BILDERSPRACHE

Sourcecode als Stadt visualisieren

Menschen sind Genies beim Erkennen von visuellen Mustern und Objekten – ganz im Gegenteil zu Maschinen. Auf diesem Gebiet der sogenannten automatischen Bildverarbeitung (Optical Character Recognition, OCR), ist noch sehr viel zu erforschen. Denn komplexe Maschinen benötigen in der Zukunft deutlich bessere visuelle Algorithmen, wenn sie sicher durch unsere menschengemachten Umgebungen navigieren möchten. Wir sind sehr visuelle Wesen. Einen Großteil der Informationen unserer Umwelt nehmen wir über dieses Sinnesorgan auf. Dafür sorgen die circa 130 Millionen Sehsinneszellen auf unserer Netzhaut. Deshalb lassen sich auch viele Informationen über den visuellen Kanal transportieren – sogar solche, die uns ansonsten nur langweilen würden, wie lange oder komplexe Zahlenreihen. Genau diese fallen bei Softwaremetriken allerdings zwangsläufig an. Die Lösung: Ergebnisse dieser Metriken zu visualisieren, mit Tools wie CodeCity und Moose.

Visualisierung bedeutet, abstrakte Daten und Zusammenhänge in eine graphische beziehungsweise visuell erfassbare Form zu bringen. Metriken zu visualisieren ist eine Mischung aus den Bereichen der sogenannten Informationsvisualisierung und Wissenschaftlichen Visualisierung. Erstere ist die Visualisierung von Daten, die nicht unmittelbar mit physikalischen Zuständen und Prozessen assoziiert werden. Wissenschaftliche Visualisierungen gehen ein Stück weiter und müssen drei Kriterien entsprechen: erstens Ausdrucksfähigkeit, zweitens Effektivität und drittens Angemessenheit. Diese Kriterien müssen zwar nicht sehr streng eingehalten werden, helfen aber beim Verständnis der Visualisierungen enorm.

Denn mit Visualisierungen kann auch viel Schabernack getrieben werden. Ein weiteres Sprichwort ist nämlich: Traue keiner Statistik, die du nicht selbst gefälscht hast. Bei Visualisierungen verhält es sich ebenso. Denn durch das Weglassen von Informationen kann ein Diagramm eine völlig andere Aussage erhalten: sehr beliebt ist es etwa, bei Skalen Achsenbeschriftungen zu unterschlagen. Wie die Visualisierungen von CodeCity und Moose diesen Dingen entsprechen, zeigt der Artikel im weiteren Verlauf.

DER FAMIX-STANDARD

Nun aber genug der Einführung zur Visualisierung. Wie schon angesprochen, geht es in diesem Artikel um die Visualisierung von Sourcecode-Metriken mit CodeCity und Moose. Es gibt zwei Hauptgründe, warum sich der Artikel beider Tools annimmt. Zum einen lassen sich so Unterschiede besser verdeutlichen. Zum anderen verwenden beide Programme den sogenannten FAMIX-Standard als Eingabeformat. Deshalb kombinieren wir gleich beide darauf aufbauende Tools.

FAMIX ist eine Familie von Metamodellen, mit der man Modelle repräsentiert, die aus verschiedenen Programmiersprachen generiert worden sind. Die Abkürzung stammt von der Namensgebung „The ***FAM***OOS Information E***x***change Model“. Aufgrund des Namens ist es sicherlich nicht falsch, in diesem Kontext von einer Metasprache oder einem Metaformat zu sprechen. Der FAMIX-Standard wurde geschaffen, um verschiedene Programmiersprachen analysieren zu können. Konkret geht es um die Analyse und Erkennung von Designfehlern in objekt-orientierten Sprachen beziehungsweise Legacy-Systemen. Daher besitzt der FAMIX-Standard ein reichhaltiges API für Abfragen und die Navigation innerhalb des Metamodells. Die Abstraktion von einer konkreten Programmiersprache erlaubt es auch, Analysetools zu implementieren, die mit einer unbekannten Anzahl von Eingabesprachen zurechtkommen. Anstatt das Werkzeug ständig zu erweitern, liegt der Fokus auf der Analyse des Metamodells. Die Überführung der konkreten Programmiersprache in das Metamodell erfolgt vorgelagert.

