Einfluss erweiterter Programmier-Paradigmen auf die Entwicklung eingebetteter DBMS

Martin Kuhlemann, Thomas Leich, Sven Apel
{*mkuhlema,leich,apel*}*@iti.cs.uni-magdeburg.de*
Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg

Zusammenfassung

Die Unterstützung der Entwicklung eingebetteter, variabler Systeme durch die Software-Technik ist derzeit problembehaftet. Paradigmen wie die Objekt-Orientierte Programmierung (OOP) erfordern zur Entwicklung angepasster Software stetige Neuentwicklungen. Weiterführende Techniken lösen einige der Probleme der OOP. In Vorbereitung des DFG-Projekts FAME werden diese Techniken vorgestellt und analysiert. Generische Programmierung (GP) verbessert die Wiederverwendbarkeit von Software. Aspekt-Orientierte Programmierung (AOP) erhöht die Wartbarkeit durch die Modularisierung von Belangen. Feature-Orientierte Programmierung (FOP) verbessert die Erweiterbarkeit. Nachfolgend werden erste Ergebnisse einer systematischen Untersuchung der Paradigmen zur Erstellung konfigurierbarer, performanter DBMS im Bereich der eingebetteten Systeme präsentiert. Untersucht wird insbesondere der Einfluss der verschiedenen Techniken auf wünschenswerte Eigenschaften des Zielsystems. Weiterführend wird eine gezielte Kombination von Techniken vorgeschlagen um Vorteile einzelner Techniken zu vereinen.

1 Einleitung und Motivation

Die Domäne eingebetteter Systeme ist im Fokus aktueller Forschung der Software-Entwicklung [8]. In diesem Bereich gelten Einschränkungen bezüglich Echtzeitfähigkeit oder Speicher-Verbrauch - auch für die Software. Die Anforderungen an Performanz und Adaptierbarkeit der Software bei simultanen Einschränkungen der Speicherkapazität sind gegenläufig. Die Software soll möglichst spät an diese speziellen Anforderungen des Anwendungszwecks angepasst werden, um viele Anforderungs-Profile erfüllen zu können und teure Neu-Entwicklungen zu vermeiden. Techniken wie die Objekt-Orientierte Programmierung (OOP) verursachen bei der Entwicklung konfigurierbarer und erweiterbarer Software Probleme: Wiederverwendbarkeit und Variabilität sind durch eine feste Kontext-Bindung und Struktur der Klassen nur eingeschränkt möglich. Im Rahmen des FAME-Projekts[1] werden Techniken analysiert, die bestimmte Nachteile der OOP mindern. Generische Programmierung (GP) ermöglicht die feingranulare Anpassung von Strukturen [3]. Aspekt-Orientierte Programmierung (AOP) modularisiert Belange, deren Code in der OOP verstreut und vermischt vorlägen [6]. Feature-Orientierte Programmierung (FOP) ermöglicht eine statische Komposition von Software-Komponenten [2].
Ziel dieser Analyse ist es, Vor- und Nachteile der einzelnen Techniken an einem durchgängigen Beispiel der DBMS-Programmierung zu zeigen. Besondere Beachtung finden dabei die Variabilität, Erweiterbarkeit und Performanz des Zielsystems. Als Analyse-Beispiel dient das Zugriffssystem einer Datenbank. Das System wird durch das bekannte Pattern `Template-Method` in den genannten Techniken implementiert [5]. Wir zeigen, dass einzelne Techniken die Bereitstellung wünschenswerter System-Eigenschaften nicht ausreichend ermöglichen. Die Techniken lösen Teilprobleme und erschweren evtl. die Umsetzung anderer wünschenswerter Eigenschaften. Als ein vielversprechender Weg um Synergien zu erzeugen werden Kombinationen der Techniken analysiert. Ein neuer Vorschlag zur Kombination von Techniken wird präsentiert und seine Anwendbarkeit diskutiert. Die neu vorgestellte Kombination vereint die Erweiterbarkeit von FOP mit der Variabilität von GP gegenüber Typen beeinflusster Variablen.

