeiten mit Legacy Code“ aufgestellt. Dort stellt er die simple Definition auf nach der „[…] Legacy Code ganz einfach Code ohne Test“ [Fea11][3] ist.

Wie der Autor bereits selbst bestätigt ist diese Definition bisher noch nicht allgemein anerkannt und es gibt in der IT Welt große Gegenwehr dagegen. Im Laufe dieses Kapitels soll aber noch herausgestellt werden, warum die Definition nach Michael C. Feathers eventuell doch die, zumindest für die Arbeit mit Legacy Code, sinnvollere Definition ist.

[2] [Ive10] Seite 72f
[3] [Fea11] Geleitwort Seite 16

Häufig wird von Gegnern dieser Definition der Punkt angebracht, dass das Vorhandensein, bzw. die Absenz von Tests keinen Einfluss auf den Code hätten [Fea11], die sie ja lediglich Beiwerk sind aber den Code selbst nicht beeinflussen. Dies ist natürlich erstmal richtig, ein vorhandener Test kann maximal das Vorhandensein von Fehlern im Code in bestimmten Szenarien und mit bestimmten Parametern feststellen, er wird aber niemals den Code selbst verändern oder gar verbessern. Daher kann also auch die Absenz von Tests keinen Einfluss auf den Code haben.

Um dieses Argument zu entkräften muss man die Motivation hinter Michael C. Feathers Definition genauer betrachten. Ihm geht es sicher nicht darum zu sagen ein Stück Sourcecode mit Test ist per se „perfekter" Code. Vielmehr geht es ihm darum eine bestimmte Eigenschaft von Code herauszustellen, die Wartbarkeit oder Veränderbarkeit. Er nimmt als Grundlage an, dass Code, der nicht veränderbar und damit auch nicht wartbar ist unter den Begriff Legacy Code fällt. Nun wäre dies als Definition relativ schwierig, da Veränderbarkeit von Code keine messbare Metrik ist, man kann kein Analyseverfahren einsetzen um hier zu einem Wert zu kommen, der angibt „Ja es ist Legacy Code" oder eben nicht. Michael C. Feathers bedient sich daher für seine Definition einer Hintertür.

Er postuliert, jeglicher Sourcecode, der keine Tests aufweist ist nicht veränderbar.

Nun mag man einwenden, dass Sourcecode auch ohne Tests jederzeit veränderbar ist, man kann problemlos Code hinzufügen, ändern, ganze Funktionen neu implementieren und so weiter. Dabei stößt man nur auf das Problem, dass man nach einer Änderung nicht mehr sagen kann, ob der Sourcecode bzw. das Programm noch genauso funktioniert wie vorher. Man muss hier den Begriff veränderbar also nicht nur als „man kann Sourcecode verändern mit ungewissem Ergebnis" sehen, sondern als „man kann Sourcecode verändern und hinterher funktioniert er wie bisher". Diese zusätzliche Hürde, die Prüfung ob der Sourcecode nach einer Änderung nach wie vor dieselbe Funktionalität ohne entsprechende Fehler aufweist, ist nur über einen Test möglich.

Daher kann also über diesen Umweg die Definition „Legacy Code ist unveränderbarer Code" um eine qualifizierbare Metrik erweitert werden zur oben postulierten Definition „Legacy Code ist Code ohne Tests".

Was versteht man eigentlich unter Tests?

An dieser Stelle sei noch ein kurzer Abstecher in den Bereich Test erlaubt. Man könnte sich fragen, was Michael C. Feathers genau mit Tests meint, ob lediglich automatisierte Unit Tests zählen, oder ob die Definition von Test entsprechend ausgeweitet werden kann. Beschäftigt man sich näher mit seinem Buch „Effektives Arbeiten mit Legacy Code", kann man zu dem Schluss kommen, dass nach Ansicht des Autors tatsächlich eher nur Unit Tests als Tests im Sinne der Definition zählen. Dies zeigt sich zum Beispiel im Kapitel 13 wo darauf hingewiesen wird, dass Teams die mit manuellen Tests arbeiten eher schlechter vorankommen als vergleichbare Teams die automatische Tests und insbesondere Unit Tests benutzen[Fea11][4].

[4] [Fea11] Kapitel 13 – Ich muss etwas ändern, weiß aber nicht, welche Tests ich schreiben soll. Seite 205

Nun könnte man einwenden, dass die Frage „manuell oder automatisch“ abhängig von den jeweiligen Gegebenheiten ist. Bei kleineren Softwareprojekten könnte es also ausreichen manuelle Tests zu verwenden. Genauso wenn man möglicherweise mehr freie Testkapazität als Entwicklerkapazität in den einzelnen Teams hat. Ab einer gewissen Größe aber wird es sicher schwer bis unmöglich den Testaufwand mit rein manuellen Mitteln zu stemmen.

Michael C. Feathers führt aber weiter aus, dass „automatisierte Test ein wichtiges Mittel [sind], aber nicht um Fehler zu suchen“. Ganz im Gegensatz dazu sollen „automatisierte Tests ein neues Ziel beschreiben oder ein bereits vorhandenes Verhalten bewahren“ [Fea11].

Zieht man diesen Zusatz in Betracht wird schnell klar, dass es damit immer schwieriger wird manuelle Tests als mögliche Option zu sehen. Bereits die ursprüngliche Definition, dass Tests die Veränderbarkeit von Sourcecode gewährleisten impliziert, dass es sich um Tests handeln muss, die nach jeder Änderung am Code ausgeführt werden müssen und bei jedem Durchlauf identisch ablaufen müssen. Dies lässt eigentlich nur den möglichen Schluss zu, dass man, um diese Abdeckung durch Tests zu erreichen nicht auf manuelle Tests setzen kann, sondern auf automatisierte Tests und hier insbesondere auf Unit Tests angewiesen ist.

Der Autor möchte vermutlich nicht manuelle Tests als Dinge der Vergangenheit abtun, die mit der Zunahme von automatisierten Unit Tests obsolet werden, sondern eher eine Distinktion zwischen zwei verschiedenen Bereichen von Test aufzeigen. Betrachtet man Tests als automatisierte Unit Tests, so sind diese ganz klar Teil der Entwicklungsteams, sie werden von Entwicklern geschrieben und sie helfen direkt den Entwicklern ihren Code zu schreiben und ihn zu verändern. Dem gegenüber sind manuelle Tests im Bereich einer – möglicherweise – dedizierten Testabteilung anzusiedeln, die natürlich weiter für ein erfolgreiches Produkt nötig sind, aber komplett andere Ziele verfolgen als Entwicklertests. Hier wird verstärkt auf die Korrektheit des Gesamtprogramms nach außen hin und die Umsetzung aus Sicht eines möglichen Kunden oder Benutzers betrachtet.

Das kann sicher als Apell aufgefasst werden, die unter Entwickler – vor allem Entwicklern die nach Wasserfall oder ähnlichen Modellen arbeiten – weit verbreitete Annahme, dass Test Aufgabe von Tester und nicht Aufgabe von Entwicklern ist in Frage zu stellen.

Wieso entsteht Legacy Code?

Um zurück zum eigentlichen Thema zu kommen, stellt sich die Frage wieso eigentlich Legacy Code entsteht. Sicher wird kein Entwickler zu Beginn eines neuen Projektes sich entschließen dieses Mal Legacy Code zu schreiben, kein Product Owner wird verlangen, dieses Feature muss in Legacy Code geschrieben werden. Setzt man ein neues Softwareprojekt auf wird es mit absoluter Sicherheit – zumindest eine Weile – nicht die Klassifizierung Legacy Code erhalten.

Warum passiert es also, dass Projekte, die ganz normal und sauber beginnen im Laufe der Zeit zu Legacy Code werden?

Um sich der Frage zu nähern kann man noch einmal auf das bereits erwähnte Beispiel des NASA Space Shuttles zurückkommen. Abseits der reinen Tatsachenberichte, dass die NASA Bodencrew gezwungen war sich nach alternativen Quellen für ihre in die Jahre gekommene

Hardware umzusehen gibt es auch einige Studien die sich damit beschäftigt haben wie es überhaupt soweit kommen konnte. Eine davon ist ein Paper von Steven E. Gemeny von der Johns Hopkins University aus dem Jahr 2003 mit dem Titel „Longevity Planning, A Cost Reduction Strategy for Ground systems of Long Duration Space Missions“. In diesem Paper kommt Steven E. Gemeny unter anderem zu dem Schluss, dass Legacy Probleme und damit erhöhter Wartungsaufwand und steigende Kosten mitunter durch fehlende Langzeitplanung verursacht werden. Er fasst dies gut mit seiner Aussage zusammen, dass „this lack of planning and the belief that “maintenance of a system is not an issue for the design team” has lead [sic] to increased cost and decreasing system reliability in numerous situations”[Gem03][5]. Dieses Verhalten lässt sich sehr häufig in der Softwareentwicklung und vermutlich auch in vielen anderen Ingenieursdisziplinen beobachten. Software wird häufig mit dem Mindset entwickelt wie man das aktuelle Szenario möglichst perfekt abbilden kann. Dabei wird dann kaum Rücksicht darauf genommen, dass die Software unter Umständen mehrere Jahre oder gar Dekaden laufen muss und sich in dieser Zeit das Szenario und die umgebenden Parameter verändern können und häufig auch werden. Verstärkt wird das natürlich auch noch durch ein betriebswirtschaftlich orientiertes Management, welches vielmals mehr Wert darauf legt die Software möglichst schnell und möglichst kostensparend an den Kunden zu bringen um den Gewinn zu steigern. Beides natürlich sehr verständliche Entwicklungen wenn man sich das heutige, sich sehr schnell verändernde Softwareumfeld ansieht, in dem Software manchmal doch bereits nach wenigen Jahren oder Monaten obsolet ist und durch einen Nachfolger ersetzt wird. Insgesamt fehlt dann häufig die Langzeitplanung, die dafür sorgen würde Wartbarkeit und Veränderbarkeit als wichtigeren Posten in der Softwareentwicklung zu fixieren.

Man kann nun sicher vortrefflich darüber streiten, ob in der heutigen Welt Langzeitplanung für Software nötig ist oder nicht. Vermutlich muss man sich hier sehr stark auf das Umfeld beziehen für das ein Softwareprodukt gedacht ist. Vor allem im Consumer-Bereich findet man heute häufig Software die als äußerst kurzlebig ausgelegt ist, seien es nun verschiedene Computerspiele, Mobile-Anwendungen oder auch Office-Anwendungen bzw. Grafikprogramme, die einen enorm schnellen Releasezyklus haben. Ist man für ein solches Softwareprojekt verantwortlich, dass nur kurze Zeit betreut werden muss, ist es sicherlich weniger wichtig die auf lange Sicht wichtige Wartbarkeit zu berücksichtigen. Für alle anderen Softwareprojekte, seien es nun langlebige Business-Anwendungen oder Softwareprodukte die von Version zu Version immer aufeinander aufbauen, sollte man sich sehr genau überlegen wie man die Veränderbarkeit der Software und damit auf lange Sicht auch die Stabilität und Funktionsfähigkeit erhöhen kann. So werden im Behörden-, Point of Sale- oder Firmenumfeld häufig Softwareprodukte verwendet, die über Dekaden hinweg auf derselben Codebasis aufbauen und lediglich erweitert werden. Für solche Fälle ist es imminent wichtig, dass die Software auch nach Jahren noch möglichst einfach verändert werden kann.

Wie verhindert man Legacy Code?

In vielen anderen Bereichen der Softwareentwicklung hat man bereits erkannt, dass man bestimmte Eigenschaften von Software nicht nachträglich einbauen kann. Nimmt man als Beispiel den Bereich Sicherheit von Software, so wird man sehr schnell erkennen, dass man

[5] [Gem03] Seite 1

Sicherheit von Anfang an bei der Arbeit mit einem Softwareprojekt beachten muss. Sicherheit – wenn sie funktionieren soll – greift so tief in alle Aspekte einer Software ein, dass es nahezu unmöglich ist bei einem Projekt, welches von Anfang an nicht auf Sicherheit geachtet hat, diese noch nachträglich einzubauen. Ähnlich verhält es sich beim Themenkomplex Legacy Code. Auch hier muss man von Anfang an – und kontinuierlich bei jeder Änderung – darauf achten den Code entsprechend zu strukturieren und zu schreiben um von vorne herein ein hohes Maß an Veränderbarkeit und Sauberkeit zu erreichen, dass auf lange Sicht das Risiko irgendwann Legacy Code zu erhalten verringert.

Entsprechende Techniken wie man Code von Projektstart an sauber schreibt gibt es bereits seit langer Zeit, seien es nun Clean Code Praktiken, bestimmte Architektur Pattern oder Ansätze wie Test-Driven Development. Konsequent und von Anfang an umgesetzt können diese und ähnliche Techniken es enorm verzögern, wenn nicht gar verhindern, dass Sourcecode zu schlecht wartbarem Legacy Code degeneriert.

Das neu entstehen von Legacy Code könnte damit also eigentlich als gelöstes Problem angesehen werden. Leider trifft das aber nur theoretisch zu, in der Praxis werden sich vielfältige Gründe finden lassen, warum man innerhalb eines Softwareprojekts diese Techniken nicht einsetzen kann oder will. Dies kann alles von schlecht sensibilisierten Entwicklern über mangelnde Infrastruktur und Toolunterstützung bis hin zu falschem Management sein. In der Praxis wird man also häufig nicht nur auf schon alten und vorhandenen Legacy Code stoßen sondern auch viele Ecken finden in denen neuer Legacy Code geschrieben wird.

Aber auch für diese Fälle gibt es sinnvolle Ansätze um das Problem abzumildern oder gar ganz zu beseitigen. Wie bereits erwähnt ist es häufig schwierig bis nahezu unmöglich gewisse Aspekte von Software nachträglich noch hinzuzufügen – oben wurde dafür der Bereich Sicherheit angesprochen. Ähnlich verhält es sich auch für den Bereich Legacy Code, diesen von Anfang an zu vermeiden ist weit einfacher als hinterher zu versuchen den Code wieder aufzuräumen. Dennoch gibt es auch hier einige Techniken die darauf abzielen bereits vorhandenen unsauberen Code zu verbessern und zu säubern. Wenngleich diese Herangehensweise weitaus schwieriger ist, so ist sie dennoch nicht unmöglich, wie zum Beispiel auch das in dieser Arbeit besprochene Analyseframework zeigen soll.

Eine Auswahl interessanter Metriken

In der Softwareentwicklung werden häufig verschiedene Metriken und Analyseverfahren eingesetzt um grundlegende Aussagen über die Beschaffenheit und bestimmte Eigenschaften des Sourcecodes zu erhalten. Diese Metriken und Analyseverfahren werden häufig für Codebasen verwendet, die relativ sauber und gut strukturiert sind und sind daher auch entsprechend auf diesen Einsatzzweck zugeschnitten.

Nun findet man allerdings auch häufiger, wie bereits im ersten Kapitel beschrieben in der freien „Wildbahn“ Software die nicht einem gewissen Maß an Sauberkeit und Struktur genügen – eben Legacy Code. Der Schluss liegt nun Nahe diese Metriken und Analyseverfahren auch auf den Bereich Legacy Code anzuwenden.

Wie sich aber herausstellt ist dies häufig nur indirekt möglich, da viele der Metriken einfach nicht auf Sourcecode zugeschnitten sind, der bestimmten Anforderungen nicht erfüllt. Zusätzlich passiert es auch häufig, dass Metriken und Analyseverfahren zwar prinzipiell für Legacy Code angewendet werden könnten, ihre Aussagekraft dann allerdings recht gering ist oder sie nicht darauf zugeschnitten sind die Kernprobleme in Legacy Code zu erkennen.

Im Folgenden soll eine kleine Auswahl dieser Metriken vorgestellt werden und darauf eingegangen werden wo die Schwächen dieser Metriken im Umgang mit Legacy Code liegen.

Lines of Code

Wenn man sich mit Metriken in der Softwareentwicklung beschäftigt wird man zwangsläufig auf die Metrik Lines of Code oder kurz LoC stoßen. Obwohl es die wohl älteste Metrik ist, die in der Softwareentwicklung eingesetzt wird, sie geht wohl bis in die 1960er Jahre zurück, wird sie dennoch auch heute noch in vielen Bereichen verwendet. Ursprünglich stammt die Metrik noch aus einer Zeit in der zeilenbasierte Programmiersprachen wie FORTRAN oder Assembler dominierten und die Softwareentwicklung noch auf Basis von Lochkarten stattfand. Unter dieser historischen Verwendung muss man auch die Aussagekraft der Metrik selbst einordnen, die zu Zeiten als die Softwareentwicklung noch in den Kinderschuhen steckte ungleich größer war als sie das heute ist. Wurde sie früher größtenteils auch eingesetzt um die Produktivität eines Entwicklers in Zeilen pro Zeiteinheit zu messen oder die Effizienz des Entwicklungsprozesses in Kosten pro Zeile zu erfassen, wird sie heute kaum noch als einzelne Metrik eingesetzt, sondern vielmehr im Zusammenhang mit anderen Metriken als „Unterstützungsmetrik“ oder als erster Einstieg in eine Codebasis verwendet.

Trotz dieses Rückgangs an Bedeutung für die Lines of Code Metrik soll sie hier alleine schon wegen ihrer historischen und fast schon ikonischen Bedeutung erwähnt werden.

Wie bereits erwähnt ist die Lines of Code Metrik schon bei sauberen und gut strukturierten Codebasen eher weniger aussagekräftig, daher stellt sich die Frage, wie sinnvoll ist sie für Legacy Code, eigentlich nicht. Der Vollständigkeit halber soll hier aber dennoch kurz darauf eingegangen werden.

Im Bereich Legacy Code wird man häufig auf sehr große Codebasen treffen, denn je kleiner eine Codebasis, desto einfacher ist es sie strukturiert zu halten und somit die Chance auf

Legacy Code zu minimieren. Große Schwierigkeiten mit Legacy Code haben dagegen häufig eher größere Softwareprojekte, die mitunter Hunderttausende oder sogar mehrere Millionen Lines of Code haben. Die Metrik Lines of Code würde einem hier aber lediglich sagen können wie groß genau das Softwareprojekt eigentlich ist, macht aber naturgemäß keine Aussage über die Probleme die im Code stecken und wo diese stecken. Man könnte die Metrik im Bereich Legacy Code also höchstens dazu benutzen vorher abzuschätzen, wie hoch die Chance ist in einem Softwareprojekt auf Legacy Code zu treffen, je höher die Anzahl an Codezeilen, desto höher die Chance auf Legacy Code. Da es sich hier aber nicht um eine direkte Abhängigkeit handelt und das Vorhandensein von Legacy Code von weit mehr Faktoren abhängt ist die Aussagekraft auch hier eher gering.

Von Interesse wird die Metrik allerdings, wenn man sie in Relation zu bestimmten Code Merkmalen setzt, wenn man also beispielsweise betrachtet wie viele Codezeilen haben einzelne Programmfunktionen oder Klassen. Über diese Relation lässt sich dann direkter feststellen an welchem Bereich im Code es zu Problemen kommen könnte. Diese Kombination aus der Lines of Code Metrik heruntergebrochen auf einzelne Codeabschnitte soll daher auch für das Analyseframework im Hinterkopf behalten werden.

Cyclomatic Complexity

Eine weitere Metrik, die nach wie vor breite Verwendung findet, wurde 1976 von Thomas McCabe[McC96] entwickelt und nach ihm benannt. Die McCabe‘s Complexity Metrik, häufig auch Cyclomatic oder Conditional Complexity genannt, betrachtet die interne Komplexität eines Programmes. Dabei wird der Code als Control Flow Graph dargestellt und anschließend die Anzahl der Kanten und Knoten bewertet. Im einfachsten Fall berechnet sich die Komplexität dann als

Anzahl der Kanten – Anzahl der Knoten + 2*Anzahl der Endpunkte

Man kann also vereinfacht sagen, dass die Komplexität steigt je mehr Kanten es im Verhältnis zu Knoten, das heißt je mehr Verzweigungen, wie if/else, switch usw. es im Code gibt. Zusätzlich steigt die Komplexität mit jedem zusätzlichen Endpunkt, in der Regel also returns im Code.

McCabe selbst hat als Obergrenze für die Komplexität einen Wert von 10 postuliert. Dieser Wert wird auch in vielen anderen Publikationen unterstützt, wenngleich es auch Teams gibt, die Höchstwerte von bis zu 15 erfolgreich verwalten können.

Bei diesem Beispiel handelt es sich sogar um eine Metrik die sehr gut geeignet wäre mögliche Schwachstellen im Code aufzuzeigen, indem es Code identifiziert der sehr komplex und damit schlecht wartbar ist. Um sie allerdings sinnvoll im Legacy Code Umfeld zu nutzen müsste man die Maßzahl wieder herunterbrechen auf kleinere Ebenen, die dann auch entsprechend behandelt werden können. Eine Cyclomatic Complexity von 30 für ein ganzes Projekt zeigt erstmal nur, dass es grundsätzlich ein Problem mit dem Projekt gibt – im Falle von Legacy Code dürfte das auch vorher schon sichtbar gewesen sein – aber erst wenn man es herunterbricht und zum Beispiel einzelne Funktionen mit einer Complexity von 30 identifizieren kann, hat man einen Ansatzpunkt für entsprechende Verbesserungen.

Bugs per line of code

Abseits dieser codenahen Metriken, die sich direkt und ausschließlich auf den Sourcecode beziehen gibt es zudem weitere Metriken, die andere Aspekte der Softwareentwicklung miteinbeziehen. Zu dieser Gruppe zählt beispielswiese die Metrik Bugs per Line of Code, die unter anderem auch von Steve McConnell in seinem Buch Code Complete[McC04] besprochen wird. Hier wird die Anzahl der gefundenen Softwarefehler mit der Gesamtanzahl an Lines of Code ins Verhältnis gesetzt. Übliche Standards sind dabei zum Beispiel für Microsoft Anwendungen 0.5 Fehler pro KLoC, also Kilo Lines of Code oder durchschnittlich 0.69 Fehler pro KLoC bei Open Source Software laut einer Studie durch die Firma Coverity[Cov06]. Hier können natürlich Abweichungen in beide Richtungen auftreten, zum Beispiel nach unten von 0 Fehler auf 500 KLoC für die Software im bereits erwähnten NASA Space Shuttle. Abweichungen nach oben finden sich dabei häufig auch in Bereichen mit Legacy Code.

Auf den ersten Eindruck klingt diese Metrik sehr Interessant für die Arbeit mit Legacy Code. Legacy Code ist in der Regel schlecht lesbar und damit schlecht verstehbar, was die Chance auf unentdeckte Fehler im Code enorm steigert. Eine Metrik zu haben, die einen Überblick bietet, wie fehlerhaft der vorliegende Code in etwa ist wäre da sicherlich von großem Vorteil.

Nur wie kommt man zur Anzahl der Bugs, bzw. Fehler im Code? Man könnte hierbei verschiedene Ansätze zu Hilfe ziehen, die Anzahl der von Benutzern aufgemachten Fehlertickets, die Anzahl der fehlgeschlagenen Unit Tests oder möglicherweise die Anzahl von, durch statische Code Analyse gefundene Fehler und Warnings. All diese Ansätze mögen bei gut strukturiertem Code durchaus funktionieren, aber bei der Verwendung im Zusammenhang mit Legacy Code ergeben sich dabei unweigerlich Probleme. Die Anzahl der fehlgeschlagenen Unit Tests wird bei Legacy Code in der Regel schwer zu ermitteln sein, da es einfach keine oder nur wenige schlechte Unit Tests gibt. Ebenso gibt die Anzahl der Fehlertickets kaum einen tatsächlichen Überblick über die Fehlersituation in der Software, da einerseits wesentlich weniger Tickets geöffnet werden, als tatsächlich Fehler im Code vorhanden sind und sich die Fehlertickets andererseits auch in den Bereichen konzentrieren werden, die von den Benutzern häufig benutzt werden und in denen Fehler einfach sichtbar sind. Zudem ist es auch relativ schwierig Fehlertickets als Grundlage einer Metrik zu benutzen, da diese vorher zuerst mühsam sortiert und klassifiziert werden müssten. Eine Tätigkeit die schwer von einem Automatismus erledigt werden kann.

Lediglich die durch statische Code Analyse gefundenen Fehler wären einfacher als Grundlage zu verwenden, da sie automatisch erfasst werden können. Unter statischer Code Analyse versteht man dabei die Analyse des Codes in seinem ursprünglichen Zustand und nicht zur Laufzeit. Dies kann von relativ einfachen Analysen, wie zum Beispiel die Prüfung auf unerreichbaren Code oder ungenutzte Variablen bis hin zu komplexen Analysen wie der Taint Analyse, eine vor allem in Websprachen verbreitete Form, bei der versucht wird Variablen zu identifizieren, die vom Benutzer verändert werden können, reichen.

Allerdings ist auch bei der statischen Code Analyse die Aussage der gefundenen Fehler relativ gering, da die statische Code Analyse nur bestimmte Fehlertypen finden kann. Bestimmte Fehlertypen wie zum Beispiel logische bzw. inhaltliche Fehler lassen sich hier gar nicht finden.

Dieses Beispiel zeigt also, dass es durchaus auch Metriken gäbe, die sehr interessant für den Umgang mit Legacy Code wären, bei denen allerdings die Datenerhebung für Legacy Code schwierig bis unmöglich ist.

Motivation

Wie die kleine Auswahl an Metriken bereits zeigt ist es sehr schwierig mit vorhandenen Mitteln einen Einstieg in Legacy Code zu erhalten. Viele vorhandene Metriken oder Analyseverfahren verhalten sich ähnlich, wie die im vorherigen Kapitel vorgestellten, sie sind prinzipiell nicht schlecht – vor allem wenn man sie für sauberen Code mit hoher Testabdeckung einsetzen will – können aber nicht wirklich eins zu eins so auch für Legacy Code eingesetzt werden, zumindest nicht, wenn die eigentliche Motivation für den Einsatz die Verbesserung des Codes und nicht nur die Qualifikation als Legacy Code ist.

Diese Erkenntnis dürfte auch nicht verwundern, da es natürlich äußerst schwierig ist in einem so komplexen und diversen Umfeld wie der Softwareentwicklung einen One Size Fits All Ansatz zu versuchen. Man wird immer Metriken und Analyseverfahren finden, die besser für das jeweilige Aufgabengebiet passen oder eben nicht, aber nie eine Lösung, die immer und überall die richtige Wahl ist. Man sieht aber auch, dass es selbst bei Metriken, die auf den ersten Blick ungeeignet scheinen, und es, setzt man sie in Reinform ein auch sind, trotzdem die Möglichkeit gibt diese abgewandelt doch wieder sinnvoll nutzen zu können. Als Beispiel mag hier die bereits erwähnte Line of Code Metrik dienen, die bei ursprünglicher Verwendung, als Lines of Code für das ganze Projekt oder als Produktivitätsmetrik für einen Entwickler wenig Aussagekraft hat. Bricht man sie aber herunter und betrachtet Lines of Code eines einzelnen Blocks, beispielsweise einer Funktion oder einer Schleife, dann steigt die Aussagekraft und man kann so zum Beispiel schlecht wartbaren Code identifizieren.

Das wird auch in der Regel die übliche Vorgehensweise sein, wenn man ein vorliegendes Softwareprojekt betrachten möchte, man muss genau überlegen wie kann man eine gegeben Metrik sinnvoll auf die vorliegenden Gegebenheiten anpassen. Häufig setzt es sogar noch früher an und man muss sich damit beschäftigen welche Metriken und Analyseverfahren man überhaupt grundsätzlich für das entsprechende Projekt anwenden kann.

