INF

Studiengang
Medien- und
Kommunikationsinformatik

Hochschule Reutlingen
Reutlingen University

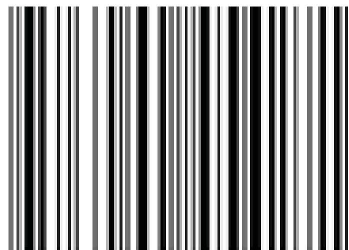
Uwe Kloos, Natividad Martínez, Gabriela Tullius (Hrsg.)

Informatics Inside:

Human-Centered Computing

Informatik-Konferenz an der Hochschule Reutlingen
30. April 2014

ISBN 978-3-00-045427-1



Impressum

Anschrift:
Hochschule Reutlingen
Reutlingen University
Fakultät Informatik
Medien- und Kommunikationsinformatik
Alteburgstraße 150
D-72762 Reutlingen

Telefon: +49 7121 / 271-4002
Telefax: +49 7121 / 271-4042

E-Mail: infoinside@reutlingen-university.de
Internet: http://www.infoinside.reutlingen-university.de

Organisationskomitee:
Prof. Dr. Gabriela Tullius, Hochschule Reutlingen
Prof. Dr. Natividad Martínez, Hochschule Reutlingen
Prof. Dr. Uwe Kloos, Hochschule Reutlingen

André Antakli
Thomas Bauer
Olaya De la Rosa Avitia
Matthias Gutekunst
Viktoria Hoffmann
Johannes Kartheininger
René Mangold
Stanislas Mauser
Lars Schneider
Arkadius Weister
Anna Wellerdiek

Hochschule Reutlingen
Reutlingen University

Copyright: © Hochschule Reutlingen, Reutlingen 2014
Herstellung und Verlag: Hochschule Reutlingen
ISBN 978-3-00-045427-1

Inhaltsverzeichnis

Gestenerkennung & Augumented Virtuality

Thomas Bauer
Anforderungsanalyse zur computergestützten Erkennung der Deutschen Gebärdensprache ... 8

Matthias Gutekunst
Augmented Virtuality zur Steigerung der Immersion in virtuellen Umgebungen ... 26

Stanislas Mauser
Analysis of Finger- and Palm-based interaction paradigms for Touch-Free Gesture-Based Control of Medical Devices with the Leap Motion Controller ... 34

Softwaretechnik

René Mangold
Selektion von Szenarien zur Optimierung von Simulationen im präventiven Krisenmanagement ... 46

Arkadius Weister
Language Oriented Programming: Modulare domänenspezifische Sprachen ... 54

Entwicklung Mobiler Anwendungen

Olaya De la Rosa Avitia
Strategy to Test Mobile Apps ... 70

Viktoria Hoffmann
Optimierung der Usability von digitalen Fahrtenbüchern durch automatisches Erfassen von fahrzeugspezifischen Daten ... 80

Johannes Kartheininger
Vergleich der Single Sign On Verfahren SAML und OpenID Connect ... 92

Virtuelle Welten

André Antakli
Umgebungswahrnehmung von agentenbasierten simulierten Menschmodellen in virtuellen Welten im Kontext C3D ... 100

Language Oriented Programming: Modulare domänenspezifische Sprachen*

Arkadius Weister
Reutlingen University
Arkadius.Weister@Student.Reutlingen-University.DE

Abstract

Beim Language Oriented Programming (LOP) erstellt der Entwickler eine Programmiersprache, um ein Problem oder eine Aufgabe in einer bestimmten Domäne zu lösen. Dabei wird die Sprache so entwickelt, dass sie das konzeptuelle Modell des Entwicklers ohne Umdenken umsetzen kann. Diese Sprachen nennt man domänenspezifische Sprachen (DSL). Zur Entwicklung dieser Sprachen werden sogenannte Language Workbenches (LWB) verwendet. Diese Arbeit befasst sich mit der Entwicklung von DSLs als ein Mittel zur Umsetzung von LOP. Durch die Nutzung der LWBs kann man DSLs mit relativ kleinem Aufwand erstellen und einsetzen. Im Fokus dieser Arbeit steht die Entwicklung von "Modularen DSLs". Hierbei werden Kriterien und Voraussetzungen für die Modularisierung betrachtet. Zum Abschluss werden drei Konzepte bestehender Systeme anhand dieser Kriterien betrachtet und bewertet.

Schlüsselwörter

Domain-specific Languages, Language Workbenches, Concrete/Abstract Syntax, Design, Generation, Transformation

*

Betreuer Hochschule: Prof. Dr.-Ing. Peter Hertkorn
Hochschule Reutlingen
Peter.Hertkorn@Reutlingen-University.de

Informatics Inside 2014
Wissenschaftliche Vertiefungskonferenz
30. April 2014, Hochschule Reutlingen
Copyright 2014 Arkadius Weister

CR-Kategorien

D.2.6 [**Software**]: Programming Environments—*Programmer workbench*; D.2.11 [**Software**]: Software Architectures—*Domain-specific architectures, Languages*; D.3.2 [**Programming Languages**]: Language Classification—*Extensible languages, Design languages*

1 Einleitung

Das Konzept des Language Oriented Programming wurde bereits 1994 von M.P. Ward [1] in seinem gleichnamigen Artikel erläutert. Damals hat er es als einen neuen Weg zur Organisierung der Entwicklung von großen Softwaresystemen bezeichnet. Der Ansatz der sprachenorientierten Programmierung beginnt mit dem Entwickeln einer formal spezifizierten und domänenorientierten sehr hohen (very high-level) Programmiersprache. Diese Sprache soll für die Entwicklung in der entsprechenden Domäne zugeschnitten und entwickelt sein [1, S.3]. Um das zu erreichen, wird das Domänenwissen im Design der Sprache so gekapselt, dass der Entwickler der Domäne die Sprache bequem und selbstdokumentierend benutzen kann. Laut Ward ist eine passende Sprache ein gutes Mittel um Domänenwissen greifbar zu machen und den Entwicklungsaufwand dramatisch zu reduzieren. Gleichzeitig wird dadurch die Wartbarkeit und die Wiederverwendbarkeit erhöht. Die Nutzung einer höheren Programmiersprache führt zu weniger Zeilen von Code. Außerdem ist der Code einfacher zu lesen, zu analysieren, zu verstehen und zu modifizieren [1, S.5]. Diese

Sprachen werden "Domänenspezifische Sprachen" genannt (engl: Domain Specific Language, kurz: DSL). Eine Möglichkeit um LOP umzusetzen, ist die Nutzung von Language Workbenches (LWB) [2]. Im nächsten Kapitel wird das Language Oriented Programming detaillierter betrachtet und die grundlegenden Begriffe der Sprachentwicklung erläutert. Die nächsten Abschnitte erläutern die Vorteile und Gründe für domänenspezifische Sprachen sowie Sprachbaukästen (Language Workbenches) und die Modularität bei DSLs. Anhand klassischer Programmiersprachen werden Kriterien für modulare Sprachen ermittelt und erfasst. Im Kapitel Language Workbenches werden die Eigenschaften aufgezählt, die eine LWB besitzen muss, um Modularität bei Sprachen gewährleisten zu können. Hinsichtlich der Kriterien zum Erreichen von Modularität werden drei Konzepte von Language Workbenches betrachtet und bewertet. Das Ergebnis der Bewertung soll Aufschluss darüber geben, ob es möglich ist mithilfe der drei Sprachbaukästen Modulare DSLs zu entwickeln. Es handelt sich dabei um das Jetbrains Meta Programming System (MPS), Xtext und Spoofax.

