Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Moderne Modularisierungstechniken und ihre Bedeutung für qualitativ hochwertige Software

Verfasser:

Martin Kuhlemann

25. August 2006

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Wirtsch.-Inf. Thomas Leich,
Dipl.-Inf. Sven Apel

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Kuhlemann, Martin:
Moderne Modularisierungstechniken und ihre Bedeutung für qualitativ hochwertige Software
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2006.

Danksagung

An dieser Stelle möchte meinen Dank all jenen aussprechen, die mich bei der Erstellung dieser Arbeit unterstützt haben.
Mein besonderer Dank gilt meinen Betreuern Prof. Dr. rer. nat. habil. Gunter Saake, Dipl.-Wirtsch.-Inf. Thomas Leich und Dipl.-Inf. Sven Apel für die fruchtbaren Diskussionen und wichtigen Hinweise. Ich möchte mich auch für die gute Zusammenarbeit bei der Anfertigung mehrerer gemeinsamer Publikationen bedanken.
Mein weiterer Dank gilt meiner Familie, die mich mit Kritik und Anmerkungen zur Arbeit unterstützte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Abkürzungen

ADT	Abstrakter Datentyp
AML	Aspekt-Mixin-Schicht (engl. aspectual mixin layer)
AOP	Aspektorientierte Programmierung
AOSD	Aspektorientierte Software-Entwicklung (engl. aspect-oriented software development)
API	Programmierschnittstelle (engl. application programming interface)
FIP	Problem korrelierender Merkmale (engl. feature interaction problem)
FOP	Merkmalorientierte Programmierung (engl. feature-oriented programming)
GP	Generische Programmierung
LoD	Gesetz von Demeter (engl. Law of Demeter)
MDSOC	Mehrdimensionale Aufteilung von Belangen (engl. multi-dimensional separation of concerns)
OOP	Objektorientierte Programmierung

Hervorhebungen im Text werden *kursiv* dargestellt.

Kapitel 1

Einleitung

Die Entwicklung qualitativ hochwertiger Software ist erstrebenswert [Mey97].
Qualität und ihre Bedeutung wurden auch in anderen Domänen als der Software-Entwicklung erkannt [RM95]. Qualität dient der Aufwands- und Kostenersparnis auf der Entwicklerseite. Somit erhöht Qualität die Kundenzufriedenheit durch die Verringerung von Entwicklungskosten und Entwicklungszeiten für Software mit gleichen oder ähnlichen funktionalen Anforderungen.
Die Umsetzung qualitativ hochwertiger Software liegt nicht allein in der Erfüllung der gestellten Aufgabe und der Bereitstellung von Systemfunktionen (wobei dies unbestritten die Hauptaufgabe ist), Qualität misst sich weiterhin an der Erfüllung von verschiedenen Anforderungen an das Produkt "Software". Neben der Anforderung an die Software, bestimmte funktionale Merkmale bereitzustellen, können auch Kriterien bestehen, die zur Auswahl aus einer Menge von funktional gleichwertigen Software-Produkten dienen. Derartige Anforderungen betreffen die Entwicklungskosten eines Software-Produkts, seine Performance oder seinen Ressourcenverbrauch. Die Erfüllung dieser Anforderungen an die Software kann auch durch den Entwickler beeinflusst werden.

Eine wichtige Rolle im Zusammenhang mit qualitativ hochwertiger Software spielt die Möglichkeit zur Modularisierung von Software-Produkten, d. h. die Aufteilung der Software in unabhängige Einheiten. Wirkte sich Modularität bisher meist nachteilig auf andere Anforderungen aus, so versprechen nun einige Ansätze eine Verbesserung der Modularisierbarkeit eines Software-Produkts ohne negative Seiteneffekte. Dieser Anspruch soll in der vorliegenden Arbeit überprüft und anhand von mehreren Fallbeispielen analysiert werden. Ziel der Arbeit ist es, die unterschiedlichen Einflüsse der Programmiertechniken auf die Qualität einer Software in einen umfassenden Zusammenhang zu bringen. Vor- und Nachteile der Programmiertechniken sollen bewertet und gegeneinander aufgewogen werden. Es soll festgestellt werden, welche Programmiertechnik für die Erfüllung welcher Qualitätsanforderung am besten geeignet ist. Weiterhin sind mögliche Lösungen für Probleme aufzuzeigen, die bei der Anwendung der Programmiertechniken entstehen.

