Bessere Software durch Querschneidende Module

Klaus Ostermann
Technische Universität Darmstadt
ostermann@informatik.tu-darmstadt.de

Abstract: Gute Separierung der Belange in Softwaresystemen ist der Schlüssel, um mit wachsender Komplexität umzugehen. Die wichtigste Aufgabe von Programmiersprachen in Bezug auf dieses Ziel ist die Bereitstellung von geeigneten Mitteln, um das mentale Modell eines Domänenexperten so direkt wie möglich in einer Programmiersprache festhalten zu können und damit die *intellektuelle Distanz* zwischen dem mentalen Modell und dem Programmcode so gering wie möglich zu halten. Dies ist der Kontext, im dem die hier zusammengefasste Dissertation versucht, den aktuellen Stand der Technik voranzubringen. Ein besonderer Schwerpunkt liegt dabei auf Sprachkonzepten für die Unterstützung sogenannter "querschneidender Modelle", die aus unterschiedlichen Sichten auf ein Softwaresystem resultieren und mit konventioneller Technologie zu Programmen mit schlechten softwaretechnologischen Eigenschaften wie Wartbarkeit und Wiederverwendbarkeit führen. Die Resultate dieser Arbeit bilden die Grundlage der Programmiersprache *Caesar*, die zur Zeit in mehreren industriellen Projekten evaluiert wird.

Neue Sprachkonstrukte für eine bessere Separierung der Belange (*separation of concerns*) in der Softwareentwicklung werden benötigt! Diese These ist der Ausgangspunkt der diesem Dokument zugrundeliegenden Dissertation [Os03]. Zunächst einmal: Was ist gemeint mit "guter Separierung der Belange"?

Ein Belang (*concern*) ist jede kohärente Angelegenheit in einer Problemdomäne. Ein Belang kann ein beliebig komplexes Konzept wie "Sicherheit" oder "Lohnbuchhaltung" sein, oder primitive Konzepte wie "warten auf ein Mausereignis".

Separierung der Belange ist eng verwandt mit Kompositions- und Dekompositionsmechanismen in Programmiersprachen, also die Partitionierung eines Softwaresystems in kleine Teile (Dekomposition) und das Zusammenbauen eines Softwaresystems anhand dieser kleinen Teile (Komposition). Diese Definition sagt jedoch nichts darüber, *wie* ein Softwaresystem dekomponiert werden soll. Zum Beispiel könnte ein Programm in zwei Teile zerlegt werden, bei dem ein Teil die geraden Zeilennummern und ein anderer Teil die ungeraden Zeilennummern enthält. Offensichtlich macht diese Dekomposition jedoch keinen Sinn. Wir benötigen *Kriterien*, die benutzt werden können, um geeignete Dekompositionen eines Systems zu finden.

Dies ist der Punkt, an dem sich das Konzept der Separierung der Belange, welches auf Parnas [Pa72a] und Dijkstra [Di76] zurückgeht, als nützlich erweist. Dieses Konzept postuliert, dass jeder Teil eines dekomponierten Softwaresystems für einen wohldefinierten Aspekt eines Systems verantwortlich sein sollte. Jeder Teil sollte sowenig Wissen wie möglich über andere Teile eines Systems haben, so daß es möglich ist, jeden Aspekt einzeln und isoliert von allen anderen Aspekten zu betrachten.

In konventionellen Programmiersprachen wird dieses Konzept durch Module unterstützt, die eine *hierarchische* Aufteilung der Softwaredomäne ermöglichen und damit eine direkte Abbildung hierarchischer Modelle ermöglichen. Die dieser Arbeit zugrunde liegende These ist, daß neben einer Unterstützung für hierarchische Modelle noch eine zweite Dimension von Modellen durch Programmiersprachen unterstützt werden sollte, nämlich sogenannte *querschneidende* Modelle, die durch unterschiedliche Sichten desselben Systems zustandekommen.