Das hat aber auch zur Folge, dass ein gesonderter FAMIX-Generator für die genutzte Programmiersprache vorhanden sein muss. Dieser Artikel konzentriert sich auf die Programmiersprache C# und nutzt den Famix-Generator [1] von Thomas Haug [2], um das Metamodell für eine Assembly zu erzeugen. Als Quellprojekt für die Analyse dient scriptcs [3], ein Projekt, mit dem sich lose C# Skriptdateien mit einem simplen Texteditor schreiben und kompilieren lassen. Ebenfalls vom gleichen Autor ist das Analysetool mit dem Namen Workbench [4], das zur sogenannten SharpMetrics Tool Suite gehört. Auch damit ist eine Analyse von C# Code möglich.

Um das FAMIX-Modell zu erzeugen, sind nur wenige Schritte auf der Kommandozeile notwendig. Die folgende Anweisung schnappt sich die Assembly ScriptCs.Core.dll und erzeugt daraus zwei FAMIX-Modelle.

```
FamixGenerator.exe --Inputfile
„ScriptCs.Core.dll“ --Outputfile
ScriptCs.Core-FAMIX.mse
```

Abbildung 1 zeigt den Prozess inklusive Ausgaben auf der Konsole. Allerdings passiert nicht allzu viel, da die Generierung recht schnell über die Bühne geht. Interessant ist dagegen, dass gleich zwei FAMIX-Modelle generiert werden: ein Metamodell nach dem FAMIX 2.1-, das andere nach dem FAMIX 3.0-Standard. Der Hintergrund ist, dass Moose FAMIX 3.0 und CodeCity FAMIX in Version

2.1 erwartet. So lässt sich eine Assembly mit nur einem Aufruf ohne Umwege direkt in beiden Programmen analysieren. Die generierten Dateien sind, im Falle der Assembly ScriptCs.Core.dll für Version 2.1 circa 760 Kilobyte und für Version 3.0 des FAMIX-Standards circa 720 Kilobyte groß. Schon recht umfangreich für eine circa 50 Kilobyte große Eingabedatei.

Bei den FAMIX-Dateien handelt es sich um reine Textdateien. Sie lassen sich also einfach über einen Texteditor wie beispielsweise Notepad++ öffnen. Der Inhalt besteht aus einer Reihe von Entitäten: Namensräume, Dateien, Klassen, Felder und so weiter, natürlich inklusive Querbeziehungen in Form von Referenzen. Gerade letztere sind für eine Analyse von immenser Bedeutung.

Der genaue Aufbau der Dateien ist in Form von Spezifikationen verfügbar. Für die Version 2.1 unter [5], für Version 3.0 unter [6]. Auf diese Dokumente sei an dieser Stelle für Interessierte lediglich verwiesen. Für eine komplette Erklärung sind die Spezifikationen deutlich zu umfangreich.

MOOSE

Moose [7] ist das erste Analyse-Tool, das den FAMIX-Standard als Eingabe nutzt, das wir uns ansehen. Genauer genommen ist es ein kostenfreies Open Source-Tool, das in Pharo [8] entwickelt wurde. Hinter dem Namen Pharo steht eine Open Source-Implementierung der Programmiersprache und der Umgebung Smalltalk.

Die Entwicklung von Moose startete schon 1996 an der Universität von Bern in einem Projekt namens FAMOOS. Dahinter verbirgt sich ein Europäisches Projekt, das es zum Ziel hatte Methoden und Tools zu entwickeln, um Designprobleme in objekt-orientierten Systemen zu

Abbildung 2: Moose direkt nach dem Start

finden. In diesem Rahmen ist auch FAMIX entstanden.