Es kann viele Gründe geben, warum bestimmte Metriken oder Analysen für das jeweilige Softwareprojekt nicht geeignet sind. Das können technische Gründe sein, so macht beispielsweise eine Taint Analyse, bei der Variablen, die von Benutzern geändert werden können untersucht werden, bei einer reinen Backend Anwendung wenig Sinn. Es können aber genauso gut inhaltliche Gründe sein. So könnte es sein, dass ein Projekt möglicherweise schon überwiegend gut strukturierten und vor allem „kurzen“ Code hat und daher eine entsprechende Analyse auf Complexity oder Lines of Code nicht nötig ist, oder zumindest hinten angestellt werden kann, während zuerst die drängenderen Probleme analysiert werden.

In der Praxis wird diese Bewertung immer schwierig sein, da die Fälle, bei denen man klar feststellen kann, dass eine bestimmte Metrik perfekt geeignet ist oder eben nicht, relativ selten sind. In den meisten Fällen wird man eher entscheiden müssen, ob die vorliegende Metrik in der jeweiligen Situation den höheren Nutzen bringt, bei geringeren Kosten als eine andere Metrik. Man wird also häufig eine Kosten-Nutzen Abschätzung durchführen müssen um sich für oder gegen bestimmte Metriken zu entscheiden.

Dies wird noch dadurch verstärkt, dass man zwar theoretisch beliebig viele Metriken einsetzen könnte, aber in der Praxis dabei durch die vorhandenen Ressourcen beschränkt wird. Der Einsatz jeder weiteren Metrik wird Ressourcen bei der Implementierung, bei der Erhebung und bei der Auswertung kosten, Ressourcen die man häufig nicht zur Verfügung hat. Zusätzlich erhöht auch jede weitere Metrik die Komplexität des ganzen Systems und man verbraucht zusätzliche Ressourcen um mit den gewonnen Zahlen zu jonglieren und daraus entsprechende Prioritäten abzuleiten.

Diese Auswahl der Metriken und Analyseverfahren ist dabei allerdings der wichtigste und häufig auch zeitaufwändigste Schritt im ganzen Prozess. Obwohl es hier vielleicht die meisten Einsparungsmöglichkeiten gäbe ist genau dieser Schritt leider auch nicht automatisierbar.

Ein weiterer Punkt den man beim Umgang mit Metriken beachten muss liegt in der Definition der Metrik selbst begründet. Metriken liefern per Definition nur Maßzahlen, also einen Zahlenwert der eine bestimmte Eigenschaft des Sourcecodes quantifiziert. Nun sind Maßzahlen als Mittel im Controlling natürlich hervorragend geeignet um Vergleiche anzustellen oder Trends zu beobachten. Für den einzelnen Entwickler, der versucht mit dem ihm vorliegenden Sourcecode zu arbeiten und eventuelle Schwachstellen oder sogenannten Code Smell zu finden sind sie allerdings eher weniger aussagekräftig.

Für einen Entwickler, der versucht vorliegenden Legacy Code zu verbessern wäre es wichtiger Stellen zu identifizieren, die Code Smell aufweisen und die daher eventuell behandelt und verbessert werden müssen. Als Code Smell werden dabei Stellen im Sourcecode bezeichnet, die – zumindest im Moment – kein Fehler sind und das Programm nicht beeinflussen, aber Ausdruck für unsauberen Code sind. Diese Stellen können, wenn man sie nicht regelmäßig entfernt, zu sinkender Wartbarkeit und auf längere Sicht damit auch zu Fehlern im Code führen. Gute Beispiele für Code Smell sind dabei lange Funktionen, man empfiehlt hier häufig Längen von maximal 10 Zeilen bis hin zu maximal eine Fensterhöhe, oder tiefe Verschachtelungen innerhalb einer Funktion, auch hier wird meist weniger als 4 Ebenen tief empfohlen.

Sobald solche Stellen identifiziert sind, ist es natürlich weiterhin von Bedeutung zu wissen, worauf man bei der Bereinigung des Code Smells achten muss. Häufig wird es in größeren und komplexeren Codebasen schwierig werden alle Abhängigkeiten einer bestimmten Funktion zu kennen. Das heißt hat man Code Smell in einer Funktion lokalisiert und möchte diesen beheben, dann sollte man sich vorher im Klaren sein welche anderen Bestandteile des Codes von genau dieser Funktion abhängen, um sicherzustellen, dass es durch die Änderungen zu keinen Fehlern in anderen Funktionen kommen kann.

Genau an diesem Punkt möchte das Codeanalyse Framework, dass Ziel dieser Arbeit ist, ansetzen. Es soll dem Entwickler ein Werkzeug in die Hand gegeben werden, dass ihm erlaubt Code Smell oder generell Stellen im Code zu finden, die für ihn von Interesse sein könnten um diese dann entsprechend zu beheben. Um die entsprechende Stelle zu beheben, sollen darüber hinaus auch die entsprechenden Abhängigkeitsinformationen zur Verfügung gestellt werden.

Da, wie erwähnt, es keine statische Auswahl von Metriken geben kann, die das Framework benutzen soll, sondern die Auswahl stark auf das Projekt bezogen erfolgen muss, soll hier auch

größtmögliche Flexibilität geschaffen werden. Den Entwicklern muss es möglich sein jede beliebige Art von Metrik auswählen und für die spätere Analyse benutzen zu können. Zusätzlich soll das Framework auch unabhängig von Programmiersprachen und Techniken verwendbar sein. Der Entwickler muss das Framework für jede Programmiersprache und im Zusammenhang mit jeder beliebigen Metrik oder Analysetechnik verwenden können.

Diese Flexibilität hat aber natürlich auch einen Preis. Je mehr Anpassungs- und Erweiterungsmöglichkeiten für den Benutzer verfügbar sein sollen, desto mehr Einrichtungsaufwand wird sich auch für den Nutzer ergeben. Im vorliegenden Fall bezieht sich die Änderbarkeit auf die zu analysierenden Sprachen und die Art der Analyse bzw. der Metriken die verwendet werden sollen. Wenn man als groben Workflow für das zu entwickelnde Framework annimmt, dass zuerst der Sourcecode analysiert wird, danach das Analyseergebnis aufbereitet und anschließend visualisiert werden soll, dann kann man annehmen, dass die Auswahl der Programmiersprache und der Metriken hauptsächlich im ersten Schritt von Bedeutung ist. Da ferner, aufgrund des Umfangs nicht alle möglichen Programmiersprachen und Metriken direkt im Framework definiert werden können, muss dafür eine Art Plug-In Mechanismus realisiert werden. Dem Benutzer muss es also möglich sein je nach Bedürfnis andere Sprachen und Metriken in Form eines Plug-Ins im Zusammenhang mit dem Framework zu benutzen.

Das kann dann einerseits bedeuten, dass der Entwickler entsprechende, bereits fertige Plug-Ins bei Bedarf nachladen kann. Es kann aber andererseits auch bedeuten, dass er, wenn seine Bedürfnisse spezieller sind, diese Plug-Ins auch selbst entwickeln muss. Diesen Umfang an Eigenarbeit für die Benutzung eines Tools wird man normalerweise ungern voraussetzen, da es Zeit und entsprechende Kenntnisse beim Benutzer erfordert und dies bei Standardbenutzern häufig nicht gegeben ist. Für das vorliegende Framework sind die Zielgruppe allerdings andere Entwickler und somit kann hier auch ausnahmsweise ein gewisses Maß an Eigenarbeit verlangt werden.

Zusammenfassend lässt sich also sagen, dass es leider schwer ist den Auswahlprozess der Metriken und die dafür nötige Kosten-Nutzen Abschätzung in sinnvoller Weise durch entsprechende Tools zu unterstützen oder gar komplett zu automatisieren. Daher soll hier wenigstens versucht werden dem Entwickler für die Arbeit mit den ausgewählten Metriken ein Tool an die Hand zu geben, was ihm erlaubt, unabhängig von seiner Auswahl sinnvoll mit den Metriken zu arbeiten und entsprechende Codearbeiten möglichst sicher durchzuführen. Es soll also ein Framework entstehen, dass den Entwickler einerseits im Grundzustand bei seiner Arbeit mit Metriken unterstützen kann, ihm andererseits aber auch genug Möglichkeit gibt Anpassungen nach seinen Bedürfnissen durchzuführen und sich die Arbeit damit noch mehr zu erleichtern.

Konzeption

Die Motivation ist also dem Entwickler ein Tool an die Hand zu gehen mit dem er Sourcecode allgemein – und Legacy Code im besonderen – sinnvoll erfassen und durch geeignete Metriken analysieren kann um schließlich zu versuchen den Code zu verbessern.

Sicherlich gibt es viele Ansätze wie man dieses Ziel vor allem im Hinblick auf Legacy Code erreichen kann. Der Ansatz, der hier im Folgenden vorgestellt werden soll ist bei weitem nicht der einzige Mögliche und sicherlich auch kein Wundermittel, aber er ist ein möglicher Ansatz, der speziell auf Legacy Code zugeschnitten ist und daher gute Ergebnisse liefern kann. Am Rande sei noch erwähnt, dass der folgende Ansatz keine Metrik an sich ist, sondern ein Analyseverfahren, dass mitunter verschiedene Metriken einsetzt.

Darstellen der Anforderungen an das Codeanalyse Framework

Der zentrale Bestandteil dieser Arbeit ist es ein Framework zu schaffen, dass die Analyse verschiedener Codebases, vor allem in Hinblick auf Legacy Code ermöglicht. Dabei ist besonders Wert darauf zu legen, dass das Framework universell einsetzbar ist, sowohl bezüglich der zu analysierenden Programmiersprachen als auch mit der Option verschiedene Visualisierungsmöglichkeiten zu verwenden. Zusätzlich soll ein geeignetes Format entworfen oder vorgeschlagen werden, dass die einfache und möglichst menschenlesbare Speicherung der Analyseergebnisse ermöglicht.

Da jede Programmiersprache unterschiedlich aufgebaut ist, unterschiedliche Keywords verwendet oder nach unterschiedlichen Paradigmen arbeitet, ist es notwendig diese unterschiedliche Ausgangsbasis in eine generische Repräsentation des Codes zu überführen, die dann weiterverwendet werden kann. Um dieser Problematik zu begegnen soll das Programm in zwei Bereiche aufgeteilt werden, einerseits das zentrale Framework selbst, das die Aggregation der Ergebnisse und die letztliche Speicherung übernimmt und andererseits den sprachspezifischen Plug-Ins, die die eigentliche Analyse für ihre jeweilige Sprache übernehmen.

Das zentrale Framework muss daher eine Möglichkeit bieten die eigenständigen Sprach-Plug-Ins dynamisch, zum Beispiel per Reflection einzubinden. Dazu muss ein Interface geschaffen werden, dass dann von den einzelnen Plug-Ins zu implementieren ist.

Als erster Analyseschritt soll der generelle Aufbau des Codes betrachtet werden um daraus weitere Analyseergebnisse abzuleiten. Der Code muss dazu in eine passende Struktur überführt werden, die das einfache Weiterarbeiten ermöglicht. Hierzu bieten sich generell Abstract Syntax Trees an, die unter Umständen abgewandelt und um zusätzliche Bereiche erweitert werden müssen.

Für die einzelnen Knotenpunkte der ASTs sollen dann weitere Informationen gesammelt werden, als erste Beispielimplementation die Anzahl der Codezeilen. Im Idealfall sollten diese Zusatzinformationen dynamisch von den einzelnen Sprach-Plug-Ins gesammelt werden können, je nachdem welche Informationen für die jeweilige Sprache sinnvoll sind um sie dann in einer generischen Form ans zentrale Framework zu übergeben. Bei den gesammelten Informationen kann es sich sowohl um qualitative – wie zum Beispiel etwaig auftretende NullReferences – als

auch um quantitative – wie beispielswiese die Anzahl an Zeilen einen Funktion – Informationen handeln.

Diese gesammelten Zusatzinformationen sollen dem Entwickler besonders helfen Code Smell zu identifizieren und diesen entsprechend zu beseitigen. Dazu muss zusätzliche zu den Informationen selbst eine Klassifizierung geschaffen und gespeichert werden, ob ein Treffer nun Code Smell darstellt oder nicht. Bei qualitativen Informationen wird dies ein einfaches Ja/Nein sein, bei quantitativen Informationen muss hier in der Regel ein Grenzwert hinterlegt werden, also beispielsweise ab X Zeilen Code pro Funktion liegt Code Smell vor.

Die Knotenpunkte selbst sollen darüber hinaus auch eine Typeigenschaft erhalten, die angibt, ob es sich bei einem Knoten um eine Programmfunktion, ein Schleifenkonstrukt oder eine Bedingung handelt. Dadurch soll es später möglich sein Teilbäume zu erstellen, die beispielswiese nur Funktionen beinhalten um damit den Funktionsfluss darzustellen.

Ein zusätzliches Problem, dass bei der Erstellung der Abstract Syntax Trees auftreten kann ist durch die Art der Darstellung als Baum begründet. Jeder Knoten in einem Baum – der Ursprungsknoten natürlich ausgenommen – kann, per Definition nur jeweils genau einen Parent haben. Übertragen bedeutet das also, dass eine Programmfunktion, die ja auch als Knoten innerhalb des Baums abgebildet werden soll nur einen Parent haben kann, also nur eine sie aufrufende Funktion. In der Realität kommt es natürlich häufig vor, dass eine Funktion von mehreren anderen Funktionen aufgerufen wird. Für diesen Anwendungsfall muss eine entsprechende Lösung implementiert werden, die es erlaubt auch Funktionen darzustellen, die mehrmals aufgerufen werden. Dies könnte dabei beispielsweise durch Aufteilung in mehrere Bäume, Duplikation von Knoten oder der Darstellung als Graph anstatt als Baum gelöst werden.

Abschließender Bestandteil des Frameworks ist die Speicherung der Analyseergebnisse. Dazu soll ein möglichst menschenlesbares Format gewählt werden, dass es ermöglicht die Baum- bzw. Graphenstruktur zusammen mit allen sprachspezifischen Zusatzinformationen zu Speichern. Hierbei bietet sich wohl am besten ein XML-ähnliches Format an, da es einerseits menschenlesbar ist und damit auch dem Modularitätsgedanken folgt. Das heißt es ist jederzeit möglich sich das Analyseergebnis in Form der XML Datei zu nehmen und – ohne sich mühsam in das Speicherformat einarbeiten zu müssen – eine eigene Visualisierungsstruktur damit zu bestücken, oder auch die XML Datei als Basis für Gated Check-In Prüfungen in einem Sourcecontrol System zu benutzen. Anderseits ist XML auch sehr gut dazu geeignet baumartige Strukturen durch die Verschachtelung einzelner XML Knoten darzustellen.

Zum Schluss sollen hier noch ein paar Worte zur Mehrsprachigkeit verloren werden. Prinzipiell ist es natürlich möglich ein Softwareprojekt in mehreren Programmiersprachen zu realisieren, in vielen Bereichen, wie etwa im Web mit PHP und JavaScript, wird dies auch häufig so gemacht. Trotzdem soll die Analyse mehrsprachiger Projekte im Rahmen dieser Arbeit außen vor bleiben, da es sonst den Rahmen sprengen würde. Dennoch lässt sich sagen, dass das Codeanalyse Framework selbst keinerlei Probleme bei der Arbeit mit mehrsprachigen Projekten haben dürfte, da – mit Ausnahme der Sprach-Plug-Ins – alle Bestandteile mit sprachunabhängigen Daten arbeiten. Es sollte daher prinzipiell kein Problem sein ein Multisprach-Plug-In zu schreiben, dass mehrere Sprachen bearbeiten kann um die daraus gewonnen Ergebnisse dann ins

Framework einzuspeisen. Die Machbarkeit dessen wird hier allerdings nur theoretisch erwähnt, eine praktische Umsetzung wird nicht erfolgen.

Definition der Visualisierungs- und Filtermöglichkeiten

Aufbauend auf dem Codeanalyse Framework sollen die nachfolgend beschriebenen Beispielvisualiserungen und Filtermöglichkeiten implementiert werden, die dem Entwickler bei der Analyse von und der Arbeit mit Legacy Code unterstützen sollen. Auch hier soll großer Wert auf eine möglichst hohe Modularisierung und einfache Möglichkeiten zur Erweiterung des Tools gelegt werden. Als Basis für das Visualisierungstool dienen die vom Codeanalyse Framework gewonnen und gespeicherten Daten.

Abhängigkeitsansicht

Wie bereits angesprochen ist eins der grundlegenden Probleme bei der Arbeit mit Legacy Code die fehlende Struktur und die nicht klar abgegrenzten Abhängigkeiten. Für den Entwickler heißt das, dass er bei jeder Änderung besonders Sorge tragen muss, dass durch seine Änderung keine Fehler in abhängigen Komponenten verursacht werden. Während diese Sorgfalt bei sauber strukturierter Software auch von Nöten ist, ist es dort relativ einfach sicherzustellen, da es klare Strukturen gibt, mit denen Probleme einfach erkannt werden können und verschiedene Tests, darunter Unittests, die eventuellen Fehler frühzeitig finden.

Bei Legacy Code hingegen gibt es in der Regel kein Sicherheitsnetz durch Tests und auch keine saubere Struktur, die den Entwickler an dieser Stelle unterstützen würden. An dieser Stelle soll die erste Visualisierung ansetzen und dem Entwickler ermöglichen alle Abhängigkeiten innerhalb der Codebase herauszufinden.

Einerseits ist dabei natürlich grundlegend von Interesse welche Funktionen in welchen Klassen enthalten sind und andererseits welche Funktionen oder Methoden konkret von welchen anderen Funktionen aufgerufen werden. Als Visualisierung soll hier ein Netzplan dienen, bei dem alle Klassen bzw. Funktionen als Knoten und alle entsprechenden Abhängigkeiten als Kanten eingetragen werden. Die Kanten können dabei sowohl unidirektional als auch bidirektional wirken und sollen als solche gekennzeichnet werden. Zwischen der Ansicht für Klassen und Funktionen soll beispielsweise mittels eines Schalters gewechselt werden können.

Filtermöglichkeiten

Da bei zunehmender Größe der Codebase, was bei Legacy Code nicht selten der Fall sein dürfte, ein kompletter Abhängigkeitsplan recht schnell unübersichtlich und dadurch weniger nützlich wird, sollen hier zusätzliche Filtermöglichkeiten geschaffen werden. Dabei ist zu beachten, dass sich viele Programmiersprachen darin unterscheiden welche Strukturelemente und Bausteine sie zulassen. Bei Sprachen wie beispielsweise Java oder der .NET Familie könnte man zwischen „eigenem“ Code und Framework Code unterscheiden und diesen entsprechend filtern. Diese beiden Sprachen würden zudem eine Filterung nach Namespaces zulassen, während andere Sprachen, wie beispielsweise PHP vor Version 5.3 das Konzept eines Namespaces gar nicht kennen und daher eine Filterung danach unmöglich machen. Um diese Problematik zu lösen muss, wie auch schon im Framework selbst, die Implementierung eine Trennung zwischen dem eigentlichen Filtermechanismus und der Ausprägung des Filters

selbst vorsehen. Dadurch können die sprachspezifischen Filter als eine Art Sprach-Plug-In für jede gewünschte Sprache implementiert werden und an den zentralen Filtermechanismus andocken.

Übergreifend sollen die Filtermöglichkeiten primär dem Entwickler helfen die hohe Komplexität und Unübersichtlichkeit des kompletten Abhängigkeitsplans auf eine überschaubare Teilmenge herunterzubrechen, die sich auf den aktuell vom Entwickler bearbeiteten Bereich bezieht. Für die Funktionsansicht sollen daher Filter auf Klassenebene sowie für die Unterscheidung zwischen „eigenem“ und Fremdcode zur Verfügung gestellt werden.

Detailansicht

Um den Entwickler zusätzlich auch im Hinblick auf Code Smell zu unterstützen werden vom Analyseframework zusätzliche Informationen über den Code gesammelt. Diese sollen in einer eigenen Detailansicht dargestellt werden und damit dem Entwickler Stellen aufzeigen, an denen Refactoring durchgeführt werden sollte. Diese Zusatzinformationen müssen für die einzelnen Knoten aus dem Analyseergebnis ausgelesen und dann entsprechend auf Klassen- und Funktionsebene aggregiert werden. In der Detailansicht können diese einzelnen Werte dann zur jeweils ausgewählten Klasse oder Funktion angezeigt werden.

To-do-Ansicht

Zusätzlich zur klassenzentrierten Ansicht soll eine globale Ansicht implementiert werden, in der die „schlimmsten“ Klassen und Funktionen als eine Art To-do-Ansicht dargestellt werden. Dazu muss für jeden Klassen- und Funktionsknoten analysiert werden wie viele der gesammelten Zusatzinformationen als „potentieller Code Smell“ klassifiziert sind. Die Klassen und Funktionen mit den meisten Treffern sollen dann als Liste ausgegeben werden, um dem Entwickler Stellen aufzuzeigen an denen sich unter Umständen Refactoring besonders lohnt.

Testszenarien

In der abschließenden Testphase muss berücksichtigt werden, dass das Analysetool einen Plug-In-ansatz verfolgt und man daher zusätzliche Abhängigkeiten schafft, die getestet werden müssen. Der eigentliche Test muss also sinnvollerweise in mehrere Teile aufgespalten werden, um die einzelnen Bestandteile für sich und dadurch das Gesamttool zu testen.

Testbasis

Für einen sinnvollen Test ist es zuerst nötig eine Testbasis zu schaffen, gegen die getestet werden kann. Da es sich beim vorliegenden Programm um eines handelt, dass größere Datenmengen – Legacy Code ist in der Regel doch umfangreicher – bearbeiten muss, ist es sinnvoll als Basis eine entsprechend umfangreiche Codebase auszuwählen. Der Nachteil dabei ist, dass bei der umfangreichen Eingabemenge auch eine relativ große Ausgabemenge produziert wird, die man nicht sinnvoll komplett validieren können wird. Aus diesem Grund soll hier ein zweigleisiger Ansatz benutzt werden, der sich auch in der Testbasis widerspiegelt.

- Schaffung einer eigenen kleinen Codebase mit bis zu 300 Lines of Code, die alle relevanten Eigenschaften beinhaltet, die das Analyseframework auslesen soll. Diese muss dabei keinen sinnvollen Code enthalten, sondern lediglich entsprechende

Funktionsdefinitionen, Funktionsaufrufe, Verschachtelungen und ähnliche Kontrukte, die ausgelesen werden sollen. Hierbei kann die komplette Ausgabemenge validiert werden.
- Heranziehen einer umfangreichen Open Source Codebase mit mindestens 5000 Lines of Code aus Quellen wie etwa github, bei der nur noch stichprobenartig die Ausgabemenge validiert wird. Auf Basis dieser Codebase lassen sich dann auch das Verhalten mit größeren Datenmengen und die Performanz beurteilen.

Bei Bedarf müssen hier entsprechend mehrere Testbasen der jeweiligen Kategorie benutzt werden um unterschiedliche Sprachen und ähnliches abzudecken.

Framework Test

Als erster Schritt soll das zentrale Framework getestet werden. Hierbei ist zu beachten, dass das Framework externen Code in Form der einzelnen Sprach-Plug-Ins aufruft und die Rückgabewerte weiter verarbeitet. Da diese Plug-Ins auch von anderen Entwicklern geschrieben werden können und auch sollen muss an dieser Stelle verstärkt geprüft werden, ob das Framework auch mit fehlerhaften Rückgabewerten umgehen kann.

Die Testcases für das Framework müssen daher diese Szenarien enthalten:

- Aufruf mit korrektem Plug-In und dazu passender Programmiersprache
- Aufruf mit einem fehlerhaften Plug-In, dass die benötigten Interfaces und Funktionen nicht implementiert
- Aufruf mit einem korrekten Plug-In und einer nicht zum Plug-In passenden Programmiersprache

Für jeden dieser Testcases, mit Ausnahme des Ersten muss der entsprechende Fehler erkannt werden und eine passende Fehlermeldung ausgegeben werden. Unabhängig vom aufgetretenen Fehler darf das Programm allerdings nicht abstürzen oder in einem instabilen Zustand zurückgelassen werden.

Zusätzlich müssen auch die eigentlichen Funktionen des Frameworks getestet werden. Um den Test unabhängig von externen Sprach-Plug-Ins zu halten soll dafür ein Dummy-Plug-In geschrieben werden, dass ein vorher definiertes gültiges Ergebnis an das Framework zurückgibt. Nach der Speicherung durch das Framework kann dann geprüft werden, ob die erwartete Ausgabe für die vorliegende Eingabe korrekt ist.

Plug-In Test

Anschließend kann die Funktion der einzelnen als Beispielimplementationen vorliegenden Sprach-Plug-Ins getestet werden. Dazu werden wieder die oben definierten Testbasen herangezogen und damit folgende Testcases für jedes Plug-In ausgeführt:

- Analyse einer kleinen Codebasis und Speicherung des Ergebnisses. Hierbei kann komplett geprüft werden, ob alle Punkte aus der Codebasis korrekt erkannt und in der Ergebnisdatei gespeichert wurden.
- Analyse einer großen Codebasis mit anschließender Speicherung. Hier soll nur noch stichprobenartig geprüft werden, ob verschiedene Punkte korrekt bearbeitet wurden.

Zusätzlich soll das Verhalten und die Performance bei größeren Codebases geprüft werden.

Visualisierungstool

Der Test des Visualisierungstools soll analog zum Analysetool mit zwei Testbasen unterschiedlicher Größe durchgeführt werden. Der Einfachheit halber können an dieser Stelle die validierten Ergebnisse des vorherigen Tests wiederverwendet werden.

Da das Visualisierungstool weitestgehend mit sprachunabhängigen Daten arbeitet ist es an dieser Stelle nicht zwingend notwendig das Tool gegen jede als Plug-In implementierte Sprache zu testen. Vielmehr soll der Test einerseits jeweils mit mehreren kleinen Testbasen durchgeführt werden, die es erlauben alle vorhandenen Features komplett zu testen und anderseits mit mehreren großen Testbasen um wiederum das Verhalten mit großen Datenmengen zu prüfen.

Mit den einzelnen Testbasen sollen dabei jeweils diese Testszenarien getestet werden:

- Navigieren in der Abhängigkeitsansicht durch mehrere, voneinander abhängige Knoten
- Wechseln zwischen den verschiedenen Ansichten auf Funktions- und Klassenebene
- Prüfen der Filtermöglichkeit mit verschiedenen „Sprachgruppen“, die die Filterung nach Namespaces, Dateiebene oder nach Eigen- und Fremdcode zulassen
- Aufruf der Detailansicht für verschiedene Knoten mit unterschiedlichen gefundenen Zusatzinformationen
- Prüfen der To-do Ansicht durch eine Testbasis mit vorher festgelegten Werten für die zu findenden Zusatzinformationen

Codeanalyse Framework - Basis

Im folgenden Abschnitt soll näher auf den ersten Bereich des Codeanalyse Tools eingegangen werden, das Framework. Hier sollen einige der wichtigeren Schlüsselkonzepte, sowie das Speicherformat und die grafische Oberfläche näher erläutert werden.