2 Language Oriented Programming

Die Frage, was Language Oriented Programming ist, lässt sich nicht einheitlich beantworten. Sergey Dmitriev beschreibt LOP als das nächste Programmierparadigma. Er fasst Ansätze wie z.B. Intentional Programming, MDA oder auch generative programming alle unter dem Begriff Language Oriented Programming zusammen [3]. Aber was unterscheidet LOP zu herkömmlichen Programmiermethoden? Bei einer herkömmlichen Programmierung erstellt der Programmentwickler zunächst ein konzeptuelles Modell (mentales Modell), um ein gegebenes Problem zu lösen oder eine Aufgabe zu erfüllen. Dieses Modell muss im nächsten Schritt in einer Programmiersprache abgebildet bzw. implementiert werden. Dazu wählt der Programmierer eine Allzwecksprache (z.B. Java) aus. Der nächste Schritt ist das Übertragen (Mapping) vom konzeptuellen Modell in die Programmiersprache. Dazu kann der Entwickler nur die Mittel nutzen, die von der Programmiersprache bereitgestellt werden. Dieser letzte Schritt kann sich als sehr schwierig erweisen, da das konzeptuelle Modell in den meisten Fällen nur mit Umwegen übertragen werden kann. Das kann man auf die Vielfalt der Ausdrücke einer natürlichen Sprachen zurückführen. Einem anderen Programmierer kann man das Modell in einer hohen Sprache erklären. Ein Computer benötigt zum Verständnis jedes Detail. Bei LOP baut der Programmierer wie zuvor zunächst das mentale Modell für das gegebene Problem. Anschließend wählt der Programmierer eine spezialisierte domänenspezifische Sprache aus, um das Modell umzusetzen. Sollte es keine passende DSL geben, so entwickelt der Programmierer eine DSL, die zur Lösung des Problems geeignet ist. Die spezialisierte DSL ermöglicht dem Programmierer eine einfache Übertragung vom mentalen Modell in die DSL, da er hier nicht gezwungen ist seine Ideen umständlich in Notationen zu fassen, die von der Sprache verstanden werden können. Zur Umsetzung von LOP benötigt man also DLSs [3, S.1ff]. Demnach werden beim LOP nicht nur Programme geschrieben, sondern auch die Sprachen entwickelt, mit denen sie geschrieben werden.

> „Language Oriented Programming will not just be writing programs, but also creating the languages in which to write our programs“
>
> Dmitriev (zitiert nach [3, S.5])

Abbildung 1 zeigt den Ablauf der Entwicklung in der herkömmlichen Programmierung mit einer Allzwecksprache (engl. General Purpose Language, kurz: GPL). In Abbildung 2 kann man den Entwicklungsablauf in LOP sehen. Die Vorgehensweise von LOP soll das Erstellen von spezialisierten DSLs erleichtern, die wiederum das Schreiben von Programmen erleichtern [3, S.6]. Mithilfe von LOP können eine Reihe von Problemen innerhalb großer Softwaresysteme in ihrem Ausmaß verringert werden.

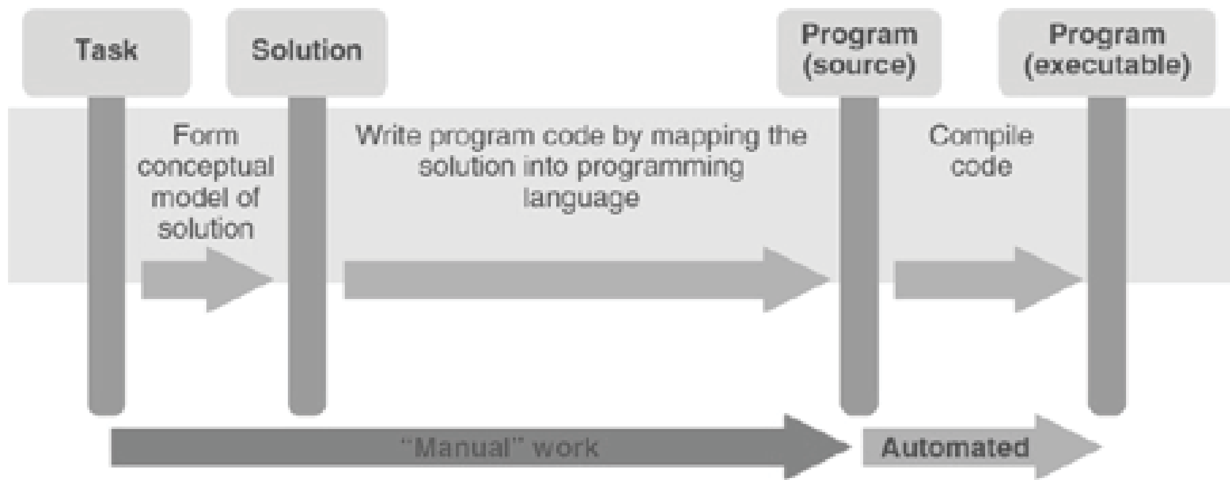


Abbildung 1: Programmierung mit Allzwecksprachen (GPL) [3]

2.1 Was ist eine Sprache?

Die Sprache (engl. Language) ist der zentrale Teil im Language Oriented Programming. Um Sprachen entwickeln zu können, muss man zunächst beantworten, was eine Sprache ist. Nach Anneke Kleppe [4] ist eine Sprache eine Reihe von „Sprachlichen Ausdrücken“: „*(Language) A language L is a set of lingusitic utterances*“ [4, S.23]. Dabei ist ein sprachlicher Ausdruck ein Bestandteil der Sprache, der durch die Sprachspezifikation definiert wird. Es kann sich dabei um ein Modell, ein Programm, ein Datenbankschema, eine Anweisung (engl. Statement) oder aber auch um einen Ausdruck in der Mathematik handeln [4, S.24ff]. Die Sprachspezifikation (engl. language specification) einer Sprache ist eine Menge von Regeln für die Ausdrücke dieser Sprache. Dabei wird die Struktur der Ausdrücke beschrieben und mindestens gezeigt, welche sprachlichen Ausdrücke genutzt werden dürfen und gegebenenfalls eine Beschreibung beziehungsweise Definition für diese bereitgestellt [4, S.39f]. Eine Sprache besteht aus einer konkreten Syntax, einer abstrakten Syntax, einer statischen Semantik und einer Ausführungssemantik. Die konkrete Syntax ist die Notation, durch die der Benutzer Programme ausdrücken/schreiben kann [5, S.26]. Die Abbildung der Eingaben eines Benutzers in die abstrakte Syntax wird von der konkreten Syntax bestimmt. Die abstrakte Syntax definiert die Beziehungen der Elemente der Sprache untereinander fest. Oft

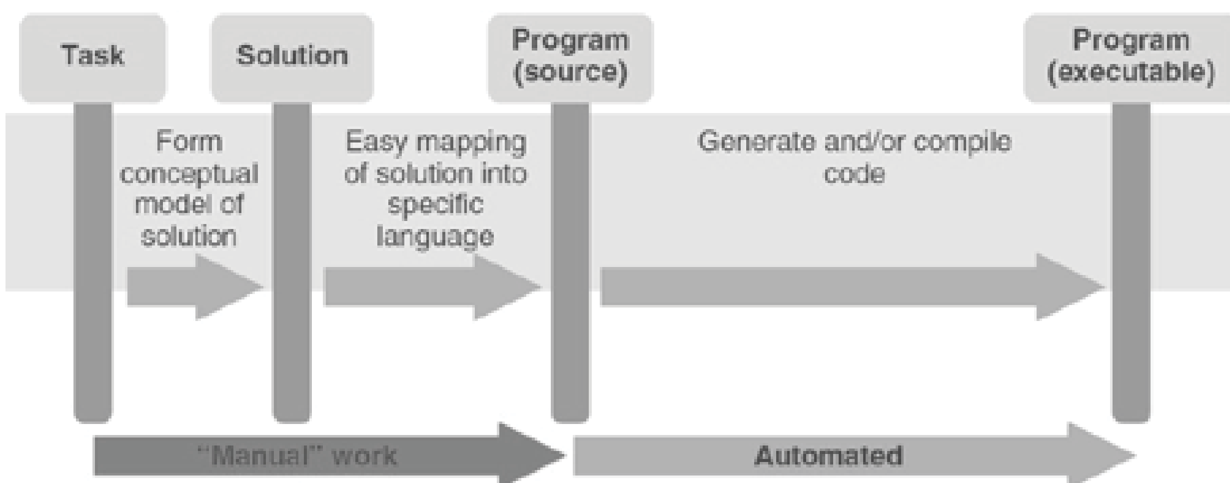


Abbildung 2: Language Oriented Programming mit DSLs [3]

wird die abstrakte Syntax in Form eines Metamodells definiert [6]. Die statische Semantik einer Sprache ist eine Sammlung von Einschränkungen und Regeln, welche von den Programmen erfüllt werden müssen. Zusätzlich muss das Programm eine korrekte Struktur in Bezug auf die konkrete und abstrakte Syntax vorweisen. Die Ausführungssemantik bezieht sich auf Programme, die ausgeführt werden und wird durch eine „execution engine" realisiert. Eine„execution engine" kann ein Generator oder Interpreter sein [5, S.26]. Beim LOP wird eine Sprache laut Sergey Dmitriev [3, S.6] mit drei Hauptbestandteilen definiert und kann weitere Aspekte wie Typsysteme und Einschränkungen besitzen:

- **Struktur**: Beschreibt die abstrakte Syntax einer Sprache, die unterstützten Konzepte und die Nutzung dieser.
- **Editor**: Beschreibt die konkrete Syntax einer Sprache. Dazu gehört die Beschreibung, wie sie verarbeitet und genutzt wird.
- **Semantik**: Beschreibt das Verhalten der Sprache. Wie wird die Sprache interpretiert und wie wird sie in ausführbaren Code transformiert.