Der Rest dieses Artikels ist wie folgt strukturiert: In Abschnitt 1 wird kurz die Bedeutung guter Separierung der Belange für die Softwareentwicklung erläutert. In Abschnitt 2 wird der Begriff der querschneidenden Modelle eingeführt und dessen Bedeutung für die Softwaretechnologie erläutert. Schliesslich wird Abschnitt 3 einen kleinen Überblick darüber geben, wie querschneidende Modelle in der Programmiersprache Caesar unterstützt werden.

1 Vorteile guter Separierung der Belange

Eine gute Separierung der Belange hat eine Reihe wichtiger Vorteile in Bezug auf unterschiedliche Qualitätsaspekte von Software.

Der offensichtlichste Vorteile einer guten Separierung der Belange ist, dass es einfacher wird, eine gegebene Software zu *verstehen*. Wenn jeder Programmteil unabhängig von anderen Teilen begriffen werden kann, muss man nicht die gesamte Struktur eines Softwaresystems kennen, um diesen Teil zu kennen.

Wiederverwendbarkeit von Software bedeutet, dass ein Stück Software in vielen unterschiedlichen Stellen benutzt wird. Die Beziehung zur Separierung der Belange ist wie folgt: Je mehr sich ein Stück Software auf einen einzelnen (oder wenige) Belange konzentriert, um so wahrscheinlicher ist es, dass dieses Stück Software in unterschiedlichen Kontexten wiederverwendet werden kann. Ein Stück Software ist umso wiederverwendbarer, je weniger Abhängigkeiten an den Benutzungskontext bestehen, und dies ist genau dann der Fall, wenn sich dieser Teil der Software nur auf einen einzelnen Belang konzentriert.

Allgegenwärtige Wiederverwendung führt zur alten Vision der Software, die einfach aus eine Reihe gekaufter Softwarekomponenten zusammgesteckt wird; eine Idee, die sich bis in die 60'er Jahre zurückverfolgen läßt [Mc68]. Der enorme Aufwand, der in kommerzielle Komponentenmodelle wie Enterprise Java Beans (EJB) gesteckt wurde, verdeutlicht die kommerzielle Bedeutung von wiederverwendbarer Software. Ohne eine gute Separierung der Belange ist der Traum von der Lego-artigen Softwarekomponente jedoch zum Scheitern verurteilt.

Die *Skalierbarkeit* eines Softwareprozesses bezieht sich auf eine vernünftige Relation zwischen Kosten und Größe eines Softwaresystems, d.h., die Kosten explodieren nicht in einem großen Projekt. Wenn es möglich ist, Belange einzeln zu behandeln und zu implementieren, kann man Domänenexperten auf einen spezifischen Teil einer Software ansetzen und man benötigt keine allwissenden Entwickler. Die Arbeit kann also in kleine Teile

aufgeteilt werden die parallel von unterschiedlichen Teams erledigt werden. Die Tatsache, daß dies mit konventioneller Technologie nur schwer möglich ist, hat auch zum berühmten "Brooks's Law" [Br75] geführt, welches aussagt, dass das Hinzufügen von Entwicklern zu einem laufenden Projekt das Projektende nach hinten verschiebt.

Es ist ein allgemein bekannter Fakt, dass die *Wartung* eines Softwareprojektes häufig den größten Teil des Budgets verschlingt. Wartung bedeutet meist das Hinzufügen, Entfernen, oder Ändern eines speziellen Belanges der Software. Wenn ein Belang der Software geändert werden soll, gibt es zwei Probleme: a) *lokalisieren* des Belangs im Code und b) *aktualisieren* der Implementation. Wenn die Implementation eines Belangs nicht klar vom Rest der Software getrennt sondern über größere Teile des Systems verstreut ist, wird die Lokalisierung sehr schwierig, weil alle Code-Teile, die zu dem Belang beitragen, gefunden werden müssen. Der gesamte Prozess des sogenannten *reverse engineering* wird sehr viel komplexer wenn keine gute Separierung der Belange vorliegt.