Abstract Syntax Tree

Da das Analyse Framework prinzipiell für viele verschiedene Programmiersprachen verwendet werden soll, muss, wie eingangs erwähnt, eine Möglichkeit geschaffen werden die zu analysierenden Daten von der jeweiligen Programmiersprache zu abstrahieren. Dafür sollen sogenannte Abstract Syntax Trees, kurz AST, verwendet werden. Unter einem AST versteht man die Darstellung der syntaktischen Struktur eines Stück Sourcecodes als Baum. Abstrakt heißt in diesem Zusammenhang, dass nicht jede Einzelheit aus dem Sourcecode dargestellt wird, sondern nur die wesentlichen Elemente. Es werden also beispielsweise Schleifen, Funktionsaufrufe oder Variabelzuweisungen aufgenommen, nicht aber zum Beispiel Klammern oder Access Modifier wie public und private. Durch diese Abstraktion kann ein AST allerdings auch unabhängig von der zugrunde liegenden Programmiersprache gehalten werden. Dazu muss lediglich jeder programmiersprachen-spezifische Begriff sofern möglich in einen entsprechenden generischen Term übersetzt werden.

Da es für das vorliegende Framework keine Einschränkung bezüglich der möglichen Programmiersprachen geben soll, müsste man an dieser Stelle normalerweise ein Superset an generischen Begriffen verwenden um alle potentiellen sprachspezifischen Begriffe abdecken zu können. Für diese Version des Analyseframeworks ist allerdings genau definiert welche Visualisierungen es genau geben soll und damit auch welche Begriffe nur gesammelt werden müssen. Daher muss nur ein relativ kleines Subset an generischen Begriffen definiert werden und alle übrigen sprachspezifischen Begriffe werden auf einen Default Wert abgebildet.

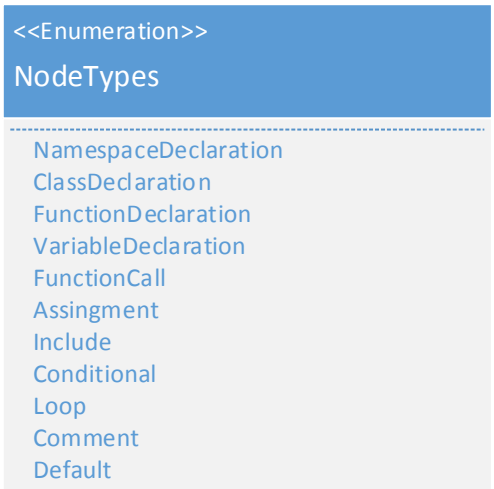

Abbildung 1 - Enumeration für verschiedene Knotentypen

Die erste Gruppe an generischen Begriffen die gebraucht werden, bezieht sich auf die hierarchische Struktur und Gliederung des Codes, also Blöcke wie Klassen oder Methoden. Für das vorliegende Framework wird die Anzahl der verwendeten Blockhierarchien auf drei

festgelegt, jeweils eine Namespace-, eine Klassen- und eine Funktionsebene. Alle zusätzlich existierenden Ebenen, wie beispielsweise regions in .NET oder eine Dateiebene werden hier ignoriert, das heißt auch nicht auf einen Default Wert abgebildet.

Ein weiteres Strukturelement welches erfasst werden muss betrifft externen Code, der in Form externer Bibliotheken oder Include-Dateien eingebunden wird. Dazu werden alle externen, eingebundenen Bestandteile in einem Knoten vom Typ Include hinterlegt, der jeweils den Pfad zum externen Code als erweitertes Attribut enthält.

Zusätzlich müssen die üblichen Bestandteile von Sourcecode, wie etwa Variablendeklarationen und -zuweisungen, Kontrollstrukturen oder Schleifen erfasst werden. Jeder dieser Programmbestandteile wird jeweils in einem Knoten mit eigenem Typ erfasst. Darüber hinaus können auch Kommentare und Anmerkungen als Knoten vom Typ Comment erfasst werden.

Ein Problem bei der Verwendung von Abstract Syntax Trees, wie bereits weiter oben erwähnt ist die Darstellung von Funktionsaufrufen als Baumknoten. Funktionen werden in der Regel an vielen Stellen innerhalb eines Programms aufgerufen, können aber – so will es die Definition als Knoten eines Baums – nur an einer Stelle im Baum hängen, also nur einen Parentknoten haben. Um diese Problematik zu umgehen wird eine Funktion nur jeweils an einer Stelle im Baum eingetragen, an genau der Stelle an der die Funktion selbst definiert wird. Jeder Funktionsaufruf wird dann als Knoten vom Typ FunctionCall hinterlegt, der den eindeutigen Namen und den Pfad der aufgerufenen Funktion als Attribut enthält. In der Abbildung unten ist diese Variante für einen rekursiven Funktionsaufruf dargestellt. Hier hat die Funktionsdeklaration die ID 1 und als untergeordnetes Element einen Funktionsaufruf. Über das ExtendedAttribute ist dann hinterlegt, welche Funktion aufgerufen wird, in diesem Fall die Funktion mit der ID 1. Diese Verbindung ist allerdings nicht als Verbindungslinie im Baum modelliert.

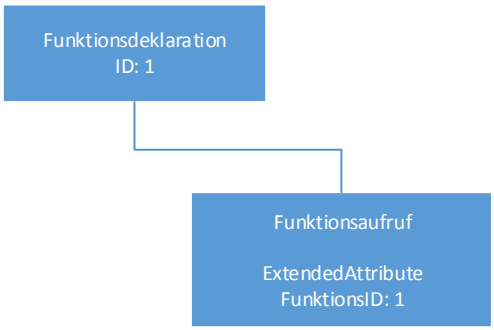

Abbildung 2 - TreeNodes für einen rekursiven Aufruf

Auf diese Weise ergeben sich keine unerlaubten Zirkularitäten in den Abstract Syntax Trees und die Funktionsverweise können bei der Visualisierung über die Pfade einfach aufgelöst werden. Bei der Analyse muss hier allerdings genau darauf geachtet werden die Pfade der einzelnen Funktionen korrekt zu ermitteln, falls beispielswiese Funktionen aus anderen Bibliotheken oder gleichlautende Funktionen mit verschiedenen Übergabeparametern verwendet werden.

Abschließend soll noch ein letzter Begriff für alle übrigen Elemente im Sourcecode definiert werden. Unter Knoten vom Typ Default werden alle Elemente aufgehängt, die zwar für die Visualisierung von Interesse sind, aber keinem eigenen Knotentyp zugewiesen werden können.

Diese Begriffe werden als Enumeration definiert und dann jeweils als Knotentyp-Attribut in jedem Knoten vergeben.

Erweiterte Attribute

Im vorherigen Kapitel wurde bereits erwähnt, dass für einige Knoten sogenannte erweiterte Attribute gespeichert werden müssen, beispielsweise der Pfad für eine externe Bibliothek in einem Include Knoten. In diesen Fällen handelt es sich um Informationen die essentiell für den jeweiligen Knoten sind und direkt zur weiteren Analyse oder Visualisierung benötigt werden – so ist ein externer Pfad wichtig um die genauen Namen und Pfade von extern aufgerufenen Funktionen zu ermitteln.

Darüber hinaus soll das Framework aber auch in der Lage sind zusätzliche Merkmale des Codes zu sammeln, die nicht direkt für die Funktionalität des Programms nötig sind. Dabei kann es sich etwa um die Anzahl an Codezeilen in einem Blockelement oder die Anzahl von Übergabeparametern einer Funktion handeln.

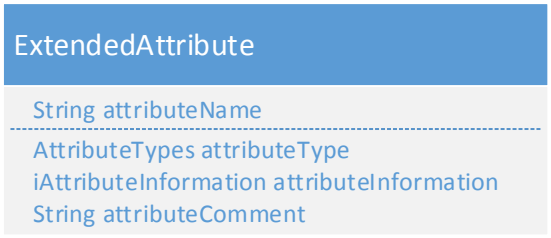

Abbildung 3 - Klassendefinition für erweiterte Attribute

Diese erweiterten Attribute werden in einer Liste von ExtendedAttributes zu jedem Knoten gespeichert. Jedes Attribut hat dabei einen je Knoten eindeutigen Namen, einen Typ und die jeweilige Information selbst. Der Typ gibt dabei an um welche Art von Attribut es sich dabei handelt, beispielsweise eine zusätzliche Information zum NodeType oder eine Eigenschaft des Codes, wie die Zeilenanzahl. Da es sich bei der Information selbst also um verschiedene Datentypen wie Zahlenwerte, Pfade oder Zeichenketten handeln kann wird das Informationsattribut über ein Interface abstrahiert.

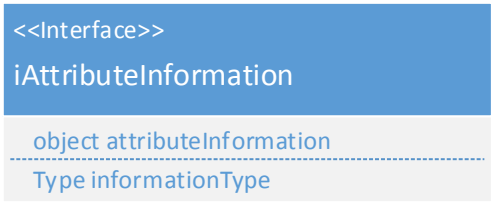

Abbildung 4 - Interface für verschiedene Attributinformationen

Für jede mögliche Ausprägung von Zusatzinformation muss daher eine eigene Klasse implementiert werden, die jeweils die Information im entsprechenden Datentyp enthält sowie die

Information um welchen Datentyp es sich handelt. Zusätzlich kann ein Kommentar angegeben werden, der eventuell nötige Zusatzinformationen enthält, die dann im Visualisierungstool für den Benutzer angezeigt werden können.

Abstract Syntax Tree - Knoten

Alle gesammelten Informationen und erweiterten Attribute zu jedem Knoten werden in einer eigenen TreeNode-Klasse gekapselt. Diese ist über eine global eindeutige ID identifizierbar und enthält den jeweiligen Typ und die Kategorie als Attribut der Klasse. Der Inhalt des Knotens, also beispielswiese der Funktionsheader oder die Variablenzuweisung wird als nodeContent gespeichert und bei der Visualisierung als eigentlicher Knotenname verwendet.

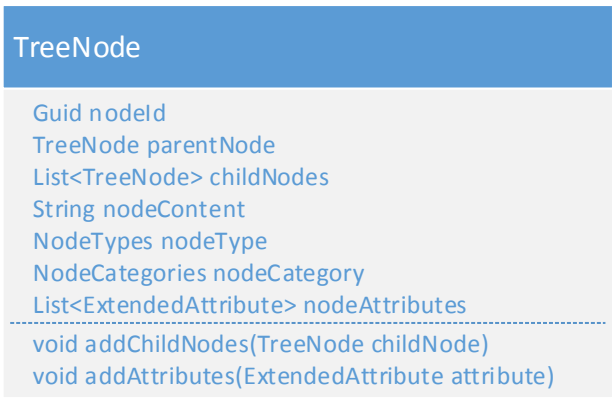

Abbildung 5 - Klassendefinition für Knoten in den Abstract Syntax Trees

Um die Baumstruktur abzubilden enthält jeder Knoten zwei Attribute die jeweils die Verbindungen im Baum nach „oben“ und „unten“ darstellen. Jeder Knoten verfügt also über genau einen übergeordneten Elternknoten und eine Liste an Kindknoten.

Würde man in einem Knoten nur die Liste der Kindknoten speichern, nicht aber den Elternknoten, so müsste beim Einfügen eines neuen Knotens nur jeweils im Elternknoten ein weiterer Kindknoten eingefügt werden. Dadurch, dass aber sowohl die Verbindungen nach „oben“ als auch nach „unten“ gespeichert werden muss bei jedem Einfügen eines neuen Knotens jeweils jeder damit verknüpfte Knoten – also nach „oben“ und nach „unten“ – angepasst werden und nicht nur der jeweilige Elternknoten. Dies führt insgesamt natürlich zu schlechterer Performance, erlaubt es allerdings über den Baum in beide Richtungen, also sowohl von „oben“ nach „unten“ als auch von „unten“ nach „oben“ zu iterieren. Dies ist für die spätere Visualisierung nötig, da dort zum Beispiel von einer gegebenen Funktion aus sowohl angezeigt werden soll welche Funktionen diese aufruft – also eine Iteration über den Baum nach „unten“ – als auch von welchen Funktionen die gegebene Funktion aufgerufen wird – eine Iteration durch den Baum nach „oben“.

Sprach-Plug-Ins

Die eigentliche Analyse des Sourcecodes und das Erstellen der oben definierten Trees soll über sprachspezifische Plug-Ins erfolgen. Dies ist nötig, da sich die zu analysierenden Programmiersprachen zum Teil erheblich unterscheiden und es daher nicht möglich ist einen Parser für alle Sprachen zu schreiben.

Die einzelnen Sprach-Plug-Ins werden in separaten Dlls gehalten und sollen komplett unabhängig vom eigentlichen Codeanalyse Framework arbeiten. Dadurch können Sprach-Plug-Ins problemlos nachträglich entwickelt oder erweitert werden, ohne dass dafür das zentrale Codeanalyse Framework verändert werden muss.

iLanguagePlugin Interface

Damit die einzelnen Sprach-Plug-Ins aber vom Codeanalyse Framework aufgerufen werden können müssen sie eine gewisse Struktur einhalten, die dem Framework bekannt ist. Dazu wird ein Interface definiert, dass jedes Sprach-Plug-In implementieren muss und welches die Schnittstelle zum Code Framework darstellt.

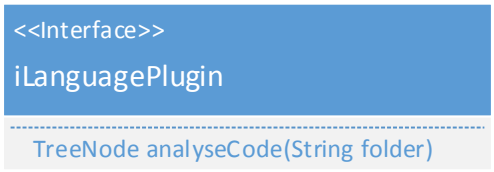

Abbildung 6 - Interface für Sprach-Plug-Ins

Dieses Interface ist sehr einfach gehalten, da der nötige Informationsaustausch zwischen Plug-In und Framework lediglich zwei Bereiche umfasst. Das Sprach-Plug-In braucht einen Dateiordner, in dem es die zu analysierenden Sourcecode Dateien finden kann und das Framework benötigt danach das entsprechende Analyseergebnis.

Zu diesem Zweck wird im Interface eine Funktion analyseCode vorgegeben, die als Übergabeparameter einen Ordnerpfad als String erwartet und schließlich das Analyseergebnis als TreeNode zurückgibt.

Code Lexer

Der erste Schritt bei der Analyse von Sourcecode ist die sogenannte lexikalische Analyse. Hierbei wird der als Plaintext vorliegende Sourcecode zeichenweise eingelesen und in logische Einheiten, die Tokens überführt. Da die lexikalische Analyse weitgehend sprachunabhängig funktioniert soll dafür ein eigenes Tool geschrieben werden, dass von den einzelnen Sprach-Plug-Ins dann aufgerufen werden kann.

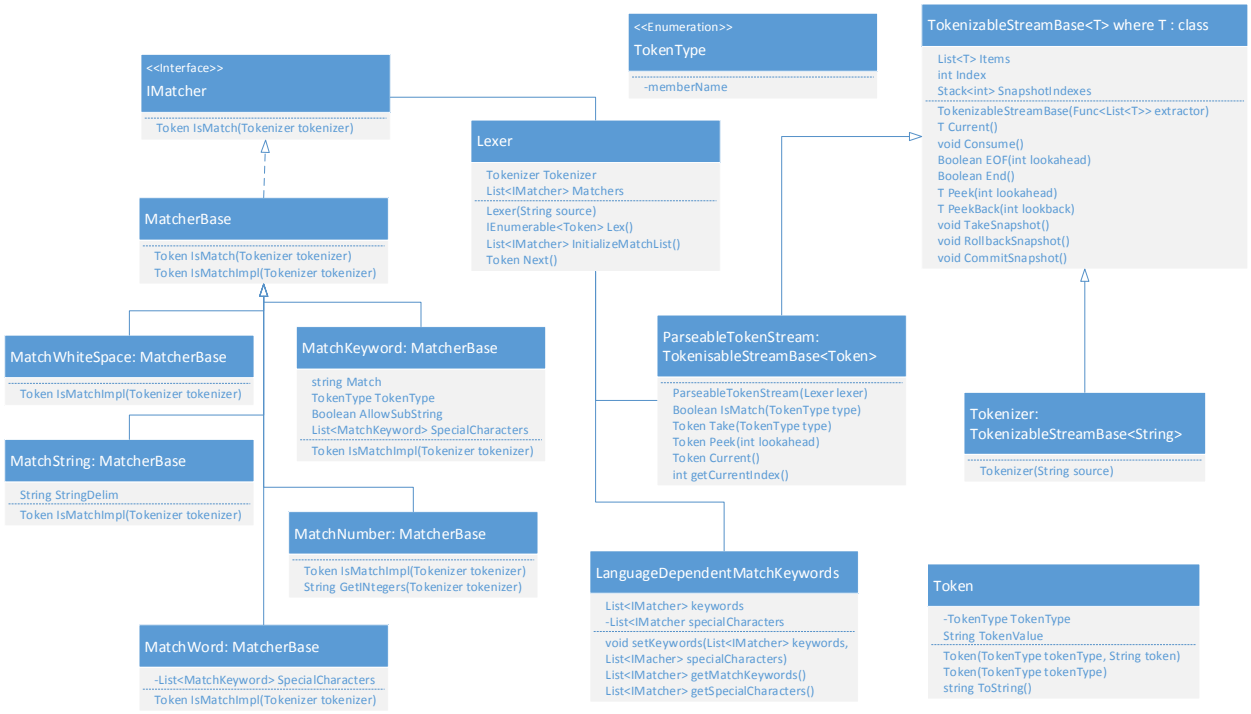

Abbildung 7 - UML Klassendiagramm für den Code Lexer

Vereinfacht gesagt wird bei der lexikalischen Analyse die Eingabe Zeichen für Zeichen betrachtet und versucht in den eingelesenen Zeichen ein bestimmtes Token zu erkennen. Token können dabei einzelne Zeichen sein, wie öffnende und schließende Klammern verschiedener Typen, mathematische Operatoren oder auch ein Semikolon. Diese Tokens können sehr leicht erkannt werden, da lediglich ein Zeichen eingelesen werden muss und es ist entweder eines dieser Tokens oder nicht. Kompliziertere Tokens können beispielsweise Zahlen, Konstrukte wie if, while und for oder primitive Datentypen wie string, int oder float sein. Bei diesen Tokens müssen jeweils mehrere Zeichen eingelesen werden, um so zu entscheiden, ob es sich um den entsprechenden Token handelt oder nicht.

Eine Spezialform an Tokens in dieser Gruppe sind die Quoted Strings, also Zeichenketten, die durch einfache oder doppelte Anführungszeichen umschlossen werden. Hier kann beim Erkennen nicht einfach Zeichen für Zeichen eingelesen werden bis man den Token erkannt hat, oder entsprechend nicht, da der Token kein festes Aussehen hat. Quoted Strings haben lediglich definierte Anfangs- und Endzeichen, dazwischen kann alles sein. Ein ähnliches Phänomen tritt bei Klammern auf, auch dort hat man eine öffnende Klammer und – hoffentlich – eine schließende Klammer und dazwischen beliebig viele andere Tokens. Im Gegensatz zu Klammerpaaren müssen allerdings Quoted Strings als eine Einheit erkannt werden, da alles was sich innerhalb der Anführungszeichen befindet entsprechend nicht als Token erkannt werden darf. Eine Zeichenkette wie “if“ ist also nicht ein als ein öffnendes Anführungszeichen, der Token if und ein schließendes Anführungszeichen zu verstehen, sondern als ein Quoted String, dessen Inhalt an dieser Stelle lediglich genauso aussieht wie ein anderer Token aber eben keiner ist. Eine zusätzliche Schwierigkeit bei Quoted Strings entsteht durch sogenannte Escape Zeichen. So kann ein Anführungszeichen, das normalerweise das Ende eines Quoted

Strings markieren würde durch ein vorangestelltes Escape Zeichen sozusagen entwertet werden und als regulärer Inhalt des Quoted Strings selbst gelten. Zum Beispiel wäre bei einer Zeichenkette wie “\“\““ das Ende nicht mit dem zweiten Anführungszeichen erreicht, sondern erst mit dem vierten, da das zweite und das dritte Anführungszeichen jeweils durch das Escape Zeichen Backslash entwertet werden. Bei der Erkennung von Quoted Strings muss also sobald ein öffnendes Anführungszeichen gefunden wurde solange weitergemacht werden, bis das entsprechende schließende Anführungszeichen gefunden wird, welches nicht durch ein Escape Zeichen entwertet wurde.

Diese verschiedenen Arten von Token sind aber jeweils weitestgehend sprachunabhängig, das heißt sie kommen in sehr vielen Sprachen vor und können dadurch universell erkannt werden.

Eine letzte Gruppe von Token sind sogenannte Keywords und Special Characters, die jeweils sprachspezifisch sind. Darunter können sich Access Modifier wie private, public oder internal befinden, Strukturelemente wie class, namespace oder function oder aber auch bestimmte Zeichenkombinationen wie „->“ oder Komparatoren wie „<>“ und „!=“. Da diese Art von Tokens sprachspezifisch ist, kann sie nicht im Lexer selbst definiert werden, sondern muss entsprechend durch das Sprach-Plug-In von außen übergeben werden.

Zusammenfassend wird also ein Tool benötigt, dem sprachspezifische Keywords und Special Characters übergeben werden und welches damit die lexikalische Analyse des zu analysierenden Sourcecodes vornimmt. Das entsprechende Ergebnis, eine Liste an erkannten Tokens wird dann an das Sprach-Plug-In zur weiteren Verarbeitung zurückgegeben.

Ergebnis der lexikalischen Analyse als ParseableTokenStream

Die erkannten Tokens in der Ergebnissemenge werden als eigene Klasse modelliert, die jeweils den Typ der Tokens und den Inhalt als Attribute enthalten.

Abbildung 8 - Klassendefinition für einen Token

Um die Arbeit mit der Ergebnismenge, die als einfache Liste an Tokens zurückgegeben werden könnte zu erleichtern, soll eine Wrapper Klasse für diese Liste geschaffen werden, die grundlegende Funktionen für das Iterieren durch die Liste bereitstellt.

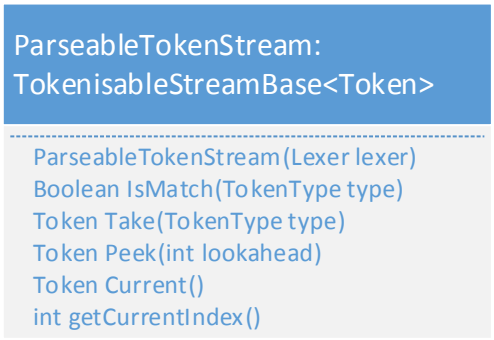

Abbildung 9 - Klassendefinition für den ParseableTokenStream

Darunter sollen einfache Funktionen sein, wie der Zugriff auf das aktuelle Token oder den Index des entsprechenden Tokens. Zusätzlich wird eine Funktion benötigt, die das weiterschreiten zum nächsten Token ermöglicht, solange noch Token vorhanden sind. Diese Zugriffsmöglichkeiten sind prinzipiell nötig um mit der Ergebnismenge zu arbeiten, würden allerdings von einer normalen Liste auch bereitgestellt werden.

Über die normalen Möglichkeiten einer Liste hinaus soll es daher noch eine zusätzlich Funktion geben, die den Abgleich von TokenTypen erleichtert. Häufig ist bei der Analyse der eigentliche Inhalt des Tokens weniger von Interesse als die Art des Tokens. So ist für die Analyse von Bedeutung ob der Token eine Funktionsdefinition oder eine Wertzuweisung ist, aber, zumindest für den Anfang nicht wie genau die Funktion heißt, oder welcher Wert zugewiesen wird. Bei einer normalen Liste müsste für diese Abfrage der entsprechende Token geholt werden und dann geprüft werden ob das TokenTyp Attribut des Tokens vom jeweiligen Typ ist. Zur Erleichterung soll dies hinter einer Funktion verborgen werden. Man kann dadurch über die IsMatch Funktion direkt prüfen, ob der aktuelle Token einen bestimmten Typ aufweist oder nicht.

Der wohl größte Unterschied zu normalen Listen liegt aber in einer Funktionsgruppe rund um das Peek Konzept. Bei der Analyse der Token ist es häufig von Bedeutung in welchem Kontext sich der aktuelle Token befindet, also welche Token befinden sich jeweils davor oder dahinter. Als einfaches Beispiel mag hier eine öffnende Klammer dienen, die jeweils eine andere Bedeutung hat, ob sie nun auf eine Funktionsdeklaration oder eine if Abfrage folgt. Nach einer Funktionsdeklaration würden in der Regel entsprechende Übergabeparameter folgen, bei einer if Abfrage die eigentliche Bedingung. Um dies zu gewährleisten werden zwei Funktionen bereitgestellt, die jeweils das „spähen“ – im englischen übersetzbar mit peek, daher die entsprechenden Funktionsnamen Peek und PeekBack – vom aktuellen Token nach vorne oder nach hinten realisieren. Aufbauend auf diesem Peek Mechanismus lassen sich weiterer Funktionen ableiten, die die Arbeit mit der Ergebnisliste an Tokens erleichtern können.

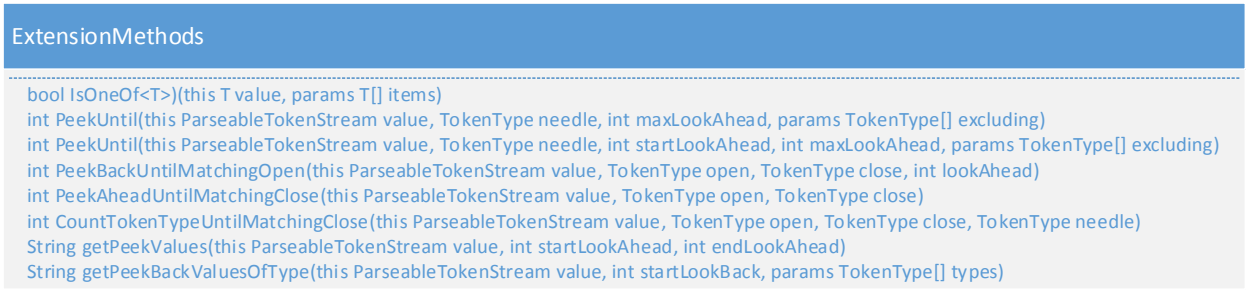

Abbildung 10 - Klassendefinition für die ExtensionMethods

Ein zentrales Strukturelement in vielen Programmiersprachen sind Blöcke, in .NET Sprachen wird dieses Konzept beispielweise als Scope bezeichnet. Dabei handelt es sich um Einheiten, die eine Ansammlung von Codeelementen gegenüber der „Außenwelt“ abgrenzen. So sind etwa Variablen, die innerhalb eines solchen Blocks deklariert werden nur dort gültig und nicht außerhalb, oder ein Block wird dazu benutzt eine aufrufbare Einheit mehrerer Codeelement zu schaffen. Das einfachste Beispiel für einen Block dürfte die Funktion sein. Sie wird in der Regel durch bestimmte Zeichen abgegrenzt, seien es nun geschweifte Klammern oder bestimmte Formen der Einrückung und stellt damit eine aufrufbare Einheit dar. Zusätzlich sind – in vielen Programmiersprachen zumindest – Variablen, die innerhalb einer Funktion definiert werden auf diese beschränkt. Neben Funktionen gibt es aber auch noch weitere Blöcke wie etwa den Code innerhalb einer if Abfrage oder einer Schleife, größere Strukturelemente wie Klassen und Namespaces oder vieles andere mehr.