Die Arbeit gliedert sich wie folgt:
Die Grundlagen der Analyse und des Vergleichs der Programmiertechniken betreffen Begriffsdefinitionen und die Vorstellung der untersuchten Techniken. Diese Grundlagen werden in Kapitel 2 vermittelt.
In Kapitel 3 erfolgt die Gegenüberstellung der Programmiertechniken zu den Kriterien, welche die Qualität von Software beschreiben.
Kapitel 4 stellt verschiedene Kombinationen der Programmiertechniken vor und untersucht ihren Einfluss auf die Erfüllbarkeit der Anforderungen an qualitativ hochwertige Software.
Die Arbeit wird in Kapitel 5 zusammengefasst. An dieser Stelle wird ebenfalls ein kurzer Ausblick auf eine mögliche zukünftige Entwicklung dieser Thematik gegeben.

Kapitel 2

Grundlagen

Dieses Kapitel stellt die Grundlagen für die Analyse und den Vergleich der Programmiertechniken vor. Das vorliegende Kapitel ist wie folgt gegliedert: In Abschnitt 2.1 werden die Anforderungen deklariert und definiert. Dem folgt eine Vorstellung der zu untersuchenden Programmiertechniken in Abschnitt 2.2.

2.1 Definition der Qualität von Software

Für die Analyse der Programmiertechniken und die Bewertung ihres Einflusses auf die Qualität der resultierenden Software ist eine Begriffsklärung wichtig. Zunächst muss der Begriff "Qualität" definiert und bewertbar gemacht werden.
Im Bereich der Qualitätssicherung ist der Begriff "Qualität" wie folgt definiert:

> Gesamtheit von Merkmalen (und Merkmalswerten) einer Einheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen [DIN98].

Im Fokus der Software-Entwicklung setzt sich Qualität aus verschiedenen Anforderungen an Programme zusammen. Deren Erfüllbarkeit ist abhängig von verschiedenen, domänenspezifischen Gegebenheiten. Die Anforderungen teilen sich nach verschiedenen Betrachtungsweisen in die des Endnutzers und jene des Entwicklers.
Kundenspezifische Produktanforderungen sind sehr vielfältig. In dieser Arbeit wird sich auf die folgenden, weithin akzeptierten Anforderungen beschränkt [Mey97, Par76, SvGB05]:

- Korrektheit,
- Geringe Entwicklungskosten,
- Später Zeitpunkt der Adaptierung,
- Hohe Performance,
- Sparsamer Umgang mit Ressourcen sowie

- Kurze Entwicklungszeit.

Die Umsetzung dieser Kundenanforderungen in Produktmerkmale erfordert aus Entwicklersicht die Erfüllung verschiedener Anforderungen durch den Quellcode. In dieser Arbeit werden die folgenden entwicklerseitigen Anforderungen betrachtet [Par76, Mey97]:

- Gute Modularisierbarkeit,
- Hohe Wiederverwendbarkeit und
- Einfache Erweiterbarkeit.

Modularität der Software verringert die Komplexität des Quellcodes und ermöglicht ein gutes Verständnis des Quellcodes [TOHS99]. Die gute Verständlichkeit wirkt gleichfalls unterstützend auf die Entwicklung fehlerfreier, korrekter Software. Dies senkt in der Folge die Kosten zur Fehlerbeseitigung.
Modularität ermöglicht eine variable Anpassung der Software an Kundenanforderungen durch das Auswählen von Komponenten. Anpassbare Software begünstigt die Wiederverwendung von Programmlogik und unterstützt so die kostengünstige Entwicklung mehrerer ähnlicher Software-Produkte sowie kurze Entwicklungszeiten für diese Produkte [Par76, SvGB05]. Entscheidungen über die Anpassung der Software sollen dabei so spät wie möglich getroffen werden.
Weiterhin ermöglicht die Modularität von Software deren variable Erweiterung. Erweiterbarkeit ermöglicht die Reduzierung des Entwicklungsaufwands durch das teilweise Wiederverwenden von Software, die nicht exakt den funktionalen Nutzeranforderungen entspricht. Erweiterbarkeit von Software trägt somit ebenfalls zur kostengünstigen und schnellen Entwicklung neuer Software bei.
Die Forderungen nach hoher Performance und geringem Ressourcenverbrauch stellen allgemeine Kundenforderungen dar, die in sehr vielen Kontexten auftreten [Mey97].