Das Aktualisieren eines solches Belangs führt zu weiteren Problemen, weil es schwierig ist, die Änderungen an den verschiedenen Stellen konsistent durchzuführen und damit die Integrität der Software zu erhalten. Analoge Beobachtungen treffen auf das Hinzufügen und Entfernen der Implementation eines Belanges zu.

2 Hierarchische und Querschneidende Modelle

In den sechziger und frühen siebziger Jahren wurde der Bedarf für eine gute Organisation großer Softwareprojekte aufgrund immer häufigerer Fehlschläge offensichtlich. Die offensichtliche Lösung schien die Übertragung des *Divide et Impera* Prinzips auf die Softwaretechnik: Ein großes Problem sollte durch Zerlegung in viele kleine Probleme gelöst werden. Wirth nannte diese Art der Softwaretechnik "*program development by stepwise refinement*" [Wi71]:

> *"In each step, one or several instructions of a given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language [...] A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible."*

Bereits 1968 schrieb Dijkstra seinen einflußreichen Artikel über die Vorzüge einer Schichtenarchitektur [Di68]. Eine Jahre später spiegelten sich die neuen Einsichten auch in den Programmiersprachen wieder und gipfelten schliesslich in der Entwicklung von Modulkonstrukten [Pa72b, Wi82, DoD83] im Rahmen der strukturierten Programmierung [DDH72].

Die Basisprinzipien von Modulen können auch heute noch in jeder Programmiersprache gefunden werden: Ein Modul ist ein Stück Software, welches von anderen, tieferliegenden

Modulen abhängt, einige interne Geheimnisse hat, und eine wohldefinierte Schnittstelle zu anderen, höherliegenden Modulen anbietet. Auch in der Softwarearchitektur spiegelt sich dieses Prinzip wieder. Beispielsweise beschreibt das *Layer Pattern* [BMR+96] eine Architektur, in der ein System in Schichten zerlegt wird, so daß Schicht n nur von Schicht $n-1$ abhängt. Hierdurch wird es sehr einfach, einzelne Schichten zu ersetzen.

Physikalische Dekomposition, also die Zuordnung von Sourcecode zu Dateien, impliziert alleine noch keine semantische Relation wie Spezialierung zwischen den Modulen. Die implizite Annahme bei hierarchischer Dekomposition ist es jedoch, daß es möglich ist, die Belange eines Systems in einer semantischen Hierarchie anzuordnen, die mehr oder weniger direkt in einer entsprechenden Modulstruktur abgebildet werden kann. In den Worten von Wegner [We90] wollen wir eine direkte Korrespondenz zwischen der *logischen Hierarchie* und der *physikalischen Hierarchie*. Winkler spricht von der *konzeptorientierten Sicht* im Unterschied zur *programmorientierten Sicht* [Wi92], Meyer postuliert das *direct mapping principle* [Me97], Larman spricht vom *low representational gap* [La01] – viele unterschiedliche Namen für dieselbe Idee.

Wenn jedoch die konzeptuelle und die physikalische Hierarchie eines Systems nicht übereinstimmen, gibt es zwangsläufig Module, die für mehr als einen Belang des Systems zuständig sind, oder umgekehrt, die Implementation einiger Belange ist verteilt über mehrere Module. Das Prinzip der Separierung der Belange ist dann also verletzt.

Ob es möglich ist, eine gute Korrespondenz des logischen Modells zur physikalischen Struktur herzustellen, hängt stark von der benutzten Programmiersprache ab. Objektorientierte Sprachen beispielsweise können durch Vererbung semantische Spezialisierungsrelationen sehr viel besser abbilden als Sprachen ohne Vererbung.

Es gibt jedoch einige Modularitätsprobleme, die nach Ansicht des Autors *niemals* zufriedenstellend mit hierarchischen Dekompositionsmechanismen gelöst werden können. Diese Behauptung wird auch durch eine Reihe anderer Arbeiten belegt ([HO93, KLM+97, TOHS99, Be90, AB92, ML98, BI94]).

