

Sprach- und bibliotheksbasierte Abstraktion

Martin Feilkas

Technische Universität München
Institut für Informatik
Boltzmannstr. 3
D-85748 Garching bei München

Abstract. Abstraktionen sind das wichtigste Instrument der Informatik. Abstraktion kann dabei auf zwei verschiedene Weisen nutzbar gemacht werden: Durch Verwendung von Abstraktionsmechanismen einer Programmiersprache oder durch Definition einer neuen (domänenspezifischen) Sprache, in die Konzepte direkt als Sprachkonstrukte eingeführt werden. In diesem Artikel sollen diese beiden Methoden zur Definition von Abstraktionen miteinander verglichen und herausgearbeitet werden, unter welchen Bedingungen welcher Weg am besten geeignet erscheint. Einer der zentralsten Unterschiede zwischen sprach- und bibliotheksbasierter Abstraktion ist die Möglichkeit, die Verwendung der Konzepte durch syntaktische Bedingungen einzuschränken. Aus diesem Grund soll abschließend der “C# Constraint Checker” als Werkzeug zur Realisierung von syntaktischen Einschränkungen auf Bibliotheken vorgestellt werden.

1 Motivation und Einführung

Der Aufwand, der mit einem Software-Entwicklungsprojekt einhergeht, wird in sehr starkem Maße davon beeinflusst, auf welchem Abstraktionsniveau mit der Entwicklung begonnen werden kann. Der Auswahl der zu verwendenden Programmiersprachen und Bibliotheken kommt aus diesem Grund eine zentrale Rolle zu. Die Definition von Abstraktionen wird in fast jeder Systementwicklung benötigt, um Wiederverwendung zu erzielen und um eine Komplexitätsreduktion zu erreichen. Heutzutage wird in vielen Projekten zwar auch über die reine Nutzung einer höheren Programmiersprache hinausgehende sprachbasierte Abstraktion genutzt, aber die Abstraktionen, die im Rahmen des Projekts neu entwickelt werden, um domänen- und projektspezifische Konzepte ausdrückbar zu machen, werden meist bibliotheksbasiert realisiert. Die Entwicklung von Sprachen gilt oftmals als Kunst, die nur Experten beherrschen. Mittlerweile gibt es aber immer mehr Bestrebungen, auch die sprachbasierte Abstraktion einfacher nutzbar zu machen. Immer häufiger werden Code-Generatoren eingesetzt, um Teile des Codes aus spezialisierten Sprachen heraus automatisch zu erstellen. Gerade die Idee der domänenspezifischen Sprachen zielt darauf ab, Systeme einer bestimmten Domäne hoch intensional zu beschreiben. Meist verbergen sich hinter derartigen domänenspezifischen Sprachen einfache deklarative Beschreibungen einer Domäne und somit Sprachen, die normalerweise keine eigenen

Abstraktionsmechanismen definieren. Die Entscheidung für bibliotheks- oder sprachbasierte Entwicklung wird derzeit meist nicht bewusst getroffen, obwohl es eine der Fragestellungen mit den weitreichendsten Folgen ist. Aus diesem Grund werden diese beiden Alternativen in diesem Artikel genauer beleuchtet.

Sprachbasierte Abstraktion ist die Definition von Konzepten durch Entwicklung einer passenden abstrakten Syntax. Hierfür wird meist ein Metamodell oder eine Grammatik erstellt (vgl. [AP04]). Die konkrete Syntax hierzu kann dabei schließlich graphisch oder auch textuell definiert werden. Die Semantik wird dabei meist durch Implementierung eines Generators und damit durch Abbildung auf eine bereits gegebene Sprache spezifiziert.

Bibliotheksbasierte Abstraktion hingegen implementiert Konzepte durch Nutzung von Abstraktionsmechanismen einer Programmiersprache. Eine Allzweck-Programmiersprache tritt somit in diesem Fall an die Stelle des Metamodells/der Grammatik. Abstraktionsmechanismen bieten die Möglichkeit, Elemente der Programmiersprache zu kapseln und an mehreren Stellen in einem Programm wiederzuverwenden.