Moose ist ein aktiv entwickeltes und gepflegtes Projekt. Auf der Download-Seite gibt es Installationspakete für Windows, Mac und Linux. Immer für die aktuelle Version 5.0. Die Anwendung an sich kommt vielleicht etwas altbacken daher, bietet aber dennoch einen großen Funktionsumfang. Abbildung 2 zeigt die Anwendung direkt nach dem Starten. Beim grauen Bereich handelt es sich um ein MDI-Fenster, da die Anwendung als MDI-Programm konzipiert ist. Das eben erzeugte FAMIX 3.0-Modell lässt sich ganz simpel über die MSE-Schaltfläche rechts oben importieren. Nach dem Importvorgang wird das geladene Modell in der Übersicht angezeigt. Abbildung 3 zeigt das Moose Panel, nachdem das importierte Modell ausgewählt wurde. Auf der rechten Seite sind alle Elemente aufgeführt, die Moose durch das Einlesen und Parsen des Metamodells gewonnen hat. Beispielsweise werden 227 Klassen, 59 Vererbungen, 1.005 Aufrufe und 534 Methoden angegeben. Dies bietet eine erste Übersicht, wie komplex das Metamodell ist

Abbildung 1: Prozess und Ausgabe auf der Konsole

und damit natürlich auch, wie komplex die Assembly ist, aus der das Metamodell generiert wurde. Natürlich ist das nur ein erster, sehr grober Blick aus der Vogelperspektive. Als Metrik sind diese Werte nur sehr bedingt zu gebrauchen.

Aber Moose kann noch mehr. Eine bessere Metrik für den Einstieg ist die sogenannte System-Komplexität. Sie zeigt, wie viele Klassen es gibt und wie diese in Beziehung zueinander stehen. Abbildung 4 enthält dazu einen Ausschnitt eines Screenshots. Denn solche Übersichten können sehr schnell sehr groß werden. So auch in diesem Fall. Die Komplexität zeigt noch mehr als nur die Klassen als Rechtecke und die Vererbungen untereinander als Verbindungen. Die Anzahl der Methoden wird auf die Höhe der Rechtecke gemappt. Je höher das Rechteck also ist, desto mehr Methoden befinden sich in den jeweiligen Klassen. Zusätzlich wird die Anzahl der Eigenschaften auf die Breite des Rechtecks übertragen. Je breiter, desto mehr Eigenschaften sind vorhanden. Zu guter Letzt spielt auch die Anzahl der Codezeilen pro Klasse eine Rolle. Diese Größe wird auf die Farbe, beziehungsweise auf die Grautöne, gemappt. Je dunkler ein Rechteck ist, desto mehr Codezeilen hat die jeweilige Klasse. Dieses Vorgehen ist typisch für die Visualisierung von Metriken. Nur die Anzahl der Klassen zu visualisieren bringt noch nicht allzu viel. Der Wert lässt sich auch leicht ablesen. Durch die zusätzliche Annotation der Visualisierung mit weiteren Daten wird allerdings die Informationsdichte deutlich erhöht, was die Grafik letztendlich nützlicher macht.

KLASSEN-BLAUPAUSE ALS METRIK

Eine weitere nützliche Metrik ist die sogenannte Klassen-Blaupause. Sie zeigt eine oder mehrere Klassen und setzt die Interna in Bezug zueinander. Abbildung 5 zeigt so eine Blaupause an einer Beispielklasse. Die Struktur einer Klasse wird dabei in fünf Schichten aufgeteilt:

1. Initialization Layer
2. Public Interface Layer
3. Private Implementation Layer
4. Accessor Layer
5. Attribute Layer

Diese Schichten sind von links nach rechts zu lesen. Links in der Grafik sind somit die Elemente zu sehen, die zur Initialisierung der Klasse dienen. Anschließend kommt die öffentliche Schnittstelle, die auch von außen von anderen Klassen genutzt wird. Diese öffentliche Schicht interagiert mit Elementen in der privaten Schicht, in der sich die Implementierungen befinden. Die öffentliche und die private Schicht greifen auf die Zugriffsschicht zu. In C# die Eigenschaften mit Gettern und Settern. Diese Schicht greift wiederum auf die Attributschicht zu, in der alle Attribute visualisiert sind. Diese Grafik visualisiert somit sehr gut, wie stark die Elemente der Klasse intern miteinander interagieren und ob beispielsweise interne Methoden häufig direkt auf Attribute zugreifen oder auch die Accessor nutzen.

Zu guter Letzt zeigt Abbildung 6 (nächste Seite) die Visualisierung von Abhängigkeiten zwischen den einzelnen Namensräumen in der Assembly ScriptCs.Core.dll. Auch hier werden wieder mehrere Metriken auf mehrere Eigenschaften der Elemente abgebildet. Die Breite der Rechtecke ergibt sich aus der Anzahl der Klassen der jeweiligen Namensräume. Die Höhe ergibt sich aus der Anzahl

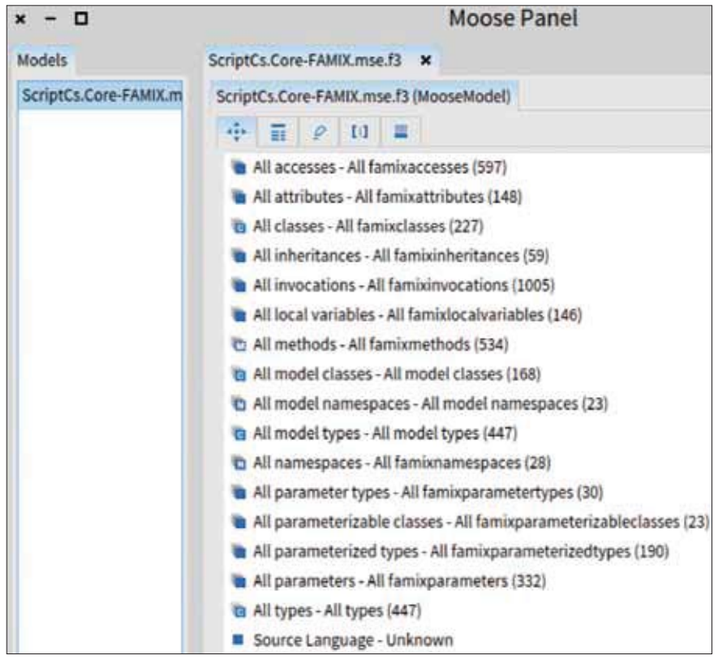

Abbildung 3: Moose Panel nachdem das importierte Modell ausgewählt wurde

Abbildung 4: Ausschnitt eines Screenshots (Moose)

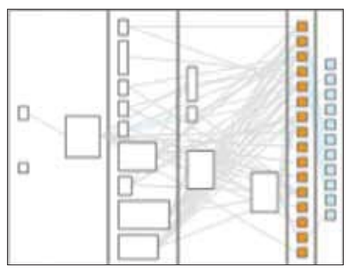

Abbildung 5: Blaupause an einer Beispielklasse (Moose)

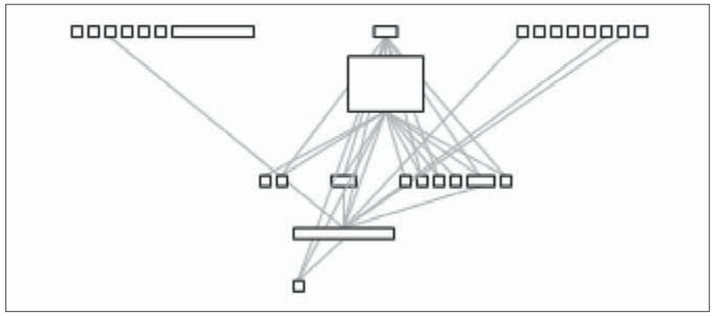

Abbildung 6: Visualisierung von Abhängigkeiten zwischen einzelnen Namensräumen

der Methoden. Und die Verbindungslinien visualisieren letztendlich die Abhängigkeiten.