Nach Meinung des Autors ist der größte Schwachpunkt hierarchischer Dekomposition, daß implizit angenommen wird, daß Konzepte aus der realen Welt in fixe, objektive Konzepthierarchien zerfallen. Tatsächlich hängt jedoch unsere Wahrnehmung der realen Welt erheblich von dem Kontext ab, aus dem wir sie betrachten. Es gibt keine konzeptuelle *lingua franca*.

Jedes Softwaresystem kann aus unterschiedlichen Perspektiven betrachtet werden, und jede dieser Perspektiven impliziert möglicherweise ein anderes Modell des Systems. Das Modell, daß wir benutzen, um unsere Software in Module zu zerlegen, hat einen großen Einfluß auf die softwaretechnologischen Eigenschaften der Software. Parnas beobachtete beispielsweise bereits 1972, daß eine datenorientierte Zerlegung es einfach macht, die Repräsentation der Datenstrukturen zu ändern als eine funktionale Zerlegung [Pa72a].

Dieses Problem kann auch graphisch repräsentiert werden. Jede Form in Abb. 1 repräsentiert einen bestimmten Belang eines Softwaresystems. Wenn wir diese Belange dekomponieren möchten, können wir dies entweder zum Beispiel anhand ihrer Form (Abb. 2) oder ihrer Farbe (Abb. 3) tun. Keine dieser Zerlegungen ist *per se* besser oder schlechter als eine andere. Mit einem hierarchischen Modellierungsansatz müssen wir uns jedoch für ei-

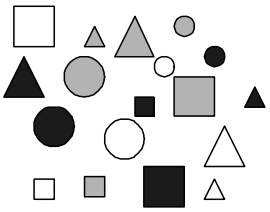

Abbildung 1: Abstrakter Raum der Belange

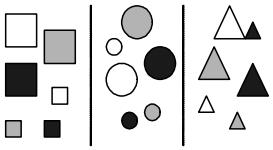

Abbildung 2: Zerlegung nach Form

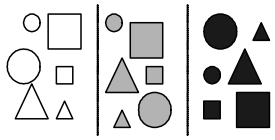

Abbildung 3: Zerlegung nach Farbe

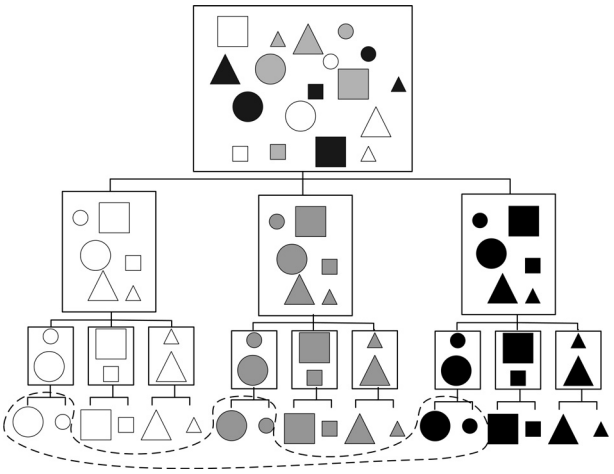

Abbildung 4: Willkürlichkeit des primären Modells

ne feste Klassifikationshierarchie entscheiden – wenn wir mehrere Klassifikationskriterien haben, müssen die Kriterien in einer Sequenz wie in Abb. 4 angeordnet werden. In diesem Beispiel war die Klassifikationssequenz *(Farbe, Form, Größe)*. Das Problem ist nun, daß nur das erste Element in dieser Liste gut modularisiert ist, während alle anderen Belange vermischt sind. Abb. 4 illustriert dies anhand des Belangs "Kreis", der in der Dekomposition querschneidend ist. Dieses Problem bezeichnen wir als *Willkürlichkeit des primären Modells*.