Bei der Analyse dieser Blöcke ist es natürlich nun wichtig zu wissen, wie weit der jeweilige Block geht, wo also sein Anfangselement und sein Endelement sitzen. Um das jeweilige passende Element zu finden muss die Liste der Tokens solange nach vorne durchlaufen werden, bis das Endelement gefunden ist. Dabei ist darauf zu achten, dass in den meisten Programmiersprachen Blöcke auch ineinander verschachtelt sein können, man muss also beim Durchgehen der Liste genau darauf achten, ob das gefundene schließende Element tatsächlich zum gewünschten Anfangselement passt, oder nur das schließende Element für einen der untergeordneten Blöcke ist. Um diese Funktionalität zur Verfügung zu stellen sollen jeweils eine Funktion geschaffen werden, die für ein gegebenes Anfangselement das entsprechende Endelement und umgekehrt finden kann.

Bleibt man bei den Blöcken, ist es häufig nicht nur von Interesse das jeweilige schließende – oder öffnende – Element zu finden, sondern man möchte auch wissen was sich innerhalb des Blocks befindet. So könnte es für die Analyse beispielsweise von Bedeutung sein wie tief die Verschachtelung innerhalb eines Blockes ist, oder wie viele if Abfragen oder Schleifen Konstrukte es in einer Funktion gibt. Zu diesem Zweck wird eine Abwandlung der oben genannten Funktionen benötigt, welche während sie das jeweilige Ende des Blocks sucht auch zählen kann wie viele Token eines bestimmten Typs im untersuchten Block vorkommen.

Zum Abschluss soll nun noch eine Funktion realisiert werden, die das „vorausspähen“ bis zum nächsten Token eines bestimmten Typs ermöglicht. Dies ist vor allem dann von Bedeutung, wenn man auf ein bestimmtes Token trifft, bei dem man weiß, dass es von einem weiteren

spezifischen Token gefolgt werden muss. Wenn man also beispielsweise in einer .NET Sprache auf einen try Block trifft, muss es auch einen darauffolgenden catch Block geben. Um den Zugriff auf diesen catch Block zu vereinfachen dient die PeekUntil Funktion, die solange nach vorne spähen soll, bis sie den entsprechenden Block gefunden hat.

Erstellen eines Sprach-Plug-Ins

Nachdem nun alle Vorarbeiten für die Sprach-Plug-Ins abgeschlossen sind bleibt noch das Erstellen des Sprach-Plug-Ins selbst. Wie bereits mehrfach erwähnt sind die Sprach-Plug-Ins selbst abhängig von der jeweiligen Sprache, die sie analysieren sollen und daher mitunter sehr unterschiedlich. Daher ist es schwierig eine allgemeine Aussage über das Erstellen der Sprach-Plug-Ins zu tätigen. Zu Demonstrationszwecken soll hier ein Sprach-Plug-In realisiert werden, das als allgemeiner Proof of Concept für das Codeanalyse Framework dient. Je nach Bedarf ist es dann möglich später Sprach-Plug-Ins, die auf die jeweiligen Sprachanforderungen der Benutzer zugeschnitten sind zu ergänzen.

Im Rahmen dieser Arbeit soll daher zuerst ein Sprach-Plug-In erstellt werden, dass die Sprache C#.NET4.0 analysieren kann. Diese Sprache wurde zum einen ausgewählt, da sie eine relativ moderne objekt-orientierte Sprache ist, die einige interessante Analyseansätze bietet. So ist C# eine Sprache, die alle bereits beschriebenen Hierarchieebenen – Funktions-, Klassen- und Namespaceebene – umsetzt. Zudem setzt die Sprache auf Techniken wie Strong Typing, Methodenüberladung und umfangreiches Exception Handling. Der zweite, pragmatischere Grund ist, dass das Analyseframework selbst in C# realisiert ist, und damit das Framework auch auf sich selbst angewendet werden kann.

Beim Umfang einer Sprache wie C# mit ihren vielfältigen Techniken sollte klar sein, dass eine komplette Abdeckung all dieser Aspekte bei der Analyse enorm umfangreich ist und – zumindest im Rahmen dieser Arbeit – nicht möglich ist. In dieser Iteration soll daher nur eine Auswahl an Elementen abgedeckt werden, die – nach Meinung des Autors – einen guten und ersten Einstieg in die Analyse von Legacy Code bieten.

Zuerst allerdings müssen die Vorarbeiten für das Sprach-Plug-In – dies ist allgemeingültig für jedes Sprach-Plug-In – durchgeführt werden. Als Anfang sollte ein neues Projekt angelegt werden, in dem das Sprach-Plug-In entwickelt werden soll. Prinzipiell können zwar mehrere Sprach-Plug-Ins in einer Dll umgesetzt werden, es empfiehlt sich allerdings aus Gründen der Modularität für jedes Plug-In eine eigene Dll zu verwenden. Anschließend wird eine Klasse benötigt, die das iLanguagePlugin Interface implementiert und die dafür notwendige analyseCode Funktion enthält. Der Name der Klasse kann beliebig gewählt werden, sollte allerdings einen sprechenden Namen haben, da der Klassenname gleichzeitig der Anzeigename des Plug-Ins im Code Analyzer GUI sein wird.

Nun kann der eigentliche Code zur Analyse der Sourcecode Dateien erstellt werden. Als erstes bietet es sich hier an das Einlesen des Sourcecodes selbst anzugehen. Für das zu realisierende C# Plug-In gibt es hier bereits die erste Besonderheit zu beachten. C# Projekte werden durch zentrale csproj Dateien definiert, in diesen wird festgehalten welche externen Referenzen eingebunden werden und auch welche Sourcecode Dateien überhaupt zum Projekt

gehören. Es müssen also hier im übergebenen Dateiordner alle csproj Dateien gefunden werden und dann alle dort definierten Sourcecode Dateien ausgelesen werden.

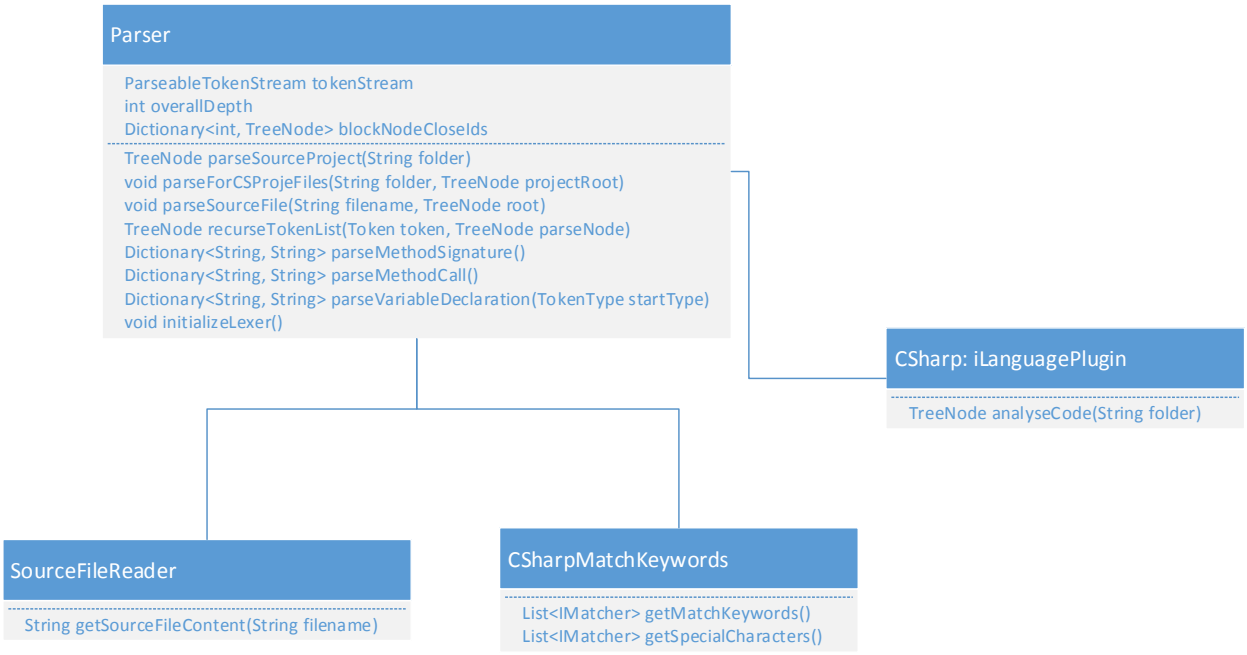

Abbildung 11 - UML Klassendiagramm für den CSharp Parser

Als erster Analyseschritt soll anschließend die Struktur des Codes erfasst werden und damit der Abstract Syntax Tree erstellt werden. Dazu kann entweder ein eigener Lexer und Parser erstellt werden, oder man bindet den in den vorherigen Kapiteln vorgestellten Standardlexer ein und übernimmt lediglich das Parsen selbst. Für das vorliegende C# Plug-In soll der vorhandene Lexer verwendet werden und lediglich das Parsen selbst im Plug-In realisiert werden.

Dadurch kann hier bereits auf eine Liste an Tokens zurückgegriffen werden und mit diesen weitergearbeitet werden. Diese Liste wird dann in einer rekursiven Funktion durchlaufen, in der für jeden Token geprüft wird, ob dieser zu einem bestimmten Muster, beispielsweise einer Funktionsdeklaration, einer Variablenzuweisung oder einer if Abfrage passt.

Für jeden Token, der auf eines dieser Muster passt wird anschließend eine Funktion aufgerufen, die sämtliche nötigen Informationen für dieses Muster ausliest. Wird beispielsweise eine Funktionsdeklaration erkannt, so wird über die bereits oben erwähnten Peek Mechanismen versucht zum Beispiel den Funktionsnamen, die Übergabeparameter oder den Rückgabetyp auszulesen. Mit diesen Informationen kann dann ein entsprechender TreeNode erstellt werden, der an den entsprechenden Platz im Abstract Syntax Tree eingefügt wird. Auf diese Weise erhält man einen kompletten Baum über den zu analysierenden Sourcecode, wie er unter anderem in der Abhängigkeitsansicht des Visualisierungsframeworks benötigt wird.

Darauf aufbauend können dann die zusätzlichen erweiterten Attribute erfasst werden. Wie erwähnt soll hier für den Anfang nur eine Auswahl an Attributen erfasst werden, die einen guten Einstieg gewährleisten. Konkret soll die Anzahl der Zeilen für jeden gefundenen Block, also

Funktionen, Klassen, Schleifen oder Abfragen, sowie die Verschachtelungstiefe der einzelnen Blöcke erfasst werden. Dies gibt bereits erste Ansätze wo es Probleme im analysierten Code gibt, da je mehr Zeilen pro Block vorhanden sind bzw. je tiefer die Verschachtelung pro Block ist, desto unwartbarer wird potentiell auch der Sourcecode.

Der dadurch entstandene Baum aus TreeNodes wird anschließend über die im Interface definierte Funktion an das eigentliche Codeanalyse Framework übergeben.

Codeanalyse Framework - Code Analyzer

Der Code Analyzer ist das zentrale Element innerhalb des Analyseframeworks und verbindet die übrigen Bestandteile. Das Projekt selbst ist grob nach MVC Prinzipien aufgebaut und besteht aus einer zentralen Controller Klasse, der eigentlichen CodeAnalyzer GUI Klasse und einigen Modelklassen.

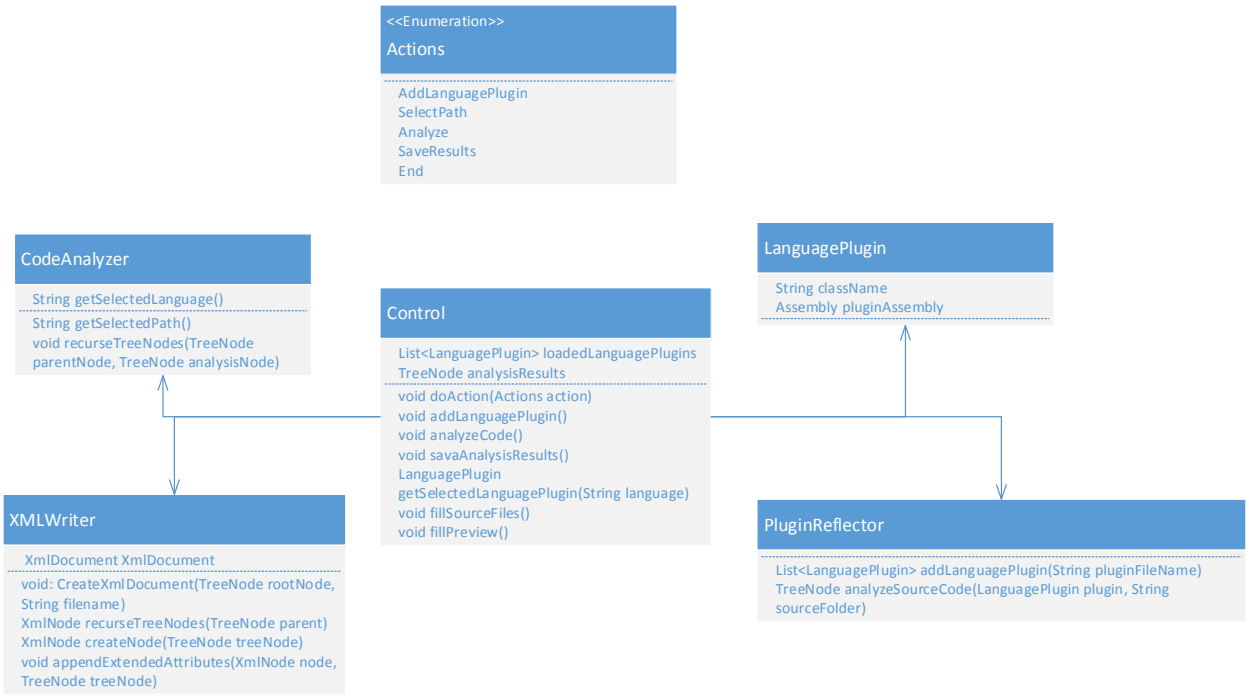

Abbildung 12 - Klassendiagramm für das Codeanalyse Framework

Unter den Modelklassen befinden sich eine Klasse, die zur Umwandlung und Speicherung der Analyseergebnisse nach XML dient und zwei Klassen, die den Plug-In Mechanismus abbilden.

XMLWriter

Der XMLWriter empfängt die Analyseergebnisse als Baum von TreeNodes und wandelt diese in die entsprechenden XML Knoten um. Dazu werden die entsprechenden XmlDocument Klassen aus dem System.Xml Namespace von .NET verwendet.

Zuerst wird der Baum rekursiv durchlaufen und für jeden TreeNode ein entsprechender XmlNode erstellt.

```
<node type="Include"
      guid="02cf2ca3-e55a-4643-b2bc-5360a4669d3f"
      content="System.Collections.Generic"
      category="Statement" />
```

Abbildung 13 - Darstellung eines XML Knotens

Anschließend werden alle zum jeweiligen TreeNode erfassten erweiterten Attribute als zusätzliche XmlNodes unter den entsprechenden Elternknoten erfasst. In den XmlNodes werden alle Eigenschaften aus den TreeNodes und ExtendedAttribute Klassen als Attribute des Xml-Knotens abgebildet.

```
<extendedAttribute type="intcodesmell"
                   name="linesofcode"
                   information="111"
                   comment="" />
```

Abbildung 14 - Darstellung eines ExtendedAttribute Knotens

Die dadurch erzeugte Xml-Struktur spiegelt dabei grob die Struktur des zuvor beschriebenen Abstract Syntax Tree wieder. An oberster Ebene befindet sich ein root Knoten, unter dem jeweils verschachtelt die einzelnen Knoten für TreeNodes und ExtendedAttributes angehängt werden. Dabei können die Knoten für TreeNodes beliebig untereinander verschachtelt werden, jeder TreeNode Knoten kann 0 – n weitere TreeNode Knoten unter sich haben. Darüber hinaus kann jeder TreeNode Knoten auch 0 – n ExtendedAttribute Knoten unter sich haben, wohingegen ExtendedAttribute Knoten selbst keine untergeordneten Knoten haben können.

```
<root>
    <node />
    <node>
        <extendedAttribute />
        <node />
        <node>
            <extendedAttribute />
            <extendedAttribute />
        </node>
    </node>
</root>
```

Abbildung 15 - Einfaches Beispiel einer möglichen XML Struktur

Das komplette XmlDocument kann abschließend unter einem gewählten Pfad und Dateinamen mittels der Save-Funktion gespeichert werden.

Plug-In Mechanismus

Für den Plug-In Mechanismus werden zwei Klassen verwendet, eine Container Klasse, die jeweils Informationen zu einem verfügbaren Sprach-Plug-In speichert und die Reflector Klasse, die die Einbindung externen Dlls übernimmt.

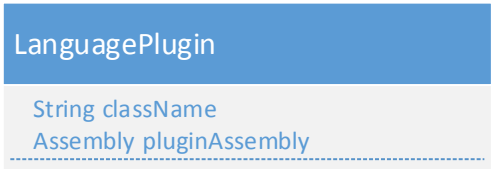

Abbildung 16 - Klassendefinition für einen LanguagePlugin Container

Die LanguagePlugin Klasse hält jeweils den Namen eines Sprach-Plug-Ins und die dazugehörige Assembly vor. Dabei ergibt sich der Name aus der jeweiligen Klasse, die das iLanguagePlugin Interface implementiert.

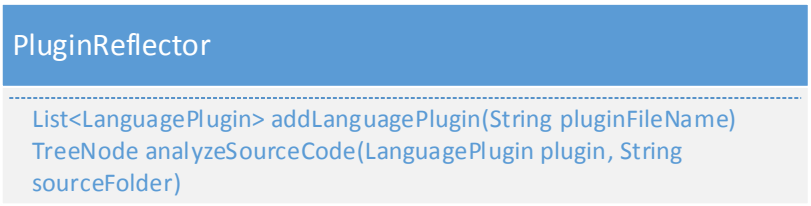

Abbildung 17 - Klassendefinition für den PluginReflector

Der PluginReflector beinhaltet zwei Methoden, die die Kommunikation mit externen Sprach-Plug-Ins bereitstellen. Mithilfe der addLanguagePlugin Methode können neue Sprach-Plug-Ins für die Analyse hinzugefügt werden. Dazu wird eine Dll über einen entsprechenden Dialog ausgewählt und jede darin befindliche Klasse untersucht. Für jede enthaltene Klasse, die das iLanguagePlugin Interface implementiert wird dann eine entsprechendes Container Objekt erstellt und das jeweilige Sprach-Plug-In kann über die Oberfläche zur Analyse ausgewählt werden.

Über die analyzeSourceCode Funktion kann dann die jeweilige sprachbezogene Analyse Methode aus dem ausgewählten Sprach-Plug-In aufgerufen werden. Dies geschieht mithilfe der Reflection Funktionalität im .NET Framework. Über einen Activator kann dabei die Instanz einer Klasse erzeugt – in diesem Fall die Instanz einer Sprach-Plug-In Klasse – und damit entsprechende Funktionen innerhalb dieser Klasse aufgerufen werden. Für diesen Fall wird entsprechend die im Interface definierte analyseCode Methode aufgerufen.

Dieser Methode wird als Parameter ein Dateiordner übergeben, in dem sich die zu analysierenden Sourcecode Dateien befinden. Als Rückgabewert wird dann das Analyseergebnis in Form eines aus TreeNodes bestehenden Baumes erwartet.

Control

Die zentrale Steuerung des Codeanalyse Frameworks wird durch die Control Klasse erledigt. Hier werden jeweils eine Liste aller geladenen Sprach-Plug-Ins und die gesammelten Analyseergebnisse vorgehalten.

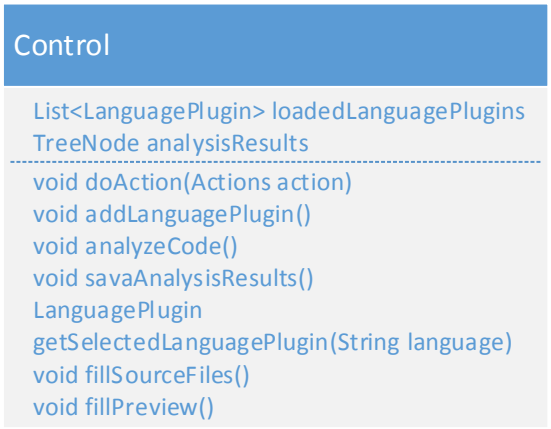

Abbildung 18 - Klassendefinition für die zentrale Control Klasse

Die Control Klasse erzeugt beim Start eine Instanz des CodeAnalyzer Views und übernimmt die weitere Steuerung der grafischen Benutzeroberfläche. Zu diesem Zweck befinden sich in der Control Klasse verschiedene Funktionen, die Teile der Oberfläche verändern oder auf entsprechende Oberflächenevents reagieren.

Sind die entsprechenden Vorbedingungen erfüllt, also beispielsweise wenn ein Dateiordner mit Sourcecode Dateien ausgewählt wurde oder wenn ein Analyseergebnis verfügbar ist, wird von der Control Klasse der entsprechende Bereich in der Oberfläche gefüllt, in diesem Beispiel also die Liste mit gefundenen Sourcecode Dateien oder die Vorschau auf das Analyseergebnis.

Um auf Events aus der Oberfläche reagieren zu können verfügt die Control Klasse über die doAction Funktion. Diese wird jedes Mal aufgerufen, sobald in der Oberfläche ein Button geklickt wird oder eine Benutzereingabe erfolgt. Jedes dieser Events ruft die doAction Methode mit einem jeweils eindeutigen Enumerationswert auf, der angibt welche Aktion ausgeführt werden soll.

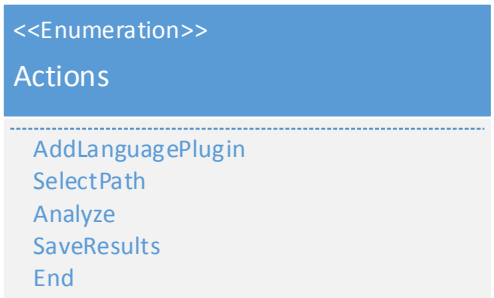

Abbildung 19 - Enumeration für mögliche GUI Events

Alle möglichen Aktionen werden in der Actions Enumeration definiert und in der doAction Methode entsprechend behandelt. Für einfache Aktionen wie beispielsweise End wird dabei einfach nur der jeweilige Befehl aufgerufen, für End ein einfaches Close(), während für umfangreichere Aktionen mehrere Hilfsfunktionen vorhanden sind. So wird für die Aktion SaveResults zum Beispiel eine Funktion aufgerufen, die den Benutzer auffordert einen

Dateiordner und Dateinamen anzugeben und anschließend das Analyseergebnis dort gespeichert.

Code Analyzer - Graphical User Interface

Die grafische Oberfläche des Code Analyzers selbst soll relativ einfach gestaltet sein und sich auf die Kernfunktionalität fokussieren. Über entsprechende Menüs in der Toolbar sollen die Speicherfunktionalität und ein Einstellungsmenü verfügbar sein. Im Einstellungsmenü können dann die Standardwerte für die einzelnen Dateipfade ausgewählt sowie neue Sprach-Plug-Ins hinzugefügt werden.

In der Hauptansicht kann anschließend der Pfad zum zu analysierenden Codeprojekt und die entsprechende Programmiersprache ausgewählt werden. Nach der Auswahl werden jeweils die gefundenen Sourcecode Dateien vorgeblendet.

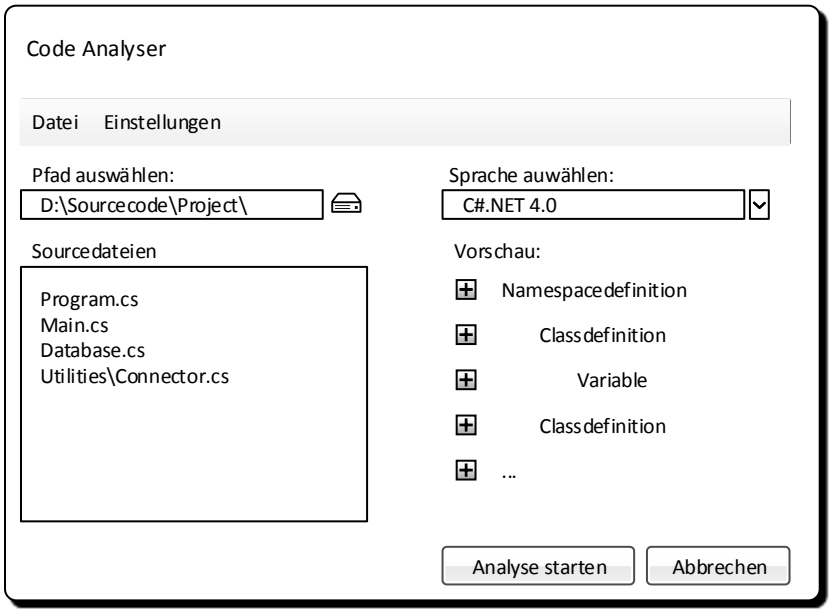

Abbildung 20 - Entwurf für die grafische Oberfläche das Code Analyzers

Über den Button „Analyse starten“ kann die eigentliche Code Analyse über das Framework mit den gesetzten Parametern gestartet werden. Nach Abschluss der Analyse wird eine vereinfachte Repräsentation des erstellten ASTs als Vorschau dargestellt. Anschließend kann das Analyseergebnis über den entsprechenden Menüpunkt für die weitere Verwendung gespeichert werden.

Visualisierungsframework

Der zweite Teil dieser Arbeit beschäftigt sich mit dem Visualisierungsframework um die gesammelten Analyseergebnisse für den Entwickler sinnvoll darzustellen. Im Folgenden soll daher näher auf die Oberflächen der einzelnen Visualisierungsansichten sowie auf die zugrunde liegende Technik eingegangen werden.

Abhängigkeitsansicht

Die Abhängigkeitsansicht soll dem Benutzer die Möglichkeit bieten die Abhängigkeiten zwischen den einzelnen Codebestandteilen, handelt es sich nun um Klassen, Funktionen oder Dateien darzustellen und entsprechend bei der Arbeit mit Legacy Code zu beachten.

Über die einzelnen Menüpunkte können in dieser Ansicht die gespeicherten Analyseergebnisse geladen, das Programm beendet und Einstellungen vorgenommen werden.

