Entwicklung von domänenspezifischen Modellierungssprachen

Christoph Lohe
218 623

Betreut von Dipl.-Inform. Christian Fuß

Zusammenfassung

Domänenspezifische Modellierungssprachen finden eine immer größere Bedeutung in der Softwareentwicklung. Domänenwissen, -semantik und -sprache bilden hierbei die Kernelemente. Obwohl domänenspezifische Modellierungssprachen seit längerem existieren, gibt es unterschiedliche Auffassungen über deren Definition. Diese Seminararbeit befasst sich mit aktuellen Ansichten bezüglich der Kernelemente von domänenspezifischen Modellierungssprachen und deren verschiedenen Entwicklungsaspekten.

Inhaltsverzeichnis

1 Einleitung

Die Definition einer domänenspezifischen Modellierungssprache wird von Experten unter verschiedenen Blickwinkeln betrachtet. Die dadurch entstehenden unterschiedlichen Ansichten erschweren eine einheitliche Definition. In der Literatur werden verschiedene Ansätze von Deursen [3], Mernik [11], Hudak [7], Spinellis [18] und Tolvanen [10] verfolgt. Alle Autoren sind in den letzten Jahren im Bereich der domänenspezifischen Modellierungssprachen aktiv gewesen.

Domänenspezifische Modellierungssprachen werden in der Softwaretechnik als Fachsprachen für Domänenexperten zur Beschreibung oder Modellierung von Domänenproblemen eingesetzt. Der Ursprung und die Analyse von domänenspezifischen Modellierungssprachen wird in Abschnitt 2 grundlegend erläutert. Abschnitt 3 befasst sich insbesondere mit den Entwicklungsmethoden nach Mernik [11], da dieser einen Überblick von allgemein angewendeten Techniken zusammenfasst. Werkzeuge zur Realisierung von domänenspezifischen Modellierungssprachen werden in Abschnitt 4 präsentiert. Abschließend wird das Thema anhand eines Praxisbeispiels verdeutlicht.

2 DSL-Grundlagen

2.1 Historie von DSLs

Seit Mitte der neunziger Jahre findet der Begriff *Domänenspezifische Modellierungssprache* (Domain Specific Modeling Language, DSML) eine zentrale Bedeutung in der Entwicklung von modellbasierten Sprachen. Alle DSMLs basieren auf *Domänenspezifischen Sprachen* (Domain Specific Languages, DSL). In der Literatur werden die Bezeichnung DSML und DSL meistens synonym verwendet. Heute ist der Begriff DSL implizit als DSML zu betrachten, da die Anwendung von Modellierungskonzepten in der Softwaretechnik eine immer wichtigere Rolle spielt.

DSLs werden seit Jahrzehnten in der Softwareentwicklung eingesetzt, jedoch unter Verwendung anderer Bezeichnungen, wie *application-oriented languages* [16] oder *little languages* [2]. Bereits zu Beginn der fünfziger Jahre wurde die Automatic Programmed Tools Language (APTL) [15] für die computerisierte numerische Steuerung von Maschinen als DSL entwickelt. APTL bildet heute oft die Basis für die Programmierung von Computerized-Numerical-Control (CNC) Maschinen.

In den sechziger Jahren wurde eine DSL zur Syntaxbeschreibung der Program-

miersprache Algol 60 entwickelt, welche unter der Bezeichnung Backus-Naur-Form (BNF) [9] bekannt geworden ist. Letztendlich handelt es sich bei der BNF um eine Chomsky-2-Grammatik, was wiederum zeigt, wie weit der Begriff der DSL in die Vergangenheit zurückgreift.

Eine weitere sehr bekannte DSL aus den siebziger Jahren ist die „Structured Query Language“ (SQL). Diese einfache, deklarative Datenbanksprache wurde von IBM konzipiert und ist heute fester Bestandteil für die Datenabfrage von relationalen Datenbanksystemen. Gegen Ende der siebziger Jahre entstand YACC (Yet Another Compiler-Compiler) zur Erstellung von Parser-Generatoren. Die YACC-Spezifikationssprache gehört ebenfalls zu der Klasse der DSLs.

Im Jahr 1989 stellte Tim-Berners Lee die „Hypertext Markup Language“ (HTML), basierend auf der „Standard Generalized Markup Language“ (SGML), vor, die eine der bekanntesten DSLs weltweit ist, da sie die Aufmerksamkeit der Öffentlichkeit erfolgreich auf das World Wide Web lenkte.

Seit den neunziger Jahren ist das Aufkommen von DSMLs durch den Übergang zu grafischen Modellierungskonzepten im Bereich der Softwaretechnik erkennbar. DSLs werden für die Modellierung von domänenbezogenen Problemen in komplexen Softwareprojekten herangezogen. Mit Hilfe einer abstrakten Beschreibungsebene wird konkret die Lösung eines Problems semantisch beschrieben. Dabei ist es wichtig diese Ebene so genau wie möglich auf das Problemfeld einzugrenzen.

Für den weiteren Verlauf der Seminararbeit werden die Begriffe DSL und DSML synonym behandelt. Grund dafür ist, dass jede DSL ein Modell textueller oder grafischer Art beschreibt. Ein Beispiel für ein Modell textueller Art bietet die DSL CHEM [2]. In Abbildung 1 wird links die grafische Darstellung des Penicillinmoleküls in CHEM beschrieben.

```
R1:   ring4 pointing 45 put N at 2
      doublebond -135 from R1.V3 ; O
      backbond up from R1.V1 ; H
      frontbond -45 from R1.V4 ; N
      H above N
      bond left from N ; C
      doublebond up ; O
      bond length .1 left from C ; CH2
      bond length .1 left
      benzene pointing left
R2:   flatring5 put S at 1 put N at 4 with .V5 at R1.V1
      bond 20 from R2.V2 ; CH3
      bond 90 from R2.V2 ; CH3
      bond 90 from R2.V3 ; H
      backbond 170 from R2.V3 ; COOH
```

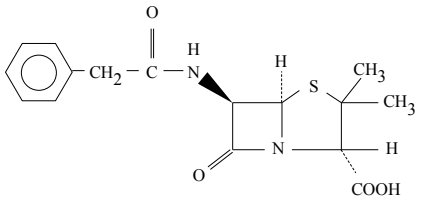

Abbildung 1: *Molekülmodell Penicillin G beschrieben in DSL CHEM*

Das Gesamtkonzept von Softwaresystemen lässt sich nur schwer auf textueller Modellebene übersichtlich beschreiben. Praktischere Anwendung findet hierbei

ein grafischer Modellierungsansatz. Meta-Metamodell Konzepte, wie das MOF-Konzept (Meta Object Facility), ein Standard der OMG[1], verfolgen das Ziel, ein standardisiertes Modellierungsverfahren zur Verfügung zu stellen, um die Integrierbarkeit von Domänenlösungen zu ermöglichen.