3 Module für Querschneidende Modelle

Dieser Abschnitt ist eine kurzer Überblick der Modulmechanismen, die die Programmierspräche Caesar zur Verfügung stellt, um die Implementierung querschneidende Modelle zu unterstützen. Die Grundidee ist es, mehrere unabhängige querschneidende Modelle in seinem Programm durch Modulkonstrukte zu unterstützen, in dem Mechanismen zur

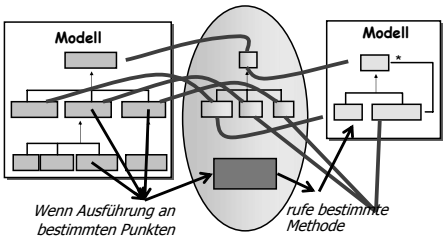

Abbildung 5: Abbildung querschneidender Modelle

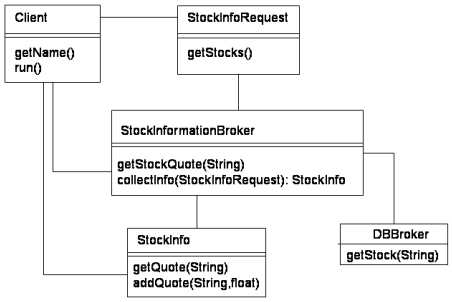

Abbildung 6: Stockbroker Anwendung

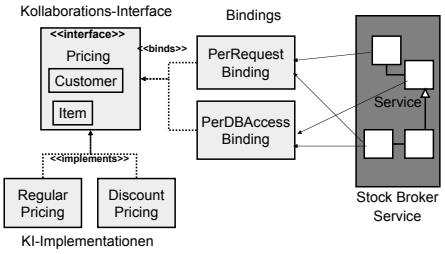

Abbildung 7: Gestaltung der Preisabrechnung mit Caesar

Verfügung gestellt werden, um querschneidende Module zueinander in Beziehung zu stellen und aneinander zu adaptieren. Dies ist in Abb. 5 illustriert.

Als Beispiel betrachten sie eine kleine Anwendung zur Abfrage von Daten über Wertpapiere (siehe Abb. 6). Ein `Client` kann ein `StockInfoRequest` Objekt an den `StockInformationBroker` schicken und erhält als Antwort ein `StockInfo` Objekt mit den angefragten Informationen.

Der Aspekt, der in diesem Beispiel betrachtet wird, ist der der Preisgestaltung: Benutzer dieses Dienstes soll ein Preis für die Benutzung in Rechnung gestellt werden. Die Gestaltung und Berechnung von Preisen ist eine Domäne, die zunächst einmal völlig unabhängig von diesem spezifischen Service ist. Aus diesem Grund macht es Sinn, Komponenten, die zur Preisgestaltungsfunktionalität beitragen, in einem zu dieser Domäne passenden Modell zu spezifizieren und nicht mit den Klassen der Stockbroker Anwendung zu vermischen.

Die Basiskonzepte von Caesar sind in Abb. 7 anhand dieses Beispiels illustriert. Von zentraler Bedeutung ist das *Kollaborations-Interface* (KI) `Pricing`. Es handelt sich hierbei um Interfaces zur Beschreibung von Abstraktionen, die für das von diesem Interface beschriebene Modell wichtig sind. In diesem einfachen Beispiel sind es die Abstraktionen `Customer` und `Item`. Komponenten aus der Preisgestaltungsdomäne können nun unter Berücksichtigung der durch dieses Modell vorgegebenen Namen und Abstraktionen spezifiziert werden, wie zum Beispiel eine Komponente, die eine reguläre Preisgestal-