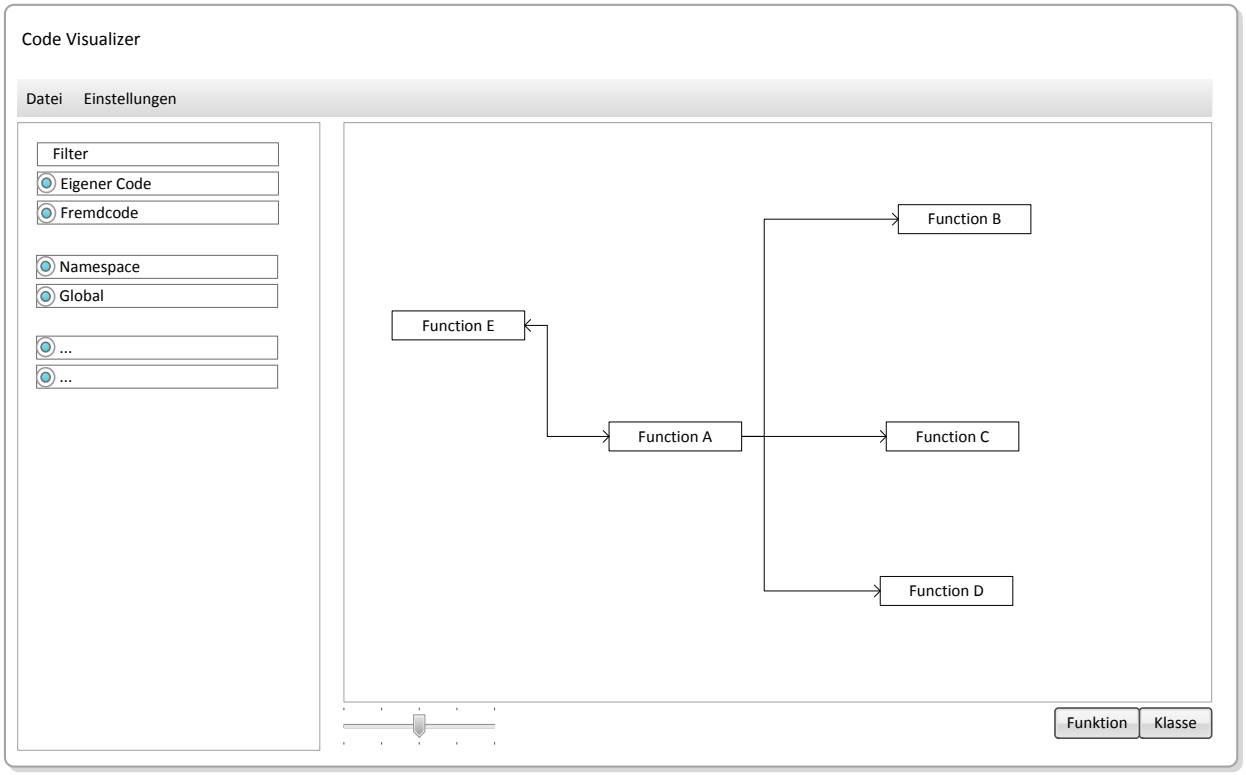

Abbildung 21 - Entwurf für die grafische Oberfläche der Abhängigkeitsansicht

Der Hauptbereich dieser Ansicht besteht aus zwei Bereichen, dem Filterbereich auf der linken Seite und der eigentlichen Baumansicht rechts. Über den Filterbereich können die bereits erwähnten Filter für die Baumansicht mit Hilfe von RadioButtons gesetzt werden.

In der Baumansicht werden die einzelnen Elemente mit ihren jeweiligen Abhängigkeiten angezeigt. Die Beziehungen werden über bi- oder unidirektionale Kanten dargestellt. Mithilfe der Kontrollelemente am rechten unteren Rand kann man zwischen der Funktions- und der Klassenansicht wechseln.

Detailansicht

Die Detailansicht wird als Erweiterung zur Abhängigkeitsansicht konzipiert. Wählt man eins der Elemente in der Abhängigkeitsansicht aus, so sollen die gesammelten Detailinformationen, die sich als erweiterte Attribute im Analyseergebnis finden dargestellt werden.

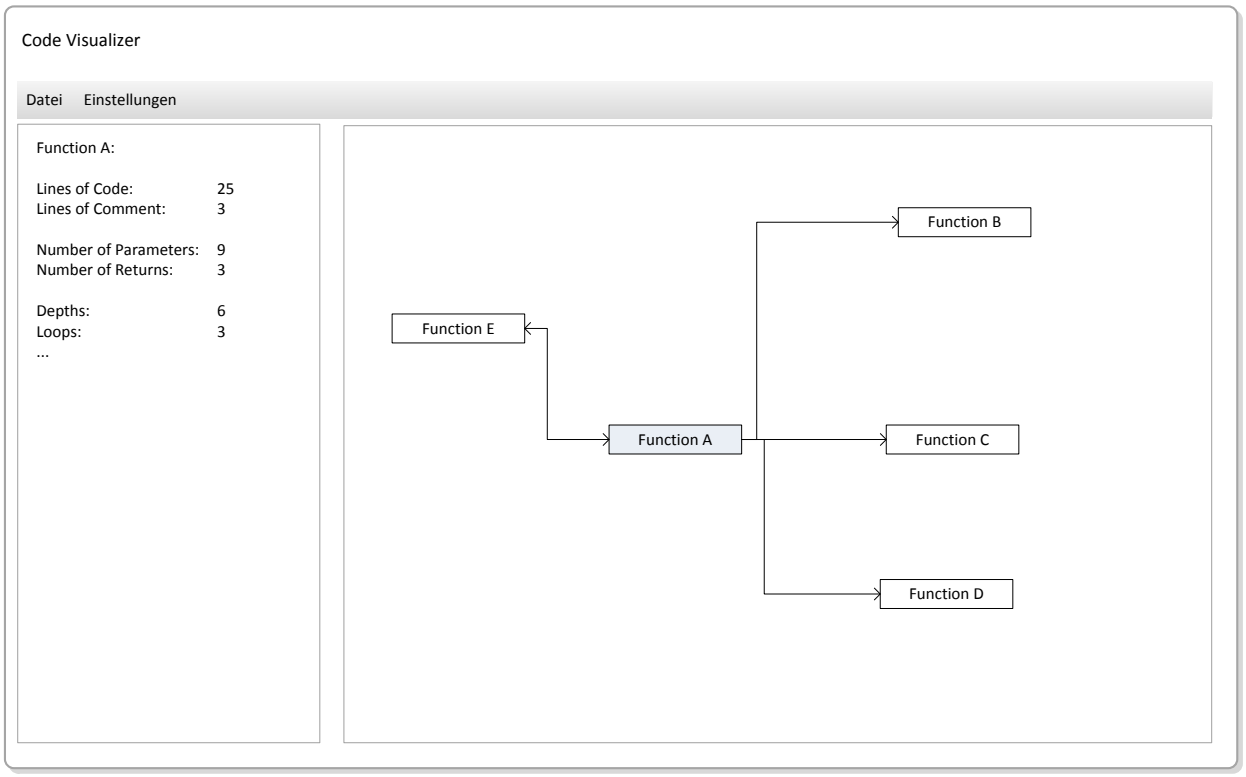

Abbildung 22 - Entwurf der grafischen Oberfläche für die Detailansicht

Das jeweils markierte Element, in obiger Abbildung also Function A wird in der Baumansicht hervorgehoben und die dazu gehörenden Detailinformationen werden in einem Panel auf der linken Seite dargestellt. Dieses Detailpanel ersetzt dabei die Filtermöglichkeiten aus der Abhängigkeitsansicht.

Die erweiterten Attribute bestehen jeweils aus den Namen der Attribute und den entsprechenden Werten, die in einer Liste dargestellt werden.

To-do Ansicht

Abschließend wird noch eine To-do Ansicht verfügbar sein. Diese soll dem Benutzer bestimmte Hot Spots in den Analyseergebnissen anzeigen, das heißt erweiterte Attribute, die quantitativ nach oben hervorstechen oder bestimmte qualitative Attribute.

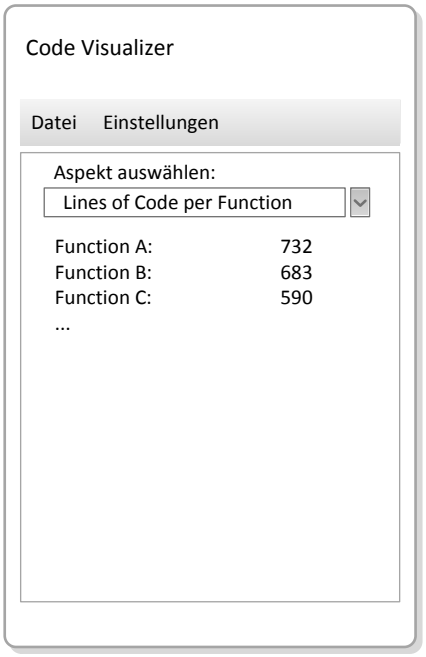

Abbildung 23 - Entwurf der grafischen Oberfläche für die To-do Ansicht

Die Art der anzuzeigenden Attribute wird über Dropdownfeld ausgewählt und die entsprechenden Ergebnisse dann in einer Liste angezeigt. Über einen Klick auf die einzelnen Einträge gelangt man dann zum jeweiligen Element in der Baumansicht.

ChartFramework

Für die Abhängigkeitsansicht wird eine Möglichkeit benötigt die einzelnen Funktions- bzw. Klassenknoten in einem Netzchart darzustellen. Hierzu wird ein Control benötigt, dass es erlaubt einzelne Knoten, in Form von Rechtecken auf einem Panel zu platzieren und diese entsprechend mit gerichteten Kanten zu verbinden. Für diese Aufgabe gibt es zwar bereits fertige Frameworks, die man einbinden könnte, diese wurden aber aus verschiedenen Gründen als ungeeignet empfunden. Viele dieser Frameworks sind als UI Control zum Zeichnen von Charts gedacht, erwarten also ihre Eingabe von einem Benutzer, der die Knoten und Kanten platziert und bieten daher keine API, die es erlauben würde Knoten und Kanten durch eine entsprechende Funktion zu zeichnen. Diese Frameworks haben in der Regel auch keine Funktionen, die die entsprechende überscheidungsfreie Platzierung der Elemente vornehmen können. Andere Frameworks dagegen wären prinzipiell mit einer API ausgestattet, die genau dies erlauben würde, haben aber dafür einen Funktionsumfang, der weit über das hinausgeht was für das vorliegende Szenario gebraucht wird und würden damit auch hohen Aufwand beim Einsatz erfordern. Aus diesen Gründen wurde sich dafür entschieden ein eigenes Tool für die Anzeige und Erstellung dieser Charts zu implementieren.

Das entstehende ChartFramework soll relativ leichtgewichtig sein und nur die allernötigsten Funktionen bieten, die für die Darstellung der Abhängigkeitsbäume nötig sind.

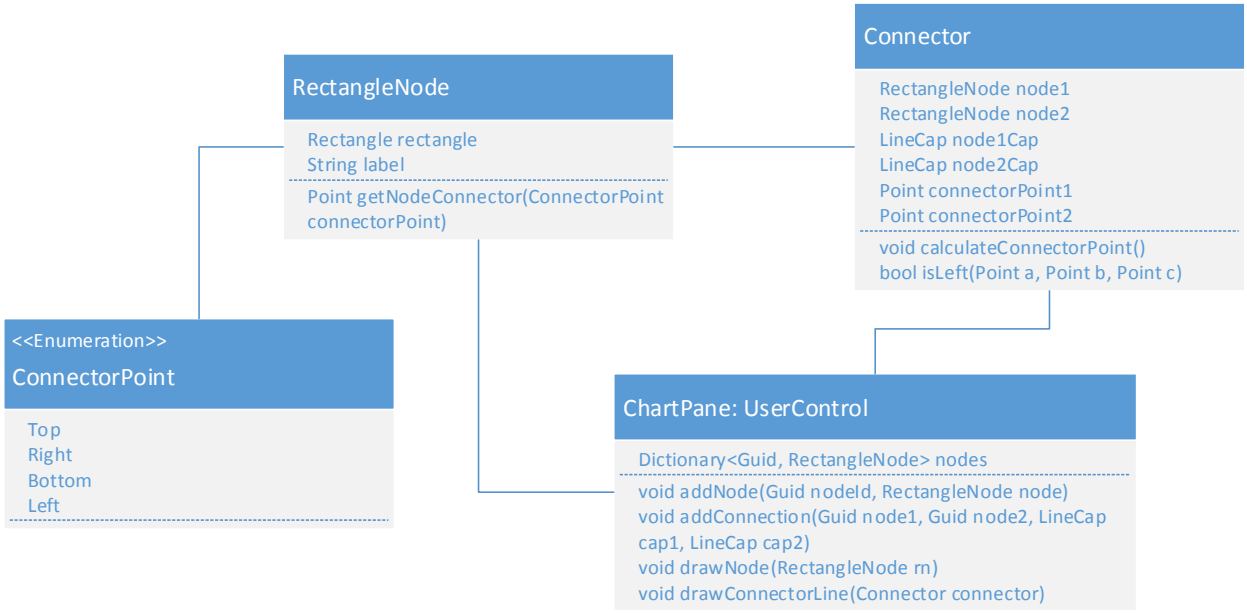

Abbildung 24 - Klassendiagramm für das ChartFramework

Im Wesentlichen werden sich auf der Zeichenoberfläche nur zwei verschiedene Typen von Elementen befinden, die jeweiligen Funktions- oder Klassenknoten, dargestellt als einfache Rechtecke und die gerichteten Verbindungslinien.

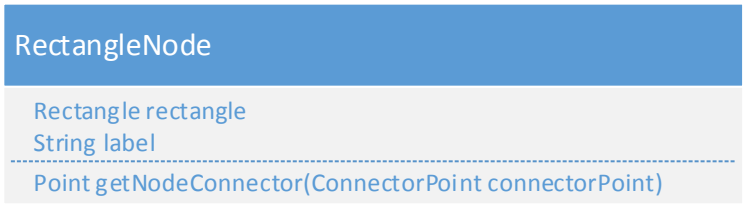

Abbildung 25 - Klassendefinition für einen Knoten

Für die einzelnen Knoten wird eine Containerklasse angelegt, in der jeweils das angezeigte Rechteck und das entsprechende Label gespeichert werden. Jeder dieser Rechtecke hat vier Ankerpunkte für Verbindungslinien, ähnlich wie dies beispielsweise in UML Tools realisiert wird, jeweils in der Mitte der vier Begrenzungslinien oben, unten, rechts und links. Über eine Funktion werden jeweils die Koordinaten des gewünschten Ankerpunktes zurückgeliefert.

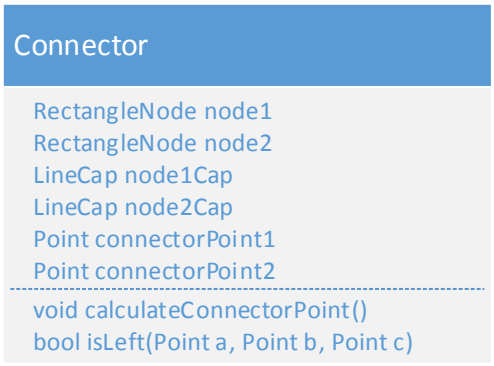

Abbildung 26 - Klassendefinition für eine Verbindungslinie

Parallel dazu werden auch die Verbindungslinien in einer eigenen Klasse definiert. Diese enthalten jeweils die beiden Knoten, die sie verbinden sollen, die Endstücke der Verbindungslinie und die beiden Ankerpunkte als Attribute. Die Endstücke der Verbindung geben dabei jeweils die Richtung der Verbindungslinie mit einem Pfeil an. Die Verbindungen können dabei einseitig gerichtet sein und damit jeweils einen Pfeil und einen Rundanker als Abschluss haben oder beidseitig gerichtet mit jeweils einem Pfeil an jedem Ende.

Die Ankerpunkte für die beiden Rechtecke werden über eine eigene Funktion berechnet. Dazu wird die Zeichenfläche ausgehend vom ersten Rechteck in vier Segmente unterteilt, jeweils an den Querachsen des Rechtecks – in der folgenden Abbildung rot dargestellt.

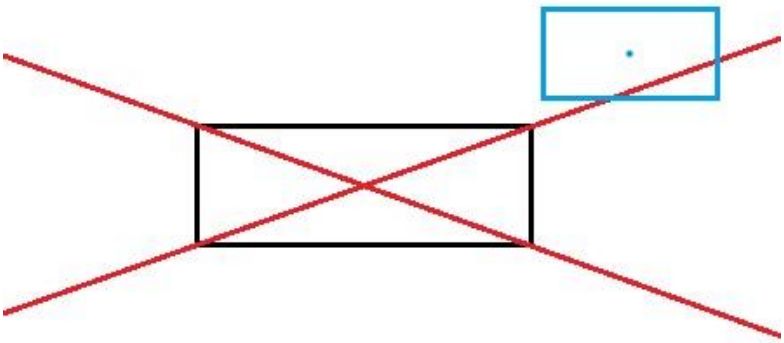

Abbildung 27 - Veranschaulichung für Berechnung der Ankerpunkte

Das zweite Rechteck anschließend platziert und geprüft, in welchem der vier Segmente die Mitte des zweiten Rechtecks – hier der blaue Punkt – liegt. Als Ankerpunkte werden dann jeweils die Punkte benutzt, die sich jeweils gegenüber liegen, wenn man das zweite Rechteck in die Mitte des Segments schieben würde, in dem es liegt. Im obigen Beispiel würde das zweite Rechteck im oberen Segment liegen, daher würde man als Ankerpunkt jeweils die Punkte Oben-Mitte von Rechteck Schwarz und Unten-Mitte von Rechteck Blau benutzen. Wäre das blaue Rechteck im rechten Segment, würde man entsprechend die Punkte Rechts-Mitte und Links-Mitte benutzen.

Als zentrale Zeichenebene schließlich dient das ChartPane. Hier werden alle gezeichneten Knoten in einer Collection vorgehalten und auf dem Zeichenpanel abgebildet.

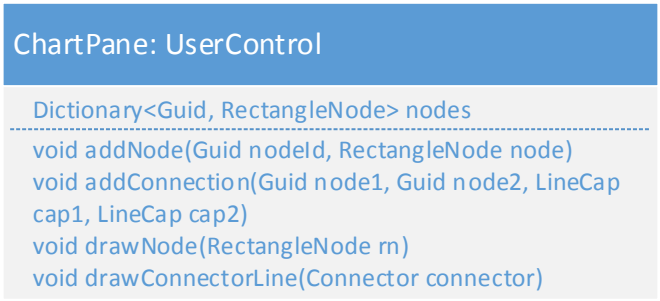

Abbildung 28 - Klassendefinition für die Zeichenebene

Über entsprechende Funktionen können neue Knoten und Verbindungslinien hinzugefügt und diese anschließend gezeichnet werden.

Die ChartPane Klasse leitet sich von einem UserControl ab und kann somit ganz einfach im Visualisierungsframework eingebunden und auf ein entsprechendes Panel gelegt werden.

Implementierung

Im folgenden Kapitel soll näher auf die Implementierung des Codeanalyse Frameworks eingegangen werden. Dazu wird zuerst ein allgemeiner Überblick über die Grundlagen gegeben um anschließen näher auf die einzelnen Teile des Frameworks einzugehen. In den jeweiligen Abschnitten zum Framework wird auch näher auf einige Algorithmen oder andere Codebereiche die von Interesse sein könnten eingegangen.

Allgemeines

Sämtliche Bereiche des Codeanalyse Frameworks werden nach objektorientierten Prinzipien implementiert. Dazu wurde als Programmiersprache C# 4.0 ausgewählt. C# ist Multiparadigmen Programmiersprache, die auf das .NET Framework von Microsoft aufsetzt. Die Version 4.0 ist dabei seit April 2010 verfügbar und inzwischen in einem sehr ausgereiften Stadium.

Als alternative Programmiersprache wurden zusätzlich Java und C++ in Betracht gezogen, diese aber aus verschiedenen Gründen wieder verworfen. C++ gelang dabei in die engere Wahl, da es dafür bereits einige Frameworks für das Parsen von Sourcecode und die Erstellung von Abstract Syntax Trees gibt. Zudem ist C++ nach wie vor eine relativ weit verbreitete Sprache, die in vielen Bereichen Einsatz findet, nicht zuletzt auch wegen ihrer guten Performance. Die sehr große Verbreitung ist auch ein Argument, dass für Java als zweite Alternative spricht. Dieser Aspekt kommt unter anderem auch dann zum Tragen, wenn man bedenkt, dass der Benutzer des Frameworks mitunter eigene Sprach-Plug-Ins schreiben muss und es daher von Vorteil ist, wenn diese in einer Sprache geschrieben werden können, die weit verbreitet und vielen Entwicklern bereits bekannt ist.

In der finalen Abwägung hat sich dann allerdings C# durchgesetzt, da es im Gegensatz zu C++ eine komplett objektorientierte Sprache ist, die zudem ein starkes Typsystem bietet. Darüber hinaus werden auch viele der Fallstricke, die es in C++ gibt in C# abgefangen, wie etwa Memory Leaks oder Garbage Collection. Die Argumente für C# und gegen Java waren dagegen unter anderem der bessere IDE Support für C# mit Visual Studio, die zusätzlichen Möglichkeiten in C# durch Techniken wie LINQ und die bessere API Dokumentation in C#. Darüber hinaus gaben auch ein – wenn auch subjektives – moderneres „Gefühl“ von C# und die höhere Familiarität des Autors mit C# den letzten Ausschlag.

Damit fiel die Wahl auf C#.NET 4.0 und für die Oberflächenkomponenten das WinForms Framework. Entsprechend dazu wurde auch als Entwicklungsumgebung dann auf Visual Studio 2010 zurückgegriffen.

Für die grobe Struktur des Codeanalyse Frameworks wird ein eigener Namespace definiert – Sca.Tools.CodeAnalysis – unter dem alle Bereiche des Frameworks zu finden sein werden. Der Namespace baut sich aus dem dreistelligen Kürzel des Autors – Sca – der groben Kategorie, in der das Projekt eingeordnet ist – Tools – und dem eigentlichen Projektnamen – CodeAnalysis – auf.

Zusätzlich soll das hier entstandene Codeanalyse Framework auch als Open Source Software zur Verfügung gestellt werden. Dazu wurde als Plattform github gewählt und ein Repository unter dem Pfad https://github.com/Schneephin/codeanalysis angelegt.

Code Lexer

Als ersten Teil des Codeanalyse Frameworks soll auf den Code Lexer eingegangen werden. Die Reihenfolge mit der hier die einzelnen Projekte besprochen werden soll auch den Workflow wiederspiegeln, mit dem der zu analysierende Sourcecode durch das Framework wandert, bis das Ergebnis schließlich in der Visualisierungsschicht dargestellt wird. Dieses Projekt soll hier aber auch vorangestellt werden, da es sich hierbei um den einzigen Teil des Frameworks handelt, bei dem externen Code verwendet wird, während die übrigen Teile komplette Eigenentwicklungen darstellen.

Der verwendete Code Lexer[Kro13] ist Teil eines privaten Experiments des Software Entwicklers Anton Kropp, der sich im Februar 2013 das Ziel gesetzt hat innerhalb weniger Wochen eine eigene Sprache mit entsprechendem Compiler zu entwickeln. Bei dem dabei entwickelten Lexer handelt es sich um einen sehr einfachen und leicht verwendbaren Lexer, der zudem recht einfach auf andere Sprachen abgewandelt werden konnte. Diese Einfachheit setzte den Lexer von Anton Kropp gegen andere frei verfügbare Lexer und AST Generatoren ab, worauf die Entscheidung fiel eben diesen zu verwenden.

Da es sich hierbei um einen kompletten Compiler musste zuerst der Lexer herausgetrennt werden, da für das vorliegende Codeanalyse Framework nur dieser Teil von Interesse war. Der nachgelagerte Parser war zu sprachspezifisch um ihn als Template für die Sprach-Plug-Ins verwenden zu können.

Daher wurde ein eigenes Projekt – Sca.Tools.CodeAnalysis.CodeAnalyser.Lexer – erstellt und alle für die Funktionalität des Lexers notwendigen Klassen übernommen. Anschließend wurden die Funktionen für die Definition der Keywords – die einzigen sprachspezifischen Komponenten des Lexers – so geändert, dass diese nicht mehr direkt im Lexer definiert sind, sondern über eine Funktion von außen übergeben werden können.

Abgesehen von dieser Änderung und einigen kleinen Fehlerkorrekturen sind alle Bereiche des oben genannten Projekts Fremdcode. Daher soll hier auch nicht näher auf die einzelnen Codebereiche eingegangen werden.

Framework Utilities

Der nächste Bereich wird von den Utilities eingenommen, die im Projekt Sca.Tools.CodeAnalysis.Utilities untergebracht sind. Hier werden hauptsächlich die Containerklassen für TreeNodes und ExtendedAttributes, dazugehörige Enumerations und das LanguagePlug-In Interface definiert. Da diese Klassen Bereits im Kapitel „Codeanalyse Framework - Basis“ beschrieben wurden und sich hier kaum Code findet, soll die Erwähnung hier genügen.

Damit bleibt im Utilities Projekt lediglich die ExtensionMethod Klasse. Diese enthält die auch bereits erwähnten Erweiterungsfunktionen für die Arbeit mit dem ParseableTokenStream. Hier bietet sich die Gelegenheit kurz auf das Konzept der ExtensionMethods in C# generell einzugehen und sie dann im Beispiel näher zu betrachten.

In C# sind Funktionen immer Teil einer Klasse und werden nach dem Schema Objekt.Funktion() – bzw. bei statischen Funktionen mit Klasse.Funktion() – aufgerufen. Um die Funktion aufrufen zu können muss sie entsprechend in der jeweiligen Klasse definiert sein. Nun kann es allerdings vorkommen, dass man für eine bereits definierte – heißt in diesem Fall in der Regel eine Klasse, die man nicht mehr ändern kann bzw. darf – eine weitere Funktion benötigen würde. Als Beispiel mag Klasse String dienen, die entsprechend durch den Benutzer nicht veränderbar ist, da es sich um eine zentrale Klasse des Frameworks handelt. Würde man nun eine Funktion Shuffle() für diese Klasse benötigen, könnte man sie nicht direkt zu String Klasse hinzufügen. Um diese Funktion dennoch nutzen zu können ist man in vielen Sprachen darauf angewiesen eine eigene Klasse – beispielsweise Utilities – zu erstellen und in dieser die entsprechende Funktion zu definieren. Die Stringinstanz, die man damit verändern möchte müsste dann als Parameter übergeben werden, wodurch der Funktionsaufruf in etwa so aussähe: Utilities.Shuffle(String). Dieser Workaround mag zwar in den meisten Sprachen funktionieren, ist allerdings eher weniger intuitiv, da man normalerweise eine Funktion von der Klasse aufrufen würde, zu der sie „gehört“.

In C# gibt es allerdings durch das ExtensionMethods Konzept die Möglichkeit genau dies zu realisieren, ohne die entsprechende Klasse verändern zu müssen. Dazu können die entsprechenden Funktionen einfach in einer separaten Klasse, häufig sprechend als ExtensionMethods bezeichnet definiert werden und anschließend so aufgerufen werden, als wären sie direkt in der, zum Beispiel Stringklasse definiert. Dazu müssen die Funktionen lediglich als static definiert sein und als ersten Übergabeparameter ein Objekt vom Typ der zugrundeliegenden Klasse enthalten. Eine mögliche Funktion für das obige Beispiel könnte als so aussehen:

```
public static String Shuffle<String>(this String value)
```

Der Aufruf würde dann wie gewohnt als Stringinstanz.Shuffle() erfolgen. Der erste Übergabeparameter muss also nicht explizit übergeben werden, sondern wird implizit aus dem aufrufenden Objekt bezogen.

Die ExtensionsMethods Klasse im Utilities Projekt setzt genau dieses Prinzip um verwendet es um hauptsächlich die Verwendung des, aus dem Lexer kommenden ParseableTokenStreams zu vereinfachen. Die erste der hier definierten Funktionen ist allerdings generischer angelegt.

```
public static bool IsOneOf<T>(this T value, params T[] items)
{
    for (int i = 0; i < items.Length; ++i)
    {
        if (items[i].Equals(value))
        {
            return true;
        }
    }

    return false;
}
```

Abbildung 29 - ExtensionMethods zur Prüfung ob ein Item in einer Liste enthalten ist

In diesem Fall wird als Typ der aufrufenden Klasse ein generisches T verwendet, das heißt diese Funktion kann als ExtensionMethod jeder beliebigen Klasse aufgerufen werden. Der zweite Übergabeparameter erfordert zudem ein Array vom Typ T, das heißt hier wird verlangt, dass die übergebenen Arraywerte vom selben Typ sein müssen wie die aufrufende Klasse.