2 Sprach- und bibliotheksbasierte Abstraktion

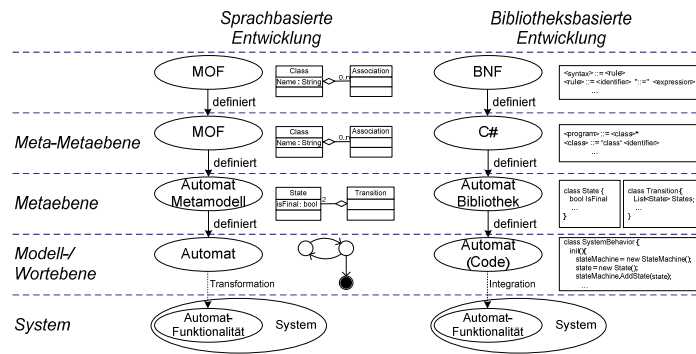


Fig. 1. Vergleich von sprach- und bibliotheksbasierter Entwicklung

Abbildung 1 zeigt beispielhaft eine Gegenüberstellung von sprach- und bibliotheksbasierter Entwicklung. Dabei wird eine Teilfunktionalität eines Systems durch einen Zustandsautomaten beschrieben. Dieser Zustandsautomat wird auf der linken Seite durch sprachbasierte Abstraktion realisiert und auf der rechten Seite durch eine Bibliothek in der Programmiersprache C#. Die beschrifteten Ebenen entsprechen den klassischen Ebenen der vierstufigen Metahierarchie [Gro]. Auf der Metaebene wird

im Falle der sprachbasierten Entwicklung MOF genutzt, um ein Metamodell eines Automaten zu definieren. Dem steht auf Seiten der bibliotheks-basierten Entwicklung eine C# Bibliothek gegenüber, welche die Funktionalität eines Automaten in sich kapselt. Auf der Modell- oder auch Wortebene wird im Fall der sprachbasierten Entwicklung ein Automat für das konkret zu entwickelnde System spezifiziert. Auf der rechten Seite muss analog dazu die generisch gehaltene Klassenbibliothek verwendet werden, um den systemspezifischen Automaten zu instanziiieren. Aus diesen beiden Spezifikationen kann schließlich das automaten-spezifische (Teil-)System erstellt werden. Der unterste im Bild dargestellte Abschnitt skizziert schließlich das fertige System (welches grundsätzlich in beiden Fällen sogar den exakt gleichen Code enthalten könnte). In der Abbildung wurde auf die Bezeichnung “Systemebene” verzichtet, da es sich streng genommen im Gegensatz zu den anderen (echten) Metaebenen nicht um eine Instanziierung, sondern um eine Transformation bzw. eine Integration handelt. Der letzte Transformations- oder auch Code-generierungsschritt erzeugt schließlich eine code-basierte Repräsentation (ggf. in C#) des Automaten, die der Metaebene und Modellebene auf der Seite der bibliotheks-basierten Entwicklung entspricht.

3 Evaluierung der Alternativen

Im Folgenden sollen die beiden Alternativen der bibliotheks-basierten Abstraktion und der Abstraktion durch Entwicklung von Sprachen anhand von drei ausgewählten Unterscheidungskriterien gegenübergestellt werden.

3.1 Syntaktische Constraints

Systeme, die in heutigen Programmiersprachen entwickelt werden, unterliegen vielen systemspezifischen Bedingungen (Constraints), die rein syntaktischer Natur sind und nicht direkt durch die verwendete Programmiersprache ausgedrückt und überprüft werden können. Diese Bedingungen werden durch domänen- oder systemspezifische Abstraktionen hervorgerufen. Sie werden oftmals durch Entscheidungen über die Strukturierung des Systems, der Architektur, eingeführt. Es handelt sich dabei einerseits um Beschränkungen, die vorschreiben, welche Komponenten nicht direkt miteinander kommunizieren dürfen, andererseits aber auch um Muster, die zwingend bei der Implementierung eingehalten werden müssen (z.B. Entwurfsmuster [GHJV95]). Aber damit noch nicht genug: Bedingungen bestehen auch innerhalb von Schnittstellen, beispielsweise wenn Funktionen immer in einer bestimmten Reihenfolge aufgerufen werden müssen.

Ein großer Teil der Informationen über derartige Constraints geht meist während der Implementierung verloren, wenn sie nicht in Form von separater Dokumentation gepflegt werden. Überprüfungen, ob derartige systemspezifische Bedingungen eingehalten werden, erfordern meist sehr viel überwiegend manuell zu erbringenden Aufwand. Der Verlust dieser Bedingungen erschwert das Verstehen des Systems und führt zu erhöhten Kosten sowie Qualitätsverlusten während der Wartungsphase.