[1] Projekt zur Entwicklung einer Software-Familie eingebetteter DBMS, unterstützt durch die DFG
http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/FAME-DBMS/index.htm

2 Analyse der untersuchten Techniken

Im folgenden Abschnitt werden die Techniken eingeführt und ihr Einfluss auf wünschenswerte Eigenschaften des DMBS wie Erweiterbarkeit, Performanz und Speicherverbrauch analysiert. Für die Untersuchung wird ein DBMS-Zugriffssystem (engl.: Data-Access-System, DAS) in den einzelnen Techniken durch Verwendung des Design-Patterns "Template-Method" implementiert [5]. Dieses Pattern ermöglicht die Variation der Implementierung konkreter Schritte eines generellen Algorithmus. Für die Aufgaben des Zugriffssystems, d. h. das Mapping einer tupelorientierten auf eine speicherorientierte Sicht, sollen variable Umsetzungen der Schritte wie "`insert`", "`remove`" oder "`modify`" von Tupeln unterstützt werden. Diese sollen ein sortiertes Auslesen von Tupeln durch deren Vorsortierung oder anfragegesteuerte Sortierung implementieren.

OOP. Die Intention der OOP ist die Kapselung von Funktionen und Eigenschaften nach ihrer Zugehörigkeit zu Realwelt-Objekten in Klassen. Erweiterungen an Klassen können nicht-invasiv durch Überschreiben, Überladen und Hinzufügen von Methoden in Subklassen eingebracht werden. Diese erben die Funktionalität ihrer Oberklasse. In der Implementierung des Zugriffssystems (vgl. Abb. 1) erben die Klassen "`ODAS`" und "`HDAS`" (Zeilen 9-13) die Beschreibung der abstrakten Klasse "`DAS`" (Zeilen 1-8) und implementieren die virtuellen Methoden "`remove`" und "`insert`" mittels Heap-Speicherung oder sortierter Speicherung. Durch das gemeinsam ererbte Interface "`DAS`" sind die Varianten nun für den übrigen Quellcode transparent ansprechbar. Erweiterungen um neue Varianten der Zugriffsstrategie können in einer Klasse gekapselt hinzugefügt werden, z. B. hash-basierte Sortierung. Die Fixierung der Strukturen und Beziehungen von Klassen im Software-Design und Quellcode erfordert invasives Eingreifen bei evolutionären Änderungen. Das verursacht Code-Replikation und im Folgenden schlecht wartbare Software.[2] Unerwartete Variationen innerhalb von Methoden-Implementierungen der Superklasse "`DAS`" - wie Indexpflege oder Logging - erfordern ebenfalls invasive Änderungen. Das modulare Hinzufügen dieser optionalen Funktionen erfordert *virtuelle* Hook-Methoden mit Default-Verhalten (Indexpflege: Zeile 8). Sie können in der Subklasse überschrieben werden. Viele OOP-Design-Pattern beruhen auf diesem Prinzip, um Variabilität und Erweiterbarkeit zu ermöglichen. Der durch virtuelle Methoden entstehende Overhead in Speicherverbrauch und Performance ist für eingebettete Systeme jedoch nicht akzeptabel [4, 9].

```
1  class DAS{ //abstract data-access-system
2   void modify(Tuple* old,Tuple* newT){
3    TID tid = remove(old);
4    insert(newT, tid);
5    modInd(newT, tid); }
6   virtual TID remove(Tuple* t)=0;
7   virtual TID insert(Tuple* t) =0;
8   virtual void modInd(Tuple* t,TID tid){}};
9  class ODAS: public DAS{//ordered insert
10  TID remove(Tuple* t){...}
11  TID insert(Tuple* newT){...}
12  TID lookup(Tuple* t){...}};
13 class HDAS: public DAS{...}//heap-insert
```

Abbildung 1: OOP-Zugriffssystem

GP. Als eine aufbauende Technik führt GP die Unabhängigkeit der Klassen und Methoden von Typen ihrer Interaktionspartner mittels Templates ein. Die als variabel deklarierten Typen werden durch die erzeugende oder aufrufende Methode durch zusätzliche Parameter zur Übersetzungszeit konkretisiert - in OOP ist diese Typisierung bereits im Design festzulegen [3]. Die Vorverarbeitung durch den Compiler macht ein einheitliches Interface mittels abstrakter Klassen unnötig. Diese Parameter ermöglichen ebenfalls eine Auswahl von Implementierungen.
Das durch Templates implementierte Zugriffssystem in Abbildung 2 verwendet die eigens zur Parametrisierung eingeführten Klassen "`HDAS`" und "`ODAS`" (Zeilen 13 und 16), um die Implementierung der Methoden des Zugriffssystems auszuwählen. Die statische Auswahl der Implementierung (Zeile 18) ermöglicht Variabilität ohne virtuelle Methoden mit ihren Nachteilen bezüglich Speicherverbrauch und Performanz. Die Vorteile dieser statischen Bindung verursachen Schwierigkeiten an anderer Stelle. Erweiterungen um zusätzliche Varianten wie hash-basierter Zugriff oder eine Veränderung Bestehender ist in diesem Beispiel nur invasiv möglich - Vererbung ver-