Wie bereits erwähnt, besteht eine Sprache aus Ausdrücken. Beim LOP werden diese Ausdrücke jedoch Konzepte genannt. Eine Sprache **L** ist demnach eine Menge/Reihe von Sprachkonzepten **C** und deren Beziehungen untereinander (siehe Abbildung 3). Der Begriff Konzept (engl. concept) bezeichnet alle Aspekte eines Elements einer Sprache. Dazu gehören alle in diesem Abschnitt besprochenen Bestandteile einer Sprache (z.B. abstrakte Syntax) [5, S.64].

2.1.1 Programm und Domänen

Eine **Domäne** ist ein Wissensbereich, für den man eine Softwareunterstützung bereitstellen möchte. Eine weitere Definition ist, dass eine Domäne durch eine Reihe/Menge von Programmen mit gemeinsamen Charakteristika oder ähnlichem Zweck identifiziert wird. Diese Definition kann man noch spezifizieren. Eine Domäne ist eine Menge von Programmen,

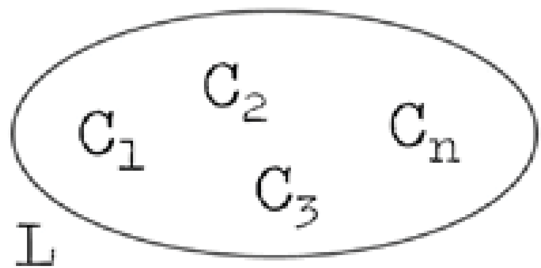


Abbildung 3: Sprache als Menge von Konzepten [5]

die in einer spezifischen Sprache geschrieben wurde. Mit dieser Definition lassen sich DSLs für eine Domäne relativ simpel erstellen, da man die Gemeinsamkeiten der Programme innerhalb der Domäne klar identifizieren kann und damit klar sagen kann, welchen Bereich eine DSL abdecken muss und welcher Code von den Programmen der DSL generiert werden muss [5, S.58f]. Abbildung 4 zeigt eine Domäne **PD** und zwei DSLs für diese Domäne (**PL1** und **PL2**). Die Sprache PL1 deckt in diesem Beispiel nicht die ganze Domäne ab und PL2 deckt zu viel ab und hat Funktionalitäten, welche nicht von der Domäne gebraucht werden.

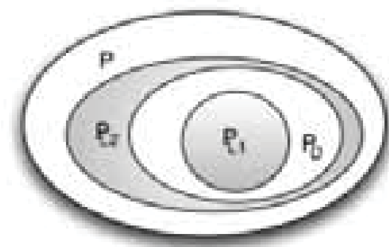


Abbildung 4: Domäne mit Sprachen [5]

Ein **Programm** ist eine konzeptionelle Darstellung einer Berechnung, die auf einem Universalrechner läuft [5, S.57]. Viele Programmierer stellen sich ein Programm als eine Reihe von Instruktionen für den Computer vor. Diese werden dann nacheinander ausgeführt. Dmitriev definiert ein Programm in LOP als eine eindeutige Lösung für ein Problem. Im Genauen heißt das: Ein Programm ist jedes präzise definierte Modell einer Lösung für ein Problem in einer Domäne, ausgedrückt mit Konzepten der Domäne [3, S.4f].

2.2 Ziele

M.P. Ward hat in seinem Artikel von Language Oriented Programming bereits die Probleme großer Softwaresysteme beschrieben und den positiven Einfluss von LOP auf diese gezeigt. Die folgende Liste ist eine kurze Zusammenfassung der Probleme, auf die LOP nach [1, S.2-3, S.20] positiven Einfluss hat:

- **Komplexität**: Große Softwaresysteme führen zu mehr Komplexität innerhalb des Systems. Dies kann zum Beispiel durch Kommunikationsschwierigkeit innerhalb von Teams und Entwicklern hervorgerufen werden. Diese können wiederum zu Kostenüberschreitungen usw. führen. Durch die Größe der Systeme gibt es eine steile Lernkurve für neues Personal und es ist kaum möglich, alle Zustände des Systems ausreichend zu visualisieren. Durch den Einsatz von LOP wird die Komplexität eines Systems in großem Maße reduziert. Die Komplexität der Domäne wird in Form von abstrakten Datentypen und Konstrukten gekapselt. Durch diese domänenspezifische Sprache werden komplexe Funktionen des Systems mit nur wenigen Codezeilen implementiert.

- **Konformität**: Die Nutzung von domänenorientierten Sprachen macht das Erreichen von Konformität einfacher. Zum Beispiel kann eine Sprache entwickelt werden, welche die aktuellen Konzepte der Steuergesetze benutzt. Mit dieser Sprache wird dann ein System zur Lohnberechnung geschrieben, welches die Anforderungen Steuerregulierungen erfüllt.

- **Änderungen**: Die einfache Verständlichkeit und Benutzbarkeit der DSL und auch die geringe Größe des Quellcodes führen dazu, dass die Sprache einfach geändert und erweitert werden kann.

- **Übersichtlichkeit**: In einer hohen domänenspezifischen Sprache wird die meiste Komplexität innerhalb der Sprache versteckt (Objekte, Konstrukte, Operationen). Dadurch lässt sich das System einfacher Visualisieren und damit besser validieren. Die problemspezifischen Aspekte stehen hier im Vordergrund.

Language Oriented Programming bietet laut [1, S.6ff] eine Reihe von Vorteilen:

- **Trennung der Bereiche**: Das Design des Systems wird vollständig von der Implementierung der Sprache getrennt. Dadurch bleibt das Design der Sprache simpel, leistungsstark und ausdrucksstark. Die Komplexität wird reduziert.

- **Hohe Produktivität in der Entwicklung**: Durch die DSL genügt eine geringe Anzahl von Codezeilen um komplexe Funktionen darzustellen. Es werden nur die Funktionalitäten implementiert, die relevant für die Domäne sind.

- **Hohe Wartbarkeit**: Die Wartbarkeit ist abhängig von der Größe des Softwaresystems. Eine geringe Anzahl von Codezeilen führt zu einer besseren Wartbarkeit.

- **Hohe Portabilität**: Die Sprache und das System lassen sich einfach von einem Betriebssystem zu einem anderen portieren. Es muss darauf geachtet werden, dass die genutzten Sprachen (z.B. Java) auf dem Zielsystem vorhanden sind.

- **Wiederverwendbarkeit**: Die Sprache kapselt ein großes Domänenwissen und kann daher für die Anforderungsanalyse eines neuen Systems genutzt werden. Außerdem kann die Sprache bei ähnlichen Projekten wiederverwendet werden.

- **Erweiterbarkeit des Systems**: Mit einer passenden Schnittstelle zu der Sprache und geeigneten Templates kann der Benutzer neue Funktionen integrieren oder bestehende erweitern.

3 Domänenspezifische Sprachen (DSL)

Der einzige Weg, um Code vollständig und ausführbar aus einem Modell zu generieren ist, wenn die Sprache und die Generatoren domänenspezifisch aufgebaut sind [7, S.23]. Ein Sprachentwickler definiert oder erweitert eine Domänenspezifische Sprache (DSL). Dabei stehen die Bedürfnisse der Benutzer im Vordergrund. Zur Generierung von Code müssen Generatoren für die DSL geschrieben werden oder bereits vorhandene genutzt werden. Dabei wird zum Beispiel der geschriebene Code der DSL in eine Zielsprache mittels Transformation generiert [7, S.24ff]. Laut Markus Völter [5] werden DSLs immer wichtiger in der Softwaretechnik und es gibt Werkzeuge, um DSLs mit relativ wenig Aufwand zu entwickeln. Eine domänenspezifische Sprache ist eine Sprache für eine Domäne, die darauf spezialisiert ist, Programme für die Domäne zu erstellen. Das macht die DSL für die Entwicklung von Programmen der Domäne effizienter als andere Sprachen. Eine DSL wird so konzipiert, dass sie so gut wie möglich in die Domäne passt. Dazu werden Abstraktionen in der Sprache genutzt, die unnötige Details vermeiden, die nicht in den Programmen gebraucht werden [5, S.59f]. Die Wahl der konkreten Syntax einer DSL soll sich nach den Benutzern in der jeweiligen Domäne richten und damit ihre Arbeit innerhalb der Domäne erleichtern [8, S.97]. Eine bekannte DSL ist zum Beispiel die Sprache für Datenbanken „SQL“ (Structured Query Language), die eine Schnittstelle für relationale Datenbanksysteme bietet. Ein weiteres Beispiel ist die Beschreibungssprache HTML für die Erstellung von Webseiten [7, S.25]. Eine DSL sollte folgende Basiskriterien erfüllen [7, S.26]:

- Die Konzepte der Sprache müssen von den Experten der Domäne in ihrem Zweck und ihrer Funktionalität verstanden werden.
- Grafische oder textuelle Notation der Sprache sollte einfach zu benutzen sein.
- Die Regeln und Grammatik der Sprache müssen ausführlich definiert sein.