2.2 Domäne und ihre Analyse

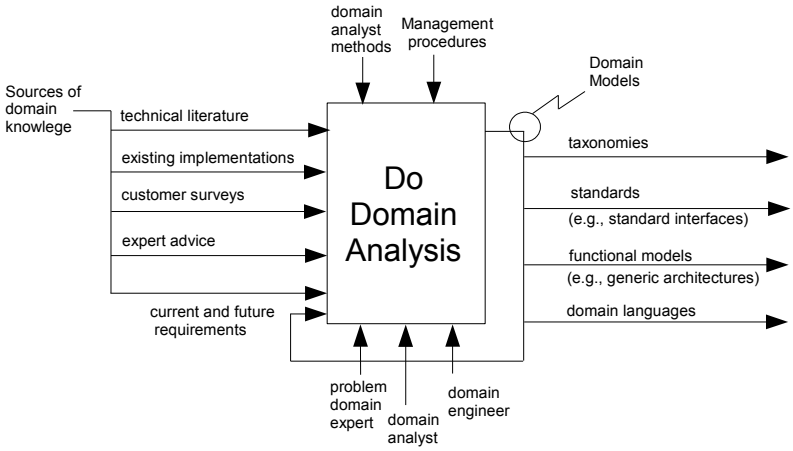

Abbildung 2: *Domänen-Analyse nach Prieto-Díaz [14]*

Eine Domäne ist ein bestimmtes Fach- bzw. Anwendungsgebiet. Dieses Fachgebiet besitzt eine eigene Syntax und Semantik zu dem Zweck gemeinsames Wissen kontextbezogen auszutauschen. Der Sprachschatz, aufbauend auf dem Domänenvokabular (Fachterminologien), ist expertenorientiert. Dieses spiegelt sich in der Fachliteratur wieder.

Der Forschungsbereich der Domänen-Analyse[2] befasst sich mit systematischen Konzepten zur Sammlung von wiederverwendaren Informationen und Strukturen über die Domäne.

Abbildung 2 illustriert die generellen Zusamenhänge in einer Domänen-Analyse nach Prieto-Díaz [14]. Eine Domänen-Analyse arbeitet auf einer Menge von Ein- und Ausgabedaten. Die domänenspezifischen Informationen bestehen in Form von Quellcodes, Dokumentationen, Benutzerhandbüchern, Testplänen und dem Domänenwissen über aktuelle und zukünftige Systeme. Domänen-

[1] Object Management Group, URL: http://www.omg.org/mof/

[2] Neighbors [12] führte als erster den Term Domänen-Analyse in Analogie zur System-Analyse ein und bezeichnete sie als eine „Aktivität zur Identifizierung von Objekten und Operationen aus Klassen von ähnlichen Systemen in einer bestimmten Problemdomäne."

experten und Domänenanalysten, verantwortlich für die Leitung der Domänen-Analyse, extrahieren relevante Informationen und Wissen. Das Ergebnis einer Domänen-Analyse sind Domänen-Modelle, Standards und Sammlungen von wiederverwendbaren Komponenten[3]. Hauptmerkmal ist das Domänen-Modell, das nach Mernik [11] aus folgenden Komponenten besteht:

- Domänendefinition, um den Domänenbereich abzugrenzen
- Domänenterminologie (Vokabular, Ontologie)
- Beschreibung der Domänenkonzepte
- Feauture-Modelle zur Beschreibung von Gemeinsamkeiten und Variabilitäten von Domänenkonzepten und ihren Zusammenhängen

Eine praktische Umsetzung einer konreten Domänen-Analyse ist *FODA* (Feature-Oriented Domain Analysis) [4] und wird u.a. in der Analysephase 3.2 bei der Entwicklung einer DSL eingesetzt. Startpunkt ist die Erstellung eines *Feature*-Diagramms.

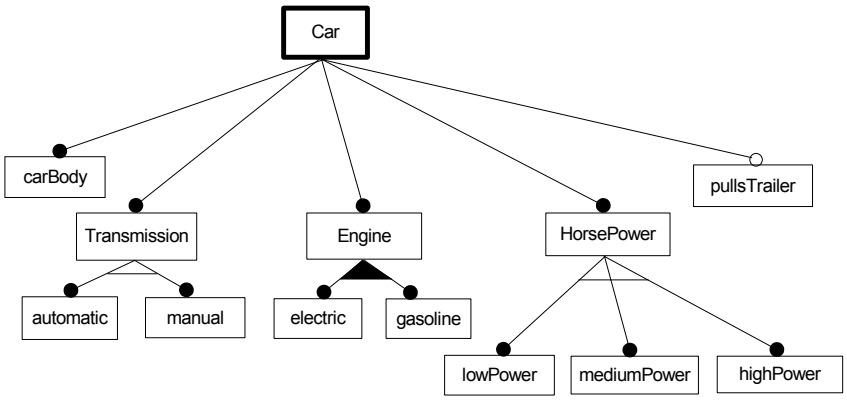

Abbildung 3: *Feature-Diagramm für ein einfaches Auto (Deursen [4])*

Abbildung 3 ist ein simples *Feature*-Diagramm für ein Auto. Absolut notwendige *Features* (*mandatory features*) werden mit einem gefüllten Punkt dargestellt. Optionale *Features* (*optional features*) sind dagegen mit einem leeren Punkt gekennzeichnet. Leere Dreiecke entsprechen einer exklusiven Auswahl von *Features*, während gefüllte Dreiecke eine nicht-exklusive Auswahl von *Features* (*alternative features*) repräsentieren. Das *Feature*-Diagram ist der Ausgangspunkt für das *Feature*-Modell. Es repräsentiert eine hierarchische Zerlegung der *Featu-*

[3]Für den Bereich der Softwaretechnik ist die Domänen-Analyse interessant für die objektorientierte Programmierung oder die Entwicklung von *Software Factories* (Produktlinien), aufgrund der intensiven (System-)Analyse von wiederverwendbaren Komponenten (Modulen, Klassen)

res in notwendige, alternative oder optionale *Features*. Aufbauend auf dem *Feature*-Diagramm wird ein *Feature*-Modell aufgestellt, welches zu den bereits erwähnten Punkten weitere Informationen über die *Feature*-Semantiken und *Feature*-Kombinationsmöglichkeiten enthält. Für die DSL-Entwicklung ist die Analyse bezüglich der Gemeinsamkeiten von Komponenten innerhalb der Domäne interessant, da sie zugleich die Hauptquelle für wiederverwendbare Komponenten (DSL-Sprachkonstrukten) sind. Als einfaches Beispiel angeführt ist in Abbildung 3 „carBody“ ein notwendiges *Feature* in der Domäne *Car*. Eine DSL für *Car* würde als Resultat einer *FODA*-Analyse die Vokabel *carBody* als domänenspezifisches Fachwort enthalten.

2.3 DSL

Eine allgemein anerkannte Definition vom Term DSL gibt es nicht. Die Definitionen unterscheiden sich in verschiedenen Punkten. Während Deursen [3] die Ausführbarkeit von einer DSL verlangt, ist für Wile [21] die Erzeugung eines lauffähigen Quellcodes nicht unbedingt Teil des DSL-Prozesses. Die Kerneigenschaften einer DSL sind charakterisiert durch:

- umfassendes/klar abgegrenztes Domänenvokabular
- aussagekräftigen Fokus auf eine Problemdomäne
- ein höheres Abstraktionsniveau zur Beschreibung des zu lösenden Domänenproblems

DSLs sind zudem keine Turing-vollständigen Programmiersprachen, sondern werden auf deklarativer Art und Weise zur domänenspezifischen Lösungsbeschreibung eingesetzt.

2.4 Vergleich DSLs mit Programmiersprachen

Für Wile [21] ist der Nutzungsvorteil einer DSL, dass der Domänenexperte seine eigene Fachsprache zur Beschreibung des Problems anwenden kann. Die Erfahrung eines Programmierers ist dabei nicht erforderlich. Wie schon in Abbildung 1 deutlich wurde, erlaubt die Ausdrucksstärke von CHEM einem Chemiker (Domänenexperte) nach kurzer Einarbeitung die grafische Darstellung von Molekülmodellen zu beschreiben. Dies gelingt durch die Anbindung von CHEM an die eigene Fachsprache.