Durch das params Flag ist es zudem möglich die entsprechenden Werte als kommagetrennte Liste zu übergeben und diese werden dann anschließend intern in ein Array umgewandelt. Obige Funktion könnte also beispielsweise so aufgerufen werden:

```
StringValue.IsOneOf<String>("private", "public", "protected", "internal")
```

Die Funktion würde dann prüfen, ob der Wert von StringValue in der übergebenen Liste enthalten ist und entsprechend true oder false zurückgeben.

```
public static int PeekAheadUntilMatchingClose(this ParseableTokenStream value, TokenType open, TokenType close)
{
    int currentLookAhead = 1;
    int depth = 0;
    while ((!value.Peek(currentLookAhead).TokenType.IsOneOf<TokenType>(close) || depth > 0) &&
        !value.Peek(currentLookAhead).TokenType.IsOneOf<TokenType>(TokenType.EOF) )
    {
        if (value.Peek(currentLookAhead).TokenType.IsOneOf<TokenType>(open))
        {
            depth++;
        }
        if (value.Peek(currentLookAhead).TokenType.IsOneOf<TokenType>(close) && depth > 0)
        {
            depth--;
        }
        currentLookAhead++;
    }
    if (value.Peek(currentLookAhead).TokenType == close)
    {
        return value.getCurrentIndex() + currentLookAhead;
    }

    return -1;
}
```

Abbildung 30 - ExtensionMethod um das passende schließende Element innerhalb eines ParseableTokenStreams zu finden

Als Beispiel für die auf ParseableTokenStreams zugeschnittenen Methoden soll die Methode PeekAheadUntilMatchingClose dienen. Diese dient, wie der Name bereits andeutet, dazu ein schließendes Element im vorliegenden TokenStream für ein gegebenes öffnendes Element zu finden. Dazu wird der Typ des öffnenden und des schließenden Tokens als Parameter übergeben. Anschließende wird solange durch den TokenStream iteriert, bis eine Token mit dem entsprechenden schließenden Typ gefunden oder das Ende des Streams erreicht wird. Der Typ des öffnenden Tokens wird benötigt um Verschachtelungen zu erkennen. Wie im Code ersichtlich ist, wird sobald ein Token vom Typ des öffnenden Tokens gefunden wird der Counter depth hochgezählt. Solange dieser Counter größer als null ist wird für den Fall eines gefundenen schließenden Tokens lediglich der Counter reduziert und nicht das Token zurückgegeben. Das heißt es müssen solange schließende Elemente gefunden werden, bis der Counter wieder bei null steht, sprich bis alle in der Zwischenzeit geöffneten, verschachtelten Token wieder geschlossen wurden. Erst danach wird das nächste schließende Element als das

zum Ursprungselement passende erkannt und entsprechend der Index des Elements zurückgegeben.

Codeanalyse Framework

An dieser Stelle im Workflow würde nun das eigentliche Sprach-Plug-In stehen, da dieses allerdings variabel und somit nicht fester Teil des Frameworks ist soll es hinten angestellt werden. Daher ist der nächste Teil das Codeanalyse Framework selbst.

Das Framework befindet sich im Projekt Sca.Tools.CodeAnalysis.Framework und umfasst das Analyse UI, die Schnittstellen zu den Sprach-Plug-Ins und die Ausgabe als XML Datei. Ein erster Entwurf des CodeAnalyzer GUIs ist bereits im Kapitel Code Analyzer – Graphical User Interface zu sehen.

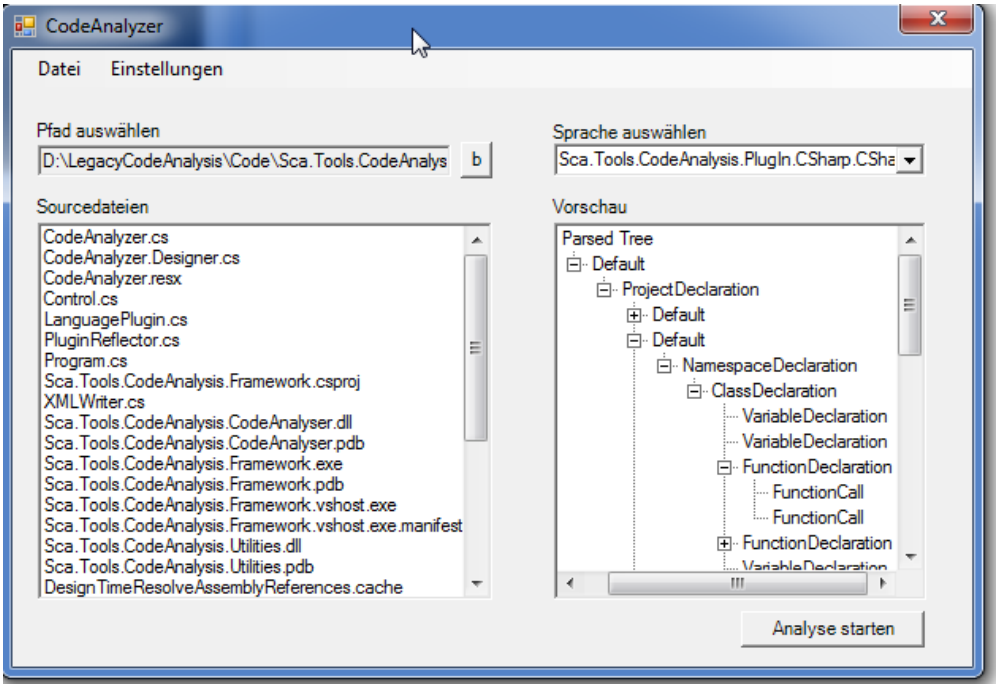

Abbildung 31 - Oberfläche des CodeAnalyzers

Die finale Version der Oberfläche ist in der obigen Abbildung zu sehen. Der Screenshot zeigt den CodeAnalyzer nachdem bereits eine Analyse durchgeführt wurde und damit die Ergebnisse im Vorschau Fenster dargestellt werden. Auf der linken Seite sieht man den ausgewählten Pfad in dem sich das zu analysierende Projekt befindet – im vorliegenden Fall handelt es sich um das Framework selbst. In der Listbox darunter sind die im gewählten Ordner gefundenen Dateien zu sehen. Auf der rechten Seite ist oben das ausgewählte Sprach-Plug-In zu sehen, hier CSharp und darunter der erstellte Abstract Syntax Tree. Im Tree ist sehr gut die Struktur der analysierten Projekts zu erkennen, oben die ProjectDeclaration, also die eigentliche csproj Datei. Darunter befinden sich eine NamespaceDeclaration und darunter die jeweiligen Klassen mit Variablen, Funktionen und Funktionsaufrufen. An dieser Stelle könnte da Ergebnis über den Menüpunkt Datei als XML Datei gespeichert werden oder beispielsweise eine neue Analyse über die Auswahl eines neuen Pfades gestartet werden.

Da das Codeanalyse Framework grob nach MVC Prinzipien aufgebaut ist, befindet sich in den UI Klassen – und somit auch im CodeAnalyzer GUI – kaum Sourcecode, abgesehen von einigen Funktionen, die für diverse Anzeigen im UI oder das Auslösen von Events nötig sind.

Die Steuerung des UIs erfolgt über eine eigene Controller Klasse, im vorliegenden Fall Control genannt. Hier werden alle Eingaben aus dem UI aufgefangen und die jeweiligen Methoden aufgerufen.

```
internal void doAction(Actions action)
{
    switch (action)
    {
        case Actions.AddLanguagePlugins:
            this.addLanguagePlugin();
            break;
        case Actions.SelectPath:
            this.fillSourceFiles();
            break;
        case Actions.Analyze:
            this.analyzeCode();
            break;
        case Actions.SaveResults:
            this.saveAnalysisResults();
            break;
        case Actions.End:
            this.codeAnalyzer.Close();
            break;
    }
}
```

Abbildung 32 - Funktion zur Steuerung der durch das UI ausgelösten Aktionen

Jedes Event im UI ruft hier die zentrale doAction Methode auf und übergibt die jeweilige Aktion, die in der Actions Enumeration definiert sind als Parameter. Je nach übergebener Aktion werden dann die entsprechenden Funktionen aufgerufen, die die Aktion ausführen. Würde zum Beispiel eine SaveResults Action übergeben, wird die entsprechende svaeAnalysisResults() Methode aufgerufen.

```
private void saveAnalysisResults()
{
    if (this.analysisResults != null)
    {
        SaveFileDialog sfd = new SaveFileDialog();
        if (sfd.ShowDialog() == DialogResult.OK)
        {
            XMLWriter.CreateXmlDocument(this.analysisResults, sfd.FileName);
        }
    }
}
```

Abbildung 33 - Methode zum Speichern der Analyseergebnisse

Für den Fall, dass bereits eine Analyse durchgeführt wurde, wird dann ein Dialog aufgerufen um den gewünschten Speicherort und Dateinamen abzufragen. Wird dieser Dialog mit einem OK beendet, wird der Dateiname und das Analyseergebnis an den XMLWriter übergeben, der anschließend das Ergebnis speichert.

Im XMLWriter wird das als Baum vorliegende Analyseergebnis rekursiv durchlaufen und in einen Baum aus XmlNodes überführt.

```
private static XmlNode recurseTreeNodes(TreeNode parent)
{
    XmlNode node = XMLWriter.createNode(parent);
    XMLWriter.appendExtendedAttributes(node, parent);

    foreach (TreeNode childNode in parent.childNodes)
    {
        node.AppendChild(XMLWriter.recurseTreeNodes(childNode));
    }

    return node;
}
```

Abbildung 34 - Rekursive Funktion zum Umwandeln von TreeNodes in XmlNodes

Der recurseTreeNodes Funktion wird dabei jeweils der aktuelle Parent übergeben und für diesen ein XmlNode angelegt. Falls der Parent zusätzliche ExtendedAttributes besitzt werden diese anschließend als untergeordnete XmlNodes angehängt. Zum Abschluss wird die Funktion wieder für alle Kindknoten im Parent aufgerufen. Dadurch werden jeweils XmlNodes entweder vom Typ node für TreeNodes oder vom Typ extendedAttribute für ExtendedAttributes angelegt.

```
XmlNode node = XmlDocument.CreateElement("node");
XmlAttribute type = XmlDocument.CreateAttribute("type");
type.Value = Enum.GetName(typeof(NodeTypes), treeNode.nodeType);
node.Attributes.Append(type);
```

Abbildung 35 - Anlegen eines neuen XmlNodes und eines dazugehörigen Attributes

Alle Attribute der TreeNodes und ExtendedAttributes werden anschließend dem erstellten XmlNode als Attribute zugewiesen – wie in obiger Abbildung beispielhaft für das Attribut type eines TreeNodes gezeigt.

Zentraler Punkt des Framework Projekts ist der PluginReflector, der das Einbinden der Sprach-Plug-Ins und den Aufruf der analyseCode Funktion übernimmt. Das Einbinden neuer Sprach-Plug-Ins wird durch die addLanguagePlugin Funktion erledigt.

```
try
{
    Assembly currentAssembly = Assembly.LoadFrom(pluginFileName);
    foreach(Type pluginClass in currentAssembly.GetTypes())
    {
        if (pluginClass.GetInterface("Sca.Tools.CodeAnalysis.Utilities.iLanguagePlugIn") != null)
        {
            plugIns.Add(new LanguagePlugin(pluginClass.FullName, currentAssembly));
        }
    }
}
```

Abbildung 36 - Laden eines neuen Sprach-Plug-Ins

Beim Aufruf wird der Funktion der Dateiname für eine Sprach-Plug-In Dll übergeben. Die entsprechende Dll wird anschließend eingelesen und geprüft, ob eine oder mehrere Klassen innerhalb der geladenen Assembly das iLanguagePlugIn implementieren. Jede Klasse, die dieses Interface implementiert wird anschließend zur Liste der vorhandenen Sprach-Plug-Ins hinzugefügt, wobei der Name der Klasse als DisplayName gesetzt wird.

ChartFramework

Der nächste Teil des Frameworks ist wiederum ein Hilfsprojekt, das ChartFramework. Dieses befindet sich im Projekt Sca.Tools.ChartFramework und soll ein Framework zum Zeichnen der Abhängigkeitsbäume bereitstellen. Das Framework selbst sollte möglichst leichtgewichtig umgesetzt werden und sich auf die wesentlichen Funktionen beschränken. Das ChartFramework befindet sich wie oben ersichtlich nicht im üblichen CodeAnalysis Namespace, da es streng genommen nicht Teil des CodeAnalysis Projekts ist, sondern lediglich eine Zeichenebene zur Verfügung stellen soll, die aber auch in anderen Projekten benutzt werden könnte.

Von Bedeutung ist hier zum einen die Connector Klasse, die die Verbindungslinien zwischen den einzelnen gezeichneten Knoten darstellt und das Zeichenpanel selbst. In der Connector Klasse befindet sich der Algorithmus zum Berechnen der beiden geeigneten Verbindungspunkte, wir er bereits im entsprechenden Kapitel unter Visualisierungsframework beschreiben wurde.

```
private bool isLeft(Point a, Point b, Point c)
{
    bool isLeft = ((b.X - a.X) * (c.Y - a.Y) - (b.Y - a.Y) * (c.X - a.X)) > 0;
    return isLeft;
}
```

Abbildung 37 - Algorithmus zur Prüfung ob ein Punkt links einer Linie liegt

Dazu wird zuerst eine Funktion benötigt, die ermitteln kann, ob ein gegebener Punkt links, oder rechts einer bestimmten Linie liegt. Hier wird die Linie durch die beiden Punkte a und b definiert und es soll die Lage für den Punkt c geprüft werden. Anschließend wird die Determinante der beiden aus a und b und aus a und c gebildeten Vektoren berechnet. Ist diese größer null, so liegt der Punkt c links der Geraden ab, ist sie kleiner null so liegt er rechts und ist sie gleich null, so liegt der Punkt auf der Geraden.

Mit Hilfe dieser Funktion kann dann berechnet werden in welchem Segment die Mitte des zweiten Rechtecks liegt.

```
private void calculateConnectorPoints()
{
    Point topleft = node1.rectangle.Location;
    Point topright = new Point(node1.rectangle.X, node1.rectangle.Y + node1.rectangle.Height);
    Point bottomleft = new Point(node1.rectangle.X + node1.rectangle.Width, node1.rectangle.Y);
    Point bottomright = new Point(node1.rectangle.X + node1.rectangle.Width, node1.rectangle.Y + node1.rectangle.Height);
    Point middle = new Point(node2.rectangle.X + node2.rectangle.Width / 2, node2.rectangle.Y + node2.rectangle.Height / 2);

    if(this.isLeft(topleft, bottomright, middle) && this.isLeft(bottomleft, topright, middle))
    {
        this.connectorPoint1 = node1.getNodeConnector(ConnectorPoint.Left);
        this.connectorPoint2 = node2.getNodeConnector(ConnectorPoint.Right);
    }
```

Abbildung 38 - Funktion zum Berechnen der beiden Verbindungspunkte

Dazu werden zuerst die jeweiligen Begrenzungspunkte des ersten Rechtecks berechnet um damit die beiden Geraden aufzuspannen. Anschließend wird geprüft, ob sich der Mittelpunkt des zweiten Rechtecks jeweils links beider Geraden, links der einen und rechts der anderen oder rechts von beiden und somit in einem der vier Segmente befindet. In obiger Abbildung wird

geprüft, ob sich der Mittelpunkt von Rechteck zwei links von beiden Geraden und somit im linken Segment befindet. Ist dies der Fall werden jeweils die Ankerpunkte Links und Rechts für das Rechteck eins und zwei verwendet. Diese Prüfung wird jeweils auch für die anderen drei Kombinationen durchgeführt.

In der ChartPane Klasse können diese Verbindungslinien und die entsprechenden Knoten dann gezeichnet werden. Dazu werden die Graphics Objekte der WinForms Bibliothek benutzt.

```
private void drawNode(RectangleNode rn)
{
    Graphics g = this.CreateGraphics();
    g.DrawRectangle(new Pen(Color.Black), rn.rectangle);
    StringFormat stringFormat = new StringFormat();
    stringFormat.Alignment = StringAlignment.Center;
    stringFormat.LineAlignment = StringAlignment.Center;
    g.DrawString(rn.label, new Font(FontFamily.GenericSansSerif, 12), Brushes.Black, rn.rectangle, stringFormat);
}
```

Abbildung 39 - Funktion zum Zeichnen eines Rechtecks mit Label

Hier ist die Funktion für das Zeichnen der Rechtecke zu sehen. Dabei wird zuerst über die CreateGraphics Methode der UserControl Klasse ein Graphics Objekt erstellt und anschließend über die DrawRectangle Funktion das übergeben Rechteck gezeichnet. Über entsprechende StringFormat Objekte kann danach die Positionierung für das Label bestimmt und dieses dann mit der DrawString Methode gezeichnet werden.

Code Visualizer

Die eigentliche Darstellung der Analyseergebnisse wird im Code Visualizer vorgenommen. Hier befinden sich sowohl die GUI Elemente selbst, darunter das zentrale Formelement und die einzelnen Panels, als auch die Steuerungslogik mit den entsprechenden Hilfsklassen.

Der erste Schritt im Code Visualizer ist das Laden der Analyseergebnisse. Dies würd über den entsprechenden Menüpunkt im Dateimenü angestoßen und durch die XMLReader Klasse erledigt. Dabei wird der Dateiname und Pfad für die ausgewählte XML Datei an den XMLReader übergeben und die entsprechende XML Struktur ausgelesen. Die Umwandlung der XML Struktur in eine Baumstruktur aus TreeNodes erfolgt analog dem XMLWriter im Codeanalyse Framework nur in entsprechend anderer Richtung. Nach dem Einlesen sollte im Code Visualizer dieselbe Struktur an TreeNodes zur Verfügung stehen wie sie im Codeanalyse Framework ursprünglich erstellt wurde.

Um die späteren Ladezeiten zu verringern wird an dieser Stelle bereits das geladene Analyseergebnis durchlaufen und verschiedene Daten, die für die Anzeige von Bedeutung sind aggregiert. Für die Anzeige wichtig sind dabei vor allem Klassen- und Funktionsknoten, sowie entsprechende Funktionsaufrufe, da diese in der Abhängigkeitsansicht angezeigt werden sollen. Zusätzlich sind die im Analyseergebnis gesetzten Filter von Interesse, da diese im FilterPanel auswählbar sein sollen.

Daher wird nach dem Laden des Analyseergebnisses der Baum rekursiv durchlaufen und alle vorhandenen Klassen-, Funktions- und Funktionsaufrufsknoten in entsprechenden Listen

gesammelt. Anschließend wird jede dieser Listen durchlaufen und für jeden Knoten geprüft, ob ein entsprechender Filter als ExtendedAttribute hinterlegt ist.

```
private void getFiltersFrom(List<TreeNode> nodes, Mode cMode)
{
    Dictionary<String, Filter> tFilters = new Dictionary<String, Filter>();
    foreach (TreeNode node in nodes)
    {
        foreach (ExtendedAttribute ea in node.nodeAttributes)
        {
            if (ea.attributeType.Equals("filter"))
            {
                if (!tFilters.ContainsKey(ea.attributeName))
                {
                    tFilters.Add(ea.attributeName, new Filter() { mode = cMode, filterName = ea.attributeName });
                    tFilters[ea.attributeName].filterValues.Add("All");
                }
                if (!tFilters[ea.attributeName].filterValues.Contains(ea.attributeInformation))
                {
                    tFilters[ea.attributeName].filterValues.Add(ea.attributeInformation);
                }
            }
        }
    }
    this.filters.AddRange(tFilters.Values);
}
```

Abbildung 40 - Funktion zum Sammeln von definierten Filtern

Filter sind jeweils als ExtendedAttribute zu einem Funktions- oder Klassenknoten hinterlegt und enthalten als attributeTyp den Wert filter. Wird ein entsprechendes ExtendedAttribute gefunden, so wird ein neues Containerobjekt vom Typ Filter erstellt. Diesem wird jeweils der entsprechende Typ, Funktions- oder Klassenknoten und der Filtername als Attribut übergeben. Zusätzlich wird ein erster Eintrag in die Liste von möglichen Filterwerten eingefügt. Bei jedem weiteren ExtendedAttribute, welches mit demselben Filternamen gefunden wird, wird dann nur noch ein eventueller neuer Filterwert nachgetragen. Ein Filter besteht dann aus einem Typ, der angibt, ob er in der Klassen- oder in der Funktionsansicht verfügbar ist, einem Filternamen und einer Liste an Filterwerten. Als Beispiel könnte es einen Filter geben, der für die Funktionsansicht gilt, den Namen Klasse trägt und als mögliche Werte alle Klassen enthält, in denen Funktionen definiert sind. Mit diesem Filter könnte dann in der Abhängigkeitsansicht, wie in der Abbildung unten zu sehen eingestellt werden, ob nur Funktionen einer bestimmten Klasse, oder aller Klassen angezeigt werden sollen.

Abbildung 41 - Filter für vorhandene Klassen in der Funktionsansicht

Sind diese Vorarbeiten erledigt, dann wird das zentrale Code Visualizer GUI geladen und die ersten Panel, initial jeweils ein FilterPanel und der TreeView platziert. Der Benutzer kann dann über den Menüpunkt View zwischen den einzelnen Panels in der linken Leiste und über die Button unten rechts zwischen Klassen- und Funktionssicht wechseln. Im zentralen Panel dagegen bleibt immer der TreeView angezeigt und es ändert sich jeweils nur der dargestellte Inhalt.

Bevor auf den dahinterliegenden Code eingegangen wird, soll nun erstmal ein Überblick über die verschiedenen Views gegeben werden.

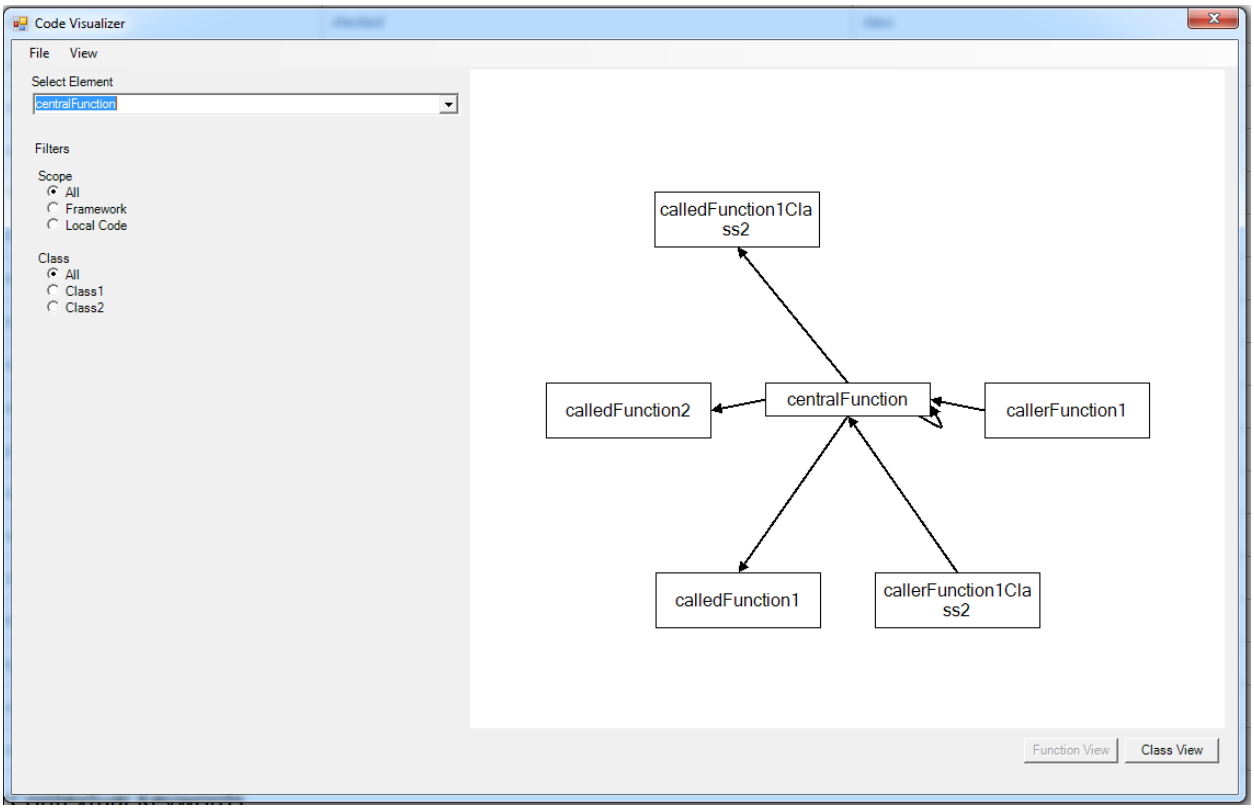

Abbildung 42 - Code Visualizer GUI mit angezeigtem FilterPanel und TreeView

In der oberen Abbildung ist das bereits erwähnte FilterPanel an der linken Seite zu sehen. Hier kann jeweils über das DropDown Control das angezeigte Element ausgewählt werden und über die darunterliegenden Filter eine entsprechende Einschränkung vorgenommen werden. An den Buttons unten rechts lässt sich erkennen, dass im Moment die Funktionsansicht aktiv ist. Daher können über das Element DropDown entsprechend Funktionen ausgewählt und darunter funktionsspezifische Filter gesetzt werden. Die ausgewählte Funktion, hier centralFunction wird dann im TreeView rechts in der Mitte angezeigt. Um die ausgewählte Funktion werden dann alle weiteren Funktionen platziert, die entweder die centralFunction aufrufen, oder von dieser aufgerufen werden. Die Richtung der Pfeile gibt dabei Auskunft über die Aufrufrichtung. Im obigen Beispiel würden jeweils die callerFunction1 und callerFunction1Class2 die centralFunction aufrufen und die centralFunction ruft im Gegenzug die übrigen Funktionen auf. Zusätzlich ist über den Pfeil, der von der centralFunction auf sich selbst zeigt zu erkennen, dass sich die centralFunction rekursiv selbst aufruft.

In obiger Abbildung stehen jeweils alle Filter auf All und somit werden alle Funktionen angezeigt. Setzt man hingegen einen Filter auf einen bestimmten Wert, wie in der Abbildung unten, wird diese Auswahl jeweils eingeschränkt.