Bei der Entwicklung einer Sprache ist es in vielen Fällen nötig, zusätzliche syntaktische Bedingungen zu integrieren, die nicht direkt in einer kontextfreien Grammatik (eines Parser-Generators) oder in einem Metamodell ausdrückbar sind. Dies wird meist durch direkte Implementierung in einem Parser oder auch über eine deklarative Beschreibung in einer Constraint-Sprache wie OCL realisiert. Im Falle bibliotheksbasierter Entwicklung ist es normalerweise nicht möglich, unerwünschte, aber in der Programmiersprache syntaktisch erlaubte Konstellationen auszuschließen. Aus diesem Grund soll in Abschnitt 4 ein Werkzeug vorgestellt werden, das es ermöglicht, derartige Einschränkungen auf Bibliotheken und Frameworks zu spezifizieren.

3.2 Verständlichkeit

Neben der abstrakten Syntax ist bei der Entwicklung einer Sprache auch die konkrete Syntax zu definieren. Diese legt die Repräsentation der abstrakten Syntax und damit auch die Art und Weise der Bearbeitung (Programmierung) von Worten der Sprache fest. Die konkrete Syntax einer Sprache ist meist entweder graphisch oder textuell.

Die Entwicklung einer eigenen Sprache, um domänenspezifische Abstraktionen nutzbar zu machen, hat den Vorteil, dass man bei der Definition der konkreten Syntax nicht durch die Syntax einer Programmiersprache eingengt wird. Somit kann die konkrete Syntax derart gestaltet werden, dass in der Domäne geläufige Symbole und Bezeichnungen in die Sprache integriert werden, um es einem Domänenexperten zu ermöglichen, die Wörter der Sprache einfacher zu interpretieren. Manche Verfechter domänenspezifischer Sprachen behaupten sogar, dass Domänenexperten selbst die Rolle des Programmierers ausfüllen und die Systeme selbst erstellen könnten [TK05].

Die Verständlichkeit und Nutzbarkeit von Abstraktionen ist eine wichtige Voraussetzung dafür, Wiederverwendung zu ermöglichen.

3.3 Kompositionalität

Im Allgemeinen ist es sehr leicht, in einem Programm mehrere Bibliotheken unterschiedlicher Domänen zu verwenden. Die Verbindung der Bibliotheken kann über die volle Ausdrucksmächtigkeit der Host-Sprache erfolgen.

Im Falle von Sprachen ist dies anders. Die gemeinsame Verwendung verschiedener Sprachen in einem System wird derzeit sehr unterschiedlich realisiert. Beispielsweise im Bereich der Web-Entwicklung werden meist unterschiedliche Sprachen zur Implementierung server- und client-seitiger Funktionalität eingesetzt. Zusammen mit HTML zur Beschreibung der graphischen Darstellung wird oftmals noch SQL zur Abfrage relationaler Datenbanken verwendet. Die Komposition wird hierbei dadurch realisiert, dass Sprachen als Zeichenkette in anderen Sprachen verwendet werden. Dadurch entstehen oftmals schwer nachvollziehbare Strukturen, die häufig erhebliche Wartbarkeitsprobleme hervorrufen.

Oftmals wird die Komposition verschiedener Sprachen auch erst auf Code-Ebene erreicht, indem verschiedene Sprachen auf die gleiche Zielsprache abgebildet werden. Ein Beispiel hierfür wäre die Verwendung eines GUI-Builders mit einer Sprache, in der ein Datenbankschema formuliert werden kann, um schließlich Datenzugriffscode zu generieren. Beide Sprachen würden schließlich Code in der gleichen general-purpose Sprache erzeugen. Die generierten Komponenten würden schließlich wiederum durch die Kompositionsmechanismen der Programmiersprache miteinander verbunden. Dieses Vorgehen hat den großen Nachteil, dass man das durch die Sprachen gewonnene Abstraktionniveau nicht durchgehend beibehalten kann und der Entwickler somit auch den generierten Code verstanden haben muss, um eine korrekte Komposition vornehmen zu können.

Alle skizzierten Möglichkeiten zur Komposition von Sprachen haben deutliche Nachteile. Somit ist die Möglichkeit der Kompositionalität von Bibliotheken ein Vorteil der bibliotheksbasierten Abstraktion.