Tabelle 1: *DSL vs. GPL*

DSL	**GPL**
textuelles / grafisches Modell	Quellcode
Zweck intuitiv besser lesbar	Zweck erfordert genauere Analyse des Quellcodes
Wissen aus Domänen	Wissen basierend auf Von-Neumann Architektur
abhängig von Domäne einsetzbar	unabhängig von Domäne einsetzbar
Anpassung erfordert Modelländerung (Semantik einfacher prüfbar)	Anpassung erfordert Quellcodeänderungen / Gefahr, dass Quellcode 'unüberschaubar' wird
DSL-Programmierer konzentrieren sich direkt auf Domänenproblem mit Hilfe der Domänensprache / des Domänenwissens	GPL-Programmierer konzentrieren sich auf Domänenproblem (Verständnis vorhanden?) und auf die komplexe Quellcodeumsetzung gleichzeitig
Domänenkonstrukte für detaillierte Problembeschreibung	Sammlung von Methoden/Funktion, direktes Mapping von Domänenkonstrukten auf APIs schwer oder gar nicht möglich

Eine Auflistung von Unterschieden zwischen DSLs und GPLs wird in Tabelle 1 aufgeführt. Sie verdeutlicht zum einen den Bezug von DSLs zu einer Domäne und zum anderen die allgemeine Anwendbarkeit einer GPL unabhängig von der Domäne. Die Domänenabhängigkeit einer DSL verleiht dem DSL-Anwender eine konzeptionelle Ausdruckskraft, die durch GPL-Sprachkonstrukte nicht erreicht wird.

3 Entwicklung von DSLs

Die Entwicklung einer DSL ist kein einheitlicher Prozess. Nach Mernik [11] besteht diese aus mehreren Phasen:

- Entscheidungsphase
- Analysephase
- Design- und Implementierungsphase

Der Entwicklungsprozess orientiert sich an der Reihenfolge dieser Phasen[4]. Sie dienen einer systematischen Aufgabenverteilung bei der Entwicklung und können sich wechselseitig beeinflussen. Ein Ergebnis aus der Analysephase kann auf die Entscheidungsphase einwirken, was sich gegebenenfalls im Design niederschlägt. Diese gegenseitigen Einwirkungen führen zu wiederholten Durchläufen der Entwicklungsphasen, um eine möglichst aussdrucksstarke DSL für die Domäne zu erhalten.

3.1 Entscheidungsphase

Mernik [11] betrachtet die Entscheidungsphase als Teil des Entwicklungsprozesses und benutzt sogenannte „Decision Patterns“ (DP) [5].

Die wörtliche Übersetzung von „Decision Patterns“ („Entscheidungsmuster“) ist irreführend. Konkreter aufgefasst sind DPs softwaretechnische Grundkonzeptionen in denen ein Einsatz von DSLs empfohlen werden kann.

Tabelle 2 führt einige DPs auf. Besonderes Augenmerk gilt bei den DPs auf die Automatisierung, Produktlinie und Notation. Diese DPs sind typische DSL-Einsatzgebiete.

3.2 Analysephase

In Abschnitt 2.2 wurde ausführlich auf eine Methodik (*FODA*) zur Vorgehensweise bei einer Domänenanalyse eingegangen. Während der Analysephase wird die Problemdomäne identifiziert und Domänenwissen angesammelt. Die Analysephase legt den Grundstein für die Entwicklung einer DSL indem sie die nötigen domänenspezifischen Terminologien und Semantiken für eine DSL-Spezifikation liefert.

3.3 Design- und Implementierungsphase

Eine Reihe von Methoden stehen DSL-Entwicklern für die Implementierung einer DSL in Abhängigkeit vom Design zur Auswahl. Die Bandbreite von Techni-

[4] Eine DSL-Entwicklung ist nicht als Wasserfallmodell durchführbar.

[5] Christopher Alexander [1] (Architekt) prägte den Begriff „pattern“. Seine „pattern language“ beschreibt Lösungsmethodiken für typische Problemen aus dem Architekturbereich. Ableitend von Alexanders „patterns“ benutzt die Informatik (besonders der angelsächsische Raum) ebenfalls diesen Begriff, um damit üblich verwendete Implementierungsverfahren zu bezeichnen (Bsp.: Model-View-Controller Pattern).

Tabelle 2: *Kurzbeschreibung der Decision Patterns nach Mernik [11]*

Decision Pattern	**Beschreibung**	**DSL**
Automatisierung	Eliminierung von wiederkehrenden Aufgaben	RoTL
Produktlinie	Spezifikation von Produkteditionen einer Produktlinie	GAL
System Front-End	Konfiguration von System Front-Ends (Bsp.: XML-Konfigurationskripte für Services)	CPL
GUI-Konstruktion	Beschreibung von GUIs	AUI
Notation	Einführung von domänenspezifischen Begriffen (Bsp.: API mit domänenspezifischen Methodennamen ausstatten)	Verischemelog
AVOPT	(Domänenspezifische Analyse, Verifikation, Optimierung, Parallelisierung, Transformation) Komplettlösung des AVOPT-Prinzips nicht praktikabel umsetzbar mittels GPLs, aber mit DSLs.	ATMOL

ken variiert zwischen einer absoluten Neuentwicklung oder der Integration von GPL-Elementen. In der Implementierungsphase wird die DSL programmiertechnisch so umgesetzt, wie zuvor in der Designphase definiert. Die meisten Methodiken sind aus dem klassischen GPL-Implementierungbereich. Erst neue DSLs in visueller Form erfordern weitere Techniken und Standards, die eine visuelle DSL-Entwicklung ermöglichen (Abschnitt 4.2). Nachfolgend werden Techniken für das Design und die Implementierung von textuellen DSLs beschrieben.

Nach Mernik [11] ist das Design einer DSL in drei Kategorien aufteilbar:

- Nutzung einer Basissprache (*Language Exploitation*)
- Neuentwicklung (*Language Invention*)
- *Common Off-The-Shelf* Prinzip (*COTS*)

Spinellis [18] unterscheidet für die Realisierung der oben genannten Kategorien zwischen verschiedenen Designtechniken, die direkten Einfluss auf die Art der Implementierungstechnik haben:

- Quelle-zu-Quelle (*Source-to-Source*)
  - lexikalische Verarbeitung
- Huckepack (*Piggyback*)

- Einschränkung (*Specialization / Restriction*)
- Erweiterung (*Extension*)
- *Pipeline*

Grundlegend ist, dass DSLs in GPLs transformiert werden. Infolgedessen ist *Source-to-Source* die generellste Form, die als Designtechnik bei einer DSL-Entwicklung angewendet werden kann. Nach Spinellis [18] ist *Source-to-Source* im Bereich der DSL-Entwicklung die Transformation einer DSL in eine Basissprache aus der softwaretechnischen Infrastruktur der Domäne. Diese liefert zugleich die Werkzeuge um die Basissprache (transformiert aus der DSL) für den softwaretechnischen Einsatz zu kompilieren.