In den Abschnitten 2.1.1 bis 2.1.6 werden die an die Software und die Programmiertechniken gestellten Anforderungen erläutert. Anzumerken ist, dass die gleichzeitige optimale Umsetzung aller genannten Anforderungen nicht möglich ist, da Wechselwirkungen zwischen ihnen bestehen. Die Erfüllbarkeit der Anforderungen ist zum Teil gegenläufiger Natur. Dieser Umstand wird in Abschnitt 2.1.7 näher betrachtet.
Die Untersuchung der Forderungen nach geringen Entwicklungskosten, kurzen Entwicklungszeiten und Korrektheit erfolgt in dieser Arbeit *indirekt* durch die Untersuchung von entwicklerseitigen Anforderungen, welche sich auf der Ebene von Programmiersprachen positiv auf die Umsetzung der besagten Nutzeranforderungen auswirken. Sie werden *nicht* explizit untersucht. Ihre allgemeine, direkte Bewertung ist nicht möglich, da die Anforderungen sehr stark von der entwickelten Anwendung und dem Entwicklungsprozess abhängen.

2.1.1 Zeitpunkt der Adaptierung

Die Anpassbarkeit von Software dient dem Erfüllen unterschiedlicher Anwendungsprofile. Dazu soll aus vorhandenen Implementierungen die für den Einzelfall beste ausgewählt werden können. Das teilweise Beibehalten einer Implementierung bei ähnlichen

Nutzeranforderungen begünstigt die Wiederverwendung und somit die Entwicklung kostengünstiger Software [Par76, SvGB05].
Adaptierbarkeit ist die Möglichkeit der nutzergetriebenen Anpassung von Software an variable und sich ändernde funktionale Anforderungen und Kontexte, sog. Konfigurierung [CE00]. Den anwenderspezifischen, variablen Gegebenheiten kann durch die Angleichung von Systemmerkmalen besser entsprochen werden. Die Entwicklung teurer Spezial-Software für jedes Anwendungsprofil wird dabei vermieden. Neben der Programmsemantik betreffen variable Nutzeranforderungen den Speicherbedarf oder die Performance der Software [KCH+90].
Entwurfsentscheidungen betreffen die Zergliederung des Software-Systems sowie den Aufbau von Komponenten und die Assoziation zwischen Komponenten. Das Verändern bereits festgelegter Entwurfsentscheidungen ist kostspielig [SvGB05]. Aus diesem Grund ist Variabilität notwendig.
Die Variabilität einer frühzeitig als veränderlich deklarierten Struktur muss in sehr vielen Entwicklungsschritten und großen Teilen der Software beachtet und gepflegt werden. Das erhöht den Entwicklungsaufwand des einzelnen Software-Produkts. Dem möglichen Zeitpunkt der Adaptierung des Systems kommt demnach in variabel anpassbarer Software eine besondere Bedeutung zu. Um den erhöhten Aufwand zu vermeiden, sollten Entscheidungen der Konfigurierung einer Software so weit wie möglich an das Ende des Software-Entwicklungsprozesses verlagert werden bzw. die Behandlung der Variabilität von Programmteilen für nicht direkt betroffenen Code transparent geschehen [SvGB05].
Eine Bewertung der Anpassbarkeit einer Software kann nach dem möglichen Zeitpunkt des Bindens ihrer Merkmale erfolgen, d. h. dem möglichen Zeitpunkt ihrer Konfigurierung. Die Bindung von Systemmerkmalen ist zum Zeitpunkt der Übersetzung (engl. compile-time), zur Ladezeit (engl. load-time) oder während der Laufzeit (engl. runtime) möglich [KCH+90]:

- *Übersetzungszeitkonfigurierung* ist die Festlegung der Systemmerkmale zum Zeitpunkt der Übersetzung einer Software durch den Compiler. Diese Entscheidung beeinflusst u. a. den Aufbau des resultierenden Software-Systems und ändert sich während der gesamten Laufzeit der Software nicht [KCH+90, CE00]. Ein Beispiel hierfür ist die Auswahl der durch den Compiler übersetzten und verknüpften Quellcodedateien.

- *Ladezeitkonfigurierung* ist die Festlegung von Systemmerkmalen und -verhalten zum Zeitpunkt des Starts der Anwendungsausführung. Die hier fixierten Merkmale können von Ausführung zu Ausführung der gleichen Software variieren, bleiben jedoch während der gesamten Ausführungsdauer konstant [KCH+90, CE00]. Ladezeitkonfigurierung ist u. a. durch die Verwendung von Bibliotheken möglich, welche zum Zeitpunkt des Ladens der Software in diese eingebunden werden.

- *Laufzeitkonfigurierung* ist die dynamische Veränderung von Systemmerkmalen während der Ausführung der übersetzten Software [KCH+90]. Für Laufzeitkonfigurierbarkeit sollen keine codebezogenen Veränderungen an der Software vorgenommen werden. Um aus den Varianten während der Laufzeit auswählen zu können, muss die Logik der Varianten in der übersetzten Einheit enthalten sein.