[2] Sowohl die Klasse als auch ihre Subklassen müssen repliziert werden um Varianten zu erhalten.

ursacht hier eine vollständige Redefinition. Die Parametrisierung im Quellcode verursacht erheblichen Pflegeaufwand bei Änderungen des Template-Interfaces. Die Reduzierung der Anforderungen an das Interface der Interaktionspartner auf die eigens verwendeten Methoden bietet eine lose Kopplung der Komponenten und Robustheit gegenüber neuen Typen. Im Gegenzug ist die semantische Prüfung auf Teilmengen-Relation nicht mehr möglich.[3]

AOP. AOP ermöglicht eine zusätzliche Modularisierung von System-Eigenschaften. OOP ermöglicht die Komposition der Komponenten nach der Teilmengen-Beziehung. Dies verursacht Code-Replikation anderer gleicher Eigenschaften der Komponenten an verschiedenen Stellen des Programms. Der Code dieser sog. "querschneidende Belange" hat zudem keine semantische Beziehung zur ihn allokierenden Klasse [6]. Durch die erweiterte Separierung der Belange werden Wartbarkeit und Verständlichkeit erhöht [3]. Der in sog. Aspekten gekapselte Code (sog. Advice) wird um Beschreibungen seiner Wirkungspunkte im Programm (sog. Pointcut) erweitert. Den Aufruf des Advice am Pointcut initiiert ein Weber [6]. Eine Vorbereitung der Komponenten auf Variabilität und Erweiterungen ist somit nicht notwendig. Variabilität wird durch Auswahl der anzuwendenden Aspekte "`ODAS`" bzw. "`HDAS`" erreicht (nicht dargestellt). Die Aspekte kapseln die variablen Implementierungen der Methoden "`hookIns`" und "`lookup`". Die zu erweiternde Klasse "`DAS`" bildet den Join-Point. Nicht benötigter Code wird nicht gewebt und benötigt keinen Speicherplatz im übersetzten System. Modulare Erweiterungen sind durch das Hinzufügen weiterer Aspekte möglich. Aspekte ermöglichen die Manipulation der Klassenhierarchien der Komponenten bis zur Endphase der Software-Entwicklung (Compilation der Software) - in OOP und GP sind diese Design-Entscheidungen zum Beginn der Software-Entwicklung zu treffen. So können variable Eigenschaften des Systems spät konfiguriert werden. Das vermeidet die Neu-Entwicklungen speziell angepasster Software.

```
1  template<class strat> class DAS{
2   void modify(Tuple* old, Tuple* newT){
3    TID tid = remove(old);
4    insert(newT, tid);
5    modInd(newT, tid); }
6   TID insert(Tuple* t){...}
7   void insert(Tuple* t, TID place){...}
8   ...
9   TID hookIns(Tuple* t);
10  void modInd(Tuple* t, TID newTID); };
11 template<class strat>
12 void DAS<strat>::modInd(Tuple* t, TID tid){...}
13 class ODAS{}; //ordered DAS
14 template<>TID DAS<ODAS>::hookIns(Tuple* newT){...}
15 template<>TID DAS<ODAS>::lookup(Tuple* t){...}
16 class HDAS{}; //heap-access
17 ...
18 DAS<ODAS> _DASInstance;
```

Abbildung 2: Zugriffssystem mit GP

Das inkrementelle Erweitern von Aspekten gestaltet sich hingegen schwierig durch die Notwendigkeit des vollständigen Wissen über die Wirkungsweise der erweiterten Aspekte.

FOP. Die FOP verbessert die Erweiterbarkeit von Software durch inkrementelle Verfeinerungen (sog. "stepwise refinement") [2]. Die eine abstrakte Eigenschaft betreffenden Code-Fragmente und Klassen werden in Schichten gekapselt. Das ausführbare Produkt ist nach Batory eine Konstante, hier "`base`" (Abb. 3, Zeile 1), auf die sukzessive Erweiterungen angewendet werden [2]. Die Erweiterungen am Beispiel des Zugriffssystems sind "`ODAS`" (Zeile 6) bzw. "`HDAS`". Durch eine Zusammenstellung der Schichten kann statische Variabilität der Klassen erreicht werden [2]. Die Klasse "`DAS`" setzt sich aus den Fragmenten der Schichten "`base`" und "`ODAS`" (Zeile 6) bzw. "`HDAS`" zusammen. Durch Austausch der Layer ist eine unabhängige Variation des Algo-

```
1  //layer base
2  class DAS{
3   void modify(Tuple* old, Tuple* newT){
4    TID tid = remove(old);
5    insert(newT); }};
6  //layer ODAS
7  refines class DAS{
8   TID lookup(Tuple* t){...}
9   TID remove(Tuple* t){...}
10  TID hookIns(Tuple* newT){...}};
```