Die oben genannten Designtechniken werden durch folgende Implementierungsmethoden angewandt, die nach Deursen [3] unterteilt sind in:

- Interpretation oder Compilierung
- Präprozessor
- Eingebette Sprachen (*Embedded Languages*) / Domänenspezifische Softwarebibliotheken
- Erweiterbarer Compiler / Erweiterbarer Interpreter

Piggyback, *Extension*, *Restriction* und *Pipeline* sind spezielle Designtechniken für eine *Language Exploitation*, die ebenfalls auf *Source-to-Source* aufbauen können (*Piggyback* 3.3.1). Anstatt der Bezeichnung *Specialization*, wie in der Literatur üblich, wird in dieser Seminararbeit die Bezeichnung *Restriction* verwendet. Der Grund dafür ist, dass Spezialisierungen durch Einschränkungen von Sprachfunktionalitäten der Basissprache umgesetzt werden.

3.3.1 Language Exploitation

Eine neue DSL der Basissprache aufzusetzen (*Piggyback*), diese zu erweitern (*Extension*) oder Sprachelemente der Basissprache einzuschränken (*Restriction*) sind Designtechniken für eine *Language Exploitation*. Nachfolgend die verschiedenen Bedeutungen dieser Designtechniken.

***Piggyback*:** Das *Piggyback*-Verfahren wird verwendet, wenn die DSL und die Basissprache gemeinsame Eigenschaften besitzen. Die DSL wird „huckepack“ der Basissprache aufgesetzt, ohne diese zu verändern. Das bedeutet, es werden DSL-Operatoren innerhalb der Basissprache integriert. LEX nutzt zur Beschreibung der Lexeme dieses Verfahren und wird auf der GPL

C „Huckepack“ aufgesetzt. Gesondert formatierte Abschnitte, die eine eigene LEX-Syntax und Semantik besitzen, werden beim Kompiliervorgang in die Basissprache (GPL C) übersetzt. Dieser Schritt entspricht einer *Source-to-Source*-Transformation. Vorteil der *Piggyback*-Methode ist die Erweiterung der Basissprachen mit High-Level-Operatoren, die domänenspezifische Konstrukte (Lexeme) implementieren.

***Language Extension*:** Die Basissprache muss für neue Funktionalitäten erweitert werden. Der Unterschied zum Huckepack-Verfahren liegt darin, dass alle DSL-Konstrukte mit Konstrukten der Basissprache implementiert werden. Im Fall von LEX ist die Basissprache C. Lexeme werden mit regulären Ausdrücken beschrieben, die nicht C-Konform sind. Wäre LEX im Sinne des Erweiterung-Verfahrens implementiert worden, müssten die Lexeme ebenfalls in Form von C-Konstrukten beschrieben werden.

***Restriction*:** Für die Entwicklung der DSL werden nicht alle Eigenschaften der Basissprachen benötigt. Diese Technik wird unter anderem verwendet, wenn unsichere Aspekte wie dynamische Speicheranforderungen oder Threads nicht Teil der DSL sein dürfen. Laut Spinellis [18] ist HTML eine Spezialisierung von SGML nach diesem Verfahren.

3.3.2 Language Invention

Klassischer Ansatz bei einer *Language Invention* ist die Implementierung eines Interpreters oder Compilers. Standardwerkzeuge aus dem Compilerbau können dazu genutzt werden speziell auf die DSL zugeschnittene Interpreter oder Compiler zu entwickeln. Dies hat den Vorteil Fehleranalysen sowie Optimierungen direkt auf der DSL-Ebene auszuführen. Nachteil ist der hohe Aufwand einer kompletten Interpreter- oder Compiler-Neuentwicklung und das Problem der Wiederverwendbarkeit der entwickelten DSL-Komponenten für die Konstruktion neuer DSLs.

3.3.3 Language Extension

Die Präprozessortechnik ist der klassische Ansatz für eine *Language Extension*. Mernik [11] zählt *Source-to-Source*, *Pipeline* und *lexikalische Verarbeitung* zu den Designmethoden, die geeignet sind für die Präprozessortechnik. Die DSL

wird über einen Präprozessor in eine GPL (*Source-to-Source*) oder eine weitere DSL (*Pipeline*) transformiert.

Die *Pipeline*-Technik transformiert/übersetzt eine DSL über mehrere DSL-Stationen in die Basissprache. CHEM [2] wurde nach diesem Prinzip umgesetzt. Im ersten Schritt wird CHEM [2] nach SCATTER [2], dann nach PIC [2] und letztendlich nach C übersetzt. Die Transformationen finden jeweils von einer DSL zu einer anderen DSL statt. (*Source-to-Source*-Transformationen bestehen aus zwei Stationen; beginnen mit einer DSL und enden mit einer GPL.)

Bekannt für die lexikalische Verarbeitung[6] ist der C-Präprozessor. Teile des Quellcodes werden durch sogenannte Direktiven markiert (in C gekennzeichnet mit #) und bei der lexikalischen Verarbeitung durch die Konstrukte aus der Basissprache ersetzt.

3.3.4 *Eingebettete Sprachen / Domänenspezifische Softwarebibliotheken*

Die *Embedded*-Methode ist erstmals von Hudak [7] beschrieben worden. Funktionale Programmiersprachen wie Haskell oder Lisp eignen sich besonders für das Prinzip der Einbettung. Die DSL nutzt dabei die vorhandene Syntax der Basissprache um domänenspezifische Konstrukte einzubetten. Listing 1 ist ein Beispiel einer eingebetteten DSL in Haskell nach Hudak [7]. Die Beispiel-DSL wurde in einem Projekt für Applikationen der amerikanischen Marine entworfen [7] mit dem Ziel eine Sprache zu entwickeln, in der Zielsysteme programmiert werden können.

Listing 1: Eingebettete DSL nach Hudak [7] für Marine-Zielsystem

```
1   -- Geometric regions are represented as functions:
2   type Region = Point -> Bool
3
4   -- so to test a point's membership in a region, we do:
5   inRegion        :: Point -> Region -> Bool
6   p `inRegion` r = r p
7
8   -- Given suitable defintions of "circle", "outside", and /\backslash:
9   circle          :: Radius -> Region     -- creates a region with given radius
10  outside         :: Region -> Region     -- the logical negation of a region
11  (/\backslash)   :: Region -> Region -> Region   -- the intersects of two regions
12
13  -- we can then define a function to generate an annulus:
14  annulus         :: Radius -> Radius -> Region
15  annulus r1 r2 = outside (circle r1) /\backslash circle r2
```

In Zeile 1-11 werden die nötigen DSL-Konstrukte auf Basis von Haskell beschrieben. Der DSL-Anwender kann diese DSL direkt benutzen, um ein Zielareal als Ring (annulus) zu beschreiben ohne tiefere Kenntnis von Haskell zu besitzen.

[6] Weitere bekannte Werkzeuge für die lexikalische Verarbeitung sind *Perl*, *Python* und *awk*.

[7] DSL-Experiment geleitet von der ARPA (Advanced Research Projects Agency), ONR (Office of Naval Research) und NSWC (Naval Surface Warfare Center)

Der Vorteil dieses Ansatzes liegt darin, dass bewährte Eigenschaften der Basissprache direkt „mitvererbt“ und in der DSL vollständig genutzt werden können. Listing 1 ist nur ein kleiner Ausschnit aus der von Hudak [7] entwickelten DSL. In der Praxis arbeiten DSL-Anwender mit einer Ansammlung von domänenspezifischen Haskell-Funktionen (domänenspezifische Bibliothek) um die softwaretechnischen Domänenprobleme zu lösen.