Bevor eine DSL definiert und entwickelt werden kann, muss eine Domänenanalyse durchgeführt werden. In der Analyse werden die Aktivitäten, Objekte und Operationen von Systemen innerhalb einer Domäne identifiziert und in einem Domänenmodell festgehalten. Die Ergebnisse einer Domänenanalyse helfen bei der Entwicklung und Definition einer neuen DSL und beim Finden von existierenden DSLs [9, S.28]. Ein Domänenmodell besteht aus den folgenden Elementen [9]:

- Definition der Domäne und ihres Anwendungsbereiches.
- Definition einer Domänenterminologie mit den wichtigsten Begriffen.
- Beschreibung der Konzepte in der Domäne.
- Beschreibung der Gemeinsamkeiten, Abhängigkeiten und Unterschiede der Domänenkonzepte .

Zur Definition und Entwicklung einer neuen DSL gehören laut Martin Fowler [2] drei Bestandteile. Einer davon ist das Definieren einer abstrakten Syntax, die das **Schema** der abstrakten Repräsentation ist. Der zweite Bestandteil ist das Definieren eines **Editors**. Dieser definiert die konkrete Syntax und ermöglicht die Bearbeitung der abstrakten Repräsentation über eine Projektion. Der letzte Bestandteil ist das Definieren eines **Generators**. Dieser beschreibt, wie die abstrakte Repräsentation der Sprache in eine ausführbare Repräsentation der Sprache übersetzt wird. Ein Generator definiert also die Semantik einer DSL. Diese drei Bestandteile sind das Minimum und können in ihrer Anzahl nach oben variieren. Es kann zum Beispiel mehr als einen Generator für die DSL geben. Damit können die DSL in mehrere Zielsprachen übersetzt werden (z.B Java und C#). Es kann auch mehrere Editoren geben. Dabei kann jeder Editor für eine Benutzergruppe zugeschnitten werden [2]. Damit sind alle, von Sergey Dmitriev genannten, Bestandteile einer Sprache im Language Oriented Programming durch eine DSL abgedeckt (siehe: 2.1).

3.1 *Gründe für DSL*

Welche Gründe bestehen für die Nutzung einer DSL ? Überwiegen die Vorteile den Nachteilen oder kann keine klare Antwort darauf gegeben werden? Den folgenden Grund für die Benutzung hat bereits die Definition von DSLs geliefert. DSLs erlauben es implizit vorhandenes Expertenwissen in kompakter Form explizit zu formulieren [6]. In Abbildung 5 wird die Entwicklung eines Produktes innerhalb einer Domäne gezeigt. Zunächst besteht eine Idee für die Lösung eines Problems der Domäne. Bei der herkömmlichen Methode wird im nächsten Schritt ein Mapping der Idee in ein Modell oder direkt in Code einer Programmiersprache durchgeführt. Beim Nutzen einer DSL kann jedoch auf das umständliche Mapping verzichtet und die Lösung direkt mit der Domänensprache in einem Domänenmodell dargestellt und daraus mithilfe von Codegenerierung ein fertiges Produkt erzeugt werden [10, S.15f].

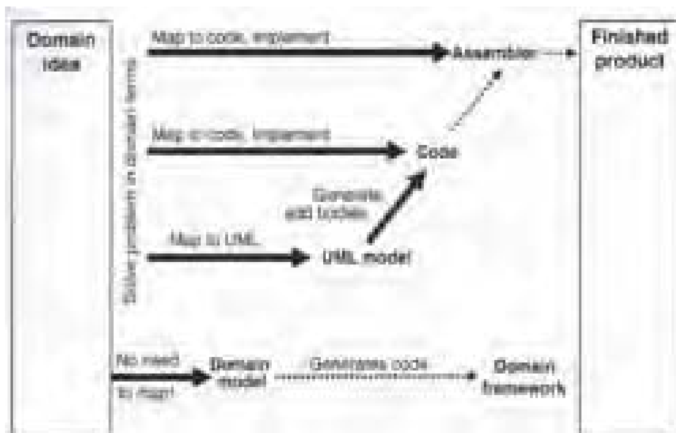


Abbildung 5: Von der Idee zum Code [10]

Die Nutzung von DSLs kann Vorteile und auch Herausforderungen mit sich bringen. In seinem Werk „DSL Engineering“ [5, S.40ff] geht Markus Velter auf die Vorteile und Herausforderungen bei DSL ein. Die folgende Liste zeigt eine Zusammenfassung davon.
Die Vorteile der Benutzung von DSL sind:

- *Steigerung der Produktivität*: Da eine DSL wenig Codezeile besitzt, wird die Komplexität gesenkt und die Arbeit effizienter.
- *Steigerung der Qualität*: Durch eine DSL entstehen weniger Fehler, bessere Konformität in der Architektur und keine Duplizierung von Code.
- *Bessere Validierung und Verifikation*: Eine DSL ist semantisch reicher als eine GPL. Dadurch ist sie nicht so überladen und Analysen und Validierung sind einfacher durchzuführen.
- *Langlebigkeit der Daten*: Die Modelle sind in einer hohen Abstraktionsebene ausgedrückt. Basierend auf den Modellen werden Analysen oder Generierungen starten. Das Modell kann auch in andere Repräsentationen transformiert werden.
- *Besseres Verständnis und Kommunikation*: Die Benutzer und Entwickler der DSL verstehen die Domäne und die Sprache sehr gut. Dadurch wird die Teamkommunikation einfacher.
- *Beteiligung*: Die Domänenexperten werden bei der Entwicklung der DSL beteiligt und können diese selbstständig benutzen, da sie wegen der Nähe zur Domäne selbsterklärend ist.
- *Werkzeuge*: Viele DSLs werden mit Werkzeugen (IDE) geliefert und bieten Debugger, Codevervollständigung, Simulatoren, Visualisierungen und vieles mehr.
- *Kein Mehraufwand*: Bei der Generierung von Code wird kein Mehraufwand in der Laufzeit geleistet. Bei Fragen der Leistung und Ressourcen ist das ein wichtiger Faktor.
- *Plattformunabhängig*: Die Anwendungslogik einer DSL ist unabhängig von der Plattform und Implementierung.

Die Herausforderungen bei DSLs sind:

- *Aufwand*: Der Aufwand bei der Entwicklung einer DSL ist hoch. Es wird Expertenwissen gebraucht und es sollte

eine Kostenanalyse durchgeführt werden. Eine DSL sollte so konzipiert werden, dass man sie wiederverwenden kann. Zur Reduzierung der Kosten werden Werkzeuge wie Language Workbenches (siehe Kapitel 5) benutzt.

- *Erfahrung und Fertigkeit*: Die Definition einer guten Sprache benötigt Erfahrung und muss durch Praxis erlernt werden.

- *Probleme bei Vorgängen*: Die Entwickler der Sprache, die Benutzer und die Domänenexperten müssen zusammenarbeiten, damit es nicht zu Problemen in den Prozessen kommt (z.B. werden mehrere Sprachen für eine Domäne entwickelt). Deshalb müssen Prozesse für die Kommunikation untereinander definiert werden.

- *Entwicklung und Wartung*: Durch die schnelle Weiterentwicklung einer DSL kann es zu Wartungsschwierigkeiten kommen. Gleichzeitig muss eine Sprache immer weiterentwickelt werden, damit sie nicht veraltet.

- *Zu viele DSLs*: Es besteht die Gefahr, dass erfahrene Entwickler immer neue DSLs erstellen, anstatt zu prüfen, ob es bereits eine existierende DSL in der Domäne gibt. Das führt dazu, dass viele halb fertige DSLs und keine vollständigen entwickelt werden.

- *Investierung*: Wenn man sich an eine Arbeitsweise gewöhnt hat und effizient darin ist, ist es schwer davon abzulassen und etwas Neues zu probieren.

- *Werkzeuge*: Es gibt grundsätzlich keine Interoperabilität zwischen DSL Tools. Dadurch ist die Entwicklung einer DSL abhängig von einem Tool (Werkzeug).