Diese drei vorgestellten Visualisierungen sind nur ein kleiner Ausschnitt aus dem, was Moose zu bieten hat. Ein guter Gesamtüberblick ergibt sich aus der Dokumentation in Form eines online verfügbaren Buches [9], genannt The Moose Book. Noch sind nicht alle Teile vollständig vorhanden. Trotzdem ist diese Online-Hilfe schon jetzt eine gute Anlaufstelle für Fragen aller Art.

CODECITY

Moose ist nicht das einzige Tool, das mit dem FAMIX-Standard umgehen kann. Wie schon kurz erwähnt, trifft das auch auf CodeCity zu. Dieses Tool arbeitet mit dem etwas älteren FAMIX-Standard in Version 2.1.

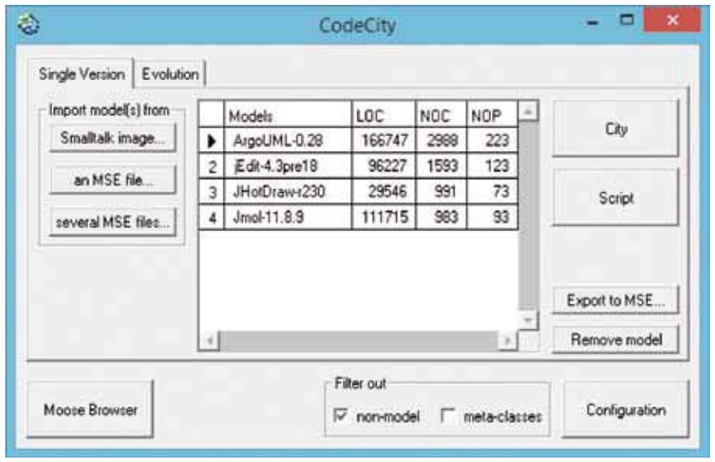

Abbildung 7: CodeCity direkt nach dem Start

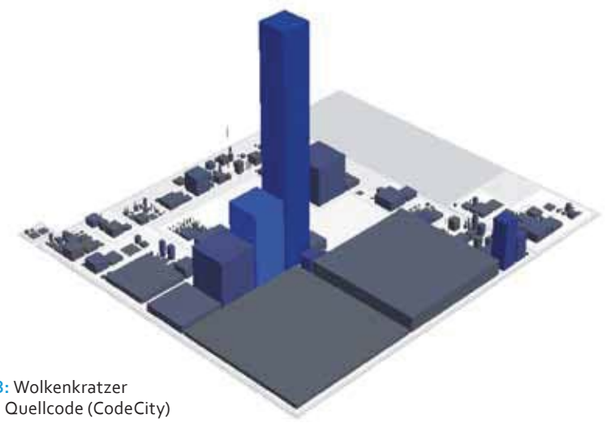

Abbildung 8: Wolkenkratzer visualisieren Quellcode (CodeCity)

Trotzdem ist CodeCity noch nicht altersschwach und gehört zu den bekanntesten Tools aus dem Bereich der Softwareanalyse.

Es macht jedoch einen etwas älteren Eindruck als Moose. Die letzte Version ist vom November 2009. Das heißt natürlich noch nicht allzu viel, da der Funktionsumfang trotzdem gut sein kann. Die aktuelle Version ist 1.4.3 und lässt sich, ebenfalls wie Moose, für Windows, Mac und Linux herunterladen [10]. Geschrieben ist es in Smalltalk, aber der ursprünglichen Version.