4 Überprüfung systemspezifischer syntaktischer Bedingungen

Wie in Abschnitt 3.1 ausgeführt, entstehen durch Nutzung bibliotheksbasierter Abstraktion eine Menge von Bedingungen bezüglich der Nutzung der Abstraktionen. Diese Bedingungen stellen Einschränkungen der Syntax der Sprache für die Nutzung bestimmter Abstraktionsmechanismen dar. Damit diese Constraints nicht für die nachgelagerten Wartungsphasen verloren gehen, ist es wichtig, diese zu dokumentieren. Am besten ist es jedoch, diese direkt in den Code zu integrieren. Damit wird der Code mit Informationen aus den der Implementierung vorangehenden Phasen (Systemarchitektur/-entwurf) angereichert und zu einer vollständigeren Beschreibung des Systems. Auch die Überprüfbarkeit der Constraints ist wünschenswert, damit Verletzungen dieser Vorgaben sichtbar gemacht werden können. Dadurch kann verhindert werden, dass Schnittstellen falsch verwendet und somit Fehler hervorgerufen werden. Zudem wird dafür gesorgt, dass Architektur- und Entwurfsentscheidungen kongruent in der Implementierung umgesetzt werden. Dies erhöht somit sowohl die Zuverlässigkeit als auch die Wartbarkeit eines Systems. Der C# Constraint Checker ist ein Framework, das es erlaubt, syntaktische Bedingungen an die Abstraktionsmechanismen der Sprache C# zu annotieren. Diese Bedingungen können schließlich auf dem abstrakten Syntax-Graphen überprüft werden. Programmieretechniken wie Annotations (Java) oder Attribute (C#) erlauben es, zusätzliche Informationen zu Programmen hinzuzufügen. In gewisser Weise ist es dadurch möglich, die Grammatik der Sprache implizit zu erweitern. Der Constraint Checker überprüft die annotierten Bedingungen durch Ausführung folgender Schritte:

1. Identifikation und Sammlung aller Constraint-Attribute (Reflection auf den Assemblies),
2. Parsen der Source-Dateien,

3. Instanziierung der Constraint-Attribute und Aufruf der Methoden zur Überprüfung der Constraints,
4. Ausgabe aller Verstöße gegen die annotierten Constraints.

Der Constraint Checker versteht sich als Framework, das Funktionalität bereitstellt, um eigene Constraints zu implementieren. Bevor jedoch damit begonnen wird, eigene Constraints zu implementieren, sollte versucht werden, den gewünschten Constraint mithilfe der bereits vorhandenen Constraint-Attribute auszudrücken, die bereits ein breites Spektrum der häufig benötigten Bedingungen abdecken. Eine genaue Beschreibung der Attribute, die durch den Constraint Checker bereitgestellt werden, als auch eine umfangreiche Abgrenzung zu Verifikationstechniken ist in [Fei07] zu finden.

Der Constraint Checker zeigt, dass unter Zuhilfenahme technischer Hilfsmittel auch eine Integration von bibliotheksspezifischen syntaktischen Bedingungen möglich ist und dies somit keinen inhärenten Nachteil der bibliotheksbasierten Entwicklung darstellt.

5 Zusammenfassung

In diesem Artikel wurde die Verwendung von bibliotheksbasierter Abstraktion der Abstraktion durch die Entwicklung eigener Sprachen gegenüber gestellt. Die Vorteile der Entwicklung eigener (domänenspezifischer) Sprachen liegen im wesentlichen darin, dass es möglich ist, die Syntax restriktiv zu formulieren und ihr eine aussagekräftige konkrete Syntax zu verleihen. Dem stehen allerdings höhere Erstellungsaufwände und ein Mangel an Kompositionalität gegenüber. Abschließend wurde kurz der C# Constraint Checker vorgestellt, ein Tool, das es ermöglicht, syntaktische Restriktionen an die Abstraktionsmechanismen der Programmiersprache C# zu annotieren und statisch zu prüfen.

References

- [AP04] Marcus Alanen and Ivan Porres. A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS, Mar 2004.
- [Fei07] Martin Feilkas. Api-constraints using annotations: The c# constraint checker. In *Submitted to ACM SIGPLAN Symposium on Library-Centric Software Design LCS D '07*, 2007.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
- [Gro] Object Management Group. Meta object facility (mof) specification.
- [TK05] Juha-Pekka Tolvanen and Steven Kelly. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In *Proceedings of the 9th International Conference on Software Product Lines, SPLC 2005*, pages 198–209. Springer, 2005.