- *Kultur*: Domänenexperten sehen sich oft nicht als Programmierer und möchten nur in Standards wie z.B. UML modellieren. Außerdem denken viele, dass die Entwicklung von Sprachen zu kompliziert ist [5, S.45]. Diese und weitere Annahmen erschweren die Einführung von DSLs.

Ob eine DSL genutzt werden sollte kommt auf die jeweilige Situation der Domäne, der Entwickler und der Benutzer an. Wenn ein Entwickler eine Domäne nur schwer versteht und/oder die Programme einer Domäne nicht implementieren kann, dann sollte eine DSL nicht entwickelt werden. Wenn die Kunden und die Benutzer keine klaren Definitionen für eine Domäne liefern können und sie ständig erweitern wollen, so können die Kosten für die Entwicklung schnell sehr hoch werden. Eine DSL sollte also nur entwickelt werden, wenn die Benutzer sie eindeutig definieren kann und der Entwickler gute Kenntnisse über Softwaretechnik, Design, Testing und die Domäne besitzt [5, S.46f].

3.2 Interne und externe DSLs

Es gibt zwei Arten von DSLs. Interne DSLS und externe DSLs. Bei einer **internen DSL** handelt es sich um eine Sprache, die in eine andere Sprache eingebettet ist. Diese andere Sprache wird Hostsprache genannt. Die interne DSL verwendet also die gleiche Syntax der Hostsprache und ist mit ihren Mitteln beschrieben. Zur Verwendung von internen DSLs bieten sich vor allem Hostsprachen mit geeigneten Erweiterungsmöglichkeiten an. Eine **externe DSL** ist eine Sprache, die von Grund auf neu erstellt wurde. Sie ist demnach nicht in eine andere Sprache eingebettet. Zum Lesen der Informationen wird ein Parser erstellt. Dadurch lässt sich die externe DSL zur Interpretation und Generierung innerhalb einer anderen Programmiersprache verwenden [8, S.98]. Der Vorteil einer externen DSL ist ihre Fähigkeit, eine Domäne in einfachster Form zum Lesen und Modifizieren darzustellen. Zusätzlich kann eine externe DSL zur Laufzeit evaluiert werden. So werden zum Beispiel Parameter verändert, ohne das Programm neu übersetzen zu müssen. Der Nachteil einer externen DSL liegt darin, dass ein Übersetzer und Parser neu erstellt werden muss und das sie nicht mit einer Basissprache

verbunden ist. Im Gegensatz dazu kann eine interne DSL alle Möglichkeiten einer Basissprache (Hostsprache) ausschöpfen. Eine interne DSL ist aber nur auf die Syntax und Struktur der Basissprache beschränkt [7, S.29f].

4 Modulare DSL

Eine wichtige Frage bei dem Einsatz von DSL ist die Möglichkeit der Reduzierung und Einsparung von Kosten und Investitionen. Dazu sollten Sprachen von Beginn an so entwickelt werden, dass sie möglichst oft wiederverwenden werden können. Dabei muss nicht die gesamte Sprache wiederverwendet werden. Die Sprachelemente und Rumpfstrukturen einer Sprache eignen sich bereits zur Wiederverwendung. Um das zu ermöglichen, müssen die einzelnen Sprachdefinitionen möglichst getrennt voneinander spezifiziert und flexibel miteinander kombiniert werden können. Diese „modulare" Entwicklung erlaubt es, neue Sprachen aus existierenden Komponenten zusammenzustellen. Eine solche DSL wird als „Modulare DSL" bezeichnet [6]. „*Die grundlegende Technik ist die modulare Entwicklung einer Sprache aus bereits existierenden Bausteinen und gegebenenfalls einer erweiterbaren Rumpfsprache*" [6]. Modulare Sprachen bestehen aus einem minimalen Sprachkern und einer Bibliothek von Sprachmodulen, die je nach Bedarf importiert und genutzt werden können. In Abbildung 6 wird eine solche modulare Sprache dargestellt. Der Kern ist eine kleine Sprache (**my L**) mit wenigen, aber starken Sprachkonzepten (α,β). Neue oder individuelle Sprachmodule (a, b, c...) können jederzeit importiert und verwendet werden. Ein Sprachmodul ist wie eine Bibliothek, besitzt aber seine eigene Syntax. Ein importiertes Sprachmodul verhält sich wie ein Teil der zusammengesetzten Sprache und kann mit anderen Modulen verbunden sein [5, S.35f]. Ein Sprachmodul kann Teil anderer Sprachen sein und selbst aus Untermodulen bestehen.

4.1 Modularität von Sprachen

Wie bereits erwähnt, macht die Modularisierung von DSLs die Softwareentwicklung effizienter und billiger. Das liegt daran, dass ähnliche und gleiche Funktionalitäten nicht

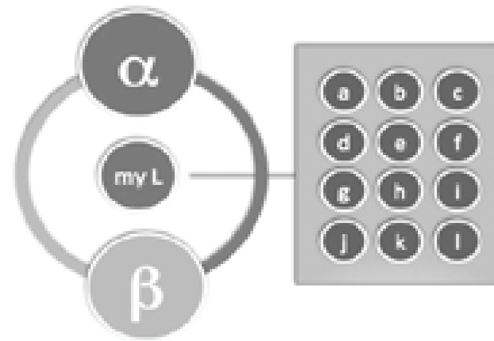


Abbildung 6: Modulare Sprache [5]

mehrfach entwickelt, sondern stattdessen wiederverwendet werden. Sprachen werden miteinander kombiniert und wiederverwendet. Welche Möglichkeiten gibt es, Modularität umzusetzen und was muss bei der Komposition beachtet werden? Bei der Komposition von Sprachen muss die Komposition der abstrakten Syntax, der konkreten Syntax, der Einschränkungen, Typsysteme und der Semantik durchgeführt werden. Bei der Komposition von Sprachen wird zwischen vier Arten der Modularisierung unterschieden: 1. Referenzierung (Referencing), 2. Wiederverwendung (Reuse), 3. Erweiterung (Extension) und 4. Einbettung (Embedding). Der Einsatz von modularen DSLs und die daraus resultierende Komposition und Wiederverwendung der Sprachen führt zur Reduzierung unnötiger halb fertiger DSLs (siehe Herausforderungen in Kapitel 3.1). Die Erweiterung von Sprachen erlaubt dem Benutzer das Hinzufügen neuer Sprachkonstrukte in die bestehende Sprache [5, S.116f].

Referenzierung (Referencing) erlaubt homogene Querverweise zwischen den Sprachen. Die referenzierende Sprache ist dann von der referenzierten Sprache abhängig. Abbildung 7 zeigt eine Sprache **l2**, die abhängig von der Sprache **l1** ist, da mindestens ein Konzept **B1** der Sprache **l2**, eine Referenz auf ein Konzept **A2** der Sprache **l1** setzt. Zum Referenzieren werden Kenntnisse über die Beziehungen und das Design der Sprachen benötigt. Die abhängigen Sprachen können, wegen ihrer Abhängigkeit zu anderen Sprachen, nicht wiederverwendet werden [5, S.118f].

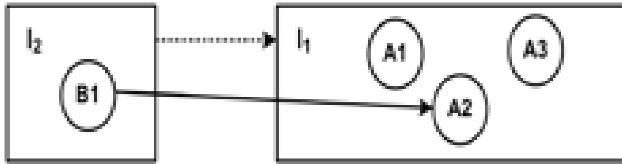


Abbildung 7: Referenzierung [5]

Bei der **Erweiterung (Extension)** erweitert eine Sprache **B** eine andere Sprache **A** um weitere Sprachkonzepte. Alle Programme, die in der Sprache **B** geschrieben werden, können alle Konzepte der Sprache **A** zusätzlich zu denen der Sprache **B** verwenden. Durch Erweiterung kann also eine Sprache um neue Funktionen und Methoden erweitert werden, um ein neues Problem innerhalb einer Domäne zu lösen, ohne dafür eine neue Sprache zu definieren. Es gibt auch die Möglichkeit eine Sprache in der Erweiterung zu beschränken (**Restriction**). Dadurch werden nicht gebrauchte Konzepte der Sprache **A** in der Sprache **B** nicht zugelassen [11]. Abbildung 8 zeigt die Erweiterung einer Sprache **l1** durch die Sprache **l2**. Das Konzept B3 erweitert das Konzept A3 so, dass es als Kind von A2 verwendet werden kann. Die Spracherweiterung ist sehr nützlich bei DSLs, die einfach starten und mit der Zeit komplizierter werden und neue Fälle und Feinheiten bekommen. Die Erweiterungen können in separaten Sprachmodulen definiert werden, welche die Kern-DSL erweitern. Die Benutzung der Module kann zusätzlich auf bestimmte Benutzer eingeschränkt werden [5, S.120f].