Die Konfigurierung von Entscheidungen, die nur bis zum Übersetzungszeitpunkt getroffen werden können, bezeichnet man aufgrund ihrer Invarianz während der Lauf-

zeit als statisch, z. B. die Auswahl der übersetzten Quellcodedateien. Die Konfigurierung von Merkmalen, die erst während der Laufzeit erfolgen muss und die daher Veränderungen während der Dauer der Ausführung ermöglicht, wird als dynamische Konfigurierung bezeichnet [CE00, CE99b]. Statische Konfigurierung ermöglicht Inter-Applikationsvariabilität, d. h. die Erzeugung unterschiedlicher Software-Produkte. Dynamische Konfigurierung ermöglicht Intra-Applikationsvariabilität, d. h. variable Merkmale des einzelnen Software-Produkts während der Laufzeit [CE99b, CE00].

Ziel bei der Entwicklung von Software ist es, Entscheidungen über Systemmerkmale so spät wie möglich zu treffen, um kostspielige Änderungen aufgrund unterschiedlicher Anwendungsprofile zu vermeiden. Im Idealbild soll die *ausgeführte* Software an unterschiedliche funktionale Anforderungen angepasst werden können [SvGB05].

2.1.2 Performance

Die Ausführungsgeschwindigkeit (auch: Performance) eines Systems ist abhängig von der Dauer der Abarbeitung einer Aufgabe. Eine performante Software benötigt eine geringe Zeitdauer für die Ausführung einer Aufgabe. Hohe Performance ist eine der Kernanforderungen an moderne Software-Systeme und erfährt besondere Bedeutung in Domänen mit hohem Datenaufkommen oder komplexen Berechnungen. Weiterhin beeinflusst die Performance eines Systems die Kundenzufriedenheit stark. Sie muss somit besondere Beachtung bei der Software-Entwicklung finden.

Die Performance ist maßgeblich abhängig von der Berechnungsgeschwindigkeit der als Plattform benutzten Hardware. Leistungsstarke Prozessoren und kurze Zugriffszeiten auf den Speicher erhöhen die Performance des Gesamtsystems. Die Verwendung leistungsstarker Hardware ist jedoch mit hohen Kosten verbunden und ist besonders zu vermeiden für Systeme, die identisch in einer hohen Stückzahl gefertigt werden. Nachteilig ist weiterhin die fehlende Erweiterbarkeit oder Anpassbarkeit von bestimmten Hardware-Bestandteilen, z. B. in eingebetteten Systemen. Veränderungen wären in diesen Domänen nur durch den vollständigen und teuren Ersatz möglich. Die Performance des Systems muss also auch bei der software-seitigen Umsetzung Beachtung finden, teure performante Hardware soll vermieden werden.
Die Voraussetzungen von performanter Software sind auf verschiedenen Ebenen angesiedelt. Aus abstrakter Sicht wird die Performance maßgeblich bestimmt von der verwendeten Implementierung, d. h. vom Algorithmus, den der Entwickler zur Lösung einer Aufgabe wählt. So gibt es meist unterschiedliche Wege, die gestellten funktionalen Anforderungen zu erfüllen. Bezüglich der Nutzeranforderungen muss jeweils die optimale Implementierung ausgewählt werden.[1] Die Entscheidungen darüber sind sehr anwendungsspezifisch und sollen nicht in dieser Arbeit betrachtet werden.
Die Performance einer Software kann auch erhöht werden, indem Programmschritte vernachlässigt werden und nicht abgearbeitet werden müssen. Auf der Ebene von Programmiersprachen, d. h. letztlich auf Ebene der durch den Prozessor abzuarbeitenden Instruktionen, kann die Performance durch verschiedene Compiler-Optimierungen,

[1] z. B. unterschiedliche Sortieralgorithmen oder Join-Strategien für Datenbanktabellen.

wie "Inlining"[2], Ergebnisvorberechnung, "Scheduling der Instruktionen"[3] oder "Loop-Unrolling"[4], verbessert werden [CE00].

2.1.3 Modularisierbarkeit

Die Modularisierbarkeit einer Software ist aus Entwicklersicht eine der wichtigsten Voraussetzungen, um Forderungen nach hoher Wiederverwendbarkeit und guter Erweiterbarkeit des Codes nachzukommen. Diese unterstützen ihrerseits die schnelle und kostengünstige Entwicklung neuer Software. Die Umsetzbarkeit von Modularität hat demnach indirekt großen Einfluss auf die Erfüllung von Nutzeranforderungen.
Modularität ermöglicht, eine Veränderung des Systems mit minimalen Aufwand umzusetzen, indem in einem atomaren Schritt *ein* lose gekoppeltes Modul ausgetauscht wird [OT00]. Die inkonsistente Umsetzung eines Belangs[5] in einem System aufgrund replizierter und gegensätzlich konfigurierter Coderepräsentationen ist nicht mehr möglich. Die Komplexität des Programmcodes sinkt so mit steigendem Grad der Modularität des Codes unabhängiger Belange. Weiterhin erhöht sich die Wiederverwendbarkeit der nicht an einen Kontext gebundenen Module [TOHS99]. Das Merkmal der Modularität erhöht die Qualität des Quellcodes.
Codereplikation widerspricht somit der Forderung nach Modularität.