Abbildung 7 zeigt die Anwendung direkt nach dem Start. Es sind schon einige Modelle geladen, die alle unterschiedlich groß und komplex sind. Über die mittlere Schaltfläche auf der linken Seite können eigene MSE-Dateien importiert werden. Leider lässt sich das über den FAMIX-Generator erzeugte FAMIX-Modell nicht importieren. Weder in Version 2.1 noch 3.0. Da der FAMIX-Generator Beta-Version ist, kann das schon einmal passieren.

Daher sind die folgenden Abbildungen aus dem Modell zu Jmol entstanden. Jmol ist ein Open-Source-Tool in Java, mit dem wir chemische Strukturen in 3D betrachten. Bekannt geworden ist CodeCity durch die Visualisierung von Quellcode mithilfe der 3D-Städtemetapher. Abbildung 8 zeigt einen entsprechenden Ausschnitt aus einem Screenshot. Wie üblich werden wieder mehrere Eigenschaften in die Elemente der Grafik gemappt. Die Farbe gibt die Anzahl der Quellcodezeilen an. Der hellblaue Wolkenkratzer ist zwar nicht der größte in der dargestellten Stadt, besitzt aber mit über 11.000 Zeilen mit Abstand den meisten Quellcode. Wohlgemerkt in einer Klasse, da alle Rechtecke in dieser Städtemetapher eine Klasse widerspiegeln. Die Höhe kommt durch die Anzahl der Methoden zustande. Hier gewinnt eindeutig der dunkelblaue Wolkenkratzer mit über 770 Methoden. Die Länge und die Breite der Rechtecke werden beide durch die Anzahl der Attribute festgelegt. Viele Attribute bedeuten demnach eine große Fläche.

Durch diese Visualisierung verschiedener Metriken lässt sich schnell sagen, welche Klassen in einem Projekt besonders hervorstechen. Nicht zwingend ist das jedoch im positiven Sinne gemeint, denn viel mehr Attribute oder gar Anzahl Zeilen Quellcode zu haben, muss nicht unbedingt gut sein. Hier sollte bei einem Refactoring angesetzt werden.

Durch eine Konfiguration, die in CodeCity angepasst und auf das aktuelle Projekt zugeschnitten werden kann, lässt sich die Visualisierung modifizieren. So kann die Anzahl der Zeilen auf die Länge und Breite gemappt werden, Attribute und die Anzahl der Methoden beispielsweise auf die Höhe und die Farbe. So lassen sich manche Konstellationen besser analysieren, weil sich die Proportionen der dargestellten Elemente ändern. Bei jeder Visualisierung ist daher darauf zu achten, allen Beteiligten die Daten der Konfiguration in irgendeiner Form mitzuteilen. Ansonsten lassen sich die Ergebnisse schwer bis gar nicht interpretieren.

VERGLEICH DER TOOLS

Ein direkter Vergleich der beiden Tools ist schwierig. CodeCity punktet mit einer wirklich sehr guten Visualisierung in Form der Städtemetapher, die zudem anpassbar ist. Allerdings verliert das Tool so langsam aber sicher an Boden, da neue Werkzeuge auftauchen, die aktueller und besser gepflegt sind.

Eines davon ist sicherlich Moose, das eine Vielzahl von Visualisierungen beherrscht. Zudem gibt es aktuelle Versionen mit weiteren Fehlerbehebungen.

Was beide Tools gemeinsam haben, ist das schon angesprochene FAMIX-Format. Auf der einen Seite ist es eine gute Idee, das zu analysierende Modell vom Sourcecode zu trennen. Analyse-Tools müssen sich somit nicht um sprachspezifische Features und Implementierungen kümmern. Das erledigt ein vorgeschaltetes Tool. Auf der anderen Seite ist das auch ein Nachteil. Denn ein Metamodell kann sich immer nur am kleinsten gemeinsamen Nenner orientieren. Alle Eingabesprachen müssen darauf heruntergebrochen werden - falls die aktuelle Sprache überhaupt unterstützt wird. Für C# war das beispielsweise hinsichtlich des FAMIX-Standards lange Zeit nicht der Fall. Erst nachträglich und deutlich später wurde eine Möglichkeit zur Generierung des Modells geschaffen.