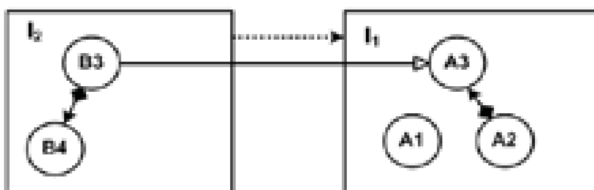


Abbildung 8: Erweiterung [5]

Die **Wiederverwendung (Reuse)** bezieht sich auf zwei voneinander unabhängige Sprachen. Die Sprachen wurden zu einem Zeitpunkt entwickelt, bei dem nicht klar war, dass sie beide in einem gemeinsamen Kontext genutzt werden würden. Die Sprachen kennen sich nicht und können auch keine Konzepte untereinander referenzieren. Eine Möglichkeit die Sprachen wiederverwendbar zu machen, ist der Einsatz einer dritten Sprache. Die dritte Sprache dient als Adapter und ermöglicht die Referenzierung und Vererbung der zwischen den beiden unabhängigen Sprachen. Die Adaptersprache ist von beiden Sprachen abhängig. Abbildung 9 zeigt die unabhängigen Sprachen **l1** und **l2** [5, S.123f].

Mithilfe der Adaptersprache **lA** wird die Referenzierung ermöglicht. Das Konzept B5 der Adaptersprache erbt von B4 und referenziert auf A3. Während Sprachreferenzierung nur die Wiederverwendung der referenzierten Sprache unterstützt, bietet Reuse zusätzlich die Unterstützung der Wiederverwendung referenzierenden Sprachen [5, S.123f].

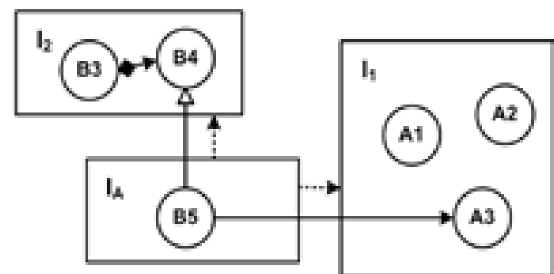


Abbildung 9: Wiederverwendung [5]

Einbettung (Embedding) ist der Wiederverwendung (Reuse) ähnlich. Es gibt wieder zwei unabhängige Sprachen. Im Gegensatz zu Reuse wird bei der Einbettung keine Referenz zwischen den Sprachen erstellt, sondern Instanzen von Konzepten der Sprache **l2** in Teile des Sprache **l1** eingebettet. Es gibt keine Abhängigkeiten zwischen der eingebetteten Sprache und der Hostsprache. Die Einbettung kann durch das Verwenden einer Adaptersprache realisiert werden. Abbildung 10 zeigt eine Adaptersprache **lA**, die von beiden Sprachen abhängig ist und das Konzept **A3** in sein Konzept **B5** einbettet. Einbettung unterstützt also die syntaktische Komposition von unabhängig entwickelten Sprachen. Eine besondere Art der Einbettung ist die Querschnitt-

Einbettung mit Metadaten (Cross-Cutting Embedding, Meta Data). Metadaten werden als Programmelemente definiert, die unabhängig von der Semantik sind. Die Daten müssen dennoch das Programm betreffen. Die Einbettung der Metadaten sollte in jeder Sprache möglich sein und dabei sollten keine Abhängigkeiten zu den Sprachen entstehen. Ein Beispiel für die Einbettung von Metadaten ist bei der Erstellung von Dokumentationen [5, 124ff].

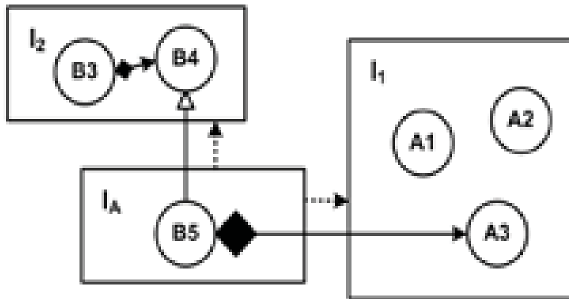


Abbildung 10: Einbettung [5]

4.2 *Kriterien*

In diesem Abschnitt werden die Kriterien und Herausforderungen aufgezählt, die eine Sprache erfüllen muss, um die vier oben genannten Arten der Modularisierung umsetzen zu können. Der herkömmliche DSL-Ansatz reicht hier nicht aus und muss um eine Modularisierungsmöglichkeit erweitert werden. Dies kann mit einer bestehenden DSL durchgeführt oder bereits bei der Definition einer neuen DSL berücksichtigt werden [12].

Syntax

Bei der *Referenzierung* und *Reuse* ist das Vermischen von konkreter Syntax nicht erforderlich, da eine Referenz zwischen Konzepten und Fragmenten der Sprache lediglich ein Bezeichner/Identifizierungszeichen ist und keine innere Struktur mit Grammatik enthält. Durch Namensauflösung werden die Querverweise in den abstrakten Syntax-Objekten gesetzt. Bei der *Erweiterung* und *Einbettung* werden Definitionen für die modulare konkrete Syntax benötigt, da zusätzliche Sprachelemente mit Programmen der Hostsprache oder Basissprache vermischt werden müssen. Das Kombinieren von unabhängigen Sprachen kann Probleme verursachen. Die Parsertechnologien der Sprachen dürfen sich gegenseitig nicht ausschließen und die kombinierten Grammatiken verstehen können. Viele parserbasierte Sprachbaukästen (LWB) haben dieses Problem bisher nicht vollständig lösen können. Projektionale Sprachbaukästen brauchen keinen Parser und haben dieses Problem nicht [5, S.127f].

Typ-System

Bei der *Referenzierung* muss die referenzierende Sprache die Regeln und Einschränkungen der referenzierten Sprache berücksichtigen. Da die referenzierte Sprache bereits bei der Entwicklung der referenzierenden Sprache bekannt ist, können die Typ-Systeme unter Berücksichtigung der referenzierten Sprache implementiert werden. Bei der *Erweiterung* müssen die Typ-Systeme der Basissprache so entworfen sein, dass in der Spracherweiterung neue Typ-Regeln hinzugefügt werden dürfen. Bei der *Einbettung* und *Reuse* müssen sich die Typ-Regeln, die das Zusammenspiel der beiden Sprachen beeinflussen, in der Adaptersprache befinden. Die Typ-Systeme der beiden Sprachen müssen erweiterbar sein [5, S.128f].

Transformation und Generierung

Bei der *Referenzierung* werden drei Fälle unterschieden. Im ersten Fall wird die Struktur der Referenzierung in das Zielfragment (Zielcode) übernommen. Die referenzierte Sprache und die referenzierende Sprache benutzen ihre Generatoren. Der Generator der referenzierenden Sprache kann der Generator der referenzierten Sprache oder nach seinem Vorbild entworfen sein. Die beiden Sprachen benutzen ihre eigenen Konzepte und Fragmente als Quelle (single-sourced) und es entstehen zwei Zielfragmente mit derselben Referenzierung wie die Quellen. Im zweiten Fall wird von mehreren Quellen ausgegangen, die ein Zielfragment transformieren. Dazu muss eine neue Transformation geschrieben werden, die das referenzierende Fragment der Sprache bei der Generierung

des Zielcodes der referenzierten Sprache berücksichtigt. Im dritten Fall wird eine Transformation vor der Generierung durchgeführt. Diese Transformation verbindet die beiden Quellfragmente der beiden Sprachen in einer neuen einzelnen Quelle. Die bereits vorhandene Transformation der referenzierten Sprachen kann genutzt werden [5, S.129f].

Bei der *Erweiterung* werden die Spracherweiterungen aus sprachlichen Abstraktionen von Ausdrücken einer Domäne definiert. Der Generator für die neuen Sprachkonzepte kann diese Ausdrücke einfach durch Assimilierung nachbauen. Zum Beispiel wird Code in der Hostsprache vom Code generiert, der in der Spracherweiterung ausgedrückt wurde. Spracherweiterungen stellen eine Gefahr zu semantischen Interaktionen dar. Die Transformationen der unabhängig voneinander erstellten Spracherweiterungen könnten miteinander interagieren. Um dieses Problem zu vermeiden, müssen Transformationen so entworfen sein, dass sie keine knappen Ressourcen verbrauchen wie z.B. vererbte Links [5, 130f].