3.3.5 Erweiterbarer Compiler / Erweiterbarer Interpreter

Eine weitere Implementierungstechnik ist die Verwendung von erweiterbaren Compilern oder erweiterbaren Interpretern. Die zuvor beschriebenen Präprozessortechniken werden direkt im Compiler oder Interpreter integriert. Dadurch kann der Übersetzungsvorgang weiter optimiert werden. In der Regel sind Interpreter leichter domänenspezifisch erweiterbar als Compiler, da ihre Erweiterung einfacher umsetzbar ist. Häufige Kandidaten für diese Technik sind DSLs, die als Basissprache Tcl [13] verwenden.

3.3.6 Common Off-The-Shelf Prinzip

Das *Common Off-The-Shelf* Prinzip bezieht sich auf die Nutzung von Werkzeugen, die standardmäßig beim Produkt vorhanden sind. Generell kommen für den *Common Off-The-Shelf* basierten Ansatz alle Produkte einer Domäne in Frage, die sich mittels Makrosprachen weiter spezialisieren lassen. Als Beispiel ist die Microsoft Office-Produktpalette (besonders Excel, Access Powerpoint) aufzuführen. SIC (Survey Instrument Creator) von Wile [22] ist eine DSL für die Erstellung eines Befragungssystems, entwickelt nach dem *Common Off-The-Shelf* Prinzip.

4 Werkzeuge

Eine effiziente DSL-Entwicklung benötigt Werkzeuge für Spezifikation, Transformation wie auch syntaxgestütztes Editieren. Hierbei helfen Sprachentwicklungssysteme und Toolkits aus der Forschung sowie dem kommerziellen Bereich. Sie bieten einen Verbund aus Werkzeugen an und haben nach Heering und Klint [6] eines gemeinsam: die Generierung von Werkzeugen anhand der Sprachspezifikation. Die verschiedenen Sprachentwicklungssysteme und Toolkits generieren wiederum eine Bandbreite von Werkzeugen:

- Compiler / Interpreter
- Konsistenzprüfer
- Editor mit integrierter Syntaxprüfung
- Analysewerkzeuge
- Entwicklungsumgebungen
- Applikations- und Codegenerator

Der Entwicklungsablauf aller Werkzeuge ist miteinander vergleichbar. Zunächst wird eine Sprachspezifikation für die DSL erstellt. Dies kann in Form von BNF-ähnlichen Metasprachen realisiert werden oder durch visuelle Metamodelle und Graphschemata. Im zweiten Schritt werden Transformationsregeln implementiert, die für die Generierung von Zwischensprachen (Pipeline-Technik in Abschnitt 3.3.3) oder der endgültigen Zielsprache (GPL) angewandt werden. Die Transformationregeln werden häufig durch Termersetzungstechniken umgesetzt.

Sprachentwicklungssysteme und Toolkits unterscheiden sich in ihrer Eignung in Bezug auf textuellen oder visuellen DSLs. Abschnitt 4.1 und Abschnitt 4.2 befassen sich mit Werkzeugen für DSLs beider Art.

4.1 Werkzeuge für textuelle DSLs

Textuelle DSLs sind deklarativ und in reiner Textform. Für eine standardisierte konzeptionelle Visualisierung sind keine DSL-Spezifikationen vorhanden. Die „Modellierung“ von textuellen DSLs benötigt Werkzeuge zur:

- lexikalischen und syntaktischen Analyse
- semantischen Transformation in eine GPL (oder eine andere DSL).

ASF+SDF [20] Meta-Environment[8] (ASM) ist ein Sprachentwicklungssystem und wird in Abschnitt 4.1.1 beschrieben. Anschließend wird das Toolkit Khepera [5] zur Konstruktion eines DSL-Compilers erläutert.

4.1.1 ASF+SDF Meta-Environment

ASM ist ein Sprachentwicklungssystem für die Beschreibung von formalen Spezifikationen und der Transformation/Analyse von vorhandenen Programmen. ASF

[8] Algebraic Specification Formalism + Syntax Definition Formalism, URL: http://www.cwi.nl/htbin/sen1/twiki/bin/view/Meta-Environment [Online-Zugriff: 22.01.2007]

definiert die DSL-Semantik mittels Termersetzungsgleichungen, während SDF die DSL-Syntax in einer BNF-ähnlichen Form beschreibt. Die Meta-Environment ist eine grafische Entwicklungsumgebung und bietet:

- Werkzeuge zur Erzeugung / zum Editieren von ASF+SDF Spezifikationen
- Testen und Ausführen der Spezifikationen
- automatisches Erstellen von Tools (Parser, Pretty Printer, Debugger)

Die Grammatik einer DSL wird durch Kombination von ASF und SDF als eine Menge von Funktionen definiert, um einen abstrakten Syntaxbaum zu konstruieren. Ziel ist es durch Transformation von abstrakten Syntaxbäumen einen DSL-Compiler zu entwickeln.

4.1.2 Khepera

Khepera ist ein Toolkit für die „Rapid Implementation“ von DSL-Compilern. Als Implementierungstechnik wird eine Source-to-Source Variante eingesetzt. Die Compiler-Schritte bestehen aus mehreren Transformationsschritten von abstrakten Syntaxbäumen. Für die Spezifikation der Transformationsschritte wird eine spezielle Sprache für Khepera verwendet.

Im Unterschied zu ASM muss für die DSL ein Parser mit LEX und YACC implementiert werden, da Khepera keinen eigenen Parser anbietet. Desweiteren werden alle angewendeten Termersetzungsregeln für die Transformation der abstrakten Syntaxbäume protokolliert. Dies ermöglicht flexibles Debugging der DSL-Programme.

4.2 Werkzeuge für visuelle DSLs

Visuelle DSLs werden grafisch modelliert. Die grafischen Modellelemente werden in Metamodellen oder Graphschemata spezifiziert. Diese grafischen DSL-Spezifikationen beinhalten die Rahmenbedingungen und Relationen zwischen grafischen Modellelementen. Implizit sind visuelle DSLs zugleich textuelle DSLs, da die grafischen Modelle in eine Textform transformiert werden können. Die transformierte Textform ist nicht geeignet für die konzeptionelle Beschreibung eines Domänenproblems aus der Sicht eines Domänenexperten, sondern spiegeln die grafische Konstruktion des Modells wieder.

Die Anwendung von Metamodellen in MetaEdit+ wird in Abschnitt 4.2.1 beschrieben. Abschnitt 4.2.2 befasst sich mit der Benutzung von Graphschemata in PROGRES.

4.2.1 MetaEdit+

MetaEdit+[9] ist eine CASE[10]-Entwicklungsumgebung bestehend aus einer Sammlung von Werkzeugen für Modellierung, Dokumentation und Codeerzeugung.

Die DSL-Entwicklung beginnt mit der Spezifikation eines Metamodells der Domäne. MetaEdit+ nutzt dazu das Metamodellierungskonzept GOPPRR (*Graph-Object-Property-Port-Role-Relationship*). Diese Basiskomponenten des Metamodells werden als Metatypen bezeichnet und dienen zur Spezifikation von Domänenobjekten. Metatypen sind:

Graph: *Graph* ist die visuelle Darstellung aller Metatypen.

Object: *Object* ist Hauptelement der Modellierung und wiederverwendbar. Bei der Modellierung können *Objects* in verschiedenen Graphen eingesetzt werden.

Property: *Property* enthält die Beschreibung eines *Objects*.

Port: Optional können *Ports* spezifiziert werden, um ein *Object* semantisch zu erweitern (*Object*: Verstärker, *Ports*: analoger oder digitaler Eingang). Alle Instanzen des erweiterten *Objects* teilen sich die gleichen *Ports*.