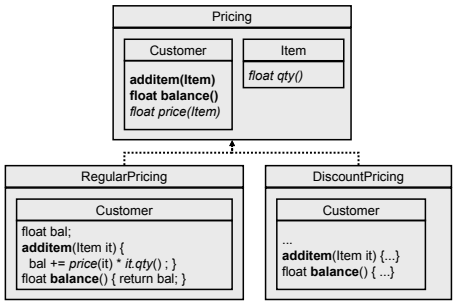

Abbildung 8: Implementation

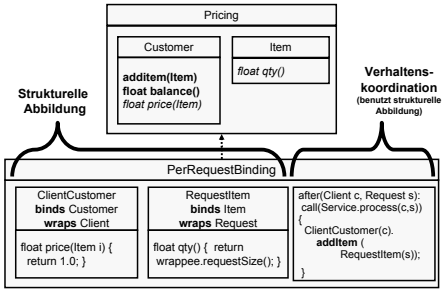

Abbildung 9: Binding

tungsstrategie (`RegularPricing`) implementiert (jedes `Item` hat einen festen Preis), oder beispielsweise eine Discount-Strategie (`DiscountPricing`). Diese Komponenten sind völlig unabhängig von ihrem Benutzungskontext, sie verweisen lediglich auf das die Domäne beschreibende Interface.

Die Abbildung des Preisbildungsmodells auf das Stockbroker Modell geschieht nun mit Hilfe eines sogenannten *Bindings*. Bei dieser Abbildung sind verschiedene Varianten denkbar. Einerseits könnte pro Anfrage an den Service abgerechnet werden (die `Item` *Rolle* wird von `StockInfoRequest` gespielt), andererseits könnte man auch pro benötigtem Zugriff auf die Datenbank abrechnen; die `Item` Rolle würde in diesem Fall von der Datenbankanfrage gespielt. Mit Hilfe der genannten Bindings lassen sich beide Abbildungen nebeneinander realisieren.

In Abb. 8 und 9 sind die genannten Interfaces, Komponenten und Bindings mit mehr Details zu sehen. Die Methoden der `Customer` und `Item` Abstraktionen aus dem Interface sind unterteilt in sogenannte `provided` Methoden (**fett** gedruckt) und `expected` Methoden (*kursiv* gedruckt). Die Aufgabe der Interface Implementierungen (Abb. 8) ist es, Implementierungen aller `provided` Methoden zur Verfügung zu stellen, wobei beliebige `expected` Methoden aufgerufen werden dürfen. Die Aufgabe der Bindings ist es, alle `expected` Methoden zur Verfügung zu stellen, wobei mit Hilfe der Schlüsselwörter `binds` und `wraps` Beziehungen zwischen den Abstraktionen aus den verschiedenen Modellen hergestellt werden können (Abb. 9).

Ein wichtiger Teil der Bindings ist die Möglichkeit, die zu kombinierenden Modelle nicht nur strukturell aufeinander abzubilden sondern auch das Verhalten zur Laufzeit zu koordinieren. Dies geschieht mit Hilfe von sogenannten *Pointcuts* und *Advices* [KHH+01]. Pointcuts sind deklarative Beschreibungen von Ereignissen im Programmablauf. In dem Beispiel bedeutet der Pointcut `call(Service.process(c,s))`, dass auf alle Aufrufe der Methode `Service.process` reagiert werden soll. Der Advice ist die Reaktion auf das Eintreten eines solchen Ereignisses. In diesem Beispiel werden die beiden aus der Stockbroker Anwendung stammenden Objekte `s` und `c` mit Hilfe eines sogenannten *Liftings* (`ClientCustomer(c)` und `RequestItem(s)`) zu ihren Rollen in der Preisbildungsdomäne umgewandelt und das neue `Item` dem Kunden in Rechnung gestellt.

Um nun eine operationales Modul zu erhalten, muss eine Interface Implementation mit

```
public deployed class PricingDeployment {
  static Map pricingMapping = new HashMap();
  static { // User <-> Pricing Mapping
    pricingMapping.put("Joe",
      new PerRequestRegularPricing());
    pricingMapping.put(„Sally",
      new PerDBAccessRegularPricing());
    }
  }
  void around( Client c ) :
    (execution( void Client.run(String []) ) && this(c)) {
      deploy( pricingMapping.get(c.getName()) )
      {
       proceed ();
      }
  }
}
```