Beim *Reuse* besitzen die wiederverwendete Sprache und die Kontextsprache bereits ihre eigenen Generatoren. Diese Generatoren müssen den Code der beiden Quellsprachen in dieselbe Sprache oder in kompatible Sprachen transformieren. Falls die Generatoren dies nicht unterstützen, funktioniert die Wiederverwendung nicht. Bei der Wiederverwendung kann es drei Fälle geben. Im ersten Fall gibt es bereits existierende Transformationen für beide Sprachen und einen Generator für die Adaptersprache. In der Transformation der Kontextsprache wurden Löcher eingebaut. Der Generator der Adaptersprache wandelt den Code der Kontextsprache so um, dass er mit der wiederverwendeten Sprache kompatibel ist, und füllt dazu die zuvor eingebauten Löcher der Kontextsprache mit Elementen der wiederverwendeten Sprache auf. Im zweiten Fall wird der bereits existierende Generator der wiederverwendeten Sprache mit Transformationscode der Kontextsprache erweitert. Dazu ist ein Mechanismus zur Komposition von Transformationen nötig. Im dritten Fall wird die Komposition der erzeugten Codefragmente von den Zielsprachen durchgeführt [5, S.131f].

Bei der *Einbettung* muss für die Transformation von eingebetteten Sprachen eine neue Transformation geschrieben werden, falls die bereits existierende Sprache nicht dieselbe Zielsprache erzeugt wie die Transformation der Hostsprache. Wenn die Transformationen dieselbe Zielsprache erzeugen, dann kann der Generator der Adaptersprache die Transformationen der Hostsprache und der eingebetteten Sprache koordinieren. In einem anderen Fall wird der Code der eingebetteten Sprache in die Hostsprache transformiert und danach zusammen mit der Hostsprache in die Zielsprache umgewandelt [5, S.132].

5 Language Workbenches

Die grundlegende Frage beim Language Oriented Programming ist nach Martin Fowler die Kosten-Nutzen-Abwägung. Welchen Gewinn bringt die Nutzung von DSLs und welche Kosten werden bei der Entwicklung von notwendigen Werkzeugen erbracht, um diese Sprache effizient zu unterstützen? Beim Nutzen von internen DSLs gibt es weniger Kosten für Werkzeuge, ist aber an die Syntax der Basissprache gebunden. Mit einer externen DSL kann der höchste Nutzen und Gewinn aus einer DSL gezogen werden. Jedoch steigen hier die Kosten bei der Entwicklung der Sprache, der Übersetzer und der Werkzeuge. Um diesen Kosten entgegen zu wirken, wurde eine neue Kategorie von Werkzeugen entwickelt. Diese Werkzeuge werden Sprachbaukästen (Language Workbenches) genannt. Diese bieten Flexibilität bei externen DSLs ohne eine semantische Barriere und machen es einfacher Werkzeuge für moderne Entwicklungsumgebungen zu bauen [2].

Martin Fowler beschreibt Language Workbenches mit den folgenden Charakteristika [2]:

1. Benutzer können frei neue Sprachen definieren, die vollständig miteinander integriert sind.
2. Die wichtigste Informationsquelle ist eine anhaltende abstrakte Darstellung.
3. Sprachentwickler definieren eine DSL in drei Hauptteilen: Schema (abstrakte Syntax), Editor (konkrete Syntax) und Generator(en).
4. Die Benutzer der Sprache manipulieren eine DSL über einen projektionalen Editor.
5. Eine Language Workbench kann aus unvollständigen oder widersprüchlichen Informationen in seiner abstrakten Darstellung bestehen.

Die Punkte 2-4 setzen die Benutzung eines projektionalen Systems voraus. Mittlerweile wird der Begriff Language Workbenches von Martin Fowler und anderen aber auch für Werkzeuge mit parserbasierten Systemen genutzt [11, S.5]. Laut Voelter spielt es keine Rolle, ob man anstatt eines projektionalen Editors einen parserbasierten Editor zum Erstellen von modularen Sprachen benutzt. Wichtig ist, dass eine Sprache modular ist. Welcher Ansatz zum Erreichen der Modularität benutzt wird, ist nicht wichtig, solange das Ziel erreicht wird [5, S.36].

5.1 Arten von Language Workbenches

Bevor auf die Arten von Language Workbenches eingegangen wird, müssen zunächst die Begriffe **Parser** und **Abstrakter Syntaxbaum (AST)** beschrieben werden. Ein Parser ist ein Stück Software, dessen Aufgabe es ist ein Modell aus seiner konkreten Syntax in seine abstrakte Syntax zu überführen [8, S.174]. Der Parser prüft die konkrete Syntax auf Korrektheit und erzeugt danach entsprechende Parsebäume aus den Ausdrücken. Diese Bäume werden abstrakte Syntaxbäume genannt und können von Interpretern oder Generatoren weiterverarbeitet werden [8, S.107].

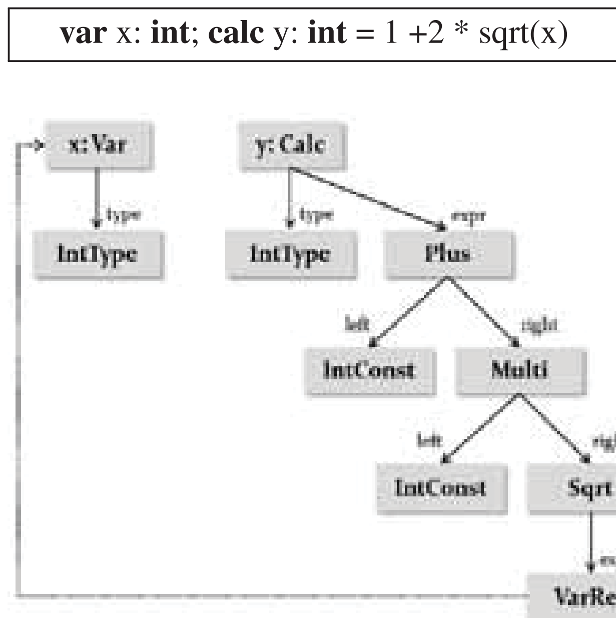


Abbildung 11: Abstrakter Syntaxbaum [5]

In Abbildung 11 ist im oberen Teil ein Programm in konkreter Syntax geschrieben und im unteren Teil der dazugehörige abstrakte Syntaxbaum. Das Programm hat eine hierarchische Struktur. Die Definitionen von **x** und **y** sind auf der obersten Ebene. Innerhalb von **y** gibt es einen geschachtelten Ausdruck, der sich in linke und rechte Schritte unterteilt. Die Boxen repräsentieren Instanzen der Sprachkonzepte.

Es gibt zwei Wege um die Beziehung zwischen konkreter Syntax (CS) und abstrakter Syntax (AS) in der Sprachentwicklung zu beschreiben. Beim Ansatz *CS-first* wird zuerst eine konkrete Syntax definiert und daraus eine abstrakte Syntax abgeleitet. Beim Ansatz *AS-first* wird zuerst eine abstrakte Syntax definiert. Danach wird eine konkrete Syntax geschrieben, die auf die abstrakte Syntax bezogen ist [5, S.178f].

Bei den Language Workbenches wird zwischen parserbasierten Systemen und projektionalen Systemen unterschieden. Im Folgenden werden beide Ansätze erläutert.

Parserbasierte Systeme:
Im parserbasierten Ansatz wird der abstrakte Syntaxbaum aus der konkreten Syntax des Programmes konstruiert. Ein Parser initiiert und füllt die abstrakte Syntax basierend auf den Informationen des Programmtextes. In diesem Ansatz kann der Benutzer den Text frei editieren, da die konkrete Syntax stark von der abstrakten Syntax getrennt ist. Beim parserbasierten Ansatz gibt es die Möglichkeit den Ansatz *CS-first* oder *AS-first* je nach LWB zu benutzen. Die Parser können selbst per Hand geschrieben werden oder von in den LWBs enthaltenen Parsergeneratoren automatisch erstellt werden. Das Parsen wird in den meisten Fällen in mehrere Prozesse aufgeteilt. Der Text wird zunächst in eine Sequenz von Zeichen/Symbolen aufgeteilt. Dieser Schritt wird von einem **Scanner** durchgeführt. Der Parser erstellt dann den AST aus der Sequenz. Der Scanner muss sich der Bedeutung der Eingaben bewusst sein. Zum Beispiel muss er alle Schlüsselwörter kennen und trotzdem dein Einsatz von Bezeichnern mit dem gleichen Namen (wie das Schlüsselwort) erlauben und durch den Kontext erkennen, dass es sich an dieser Stelle nicht um ein Schlüsselwort handelt. Viele Sprachen unterstützen diese Funktionalität nicht (z.B. Java). Deswegen gibt es neben dem normalen Parsen auch das Parsen ohne Scanner (scannerless parsing)[5, S.179ff].