Und auch der Markt der Analysetools an sich schläft nicht. Es existieren mittlerweile zahlreiche Werkzeuge, die direkt auf C# zugeschnitten sind und mindestens alle in diesem Artikel vorgestellten Analysen und Visualisierungen unterstützen. NDepend ist so ein Vertreter, der insbesondere ab Version 5 immer häufiger zum Einsatz kommt.

Welches Tool zum Einsatz kommen soll, lässt sich nach so einer kurzen Einführung kaum sagen. Die Anforderungen des eigenen Projekts sollten maßgeblich die Entscheidung beeinflussen. Gibt es eine bestimmte Metrik beziehungsweise Visualisierung in einem Tool nicht, ist es womöglich besser, auf ein anderes auszuweichen, anstatt hinterher mehrere Werkzeuge gleichzeitig zu nutzen. Der parallele Einsatz von mehreren Werkzeugen kann aber dennoch sinnvoll sein, um Metriken und Visualisierungen gegeneinander abzugleichen.

FAZIT

Visualisierungen sind für uns Menschen ein wichtiges Mittel, um Informationen und komplexe Sachverhalte einfacher darzustellen und miteinander in Kontext zu setzen. Moose und CodeCity sind dabei zwei gute Vertreter von Analysetools, um Metriken von C#-Code zu visualisieren.

Das perfekte Tool gibt es beileibe nicht. Vor dem Einsatz ist eine Evaluation ratsam, welche Anforderungen vorhanden sind und wie diese am besten erreicht werden können. Neben Moose und CodeCity lohnt sich auch sicherlich ein Blick auf NDepend und Co – Ndepend kommt jedoch nur mit Code aus der .NET-Welt zurecht.

QUELLEN

[1] Webseite zum FAMIX-Generator: www.sharpmetrics.net/index.php/famix-generator

[2] Über mich Seite von Thomas Haug: www.sharpmetrics.net/index.php/about

[3] Webseite zu scriptcs: http://scriptcs.net/

[4] Community Edition der Workbench: www.sharpmetrics.net/index.php/communityedition

[5] Informationen zum FAMIX-Standard Version 2.1: www.academia.edu/452056/FAMIX_2.1_the_FAMOOS_information_exchange_model

[6] Informationen zum FAMIX-Standard Version 3.0: http://rmod.lille.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf

[7] Webseite zu Moose: www.moosetechnology.org/

[8] Wikipedia Seite zu Pharo: http://en.wikipedia.org/wiki/Pharo

[9] Dokumentation zu Moose: www.themoosebook.org/book

[10] Download von CodeCity: www.inf.usi.ch/phd/wettel/codecity-download.html

FABIAN DEITELHOFF

lebt und arbeitet in Dortmund, der Metropole des Ruhrgebiets. Er studiert derzeit den Masterstudiengang Informatik mit dem Schwerpunkt Biomedizinische Informatik an der Hochschule Bonn-Rhein-Sieg in Sankt Augustin. Seine Schwerpunkte liegen in der Entwicklung von Visual Studio Erweiterungen, der Analyse und Beschreibung von Open Source Frameworks sowie im Rapid Prototyping. Beruflich ist er als freier Autor, unter anderem für Pluralsight, Sprecher und Softwareentwickler im .NET Umfeld tätig. Sie erreichen Ihn über seinen Blog www.fabiandeitelhoff.de, per E-Mail unter Fabian@FabianDeitelhoff.de oder auf Twitter als @FDeitelhoff.