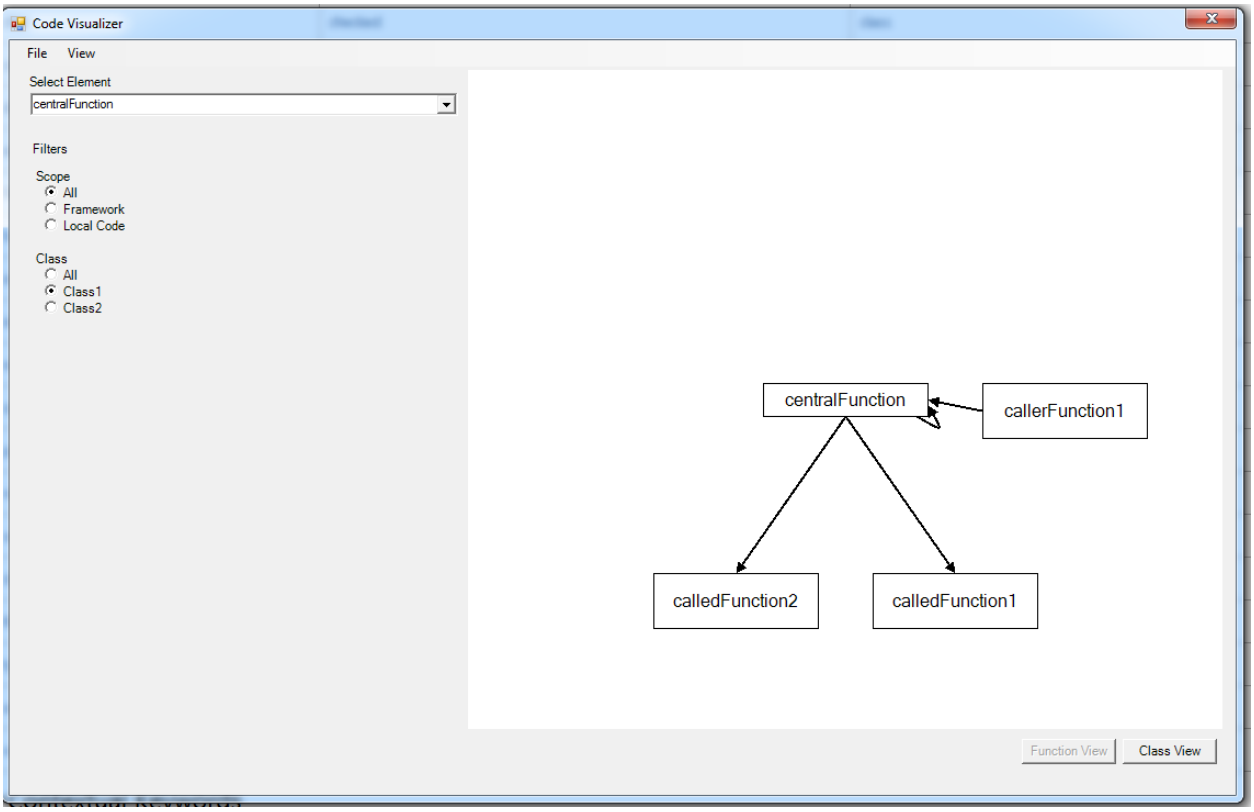

Abbildung 43 - TreeView mit gesetztem Filter

Hier wurde der Class Filter auf Class1 gesetzt und somit sind werden nur noch Funktionen, die in Class1 definiert sind angezeigt. Alle Funktionen aus Class2 – in diesem Beispiel sprechend mit dem Namensbestandteil Class2 – werden dagegen nicht mehr angezeigt.

Wechselt man über die Buttons rechts unten zur Klassenansicht, so wird links das Element DropDown auf Klassen geändert und darunter die klassenspezifischen Filter angezeigt. Im TreeView wird dann die jeweils ausgewählte Klasse mit allen darin definierten Funktionen dargestellt, in der Abbildung unten ist dies die Class1.

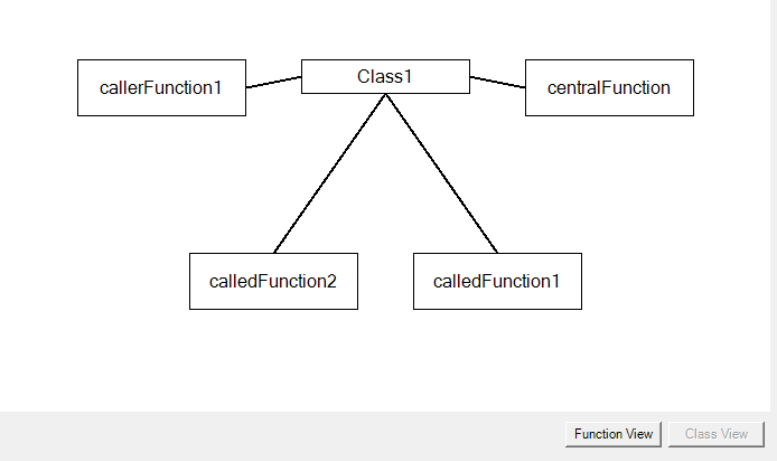

Abbildung 44 - TreeView in der Klassenansicht mit Class1

Die einzelnen Funktionen werden dabei mit ungerichteten Verbindungen mit der Klasse verbunden.

Zwischen den einzelnen Informationspanels am linken Rand kann über den Menüpunkt View gewechselt werden. Nach dem FilterPanel wäre hier das nächste das DetailPanel.

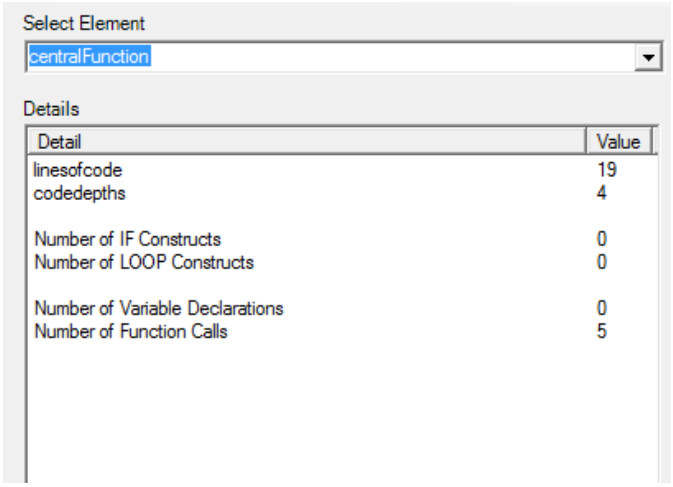

Abbildung 45 - Detailansicht für die centralFunction

Beim Wechsel des Informationspanels bleibt jeweils der TreeView erhalten, lediglich das Panel am linken Rand ändert sich. Im Detailpanel kann wieder das gewünschte Element – eine Klasse oder Funktion je nach aktuellem Modus – gewählt werden. Passend zur Auswahl wird jeweils der TreeView aktualisiert und in der Liste im linken Panel die jeweiligen Detailinformationen zum gewählten Element angezeigt. In der Abbildung oben sind dies zum Beispiel Code Smell Einträge wie die linesofcode und die codedepths oder auch die Anzahl an verschiedenen Elementen innerhalb der gewählten Funktion wie Loops, Variablen oder Funktionsaufrufe.

Das letzte Informationspanel, das über den Menüpunkt View ausgewählt werden kann ist die To-do Liste. Hier werden alle gefundenen Einträge vom Typ Code Smell zusammengetragen und in einer Top X Liste angezeigt.

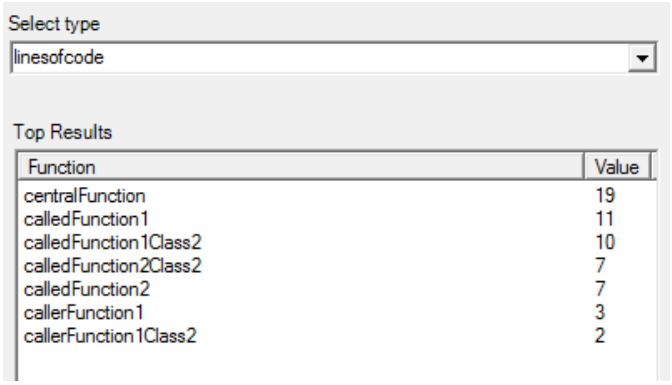

Abbildung 46 - To-do Liste mit ausgewähltem Typ Lines of Code

Wie in der Abbildung oben zu erkennen, kann hier über das DropDown die gewünschte Metrik gewählt werden, im Beispiel ist dies Lines of Code. Nach der Auswahl werden in der Liste unten die Funktionen oder Klassen angezeigt, die für diese Metrik die höchsten Werte erzielen und daher mit höherer Wahrscheinlichkeit Probleme, bzw. Verbesserungspotential aufweisen. In obiger Abbildung wäre dies die Funktion centralFunction, die mit 19 Codezeilen an der Spitze steht. Hierbei handelt es sich natürlich nur um ein Beispiel und die Zahlen für eine Legacy Code Base dürften um einiges höher ausfallen.

Nachdem nun die Oberfläche behandelt wurde, kann der eigentliche, darunter liegende Code betrachtet werden. Das Code Visualizer Projekt befindet sich im Namespace Sca.Tools.CodeAnalysis.Visualizer und ist, wie auch das Codeanalyse Framework grob nach MVC Kriterien aufgebaut. Das heißt es finden sich hier die jeweiligen Anzeigeelemente, das zentrale GUI und die oben bereits vorgestellten Panels, die jeweils nur den nötigsten Code enthalten und die eigentliche Control Klasse. Aus Sourcecode Sicht können daher die einzelnen GUI Elemente fast komplett übergangen werden, da sich dort nur wenig interessanter Code finden lässt. Einzige Ausnahme bildet vielleicht das FilterPanel. Hier werden im unteren Bereich die bereits vorgestellten Filter dargestellt. Die Schwierigkeit besteht hier darin, dass die Anzahl der Filter und die Anzahl der Werte in den einzelnen Filtern komplett variabel sind. Daher müssen die einzelnen RadioButtons dynamisch auf dem Panel platziert werden.

Die Platzierung der Filter erfolgt nach einem bestimmten Schema, für jeden Filter wird zuerst der Filtername als Label angezeigt. Darunter wird ein Panel angelegt, das genug Platz für die einzelnen RadioButtons bietet und auf diesem werden dann die einzelnen Filterwerte als RadioButton platziert.

Um diese Positionierung möglichst dynamisch zu gestalten wird die Funktion addFilters verwendet. Der Funktion wird jeweils eine Liste der gewünschten Filter Objekte übergeben und die dann gezeichnet werden sollen.

```
internal void addFilters(List<Filter> filters)
{
    int distanceToTop = 60;
    RadioButton rb;
    Panel panel;
    Label lbl;
    Boolean first;
    foreach (Filter filter in filters)
    {
        distanceToTop += 30;
        first = true;
        lbl = new Label() { Name = filter.filterName, Left = 10, Text = filter.filterName, Top = distanceToTop, Height = 13 };
        this.Controls.Add(lbl);

        panel = new Panel() { Location = new Point(10, distanceToTop + 15), Size = new Size(300, filter.filterValues.Count * 15) };
        this.Controls.Add(panel);

        int distanceToTopInPanel = 0;
        foreach (String filterValue in filter.filterValues)
        {
            rb = new RadioButton() { Name = filter.filterName + "_" + filterValue, Text = filterValue, Left = 10, Height = 13,
                Checked = first, Top = distanceToTopInPanel };
            rb.CheckedChanged += new System.EventHandler(this.radioButton_CheckedChanged);
            panel.Controls.Add(rb);
            this.filterButtons.Add(rb);
            first = false;
            distanceToTopInPanel += 15;
            distanceToTop += 15;
        }
    }
}
```

Abbildung 47 - Funktion zum dynamischen Platzieren der Filter

In der Funktion wird, wie in der dann durch die Liste an Filtern iteriert und für jeden Filter ein entsprechendes Label mit dem Filternamen angelegt. Anschließend wird ein Panel positioniert, das genau so groß ist, dass die Anzahl an vorhandenen Filterwerten als RadioButtons darauf platziert werden können. Dies ist nötig, da bei WinForms die Gruppierung von RadioButtons über die Platzierung auf einem Panel oder einem ähnlichen Element erfolgt. Das heißt alle RadioButtons, die sich auf einem Panel befinden gehören zur selben Gruppe und können daher nur jeweils einzeln angewählt werden.

Nachdem das Panel erstellt wurde wird anschließend durch die Liste der Filterwerte iteriert und die jeweiligen RadioButtons angelegt. Wichtig hierbei ist jeweils, dass die Positionen der einzelnen Controls und damit auch die Abstände zwischen den Controls korrekt erhöht werden, da sich sonst Controls überlagern oder mit unterschiedlichen Abständen angezeigt werden könnten.

Die Hauptlogik zu den einzelnen Oberflächenelementen befindet sich allerdings in der zentralen Control Klasse. Am Anfang des Kapitels wurde bereits auf die Funktionen eingegangen, die zur Initialisierung nötig waren, wie das Aggregieren der Klassen- und Funktionsknoten oder das Sammeln der vorhandenen Filter. Im Anschluss soll daher auf die restlichen Funktionen in der Control Klasse eingegangen werden, die während des Betriebs der Oberfläche von Bedeutung sind.

Das dominante Element der Oberfläche ist sicherlich der TreeView, der nicht nur immer zu sehen ist, sondern auch mitunter die wichtigsten Informationen anzeigt. Als Basis des TreeView wird eine Instanz der im vorherigen Kapitel besprochenen ChartPane aus dem ChartFramework verwendet. Auf dieser werden dann jeweils alle benötigten Knoten und die entsprechenden Verbindungen gezeichnet. Im Zentrum der ChartPane befindet sich dabei jeweils der

ausgewählte Knoten, während die übrigen, abhängigen Knoten um diesen zentralen Knoten positioniert werden. Für die Positionierung der abhängigen Knoten war es notwendig eine eigene Funktion zu entwickeln, die für jeden zu zeichnenden Knoten eine neue Position berechnen kann.

```
private RectangleNode createNextRectangle(String label)
{
    RectangleNode node = null;
    System.Drawing.Size panelSize = ((TreeView)this.panels["tree"]).chartPane.Size;
    System.Drawing.Point middle = new System.Drawing.Point(
        (int)Math.Floor(panelSize.Width / 2 + radius * Math.Cos(rectangleCount * arc * Math.PI / 180F)),
        (int)Math.Floor(panelSize.Height / 2 + radius * Math.Sin(rectangleCount * arc * Math.PI / 180F)));
    node = new RectangleNode(middle.X - 75, middle.Y - 15, 150, 50, label);
    rectangleCount++;
    if (rectangleCount % 6 == 0)
    {
        radius += 70;
        arc += 30;
    }

    return node;
}
```

Abbildung 48 - Funktion zur Berechnung einer neuen Knotenposition

Die Überlegung war es die Knoten jeweils auf einem Kreis mit Mittelpunkt im Zentrum der ChartPane zu platzieren. Dazu wird eine Position auf dem Kreis mit einem bestimmten Winkel berechnet und diese als Mittelpunkt für den neuen Knoten angenommen. Für jeden weiteren Knoten wird dann lediglich der Winkel erhöht und die Knoten somit auf einer Kreisbahn angeordnet. Pro Kreisbahn sollen dabei jeweils nur sechs Knoten angeordnet werden, da die Knoten ansonsten zu eng positioniert sind. Bei jedem sechsten Knoten wird jeweils der Radius der Kreisbahn vergrößert und der Winkel um 30° verschoben.

Diese Funktion wird dann beim Erstellen der jeweiligen TreeView Ansicht verwendet. Dies geschieht jeweils durch verschiedene Funktionen für das Zeichnen der einzelnen Ansichten.

```
private void displayDependencyViewCallingFunctions(TreeNode function, Dictionary<String, String> filters)
{
    AdjustableArrowCap bigArrow = new AdjustableArrowCap(5, 5);
    List<TreeNode> callingFunctions = this.getCallingFunctions(function, filters);

    foreach (TreeNode callingFunction in callingFunctions)
    {
        TreeNode callingFunctionDeclaration = this.getCallingFunctionForFunctionCall(callingFunction);
        if (callingFunctionDeclaration.nodeId == function.nodeId)
        {
            ((TreeView)this.panels["tree"]).chartPane.drawRecursiveCurve(function.nodeId);
        }
        else
        {
            if (!((TreeView)this.panels["tree"]).chartPane.nodes.ContainsKey(callingFunctionDeclaration.nodeId))
            {
                RectangleNode rect = null;
                do
                {
                    rect = this.createNextRectangle(callingFunctionDeclaration.nodeContent);
                }
                while (((TreeView)this.panels["tree"]).chartPane.intersectsWith(rect));
                ((TreeView)this.panels["tree"]).chartPane.addNode(callingFunctionDeclaration.nodeId, rect);
            }
            ((TreeView)this.panels["tree"]).chartPane.addConnection(function.nodeId, callingFunctionDeclaration.nodeId, bigArrow, null);
        }
    }
}
```

Abbildung 49 - Funktion zum Anzeigen von aufrufenden Funktionen auf dem TreeView

In der obigen Abbildung ist dabei exemplarisch die Funktion gezeigt, die alle aufrufenden Funktionen der ausgewählten zentralen Funktion anzeigt. Dabei werden über eine Hilfsfunktion alle Funktionen gesammelt, die die ausgewählte Funktion aufrufen. Anschließend wird für jede gefundene Funktion versucht diese zu zeichnen. Handelt es sich bei der aufrufenden Funktion auch um die aufgerufene Funktion, ist es also ein rekursiver Aufruf, dann wird der oben bereits gezeigte rekursive Pfeil gezeichnet. Andernfalls wird über den oben gezeigten Algorithmus versucht ein neues Rechteck zu generieren. Nun kann es bei obigem Algorithmus vorkommen, dass eine neue Position gefunden wurde, die eine Überschneidung mit einem bereits positionierten Rechteck bedeuten würde. Daher wird hier für jedes neu erstellte Rechteck geprüft ob eine Überschneidung vorliegt und bei Bedarf dann solange ein neues Rechteck erstellt, bis dies nicht mehr der Fall ist. Anschließend kann dann das Rechteck und der entsprechende Verbindungspfeil gezeichnet werden. Dieses Vorgehen wird dann entsprechend für alle aufgerufenen Funktionen wiederholt.

Sprach-Plug-In

Der letzte Teil im Kapitel Implementierung beschäftigt sich mit den Sprach-Plug-Ins. Diese sind wie bereits im Kapitel zum Codeanalyse Framework erwähnt kein eigentlicher Teil des Codeanalyse Frameworks sondern externe Plug-Ins, die es dem Entwickler erlauben die für sein zu analysierendes Projekt nötigen Sprachen und Metriken zu definieren. Damit sind die Sprach-Plug-Ins auch die Stelle an der die Erweiterbarkeit des Codeanalyse Frameworks ansetzen kann.

Im Rahmen der Erstellung des Frameworks wurde bereits ein Sprach-Plug-In für die Sprache C#.NET entwickelt. Daher soll dieses Kapitel einerseits dazu dienen auf die Entwicklung dieses Sprach-Plug-Ins im Besonderen einzugehen, aber auch um eine allgemeine Beschreibung zu liefern wie prinzipiell andere Sprach-Plug-Ins erstellt oder erweitert werden können.

Der zentrale Ansatzpunkt für die Sprach-Plug-Ins ist jeweils die Klasse, die das iLanguagePlugIn Interface implementiert. Im vorliegenden Fall handelt es sich dabei um die CSharpNet Klasse, die sehr einfach ausfällt und nur die für das Interface nötige Funktion enthält.

```
public class CSharpNet : iLanguagePlugIn
{
    public TreeNode analyseCode(string folder)
    {
        Parser parser = new Parser();
        return parser.parseSourceProject(folder);
    }
}
```

Abbildung 50 - Klasse, die zum Aufruf des Sprach-Plug-Ins dient

Der Name der Klasse ergibt gleichzeitig auch den Anzeigenamen des Sprach-Plug-Ins in der Code Analyzer Oberfläche und ist daher möglichst sprechend gewählt.

Bevor die eigentliche Analyse des Sourcecodes begonnen werden kann, muss hier noch kurz auf den Aufbau von C# Projekten eingegangen werden. Zentraler Bestandteil eines C# Projekts ist die csproj Datei, in der alle projektrelevanten Informationen gespeichert werden, wie etwa

alle eingebunden Sourcecode Dateien, externe Projektreferenzen und Build Events. Die csproj Datei selbst wird im XML Format gespeichert und kann daher sehr leicht ausgelesen werden.

Zu diesem Zweck wurde eine eigene Klasse geschrieben, die eine Abbildung der csproj Dateien im Code darstellt. Nach Übergabe einer entsprechenden csproj Datei wird versucht die entsprechenden Bestandteile, also Referenzen, Sourcecode Dateien und Ressourcen auszulesen.

```
private void LoadSourceFiles()
{
    SourceFiles.Clear();

    XmlNodeList objNodeListSourceFiles = m_objXmlDocProject.SelectNodes("./def:Project/def:ItemGroup/def:Compile",
        m_objXmlNameSpaceManager);
    foreach (XmlNode objNodeSourceFile in objNodeListSourceFiles)
    {
        CSProjectSourceFile objCSSource = new CSProjectSourceFile(objNodeSourceFile);
        SourceFiles.Add(objCSSource);
    }
}
```

Abbildung 51 - Funktion zum Auslesen aller Sourcecode Dateien

Das Auslesen selbst wird über die Funktionen der XmlDocument Klassen bewerkstelligt. Im obigen Beispiel sieht man das Auslesen der Sourcecode Dateien, die sich in der Xml Struktur unter dem Knoten Project – ItemGroup – Compile befinden. Für jede gefundene Sourcecode Datei wird anschließend ein Objekt vom Typ CSProjectSourceFile angelegt und der SourceFiles Collection hinzugefügt.

Im Konstruktor der CSProjectSourceFile Klasse werden dann noch zusätzliche Informationen über die entsprechende Datei ausgelesen. In der unten stehenden Abbildung sieht man das Auslesen von Informationen wie AutoGen, also ob es sich um autogenerierten Code handelt, DependentUpon, also welche andere Datei diese eventuell erfordert und den SubType der Datei.

```
foreach (XmlNode objChildNode in objNodeSourceInformation)
{
    if (objChildNode.Name == "SubType")
    {
        switch (objChildNode.InnerText)
        {
            case "Form":
                SubType = SourceSubTypes.Form;
                break;
            ....
        }
    }
    else if (objChildNode.Name == "DependentUpon")
        DependsOn = objChildNode.InnerText;
    else if (objChildNode.Name == "AutoGen")
        AutoGen = objChildNode.InnerText == "True" ? true : false;
    else if (objChildNode.Name == "DesignTime")
        DesignTime = objChildNode.InnerText == "True" ? true : false;
}
```

Abbildung 52 - Auslesen verschiedener Zusatzinformationen für Sourcecode Dateien

Diese Prozedur wird entsprechend auch für die anderen Bereich der csproj Datei, also die Referenzen und Ressourcen durchgeführt.

Dieses Vorgehen bietet sich prinzipiell für alle Sprach-Plug-Ins an, die eine Sprache mit zentraler Steuerdatei, wie im vorliegenden Fall die csproj Datei, aufweisen. Für andere Sprachen, wie etwa PHP, die dieses Konzept nicht kennen muss entsprechend auf eine andere Lösung zurückgegriffen werden. Für das Beispiel PHP wäre möglich alle Dateien auszulesen, die sich im jeweiligen Ordner befinden und beim Parsen der Dateien eventuell vorhandene Include Dateien von außerhalb des Ordners nachzuladen. Je nach Sprache sollte aber grundsätzlich möglichst früh im Analyseprozess eine Liste aller erforderlichen Dateien erstellt werden.

Bevor die Analyse der Dateien selbst durchgeführt werden kann muss zuerst der Code Lexer initialisiert werden. Dies ist natürlich nur nötig, falls auch der mitgelieferte Code Lexer verwendet werden soll, andernfalls müssen an dieser Stelle die eventuell nötigen Vorarbeiten für den verwendeten Code Lexer erledigt werden.

Da für das C# Plug-In allerdings der mitgelieferte Code Lexer verwendet werden soll wird hier eine eigene Klasse für diese Vorarbeiten angelegt. In der CSharpMatchKeywords Klasse werden alle sprachspezifischen Keywords und Special Characters definiert und anschließend an den Code Lexer übergeben.

```
List<IMatcher> keywords = new List<IMatcher>
                          {
                              new MatchKeyword(TokenType.ClassDeclaration, "class"),
                              new MatchKeyword(TokenType.NamespaceDeclaration, "namespace"),
                              new MatchKeyword(TokenType.Include, "using"),
                              ...
                              new MatchKeyword(TokenType.NotCompare, "!="),
                              new MatchKeyword(TokenType.Compare, "=="),
                              new MatchKeyword(TokenType.Equals, "="),
```

Abbildung 53 - Definieren von sprachspezifischen Keywords und Special Characters

Dazu werden jeweils eine Liste für die Keywords und die Special Characters erstellt und dort für jeden benötigten Eintrag das Mapping zwischen den im Lexer definierten TokenTypes und dem String wie er im Sourcecode anzutreffen ist hergestellt. In der obigen Abbildung sieht man zum Beispiel wie dem TokenType ClassDeclaration der entsprechende String class zugeordnet wird und die Special Characters für verschiedene Vergleichs- und Zuweisungsoperationen gesetzt werden.

Anschließend kann der Sourcecode analysiert und der entsprechende Abstract Syntax Tree erstellt werden. Dazu wird zuerst ein root Knoten angelegt unter dem alle Analyseergebnisse zusammengefasst werden.

```
TreeNode projectRoot = new TreeNode() { nodeContent = "ProjectRoot", nodeType = NodeTypes.Default };
```

Abbildung 54 - Rootknoten für den zu analysierenden Sourcecode

Unter diesen wird für jede gefundene csproj Datei ein entsprechender Projektknoten erstellt und diesem alle gefundenen Referenzen als ExtendedAttribute angehängt.

```
if (!String.IsNullOrEmpty(reference.HintPath) && reference.HintPath.EndsWith(".dll"))
{
   referencePath = Path.GetFullPath(Path.Combine(Path.GetDirectoryName(csprojFile), reference.HintPath));
}
else if (!String.IsNullOrEmpty(reference.Include))
{
    referencePath = reference.Include;
}
projectNode.nodeAttributes.Add(new ExtendedAttribute() { attributeType = "reference", attributeInformation = referencePath });
```

Abbildung 55 - Auslesen der Projektreferenzen und Anhängen als ExtendedAttribute an den Projektknoten

Nun können die eigentlichen Sourcecode Dateien für die jeweilige csproj Datei analysiert werden. Im vorliegenden Fall werden die Dateien dazu als Text eingelesen und dem Code Lexer übergeben, wodurch man eine Liste aller im Sourcecode vorkommenden Token, den ParseableTokenStream erhält. Diese Liste an Token wird dann durchlaufen und versucht dort bestimmte Muster, wie eine Klassendeklaration, Funktionsaufrufe oder if Abfragen zu erkennen.