Role: *Role* spezifiziert die Rolle eines *Objects* innerhalb von *Relationships*.

Relationship: Explizite Beziehungen zwischen einer Gruppen von Objekten sind *Relationships*, welche mit Rollenspezifikationen (*Roles*) an ein Objekt gebunden sind (Bsp.: Object X 'parent' of Object Y [Role X = parent]).

Die Erstellung des Metamodells wird durch Metatyp-Werkzeuge (MetaEdit+ Workbench) unterstützt. Eine Domänenlösung wird vom DSL-Anwender in einem Graph-Editor visuell modelliert.

MERL (MetaEdit+ generator definition language) ist eine Skriptsprache für die lexikalische Verarbeitung der textuellen Beschreibung eines grafischen Modells. Ein weiteres Werkzeug von MetaEdit+ ist der Generator-Editor. Durch Anwendung von MERL werden Teile des grafischen Modells in lauffähigen Quellcode transformiert. Hierbei wird die Expertise eines erfahrenen Programmierers benötigt.

DSL-Anwender profitieren vom domänenspezifischen Abstraktionslevel mittels der Modellierung im Graphik-Editor. Lauffähiger Quellcode wird mit Hilfe des auf MERL basierenden Codegenerators erzeugt.

[9] URL: http://www.metacase.com, CEO Juha-Pekka Tolvanen
[10] Computer-Aided Software Engineering

4.2.2 PROGRES

PROGRES[11] [17] (PROgrammed Graph REwriting Systems) ist ein High-Level Sprachentwicklungssystem für visuelle Modellierungswerkzeuge und zugleich eine „ausführbare“ Spezifikationssprache basierend auf Graphgrammatiken. Dieses Sprachentwicklungssystem besteht aus einer integrierten Programmierumgebung mit:

- Graphik- und Texteditoren für Klassendiagramme
- Graphersetzungsregeln
- Attributierungsgleichungen und imperativen Kontrollstrukturen
- Analysatoren
- Interpreter und Compiler (Codegenerierung)
- einem Tcl/Tk-basierenden Rapid-Prototyping-Generator für grafische Editoren

Analog zu MetaEdit+ 4.2.1 muss zu Beginn ein Beschreibungsmodell für die DSL-Spezifikation erstellt werden. Dies geschieht durch objektorientierte Modellierung eines Graphschemas [12]. DSLs werden im Rahmen eines Graphschemas visuell modelliert.

Transformationen werden durch Graphersetzungsregeln, sogenannte Produktionen, formalisiert. Die Graphmodelle werden in GRAS[13] (Graph-Oriented Database System for (Software) Engineering Applications), einer graphbasierten Datenbank angelegt und modifiziert. In Analogie zur Datankbanktransaktionen sind Operationen in PROGRES Transaktionen.

Vorteil von PROGRES ist die automatische Erzeugung von Bearbeitungswerkzeugen anhand der Graphschemata. Somit werden die grafischen Modellierungswerkzeuge für die Editierung, Prüfung und Dokumentation während der DSL-Entwicklung indirekt mitentwickelt. Durch die Generierung eines Editors anhand der DSL-Spezifikation (Graphschema) wird dem DSL-Anwender direkt ein DSL-Modellierungswerkzeug zur Verfügung gestellt. Dieser Editor ermöglicht die Modellierung von einem Arbeitsgraphen (domänenspezifische Lösung), der konform zum Graphschema (DSL-Spezifikation) ist. Editieroperationen für den Arbeitsgraphen werden durch Produktionen und Transaktionen definiert.

Um automatisch GPL-Code für die domänenspezifischen Plattformen zu ge-

[11] URL: http://www-i3.informatik.rwth-aachen.de/progres [Online-Zugriff am: 22.01.2007]

[12] Knoten im Graphschema sind attributiert (Variablendeklaration) und haben einen Typ (ähnlich zum Metatyp *Role* 4.2.1) während Kanten nur typisiert sind.

[13] URL: http://www-i3.informatik.rwth-aachen.de/tikiwiki/tiki-index.php?page_ref_id=346 [Online-Zugriff am: 22.01.2007]

nerieren, müssen weitere Arbeitsgraphen modelliert werden. Diese nutzen spezifischen Produktionen und Transaktionen für die Transformation in den erforderlichen GPL-Code.

5 DSL-Projekte in der Praxis

Zur Veranschaulichung, welchen praktischen Nutzen die Verwendung einer DSL mit sich bringt, wird in Abschnitt 5.2 ein erfolgreiches DSL-Projekt beschrieben. Es kann als erfolgreich bewertet werden, da es sich einen festen Platz in der Softwareentwicklung bei NOKIA gesichert hat und aktuell zur Umsetzung von domänenspezifischen Problemen angewandt wird. Anhand des in Abschnitt 4.2 vorgestellten Werkzeugs MetaEdit+ werden die einzelnen Aspekte einer DSL und deren Entwicklung beleuchtet.

5.1 Produktivität von DSLs

Die Produktivität von DSLs wird in der Literaur gemessen durch:

- relativen Vergleich zwischen der Größe des DSL-Codes und GPL-Codes
- Vergleich von Projekt-Zeitaufwänden basierend auf DSL oder GPL

In der Regel ist eine höhere Produktivität mit Kostenersparnissen verbunden. Abbildung 4 prognostiziert Hudaks [8] theoretischen Kostenverlauf einer DSL-Entwicklung im Jahr 2001. Vergleichsweise zur konventionellen GPL-Entwicklung ist hierbei die Anfangsinvestition höher. Im Laufe der Softwarenutzung rentiert sich das DSL-Konzept, da Kosten in Bereichen wie Wartbarkeit, Wiederverwendbarkeit u.a. minimiert werden.

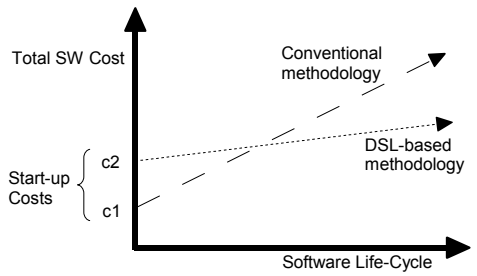

Abbildung 4: *DSL-Technologie Payoff nach Hudak [8]*

Spinellis Prognose wird von Tolvanens Erfahrungen mit MetaEdit+ (Abschnitt 4.2.1) gestützt. Tolvanens [19] DSL-Projekte erreichten Produktivitätssteigerungen bei:

- Nokia: 10-fache Produktivitätssteigerung
- Lucent Technologies: 3 bis 10-fache Produktivitätssteigerung
- US Airforce: 3-fache Produktivitätssteigerung gegenüber 'best practice' code, 50% weniger Fehler

David Narraway[14] betont (die Nutzung von MetaEdit+ zur DSL-Modellierung):

> Die Entwicklung eines Moduls, für die wir ursprünglich auch mit dem neuen Werkzeug zwei Wochen veranschlagt hatten, hat genau einen Tag gedauert: vom ersten Entwurf bis zum fertigen Produkt.

Ein Vergleich von automatisch generiertem Code anhand einer DSL-Spezifikation ergab in einem Projekt von Wile [21] ein Verhältnis von einem 50-zeiligen Code zu einer Zeile DSL-Spezifikation.

Die oben genannten Ergebnisse sind nicht repräsentativ, sondern Fakten aus erfolgreichen DSL-Projekten. Sie zeigen eine Tendenz der Produktivtät bei Einsatz von DSLs in der Softwaretechnik.