Abbildung 3: Zugriffssystem FOP

[3]Im Gegensatz zu C++ bieten Eiffel und Ada diese Funktionalität - sog. "constrained genericity" [3].

rithmus (Zeilen 3-5) und der Implementierungen seiner Schritte (Zeilen 8-10) möglich. Mixins als mögliche Umsetzung erlauben eine variable Zusammensetzung der Fragmente durch die Parametrisierung von Klassen mit ihren Superklassen. Vererbung ermöglicht diese Variation einzig durch virtuelle Methoden mit ihren inhärenten Problemen oder Code-Replikationen [4]. Die Variabilität der FOP erfordert keine virtuellen Methoden und ermöglicht eine *additive*, statische Anpassung an die konkreten Anforderungen des Produktes. Weiterhin können die statisch variablen Klassen einheitlich durch referenzierenden Code behandelt werden. Das neue Modul ist durch die Kapselung mehrerer Fragmente eine Kollaboration statt einer Klasse in OOP oder GP. Erweiterungen an bestehenden Klassen (sog. refinements) durch Member oder das Hinzufügen neuer Klassen ist durch das Anwenden weiterer Schichten möglich [2].

Die finale, verfeinerte Klasse "`DAS`" kann im mixinbasierten Ansatz durch eine zum Übersetzungszeitpunkt erzeugte Vererbungshierarchie erstellt werden oder in einem Jampack zusammengesetzt werden, d.h. einer einzelnen Klasse [2]. Die FOP ist begrenzt auf statische Variabilität und erlaubt nur eine Variante zur Laufzeit. Ist diese Einschränkung nicht möglich, z. B. für Tupel (Zeile 3), so muss auf Mittel der OOP, d.h. Vererbung und virtuelle Methoden, zurückgegriffen oder Code-Redundanz eingeführt werden. Im letzteren Fall muss jede Methode, wie z. B. "`insert`" (Zeile 5), entsprechend der verfügbaren Typen überladen werden. Code-Replikationen ergeben sich demnach aufgrund von Typ-Variationen. Weiterhin verursachen Erweiterungen vieler Klassen durch ein Code-Fragment eine Replikation des Codes [1].

	OOP	**GP**	**AOP**	**FOP**
Modularisierung	-	0	+	+
strukturelle Variabilität	0	-	++	+
sukzessive Erweiterbarkeit	0	0	-	+
Performanz	-	+	+	+

Abbildung 4: Vergleich der Techniken

3 Kombinationen

Die vorgestellten Techniken bieten Vorteile für Teilprobleme, verursachen jedoch jeweils Nachteile anderer Art. Die Hauptgegensätze der Analyse bestanden zwischen den Design-Zielen der Separierung von Belangen, der Erweiterbarkeit, Performanz und Anpassbarkeit des Systems. Eine Zusammenfassung ist in der Abbildung 4 dargestellt. Modularisierung bewertet die Fähigkeit, unabhängige System-Eigenschaften zu kapseln. Strukturelle Variabilität bewertet die variable Anpassbarkeit des Systems an Systemanforderungen und ihre Folgen auf weitere wünschenswerte Eigenschaften der Software. Sukzessive Erweiterbarkeit bewertet das wiederholte Hinzufügen von Funktionalität. Performanz drückt die Abhängigkeit der Variabilität von virtuellen Methoden aus, welche die Geschwindigkeit des Systems beeinflussen. Semantische Erweiterbarkeit und Varianz kann effizient durch AOP und FOP erreicht werden. GP erlaubt eine effiziente Behandlung von Typ-Variationen.