Die Schwierigkeit beim Erkennen dieser Muster hängt stark davon ab wie strikt sie definiert sind und ob sie eventuell eindeutige Elemente enthalten. Als Beispiel für ein einfach zu erkennendes Muster mag eine Klassendeklaration dienen. Diese wird ganz klar durch das voranstehende Keyword class definiert und hat immer in etwa denselben Aufbau:

Access Modifier class Name: Superclass, Interface {

Da der Token vom Typ ClassDeclaration auch nur in diesem Kontext vorkommen kann ist es natürlich relative einfach eine Klassendeklaration zu erkennen. Sobald der TokenType ClassDeclaration gefunden wird muss man nur noch die entsprechenden Attribute der Deklaration auslesen und den entsprechenden TreeNode erstellen.

```
private TreeNode handleClassDeclaration()
{
    String descriptor = "";
    while (!this.tokenStream.Peek(1).TokenType.IsOneOf<TokenType>(TokenType.LBracket))
    {
        this.tokenStream.Consume();
        descriptor += this.tokenStream.Current.TokenValue;
    }
    if (!this.tokenStream.Current.TokenType.IsOneOf<TokenType>(TokenType.LBracket))
    {
        this.tokenStream.Consume();
    }
    String className = descriptor.Split(':')[0];
    TreeNode classNode = new TreeNode() { nodeType = NodeTypes.ClassDeclaration, nodeContent = className };
    if (descriptor.Contains(':'))
    {
        String[] interfaces = descriptor.Split(':')[1].Split(',');
        foreach (String entry in interfaces)
        {
            classNode.addAttribute(new ExtendedAttribute() { attributeName = "interface", attributeInformation = entry });
        }
    }
    return classNode;
}
```

Abbildung 56 - Erstellen eines TreeNodes für Klassendeklarationen

Die Funktion in obiger Abbildung wird aufgerufen, sobald ein Token vom Typ ClassDeclaration gefunden wurde. Initial steht der Pointer im ParseableTokenStream auf der ClassDeclaration. Nun wird solange nach vorne durch den ParseableTokenStream iteriert, bis die öffnende geschweifte Klammer gefunden wird. Der dadurch gewonnene String entspricht dem Schema

Name: Superclass, Interface

Nachdem dieser String aufgeteilt wurde in den Klassennamen und die Interfaces kann damit dann der TreeNode angelegt und die Interfaces als ExtendedAttribute hinzugefügt werden. Zusätzlich könnten nun noch die Access Modifier oder andere Keywords ausgelesen werden und zwischen Superclass und Interface unterschieden werden, dies ist allerdings in der vorliegenden Version nicht nötig, da diese Informationen im späteren Framework nicht ausgewertet werden. Sollte der Benutzer eine Metrik verwenden wollen, die auf diese Informationen aufbaut, beispielswiese wenn es Probleme mit zu häufig verwendeten public Klassen gibt, muss das Sprach-Plug-In an dieser Stelle entsprechend angepasst werden um die benötigten Informationen zu erfassen.

Weitaus schwieriger zu erkennen sind dann zum Beispiel Funktionsaufrufe. Diese verfügen nicht über ein eindeutiges Keyword, durch das sie identifiziert werden könnten und haben auch lediglich ein sehr schwammiges Schema. Funktionen können alleine für sich aufgerufen werden, sie können so aufgerufen werden, dass der Rückgabewert einer Variable zugewiesen wird, sie können im Bedingungsteil einer if Abfrage aufgerufen werden und noch viele Möglichkeiten mehr. Je nach Art des Aufrufens ändert sich auch das grundlegende Schema. Dabei sind unter anderem folgende Schemata möglich:

```
Typ Variablenname = Funktionsname(Parameter);
Funktionsname(Parameter, Parameter);
if(Funktionsname() == Variable)
```

Führt man diese Liste fort, stellt man fest, dass die einzige Gemeinsamkeit all dieser Aufrufe der Funktionsname und das Klammerpaar ist. Alle anderen Kriterien sind optional. Man hat also prinzipiell diese beiden Ansatzpunkte um einen Funktionsaufruf zu identifizieren. Wobei beide Punkte relativ schlecht geeignet sind, da sie jeweils auch auf andere Schemata passen könnten. So könnte beispielsweise der Funktionsname, der einem Token vom Typ Word entspricht auch ein Variablenname oder ein Klassenbezeichner sein, während das Klammerpaar auch zu einer if Abfrage oder while Schleife gehören könnte.

Im vorliegenden Fall wurde sich daher spontan für das Klammerpaar, speziell für die öffnende Klammer als Ansatzpunkt für das Erkennen eines Funktionsaufrufs entschieden.

```
private Dictionary<String, String> parseMethodCall()
{
    Dictionary<String, String> methodCall = new Dictionary<String, String>();
    int openParenthPosition = 0;
    int closeParenthPosition = this.tokenStream.PeekUntil(TokenType.CloseParenth, openParenthPosition, 1000);

    if (!this.tokenStream.PeekBack(1).TokenType.IsOneOf<TokenType>(TokenType.Word))
    {
        return null;
    }

    methodCall["methodname"] = this.tokenStream.getPeekBackValuesOfType(1, TokenType.Dot, TokenType.Word);


    if (closeParenthPosition > openParenthPosition)
    {
        methodCall["parameters"] = this.tokenStream.getPeekValues(openParenthPosition, closeParenthPosition);
    }

    if (closeParenthPosition > openParenthPosition && methodCall.ContainsKey("methodname"))
    {
        for (int i = 0; i <= closeParenthPosition; i++)
        {
            this.tokenStream.Consume();
        }
    }
    else
    {
        return null;
    }

    return methodCall;
}
```

Abbildung 57 - Funktion zum Erkennen von Funktionsaufrufen

Die oben abgebildete Funktion wird daher aufgerufen, sobald im ParseableTokenStream ein Token vom Typ OpenParenth, eine öffnende, runde Klammer gefunden wird. Als erster Schritt wird anschließend die dazu passende schließende Klammer gesucht und geprüft, ob sich vor der öffnenden Klammer ein Token vom Typ Word befindet. Befindet sich vor der öffnenden Klammer kein Word, dann kann es sich nicht um einen Funktionsnamen handeln und es wird entsprechend null zurückgegeben.

Wird ein Word Token vor den Klammern gefunden, dann wird anschließend versucht der Methodenname auszulesen. Dazu wird im ParseableTokenStream solange nach hinten iteriert, bis kein Token vom Typ Dot oder Word mehr gefunden wird. Dadurch wird versucht auch verkettete Funktionsnamen nach dem Schema Klasse.Funktionsname und ähnliches zu erkennen.

Zum Schluss werden noch die entsprechenden Übergabeparameter ausgelesen und der ParseableTokenStream solange weiteriteriert, bis er am Ende des Funktionsaufrufs steht. Die Funktion gibt dann den gefundenen Funktionsaufruf als Dictionary zurück, oder null, falls irgendwann im Laufe der Erkennung festgestellt wurde, dass es sich nicht um einen Funktionsaufruf handelt.

```
case TokenType.OpenParenth:
    Dictionary<String, String> methodCall = this.parseMethodCall();
    if (methodCall != null)
    {
        TreeNode methodCallNode = new TreeNode() { nodeType = NodeTypes.FunctionCall, nodeContent
        parentNode.addChildNode(methodCallNode);
    }
    else
    {
        this.tokenStream.Consume();
    }
    break;
```

Abbildung 58 - Code zur Behandlung von öffnenden Klammern

Durch die Rückgabe von null, falls kein Funktionsaufruf erkannt wurde kann der Token für weitere Erkennungsversuche genutzt werden. Im oben abgebildeten Codeausschnitt sieht man, dass die Funktion parseMethodeCall aufgerufen wird und für den Fall, dass ein valider methodCall zurückgegeben wird, wird der entsprechende TreeNode angelegt. Wird allerdings null zurückgegeben, könnte an dieser Stelle eine erneute Prüfung stattfinden, beispielsweise auf eine if Abfrage. In obigem Beispiel wird allerdings keine weitere Prüfung durchgeführt und im Fehlerfall lediglich der ParseableTokenStream eins weiteriteriert.

Nach diesem Schema wird dann der gesamte ParseableTokenStream durchlaufen und nach und nach versucht jede gewünschte Signatur zu erkennen und in einen entsprechenden TreeNode umzuwandeln. Dazu muss für jede zu erkennende Signatur eine entsprechende Funktion definiert werden, die diese Signatur erkennt und diese an geeigneter Stelle beim Durchlaufen des ParseableTokenStreams aufgerufen werden. Dabei ist unter Umständen zu beachten, dass Signaturen, die sich überschneiden können, also etwa Signatur A ist Teil von Signatur B, in der richtigen Reihenfolge aufgerufen werden, damit nicht eine der Signaturen nie erreicht werden kann.

Nach dieser ersten Analyse müssen in einem nächsten Schritt die gefundenen Analyseergebnisse aufbereitet werden. Dazu zählt zu einen das Säubern der gefundenen Knoten, so müssen zum Beispiel alle gefundenen FunctionCalls aufgelöst werden in die entsprechenden tatsächlichen Funktionsnamen. Dazu müssen zuerst alle Bestandteile des FunctionCalls analysiert werden um festzustellen, ob es sich um eine lokale Funktion oder um eine Funktion aus einer anderen Klasse handelt. Für letzteren Fall müssen dann mitunter Aufrufe nach dem Schema Objekt.Funktion in das entsprechende Klasse.Funktion umgewandelt werden. Anschließend können dann aus den jeweiligen Klassen die passenden Funktionsknoten gefunden und zum FunctionCall die entsprechende Guid hinterlegt werden.

Ein weiterer Schritt ist das Sammeln der Detailinformationen und Filter. Dazu müssen die entsprechenden Werte, wie etwas Lines of Code oder Verschachtelungstiefe gesammelt und als ExtendedAttributes mit dem Typ intcodesmell an die jeweiligen Knoten angehängt werden. Parallel dazu müssen auch für die gewünschten Filter die Werte zu den entsprechenden Funktions- und Klassenknoten gesetzt werden. Das heißt für jeden Knoten muss entschieden werden ob er für einen bestimmten Filter einen bestimmten Filterwert zugeordnet bekommen soll. Dieser muss dann als weiteres ExtendedAttribute vom Typ filter unter den Knoten angehängt werden.

Als letzter Schritt im Sprach-Plug-In muss dann lediglich der erstellte Baum aus TreeNodes an das aufrufende Codeanalyse Framework zurückgegeben werden.

Testphase

Teil der Erstellung des Codeanalyse Frameworks ist auch eine umfangreiche Testphase. Diese wird, wie bereits beschrieben in mehrere Teile, analog zu den einzelnen Teilen des Frameworks aufgeteilt. Zusätzlich soll es mehrere Testbasen, also zu analysierende Sourcecode Projekte geben, gegen die getestet wird. Im Anschluss soll hier sowohl die finale Definition der Test Cases und der Testbasen erfolgen, als auch die Protokollierung des eigentlichen Tests.

Test Cases

Bei den Test Cases sollte ursprünglich jeweils zwischen den drei Teilen des Frameworks, dem Sprach-Plug-In, dem Framework und der Visualisierungsschicht unterschieden werden. Die Trennung von Framework und Sprach-Plug-In Test wird allerdings nicht mehr als nützlich angesehen. Es wurde entschieden, dass der Aufwand ein standardisiertes Sprach-Plug-In zu erstellen, das vorher definierte Rückgabewerte zurückgibt größer ist als der Nutzen es erlauben würde. Ein kombinierter Test eines tatsächlichen Sprach-Plug-Ins mit dem Framework würde dieselben Fehler finden und wäre einfacher zu realisieren. Daher soll hier lediglich zwischen Framework inklusive Sprach-Plug-In und Visualisierungsschicht unterschieden werden.

Framework

Hier soll der erste Test Case jeweils für alle vorhandenen Testbasen ausgeführt und das Ergebnis geprüft werden. Für die übrigen Test Cases reicht der Test mit einer Testbasis, da der Test unabhängig vom Inhalt der Testbasis zum selben Ergebnis kommen muss.

Test Case 1: Aufruf mit korrektem Plug-In und dazu passender Programmiersprache

Das Framework soll gestartet werden und ein Sprach-Plug-In und ein Sourcecode Projekt mit passender Sprache ausgewählt werden. Nach durchgeführter Analyse wird das Ergebnis gespeichert und die Validation auf Basis der gespeicherten XML Datei durchgeführt.

Es wird erwartet, dass alle Kennzeichen aus dem Sourcecode Projekt richtig in der XML Datei gespeichert wurden

Test Case 2: Aufruf mit fehlerhaftem Plug-In, das die benötigten Interfaces und Funktionen nicht implementiert

Das Framework soll gestartet werden und eine Dll ohne Sprach-Plug-In Interface geladen werden.

Es wird erwartet, dass eine entsprechende Fehlermeldung ausgegeben wird und kein Sprach-Plug-In zur Auswahl steht

Test Case 3: Aufruf mit Plug-In, das nicht zur zu analysierenden Programmiersprache passt

Das Framework soll gestartet und ein gültiges Sprach-Plug-In geladen werden. Anschließend soll eine Codebasis in unpassender Programmiersprache analysiert werden.

Es wird erwartet, dass das Analyseergebnis leer ist und eine entsprechende Meldung ausgegeben wird.

Visualisierungstool

Die Tests zum Visualisierungstool sollen jeweils mit allen zur Verfügung stehenden Testbasen durchgeführt werden.

Test Case 4: Navigieren in der Abhängigkeitsansicht

Der Code Visualizer soll gestartet und ein vorhandenes Analyseergebnis geladen werden. Anschließend soll über die Auswahlbox im FilterPanel verschiedene Funktions- und Klassenknoten ausgewählt werden.

Es wird erwartet, dass sich der TreeView der Abhängigkeitsansicht entsprechend updatet und jeweils den gewählten Knoten mit seinen Abhängigkeiten darstellt.

Test Case 5: Wechseln zwischen Funktions- und Klassenebene

Es soll der Code Visualizer gestartet und ein Analyseergebnis geladen werden. Danach soll in den einzelnen Ansichten zwischen Klassen- und Funktionsebene gewechselt werden.

Es wird erwartet, dass sich alle spezifischen Elemente entsprechend der gewählten Ebene verändern, also die DropDowns für Funktionen und Klassen, die angebotenen Filter und der TreeView.

Test Case 6: Prüfen der Filtermöglichkeiten

Es soll der Code Visualizer gestartet und ein Analyseergebnis geladen werden. In der Filteransicht sollen nach verschiedenen Filterkriterien eingeschränkt werden. Dies soll jeweils für die Funktions- als auch für die Klassenansicht durchgeführt werden.

Es wird erwartet, dass nur noch die Knoten angezeigt werden, die den gesetzten Filtern entsprechen.

Test Case 7: Test der Detailansicht

Es soll der Code Visualizer gestartet und ein Analyseergebnis geladen werden. Nach Wechsel in die Detailansicht sollen verschiedene Funktions- und Klassenelemente ausgewählt werden und die angezeigten Details geprüft werden.

Es wird erwartet, dass alle angezeigten Details korrekt sind und alle im Analyseergebnis definierten Details angezeigt werden.

Test Case 8: Prüfen der To-do Ansicht

Der Code Visualizer wird aufgerufen und ein Analyseergebnis geladen. Nach Wechsel in die To-do Ansicht wird zwischen den einzelnen Kriterien gewechselt und jeweils die Liste der gefundenen To-do Einträge geprüft.

Es wird erwartet, dass alle To-do Einträge, die im Analyseergebnis definiert sind korrekt ausgelesen und angezeigt werden.

Testbasis

Für den eigentlichen Test werden zwei verschiedene Testbasen benötigt. Um einen Test über das komplette Analyseergebnis durchzuführen wird eine Testbasis benötigt, die entsprechend klein ist um eine übersichtliches Ergebnis zu erlangen. Um den Test unter realen Bedingungen abzubilden soll zusätzlich eine große Testbasis benutzt werden, bei der dann nur noch stichprobenartig die einzelnen Ergebnisse geprüft werden.

Testbasis 1

Als erste Testbasis wird ein eigenes Projekt erstellt, welches drei Klassen und jeweils zehn Funktionen erhält. Diese Funktionen sollen sich mehrfach gegenseitig oder auch rekursiv selbst aufrufen. Zusätzlich sollen in einzelnen Funktionen if, while und for Konstrukte zu finden sein, die auch mehrere Ebenen tief verschachtelt sein können.

Die eigentliche Funktionalität dieser Testbasis ist irrelevant, es muss lediglich gewährleistet sein, dass sie inhaltlich fehlerfrei ist und entsprechend übersetzt werden kann.

Testbasis 2

Als zweite Testbasis soll das Analyseframework selbst benutzt werden. Da das realisierte Sprach-Plug-In für die Sprache C# gedacht ist und das Analyseframework in dieser Sprache geschrieben ist, bietet es sich entsprechend an die Analyse auch auf sich selbst durchzuführen.

Testbasis 3

Als dritte Testbasis soll ein Open Source Projekt benutzt werden. Da mit dem Code Lexer für das Codeanalyse Framework bereits ein entsprechendes Open Source Projekt im Einsatz ist soll dieses als dritte Testbasis benutzt werden. Dabei soll jedoch das komplette Projekt inklusive der im Codeanalyse Framework nicht benutzten Teile als Testbasis benutzt werden.

Testdurchführung

Der Test wurde jeweils für die einzelnen Test Cases unter Verwendung der vorhandenen Testbasen durchgeführt. Gefundene Fehler wurden entsprechend behoben und anschließend ein erneuter Test durchgeführt. Dies wurde solange wiederholt, bis kein kritischer Fehler mehr gefunden wurde. Nachfolgend sind die einzelnen Test Cases mit den Ergebnissen je Testbasis aufgeführt.

Test Case	Testbasis 1	Testbasis 2	Testbasis 3
Test Case 1	Erfolgreich	Erfolgreich	Erfolgreich
Test Case 2	Erfolgreich	N. A.	N. A.
Test Case 3	Erfolgreich	N. A.	N. A.
Test Case 4	Erfolgreich	Erfolgreich	Erfolgreich
Test Case 5	Erfolgreich	Erfolgreich	Erfolgreich
Test Case 6	Erfolgreich	Erfolgreich	Erfolgreich
Test Case 7	Erfolgreich	Erfolgreich	Erfolgreich
Test Case 8	Erfolgreich	Erfolgreich	Erfolgreich

Nach erfolgtem Test und Beseitigung aller gefundener Fehler ist nunmehr die Entwicklung dieses Projekts abgeschlossen, die möglichen Erweiterungen, die sich bieten werden hoffentlich folgen.

Fazit

Im Laufe dieser Arbeit hat sich gezeigt, dass das Feld Legacy Code noch weit stiefmütterlicher behandelt wird als ursprünglich angenommen. Gemessen an der Gesamtzahl befassen sich relativ wenige Publikationen direkt und intensiver mit der Thematik Legacy Code, besonders mit dem Komplex wie man Legacy Code behandelt. Zum Bereich Vermeidung von Legacy Code gibt es dagegen etwas mehr Lesematerial. Dies ist umso mehr Verwunderlich, wenn man sich vor Augen hält wie weit verbreitet das Phänomen Legacy Code tatsächlich ist und wie allumfassend seine Auswirkungen sind.

Legacy Code ist jeweils nicht nur ein Phänomen, das in der Entwicklung selbst auftritt und dann darauf beschränkt bleibt. Legacy Code ist vielmehr ein Phänomen, das in jeder Phase der Softwareentwicklung – von der Konzeption über die eigentliche Entwicklung bis hin zum Support – zum Tragen kommt. Bereits bei der Konzeption können dabei die Weichen gestellt werden, ob später, mit höherer Wahrscheinlichkeit Legacy Code entstehen wird oder nicht. Fehlt in den Anfängen eines Projekts bereits eine gute Planung, auch in Hinblick auf die Lebenszeit und die möglichen Erweiterungen des Projekts, so steigt das Risiko im Laufe der Zeit Legacy Code zu erhalten enorm an. Genauso kann auch ein schlechter Support- und Fehlerbehebungsprozess ein Auslöser für Legacy Code werden. Wird bei gefundenen Fehlern nicht darauf geachtet, wie die Fehlerbehebung durchgeführt wird, sondern einfach irgendwelche Workarounds und Dirty Fixes im Code hinterlassen, so kann sich auch hier noch aus einer anfangs sauberen Codebasis ein Projekt mit Legacy Code entwickeln.

Aber nicht nur der Auslöser für Legacy Code kann im gesamten Spektrum der Softwareentwicklung gefunden werden, sondern auch die Auswirkungen sind überall zu spüren. Legacy Code lässt sich naturgemäß schlecht warten und entsprechend ist die Fehlerbehebung und damit der komplette Supportprozess entsprechend schwieriger und kostenaufwändiger. Aber auch für die Konzeption von gewünschten Erweiterungen stellt Legacy Code ein Problem dar, da für jede potentielle Änderung möglicherweise einiges an der Codebasis geändert werden muss oder entsprechende Workarounds eingebaut werden müssen um ein neues Feature überhaupt erst anbauen zu können.

Bereits bei der Entwicklung des doch relativ kleinen Codeanalyse Frameworks konnten diese Effekte sehr gut beobachtet werden. Es mussten während der Entwicklung immer wieder kleinere Refactoring Aktionen durchgeführt werden um die angestrebte Modularität des Frameworks zu erreichen. Und selbst nach Fertigstellung finden sich noch immer stellen, die wenn sie nicht bereits Legacy Code sind, so zumindest erste Ansätze für Legacy Code darstellen.

Rückblickend können vielleicht erste Gründe für diese Tendenz zu Legacy Code identifiziert werden. Ein wichtiger Grund wird wohl bereits in der Tätigkeit selbst liegen. Die Konzeption eines komplexen Softwareprojekts in einem Wasserfallansatz ist sicher nicht einfach. Vor allem für unerfahrenere Entwickler ist es schwierig alle Aspekte eines Softwareprojekts schon zu Beginn, bevor noch eine Zeile Code geschrieben ist abzuschätzen und entsprechend alle Eventualitäten einzuplanen. Je größer und komplexer das Softwareprojekt und je unerfahrener

der Entwickler umso mehr Punkte werden wohl vergessen werden, die dann im Nachhinein noch möglichst sauber in das bereits vorhandene Konstrukt eingepflegt werden müssen.

Ein weiterer Punkt mag der schlechte Toolsupport während der Entwicklung sein. Mit wachsender Codebasis wird es immer schwieriger den Gesamtüberblick über das Projekt zu behalten und Refactoring entsprechend sauber und koordiniert durchzuführen. Häufig wird man Codeduplizierung erzeugen, weil man einfach nicht mehr im Blick hat, dass es diese Funktion bereits in leicht anderer Form irgendwo gibt. Fehlt dann ein geeignetes Tool um den Entwickler darauf aufmerksam zu machen und ihm bei der Behebung zu unterstützen, so driftet der Code sehr leicht in die Legacy Code Ecke.

Als letzten Punkt schließlich könnten Einschränkungen in der verwendeten Programmiersprache selbst ein Auslöser sein. Setzt man, wie im Codeanalyse Framework eine Objektorientierte Programmiersprache ein, so kann man auf Situationen treffen, bei denen möglicherweise eine Funktionelle Sprache besser geeignet wäre, oder entsprechend umgekehrt. Ans diesen Punkten stellt sich dann die Frage, ob man die Sprache wechseln sollte, ob man dies überhaupt kann, oder ob durch den Wechsel die Komplexität zu sehr steigen würde. Je nachdem für was man sich entscheidet muss man aber dennoch Abstriche in der Sauberkeit des Codes hinnehmen und bietet damit einen weiteren potentiellen Ansatz, an dem sich Legacy Code ausbreiten kann.

Man kann aus dem Codeanalyse Projekt aber auch Punkte ableiten, die halfen Legacy Code zu mindern. Ein großer Punkt ist hier sicherlich die Anzahl der beteiligten Entwickler – nämlich genau einer, der Autor – die es sehr leicht machte eine umfassende Kenntnis der Codebasis und der Planung zu gewährleisten. Sind dagegen mehrere Entwickler beteiligt muss man auch entsprechend mehr Aufwand in die Kommunikation zwischen den Entwicklern investieren um alle Entwickler in die Planung einzubeziehen und den Kenntnisstand über die Codebasis außerhalb des selbstgeschriebenen Codes zu erhöhen. Kennen einzelnen Entwickler dagegen nur ihre jeweilige Nische, in der sie arbeiten so erhöht sich auch hier wieder sehr leicht die Wahrscheinlichkeit Legacy Code zu entwickeln.

Ein weiterer Punkt der das Risiko auf Legacy Code zumindest mitigiert hat ist die Fokussierung auf genau diesen Aspekt der Softwareentwicklung. Bei der Entwicklung des Frameworks wurde – dem Thema entsprechend – besonderes Augenmerk auf die Vermeidung von Legacy Code gelegt. Trotz dessen konnte es nicht einmal hier komplett vermieden werden auch Stellen mit Legacy Code zu erhalten. Bei Projekten, bei denen der Fokus entsprechend weniger oder möglicherweise gar kein Fokus auf die Vermeidung von Legacy Code liegt wird es vermutlich noch wesentlich einfacher zur Bildung von Legacy Code kommen.

Aufgrund dieser Einfachheit mit der Legacy Code entstehen kann und der relativen Schwere mit der man ihn wieder eindämmt ist es nach Ansicht des Autors besonders wichtig entsprechende Tools – wie auch dieses Codeanalyse Framework – zu besitzen um dieser Problematik zumindest im Ansatz zu begegnen. Das vorliegende Codeanalyse Framework ist dabei sicher nicht mehr als ein Tropfen auf dem heißen Stein, da es alleine auf die Entwicklungskomponente abzielt und alle umliegenden von Konzeption bis Support außer Acht lässt. Dennoch ist es ein Start der gemacht werden muss, vor allem auch an einer Stelle an der mit Toolsupport etwas

bewegt werden kann. Sicher müssen alle Bereiche der Softwareentwicklung stärker für das Thema Legacy Code sensibilisiert werden, aber dabei handelt es sich um Ausbildungsmaßnahmen, Prozessverbesserungen und ähnliches. Dies sind keine Themen, die sich durch Tools verbessern lassen. Tools können aber direkt beim Entwickler ansetzen und ihn unterstützen.

Wie wichtig dieses Thema ist zeigt sich auch gerade an einem Beispiel aus der Praxis. Im vorliegenden Fall handelt es sich um eine Codebasis, die mehrere Millionen Lines of Code enthält und inzwischen eine Komplexität erreicht hat, bei der der einzelne Entwickler keinen Überblick mehr hat welche Auswirkungen seine Änderungen haben könnten. Es entstand daher die Idee das im Rahmen dieser Arbeit entwickelte Codeanalyse Framework zu benutzen um genau an dieser Stelle den Entwickler zu unterstützen. Es soll daher bei jedem Check-In eines Entwicklers ins zentrale Sourcecode Repository über das Codeanalyse Framework geprüft werden, welche anderen Funktionen von der Änderung betroffen sind und diese entsprechend auch für den Build und anschließenden Test vorgemerkt werden.

Mit dem Einsatz in der Praxis wäre dann auch die größtmögliche Form des Erfolgs für das hier entstandene Projekt erreicht.

Anhang

Bibliographie

[Aro89]
Proceedings of the 1978 Army Numerical and Computers Analysis Conference.
US Army Research Office 1978.

[Tin01]
Paul C. Tinnirello. New Directions in Project Management (Best Practices)
Auerbach Publications 1. Ausgabe September 2001 ISBN 978-0849311901

[Ive10]
William Noel Ivey, Marieke Lewis. Astronautics and Aeronautics: A Chronology, 2001 – 2005.
Nasa History Division February 2010.

[Fea11]
Michael C. Feathers. Effektives Arbeiten mit Legacy Code – Refactoring und Testen bestehender Software.
mitp 1. Auflage 2011. ISBN 978-3-8266-9021-1

[Gem03]
Steven E. Gemeny. Longevity Planning, A Cost Reduction Strategy for Ground Systems of Long Duration Space Missions.
The Johns Hopkins University Applied Physics Laboratory 2003

[McC96]
Arthur H. Watson, Thomas J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.
NIST Special Publication 235 September 1996

[McC04]
Steve McConnell. Code Complete: A Practical Handbook of Software Construction
Microsoft Press 19.06.2004

[Cov06]
Coverity. Coverity Scan Project
Coverity and U.S. Department of Homeland Security May 2013

[Kro13]
Anton Kropp. Building a Custom Lexer & A Handrolled Language Parser
http://onoffswitch.net/building-a-custom-lexer/
http://onoffswitch.net/a-handrolled-language-parser/
https://github.com/devshorts/LanguageCreator

[-]
Repository des Codeanalyse Framework
https://github.com/Schneephin/codeanalysis