5.2 Symbian S60 GUI-Programmierung

Symbian[15] OS ist ein speziell für Mobilfunktelefone entwickeltes Betriebssystem. Das SDK[16] S60[17] wird von Nokia zur Verfügung gestellt und bietet für Entwickler eine Sammlung von Funktionen, um ihre Applikationen vorab in einem Emulator testen zu können.

Abbildung 5 zeigt die Modellierung eines Restaurantfinderprogramms (RP) für die Anfrage an einen SMS-Service. Im Kontext grün markiert ist der automatisch generierte Python-Quellcode für das Pop-Up-Menü „Select restaurant“ (Metatyp *Object*: Popup_menu). Das Pop-Up-Menü enhält drei Auswahlkriterien (Metatyp *Properties*: 'by name', 'by type', 'exit') und drei rollenbezogene Beziehungen (Metatyp *Role*: 'fromChoice') in Abhängigkeit von den *Properties*. Die

[14] Projektmanager bei Nokia, URL: http://www.metacase.com/de/cases/nokia.html [Online-Zugriff: 22.01.2007]

[15] URL: http://www.symbian.com [Online-Zugriff: 22.01.2007]

[16] Software Development Kit

[17] URL: http://www.forum.nokia.com/main/platforms/s60/index.html [Online-Zugriff: 22.01.2007]

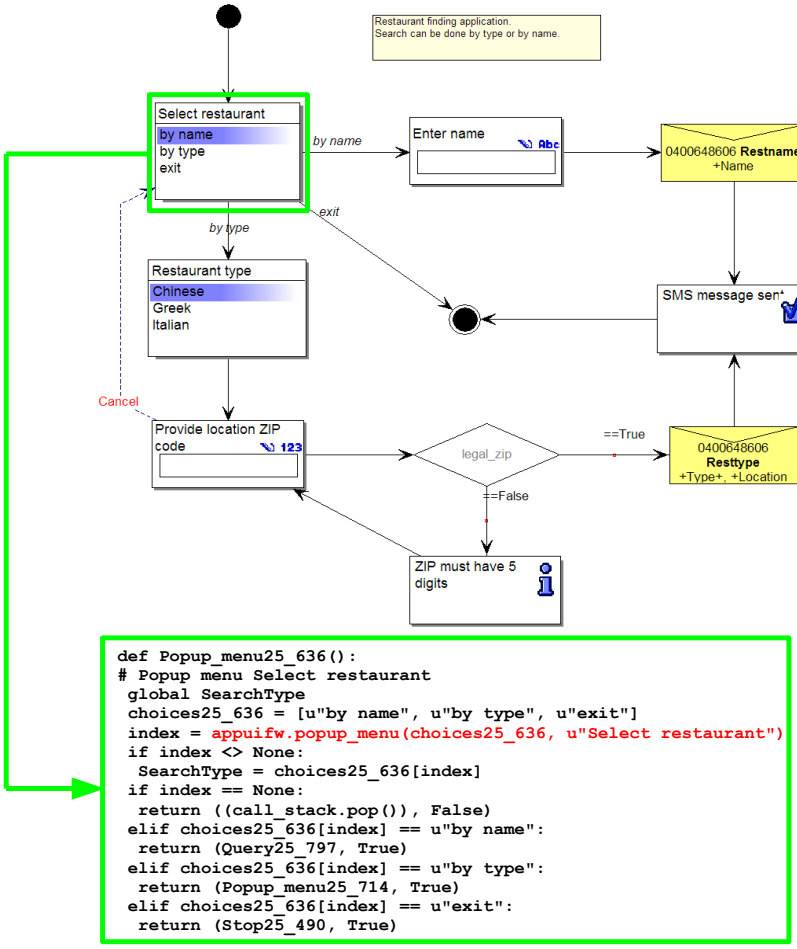

Abbildung 5: *DSL für S60 Smartphones entwickelt in MetaEdit; Implementierung eines Restaurantfinders; Python-Quellcode erzeugt durch automatischen Codegenerator*

Anbindung an das verwendete domänenspezifische S60 SDK geschieht über den Methodenaufruf (in rot markiert):

```
appuifw.popup_menu
```

Das SDK-Modul *appuifw* stellt Klassen zur Verfügung für native S60 GUI-Objekte wie Textdisplay, Diaologboxen und Menüs.

Über ein Menü wird eine Anfrage zur Restaurantsuche gestartet und es kann per Restaurantname oder Restauranttyp gesucht werden. Bei Auswahl einer Suche

nach Restaurantname benötigt das Programm die textuelle Eingabe des gesuchten Restaurants. Anschließend wird der eingegebene Restaurantname an einen SMS-Service verschickt. Wurde eine Auswahl nach Restauranttyp getätigt, so erhält der Benutzer eine Auswahlliste bezüglich der Restauranttypen (Chinesisch, Griechisch, Italienisch) und muss danach eine Eingabe in Form eines ZIP-Codes vollziehen. Entspricht die Eingabe des Benutzers einem gültigen ZIP-Code wird abschließend der Restauranttyp samt gesuchten ZIP-Codebereich an den SMS-Service versandt.

Vorgänge, die den schwarzen Punkt inmitten des Modells erreichen, werden sofort gestoppt. Der schwarze Punkt symbolisiert daher den Endzustand vom RP und kann entweder direkt vom Eingangsmenü oder nach Abschluss des SMS-Versands erreicht werden.

Das Ablesen der Vorgänge bereitet für einen langjährigen Nutzer eines Mobilfunktelefons keine großen Schwierigkeiten. Die Zusammenhänge zwischen den einzelnen Vorgängen und die allgemein bekannten Vokabeln wie 'ZIP' oder 'SMS' tragen aufgrund des vorhandenen Domänenwissens des Nutzers zum einfachen Verständnis des Modells bei.

Mit MetaEdit lassen sich Generatoren modellieren, welche für die Implementierung des RP im Nachhinein benötigt werden. Unter anderem handelt es sich um Generatoren für die automatische Codegenerierung, Modellexportierung oder die Generierung eines Wörterbuchs aus den angewandten Vokabeln. Abbildung 5 zeigt den von MetaEdit automatisch generierten Python-Quellcodes für die Auswahlfunktion der Restaurantsuche (Suche nach Restaurantname oder -typ). Das SDK S60 unterstützt ebenfalls die Programmiersprachen C++, Java, Visual Basic und C# und somit ist bei einer Entwicklung entsprechender Codegeneratoren die Übersetzung von RP in andere lauffähige S60 Plattform-Sprachen möglich.

Das RP hat zwar auf den ersten Blick keine sehr komplexe Funktionalität, auf den zweiten Blick jedoch ist gerade diese Komplexität hinter der abstrakten DSL-Modellierungsansicht verborgen. Die Größe des vollständigen Python-Quellcodes beläuft sich auf 132 Zeilen. Der rasante Wachstum der Quellcodegröße bei einem komplexen S60 Programm lässt sich schnell vor Augen führen, wenn eine Applikation aus dem Bereich der Navigation betrachtet wird. Allein die flexible Schnittstellenimplementierung zu anderen Systemkomponenten, wie der Anschluss eines GPS-Empfängers oder die Bluetoothanbindung eines Headsets, benötigen eine Reihe von einzelnen weitaus größeren Programmen als RP. Die Einhaltung der Übersicht auf reiner Quellcodebasis ist ab einer bestimmten Größe und einem bestimmten Komplexitätsgrads nicht mehr möglich. Die grafische Modellierung hingegen verschafft einen raschen Gesamtüberblick für den Domänenexperten und kann bei Bedarf in weitere Komponenten aufgeteilt werden.