Um Synergien zu erzeugen, erscheinen Kombinationen der Techniken sinnvoll. Verschiedene Anstrengungen wurden in dieser Hinsicht bereits unternommen, jedoch nicht durch eine Kombination von FOP und GP [1, 7]. Um die Vorteile der Überschaubarkeit von Änderungen bei der FOP zu nutzen und ihren Nachteil bei der Verteilung identischen Codes zu überwinden, wurde in [1] eine Kombination mit AOP vorgeschlagen. Ein weiterer Ansatz um eine Vielzahl von Klassen durch ein identisches Code-Fragment zu erweitern, liegt in der Kombination von FOP oder AOP mit GP.

```
1 //layer homogen
2 template<class Subclass> class temp{
3  Subclass* foo(){...}
4 };
5 refines class Client1: public temp<Client1>{};
6 //layer homogenAllocC2
7 refines class Client2: public temp<Client2>{};
```

Abbildung 5: Symbiose von FOP und GP

In Abbildung 5 wird das zu verteilende Fragment in einem Klassen-Template gekapselt (Zeilen 2-4). Die explizite Verteilung und Anpassung des Fragments erfolgt mittels parametrisierter Vererbung durch die zu erweiternden Klassen (Zeilen 5 und 7). Die Deklaration der Erweiterungspunkte kann getrennt vom zu verteilenden Code in zusätzlichen Schichten erfolgen. Erst die Auswahl der Schichten zur Übersetzungszeit bestimmt die tatsächlichen Erweiterungspunkte des Programms. Diese Selektion kann basierend auf den speziellen Anforderungen an das System erfolgen. Später eingeführte Komponenten können unkompliziert und deklarativ das Code-Fragment übernehmen. Derartige Erweiterungen aber auch variable Änderungen der Wirkungspunkte, erfolgen additiv und verlangen keine Anpassung eines Pointcuts. Eine Erweiterung des Fragments ist durch Refinements in weiteren Schichten nicht-invasiv und modular möglich. Das Einfügen von Template-Methoden ist ebenso durch AOP möglich. Die Behandlung verschiedener Member-Typen, z.B. Indexe, kann in der AOP-Kombination abstrakte Interfaces mit virtuellen Methoden erfordern oder Code-Replikation verursachen. Weiterhin bietet eine derartige Lösung keine Robustheit des Fragments gegenüber evolutionär eingefügten Typen. Sie kann infolgedessen Code-Replikationen verursachen. Mittels der vorgeschlagenen Kombination mit parametrisierter Vererbung können Member-Typen ohne virtuelle Methoden oder Code-Redundanz durch ein erweitertes Template-Interface konfiguriert werden. Feingranulare Anpassungen des zu verteilenden Codes erfolgen somit automatisch. Als Beispiel kann eine Erweiterung der "`modInd`"-Methode betrachtet werden, welche unterschiedliche Index-Typen unterstützen soll. Im Vergleich mit AOP ist die Zuordnung der Wirkungspunkte in der Kombination von FOP und GP nicht anfällig für inkrementelle Erweiterungen, z. B. das Hinzufügen neuer Klassen. Die Wirkungsstellen des Codes der vorgestellten Lösung durch die FOP-GP-Kombination verändern sich nicht durch eine Umordnung der Schichten (so lange die relative Reihenfolge der kollaborierenden Schichten erhalten bleibt) - durch AOP können Ausführungspunkte hinzukommen oder verschwinden.

4 Zusammenfassung

In diesem Beitrag wurden die Effekte verschiedener Techniken auf Eigenschaften eines beispielhaft implementierten DBMS-Zugriffssystems untersucht. Es zeigte sich, dass die einzelnen Techniken nur Teil-Probleme wünschenswerter Software-Eigenschaften lösen. Es wurden positive Effekte durch Kombination der Techniken gezeigt. Im Speziellen wurde ein Ansatz vorgestellt, der die Flexibilität und Erweiterbarkeit gegenüber semantischen Änderungen der FOP mit der Robustheit der GP gegenüber Typ-Variabilität kombiniert. Dieser Ansatz wurde analysiert und mit anderen Vorschlägen verglichen. Es zeigte sich, dass die Vorteile der beiden Techniken FOP und GP im vorgeschlagenen Ansatz erhalten werden konnten und Nachteile durch die jeweils andere Technik ausgeglichen werden konnten.

Literatur

[1] S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proceedings of IEEE and ACM SIGSOFT 28th International Conference on Software Engineering (ICSE'06)*, May 2006. to appear.

[2] Don Batory, Jacob Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. *IEEE Transactions on Software Engineering*, 30:1278–1295, 2004.

[3] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[4] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in c++. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–323, New York, NY, USA, 1996. ACM Press.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[7] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04)*, 2004.

[8] Wolfgang Schröder-Preikschat, Daniel Lohmann, Fabian Scheler, Wasif Gilani, and Olaf Spinczyk. Static and dynamic weaving in system software with aspectc++. In *HICSS*. IEEE Computer Society, 2006.

[9] Bjarne Stroustrup. *Die C++ Programmiersprache*. Addison-Wesley Verlag, München, 4. edition, 2000.