In Abbildung 12 kann man auf der linken Seite das parserbasierte System erkennen. Der Benutzer sieht die konkrete Syntax und arbeitet auch mit dieser.

Projektionale Systeme:
Im projektionalen Ansatz wird der abstrakte Syntaxbaum direkt über Aktionen und Eingaben im Editor erstellt. Die konkrete Syntax wird vom AST mit Projektionsregeln erstellt. Während ein Benutzer das Programm editiert, wird der abstrakte Syntaxbaum direkt modifiziert. Eine Projektionsmaschine erstellt eine Repräsentation des AST. Mithilfe dieser Repräsentation kann der Benutzer interagieren und Änderungen sehen. Die Instantiierung eines Objektes des ASTs erfolgt über die Auswahl eines Sprachkonzeptes über ein Auswahlmenü bei der automatischen Codevervollständigung. Die erstellte Instanz wird in den AST eingelagert und mit einer eindeutigen Identifikationsnummer (UID) versehen. Anhand dieser UID werden Referenzen zwischen Programmelementen über Pointer hergestellt [5, S.188f].

In Abbildung 12 ist auf der rechten Seite das Verfahren beim projektionalen System dargestellt. Der Benutzer sieht eine Repräsentation des AST in Form von konkreter Syntax, arbeitet aber direkt auf dem AST.

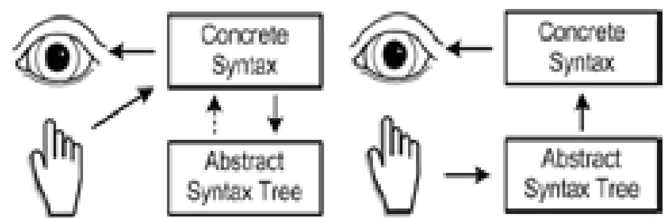


Abbildung 12: Parserbasiertes und projektionales System [5]

5.1.1 Modularität in MPS

Das Meta Programming System (MPS) von Jetbrains ist ein Beispiel für eine projektionale Language Workbench. Es kommt also kein Parser zum Einsatz. MPS benutzt einen generativen Einsatz. Sprachen können erstellt und die Generatoren für sie geschrieben werden. MPS unterstützt dabei vor allem die Erzeugung von Java-Code, kann aber auch andere Sprachen generieren. Es gibt die Möglichkeit Text zu generieren, der selbst definiert wurde. Eigene Generatoren für weitere bisher noch nicht unterstütze Sprachen können geschrieben werden. MPS besitzt eine BaseLanguage. Diese ist eine Implementierung von Java. Bei der Erstellung neuer DSLs wird erst eine abstrakte Syntax definiert, dann der Editor spezifiziert und zuletzt der Generator definiert [13]. Bei der Modularisierung unterstützt MPS die *Referenzierung*, die *Wiederverwendung (Reuse)*, die *Erweiterung* und die *Einbettung*. Bei

der Wiederverwendung müssen die Sprachen jedoch bereits bei der Entwicklung so konzipiert werden, dass eine Wiederverwendung möglich ist. Die Erweiterung in MPS ist besonders einfach, da dafür zum Beispiel die BaseLanguage erweitert werden kann. MPS bietet bereits viele Erweiterung ihrer BaseLanguage [5, S.394ff].

5.1.2 Modularität in Xtext

Xtext ist eine parserbasierte Language Workbench und ist eine Open Source Anwendung, die auf dem Eclipse Modeling Framework basiert. Die komplette Infrastruktur der Sprache, Parser, Linker, Compiler und Interpreter können umgesetzt werden [14]. Bezogen auf Modularität unterstützt Xtext die *Referenzierung*, die *Wiederverwendung (Reuse)* und die *Erweiterung*. Die *Einbettung* wird von Xtext nicht unterstützt, da die Adaptersprache von zwei BaseLanguages erben müsste. Xtext unterstützt aber nur die Erweiterung von einer Basisgrammatik [5, S.414ff].

5.1.3 Modularität in Spoofax

Die Spoofax Language Workbench ist eine Plattform für die Entwicklung von textuellen parserbasierten domänenspezifischen Sprachen und basiert auf dem Eclipse Framework. Die Spoofax-Umgebung unterstützt eine agile Entwicklung von Sprachen, indem sie die inkrementelle und iterative Entwicklung erlaubt und geeignete Editoren während der Entwicklung bereitstellt, mit denen unter anderem die abstrakte Syntax dargestellt werden kann. Spoofax benutzt die modulare Syntaxdefinition SDF, die stark modular sind [15]. Im Gegensatz zu Xtext benutzt Spoofax einen Parser ohne Scanner (scannerless parsing). Wie bereits erwähnt sind die Sprachdefinitionen in Spoofax typischerweise modularisiert. Bezogen auf Modularität unterstützt Spoofax die *Referenzierung*, die *Wiederverwendung (Reuse)*, die *Einbettung* und die *Erweiterung*. Die *Erweiterung* lässt sich in Spoofax wegen seiner modularen Strkutur über Import der bestehenden Module realisieren [5, S.428ff].

6 Fazit

Domänenspezifische Sprachen können bei richtigem Einsatz und mit guten Grundkenntnissen deutliche Vorteile in der Softwareentwicklung bringen. Durch den Einsatz von Sprachbaukästen können die Kosten in der Entwicklung gering gehalten werden. Der Einsatz von DSLs lohnt sich aber erst ab einer bestimmten Projektgröße oder in Prozessen, die sich ständig wiederholen. Das Einführen der modularen Entwicklung ermöglicht zusätzlich die Wiederverwendung und Komposition der Sprachen und senkt die Kosten weiter. Die Entwicklung von Language Workbenches wie MPS oder Xtext wird meines Erachtens weiter gehen und stark an Bedeutung zunehmen. Vor allem die Möglichkeit der Komposition und Spracherweiterung wird den Einsatz dieser Sprachbaukästen in der Praxis fördern.

Literatur

[1] M. P. Ward. Language oriented programming, 1994.

[2] Martin Fowler. Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`, letzter Aufruf am 06.02.2014, 2014.

[3] JetBrains Sergey Dmitriev. Language oriented programming: The next programming paradigm, 2004.

[4] Anneke G. Kleppe. *Software language engineering: Creating domain-specific languages using metamodels*. Addison-Wesley, Upper Saddle River and NJ, 2009.

[5] Markus Voelter. Dsl engineering. designing, implementing and using domain-specific languages. `http://voelter.de/dslbook/markusvoelter-dslengineering-1.0.pdf`, letzter Aufruf am 21.02.2014, 2013.

[6] Holger Krahn Steven Völkel Bernhard Rumpe. Mit sprachbaukaesten

zur schnelleren softwareentwicklung: Domaenenspezifische sprachen modular entwickeln, 12.08.2013.

[7] Turhan Özgür. *Domain-Specific Modeling: A practical approach: A comparison of Microsoft DSL Tools and Eclipse Modeling Framework in the context of Model-Driven Development*. Lambert Academic Publishing, Köln, 2009.

[8] Thomas Stahl. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt-Verl., Heidelberg, 2 edition, 2007.

[9] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*, volume 1 of *Aachener Informatik-Berichte, Software Engineering*. Shaker, Aachen, 2010.

[10] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: Enabling full code generation*. Wiley-Interscience and IEEE Computer Society, Hoboken and N.J, 2008.

[11] Markus Voelter. Language and ide modularization, extension and composition with mps. `http://voelter.de/data/pub/Voelter-GTTSE-MPS.pdf`, letzter Aufruf am 13.02.2014, 2011.

[12] Markus Knecht Jürg Luthiger. Modulare domänenspezifische sprachen: 33 imvs fokus report 2012. `http://www.fhnw.ch/technik/imvs/publikationen/artikel-2012`, letzter Aufruf am 21.02.2014, 2012.

[13] JetBrains, Inc, and http://www jetbrains com. Jetbrains :: Meta programming system — language oriented programming environment and dsl creation tool. `http://www.jetbrains.com/mps/`, letzter Aufruf am 06.02.2014.

[14] Sven Efftinge. Xtext - language development. `http://www.eclipse.org/Xtext/`, letzter Aufruf am 06.02.2014, 2014.

[15] William R. Cook, Martin Rinard, Siobhán Clarke, Markus Völter, and Eelco Visser. Language extension and composition with language workbenches. page 301.