Abbildung 10: Statisches und Dynamisches Deployment

einem Binding kombiniert werden. Dies geschieht mit einem speziellen Kompositionsoperator `&`, der zwei rekursive Klassenstrukturen miteinander kombiniert.

```
class PerRequestRegularPricing extends
     PerRequestBinding & RegularPricing

class PerDBAccessRegularPricing extends
     PerRequestBinding & RegularPricing
```

Diese komponierten Klassen sind nun voll funktional und können in der Anwendung instanziiert werden. Um die in einer Klasse definierten Advices zu aktivieren, gibt es in Caesar einen sogenannten *Deployment* Mechanismus. Hierbei wird unterschieden zwischen statischem und dynamischem Deployment. Statisches Deployment wird über das Schlüsselwort `deployed` als Klassen-Attribut ausgewählt, dynamisches Deployment über die Benutzung von dem Schlüsselwort `deploy` als Blockkonstrukt. Dies ist in Abb. 10 illustriert, in dem ein mittels statischem Deployment aktivierter Advice dynamisches Deployment benutzt, um die Preisgestaltungsstrategie abhängig vom Benutzer zur Laufzeit auszuwählen. Das Schlüsselwort `proceed` dient dazu, die ursprüngliche Ausführung fortzusetzen, in diesem Fall jedoch im Kontext des dynamisch ausgewählten Preisbildungsaspektes.

4 Zusammenfassung und Zukünftige Arbeiten

Wir haben in diesem Dokument einen kurzen Überblick über die Modularitätsprobleme gegeben, die durch rein hierarchische Module entstehen und vorgestellt, wie durch querschneidende Module in der Programmiersprache Caesar mit diesen Problemen umgegangen werden kann. Caesar wird zur Zeit in zwei unterschiedlichen Projekten mit Industriepartnern evaluiert. Hierbei wird in einem Fall ein reales Projekt parallel einmal mit

konventioneller Technologie und einmal mit Hilfe von Caesar implementiert, so daß ein aussagekräftiger Vergleich möglich wird. In dem anderen Projekt wird Caesar anhand einer Fallstudie mit unterschiedlichen anderen aspektorientierten Programmiersprachen verglichen.

Laufende und zukünftige Arbeiten im Kontext von Caesar befassen sich mit der formalen Semantik und dem Typsystem von Caesar [JDAO04], der Einbettung in Entwicklungsumgebungen [Ar04], mächtigeren Pointcutsprachen [EMO04], und speziellen Virtual Machines für aspektorientierte Sprachen [BHMO04] und dynamisches Deployment [BHMO04].

Literatur

[AB92] Aksit, M. und Bergmans, L.: Obstacles in object-oriented software development. In: *Proceedings OOPSLA '92. ACM SIGPLAN Notices 27(10)*. S. 341–358. ACM. 1992.

[Ar04] Aracic, I. Entwicklung eines eclipse plugins für caesar. Studienarbeit, TU Darmstadt. 2004.

[Be90] Berlin, L. M.: When objects collide: Experiences with reusing multiple class hierarchies. In: *Proceedings of ECOOP/OOPSLA '90, ACM SIGPLAN Notices 25(10)*. S. 181–193. 1990.

[BHMO04] Bockisch, C., Haupt, M., Mezini, M., und Ostermann, K.: Virtual Machine Support for Dynamic Join Points. In: *AOSD 2004 Proceedings*. ACM Press. 2004.

[BI94] Baclawski, K. und Indurkhya, B.: The notion of inheritance in object-oriented programming. *Communications of the ACM*. 37(9):118–119. 1994.

[BMR+96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., und Stal, M.: *Pattern-Oriented Software Architecture*. Wiley. 1996.

[Br75] Brooks, F. P.: *The Mythical Man-Month*. Addison-Wesley. 1975.

[DDH72] Dahl, O.-J., Dijkstra, E. W., und Hoare, C.: *Structured Programming*. Academic Press. 1972.