6 Zusammenfassung und Ausblick

DSLs sind seit Jahrzehnten Bestandteil der Softwareentwicklung. Ihre Historie wurde in Abschnitt 2 beschrieben. Eine einheitliche Definition des Begriffs DSL ist nicht in der Literatur vorhanden ([3] vs. Wile [21]). Daher wird in Abschnitt 2 eine grundlegende Definition mit Kernpunkten einer DSL beschrieben. Zu einer DSL gehört eine detaillierte Domänen-Analyse und dafür wurde der Domänenbegriff in Abschnitt 2.2 näher erläutert. FODA [4] hat eine explizite Vorgehensweise für Domänen-Analyse im Konzept fest integriert (Feature-Diagramme). Die Klasse der GPLs hat nicht an Bedeutung verloren, sondern im Gegenteil, sie werden in Zukunft weiterhin ein fester Bestandteil beim DSL-Entwicklungsprozess sein. Der letzte Schritt ist bei allen DSLs stets die Codegenerierung von GPL-Quellcode. Unterscheidungsmerkmale zwischen einer DSL und GPL wurden in Abschnitt 2.4 beschrieben.

Die Entwicklung einer DSL ist eine komplexe Aufgabe und besteht aus mehreren Phasen (Abschnitt 3), die sich auch in einem Verfeinerungsprozess wiederholen können. Ziel ist es, dass Domänenexperten ihr Domänenwissen durch Benutzung der DSL für neue Domänenlösungen einsetzen können. Eine Reihe von Werkzeugen steht für den DSL-Entwicklungsprozess zur Verfügung (Abschnitt 4).
Die in Abschnitt 5 gezeigte Abbildung 5 verdeutlicht die Stärken einer DSL. Applikationen für Mobilfunktelefone, die als Betriebssystem eine Symbianplattform besitzen, können mit einer DSL sehr effizient programmiert werden (Abschnitt 5.2). Durch den Wechsel von verschiedenen Codegeneratoren kann die GPL-Basis verändert werden ohne die bestehende DSL zu beeinflussen.

DSLs fallen in den Bereich von modellgestützten Architekturen und liegen damit im Trend. Der Erfolg von DSLs in der Vergangenheit ist nicht zu verkennen, denn selbst diese Seminararbeit wurde in LaTex geschrieben. Aktuell werden DSLs erfolgreich in folgenden Bereichen angewendet und weiter entwickelt:

- Autoindustrie
- Versicherungssektor
- Mobilfunkgeräte
- Automatisierung
- Finanzsektor
- IP-Telefonie
- Web-Applikationen

Die Vorteile einer DSL wurden in dieser Seminararbeit mehrfach erläutert, aber was sind ihre Nachteile? Im Moment bestehen trotz aller guter Eigenschaften einer DSL große Hemmungen in die DSL-Entwicklung einer Domäne zu investieren. Der Erfolg ist nicht garantiert und wenn kein Erfolg erlangt wird, ist auch gleich ein Großteil der Investitionen verloren. Denn im Gegensatz zu dem Ansatz, eine Softwarelösung mit GPLs zu realisieren, müssen bei einer DSL zusätzliche Einstiegskosten beachtet werden. Schließlich kann die neue DSL nur durch Weiter- bzw. Ausbildung der Angestellten eingesetzt werden. Ein weiterer Nachteil ist gerade die starke Domänenbezogenheit. Wenn die Domäne 'ausstirbt' findet die zuvor schwer erworbene DSL keinen Einsatz mehr.

Literatur

[1] ALEXANDER, CHRISTOPHER, SARA ISHIKAWA und MURRAY SILVERSTEIN: *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, August.

[2] BENTLEY, JON: *Programming pearls: little languages*. Commun. ACM, 29(8):711–721, 1986.

[3] DEURSEN, ARIE VAN, PAUL KLINT und JOOST VISSER: *Domain-Specific Languages: An Annotated Bibliography.* SIGPLAN Notices, 35(6):26–36, 2000.

[4] DEURSEN, A. VAN und P. KLINT: *Domain-specific language design requires feature descriptions*. Journal of Computing and Information Technology, 2001.

[5] FAITH, RICKARD E., LARS S. NYLAND und JAN F. PRINS: *KHEPERA: A System for Rapid Implementation of Domain Specific Languages*. Seiten 243–256, 1997.

[6] HEERING, JAN und PAUL KLINT: *Semantics of programming languages: a tool-oriented approach*. SIGPLAN Not., 35(3):39–48, 2000.

[7] HUDAK, PAUL: *Building domain-specific embedded languages*. ACM Comput. Surv., 28(4es):196, 1996.

[8] HUDAK, PAUL: *Modular Domain Specific Languages and Tools*. In: DEVANBU, P. und J. POULIN (Herausgeber): *Proceedings: Fifth*

International Conference on Software Reuse, Seiten 134–142. IEEE Computer Society Press, 1998.

[9] KNUTH, DONALD E.: *Backus normal form vs. Backus Naur form.* Commun. ACM, 7(12):735–736, 1964.

[10] LUOMA, JANNE, STEVEN KELLY und JUHA-PEKKA TOLVANEN: *Defining Domain-Specific Modeling Languages: Collected Experiences.* 2004. `http://www.dsmforum.org/Events/DSM04/luoma.pdf`. [Online-Zugriff: 22.01.2007].

[11] MERNIK, MARJAN, JAN HEERING und ANTHONY M. SLOANE: *When and how to develop domain-specific languages.* ACM Comput. Surv., 37(4):316–344, 2005.

[12] NEIGHBORS, J.: *Software Construction Using Components.* Doktorarbeit, 1980. `http://www.bayfronttechnologies.com/thesis.htm`. [Online-Zugriff: 22.01.2007].

[13] OUSTERHOUT, JOHN K.: *Scripting: Higher-Level Programming for the 21st Century.* IEEE Computer, 31(3):23–30, 1998.

[14] PRIETO-DÍAZ, RUBÉN: *Domain analysis: an introduction.* SIGSOFT Softw. Eng. Notes, 15(2):47–54, 1990.

[15] ROSS, DOUGLAS T.: *Origins of the APT language for automatically programmed tools.* SIGPLAN Not., 13(8):61–99, 1978.

[16] SAMMET, J. E.: *Programming Languages: History and Fundamentals.* 1969.

[17] SCHÜRR, ANDY: *PROGRES: A Visual Language and Environment for PROgramming with Graph REwrite Systems*, 1994.

[18] SPINELLIS, DIOMIDIS: *Notable design patterns for domain-specific languages.* Journal of Systems and Software, 56(1):91–99, 2001.

[19] TOLVANEN, JUHA-PEKKA, JEFF GRAY und MATTI ROSSI: *2nd Workshop On Domain-Specific Visual Languages, http://www.cis.uab.edu/info/OOPSLA-DSVL2/*, 2002.

[20] VISSER, EELCO: *A Family of Syntax Definition Formalisms.* In: *ASF+SDF'95. A Workshop on Generating Tools from Algebraic Specifications*, Seiten 89–126. Technical Report P9504, Programming Research Group, University of Amsterdam, May 1995.

[21] WILE, DAVID S.: *Supporting the DSL spectrum*. Journal of Computing and Information Technology (CIT), Seiten 263–287, 2001.

[22] WILE, DAVID S.: *Lessons learned from real DSL experiments.* Sci. Comput. Program., 51(3):265–290, 2004.