[Di68] Dijkstra, E. W.: The structure of the THE-multiprogramming system. *Communications of the ACM*. 11(5):341–346. 1968.

[Di76] Dijkstra, E. W.: *A Discipline of Programming*. Prentice Hall. 1976.

[DoD83] US Department of Defense: *Reference Manual for the Ada Programming Language*. 1983. ANSI/MIL-STD1815 A.

[EMO04] Eichberg, M., Mezini, M., und Ostermann, K. First-class pointcuts as queries. In Preparation. 2004.

[HO93] Harrison, W. und Ossher, H.: Subject-oriented programming (A critique of pure objects). In: *Proceedings OOPSLA '93. ACM SIGPLAN Notices 28(10)*. S. 411–428. 1993.

[JDAO04] Jolly, P., Drossopoulou, S., Anderson, C., und Ostermann, K.: Simple Dependent Types: Concord. In: *Workshop on Formal Techniques for Java-like Programs at ECOOP 2004*. 2004.

[KHH+01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., und Griswold, W. G.: An overview of AspectJ. In: *Proceedings of ECOOP '01*. 2001.

[KLM+97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., und Irwin, J.: Aspect-oriented programming. In: *Proceedings ECOOP'97*. LNCS 1241. S. 220–242. Springer. 1997.

[La01] Larman, C.: *Applying UML and Patterns*. Prentice Hall. 2001.

[Mc68] McIlroy, M. Mass produced software components. In P. Naur and B. Randell, Software Engineering - Report on a conference sponsored by the NATO Science Committee. 1968. Garmisch, Germany.

[Me97] Meyer, B.: *Object-Oriented Software Construction*. Prentice Hall. Second. 1997.

[ML98] Mezini, M. und Lieberherr, K.: Adaptive plug-and-play components for evolutionary software development. In: *Proceedings OOPSLA '98. ACM SIGPLAN Notices 33(10)*. S. 97–116. 1998.

[Os03] Ostermann, K.: *Modules for Hierarchical and Crosscutting Models*. PhD thesis. Technische Universität Darmstadt, Fachbereich Informatik. 2003.

[Pa72a] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. *Communications of the ACM*. 15(12):1053–1058. 1972.

[Pa72b] Parnas, D. L.: A technique for software module specification with examples. *Communications of the ACM*. 15(5):330–336. 1972.

[TOHS99] Tarr, P., Ossher, H., Harrison, W., und Sutton, S. M.: N degrees of separation: Multi-dimensional separation of concerns. In: *Proceedings International Conference on Software Engineering (ICSE) '99*. S. 107–119. ACM Press. 1999.

[We90] Wegner, P.: Concepts and paradigms of object-oriented programming. *OOPS Messenger*. 1:7–87. August 1990.

[Wi71] Wirth, N.: Program development by stepwise refinement. *Communications of the ACM*. 14(4). 1971.

[Wi82] Wirth, N.: *Programming in Modula-2*. Springer Verlag. 1982.

[Wi92] Winkler, J.: Objectivism: "class" considered harmful. *Communications of the ACM*. 35(8):128–130. 1992.

Personalien

Klaus Ostermann wurde am 6. Dezember 1974 in Cloppenburg geboren. Er erlangte sein Abitur 1994 am Copernicus-Gymnasium Löningen. Von 1995-2000 studierte er Informatik an der Universität Bonn. Das Studium wurde am 2.1.2001 mit einem Diplom in Informatik abgeschlossen. Von 2001 bis Ende 2002 war er als Doktorand in der Abteilung "Corporate Technology" der Siemens AG in München beschäftigt. 2003 wechselte Herr Ostermann als wissenschaftlicher Mitarbeiter zur Fachgruppe Softwarechnologie der TU Darmstadt unter Leitung von Prof. Mira Mezini. Dort beendete er auch erfolgreich am 9.7.2003 seine Promotion. Seit dem 1. April 2004 ist Herr Ostermann Juniorprofessor an der TU Darmstadt und vertritt dort das Fachgebiet Aspektorientierte Programmierung.