Für die Beurteilung der Qualität von Quellcode wurden bereits diverse Software-Produktmetriken[6] definiert, welche versuchen, die Erfüllung der entwicklerseitigen Anforderungen zu bewerten. Beispiele hierfür sind Anzahl der Codezeilen (LOC) oder Anzahl der kopierten Zeilen (LCC) [RDL04, Pin05]. Diverse weitere Software-Maße wurden für spezielle Programmierparadigmen vorgeschlagen, beispielhaft genannt seien hier Coupling-between-object-classes (CBO) oder Lack-of-cohesion-in-methods (LCOM) [CK94].
Des Weiteren wurden Entwurfsregeln definiert, welche die Qualität des Codes erhöhen sollen. Als Beispiel sei das Gesetz von Demeter (engl. Law of Demeter; LoD) genannt, das die Modularität durch lose Kopplung der Komponenten zu erhöhen sucht. Eine Weiterführung findet es im Gesetz von Demeter für Belange (engl. Law of Demeter for Concerns) [LH89, Lie04].
Sind diese Metriken und Entwurfsregeln teilweise nicht undifferenziert auf andere Programmiertechniken übertragbar, so lässt sich doch erkennen, dass die Module einer Software möglichst gekapselt und lose gekoppelt sowie semantisch vollständig sein sollten [LLO03]. Diese Definition der Modularität lässt sich auch auf die zu untersuchenden Programmierparadigmen wie Merkmalorientierte Programmierung übertragen.
Die Möglichkeit, semantisch zusammenhängende Software-Einheiten syntaktisch in einem Modul zu kapseln, wird als Merkmalkohäsion (engl. feature cohesion) bezeichnet. Die Zusammenstellung kann im Folgenden als einzelne, semantisch vollständige Einheit betrachtet werden [LHBC05]. Eine Zusammenstellung von Modulen kann ihrerseits

[2] Methodenrümpfe werden an die Stellen ihres Aufrufs kopiert und so Sprunganweisungen vermieden.
[3] Optimierung der Reihenfolge von Maschinenanweisungen zur Verkürzung von Speicherzugriffszeiten.
[4] Vervielfältigen von Schleifenrümpfen zur Vermeidung von Sprunganweisungen am Ende der Schleife.
[5] Ein Belang ist eine Eigenschaft einer Software [Par72].
[6] Produktmetriken messen das Software-Produkt, z. B. die Komplexität des Software-Designs. Prozessmetriken messen den Prozess der Software-Entwicklung, z. B. die Entwicklungsdauer [Mil88].

ein Modul eines abstrakteren, zusammengesetzten Moduls darstellen, sog. Umschließung (engl. closure) [LHBC05]. Aus dieser Eigenschaft folgt auch die Möglichkeit zur Zergliederung, sog. Dekomposition, einer Software.
Die Gruppierung und Zusammenstellung von Modulen bzw. ihre Dekomposition ermöglichen Abstraktionen, welche die allgemeine Verständlichkeit des Codes fördern [LLO03, Par72]. Der Programmierer kann die Software auf Abstraktionsstufen manipulieren, ohne dass die Konfrontation mit der gesamten Komplexität der Software notwendig wäre. Die Dekomposition, d. h. die Aufteilung von Software in Einheiten, die Semantik kapseln, unterstützt so die Entwicklung korrekter Software. Die Kosten zur Fehlerbeseitigung werden minimiert. Dekomposition und die damit verbundene modulare semantische Adaptierbarkeit hat eine besondere Bedeutung für die Entwicklung wiederverwendbarer Software. Die Möglichkeit der Dekomposition einer Software wird daher nicht im Zusammenhang mit Modularisierbarkeit, sondern mit Wiederverwendbarkeit behandelt.
Voraussetzungen für eine sinnvolle Modularisierung sind eine Analyse und Separation der durch die Software bereitgestellten Funktionen, ein sog. Trennen der Belange (engl. separation of concerns) [Par72]. Jedes Modul sollte folglich nur den Code eines Belanges enthalten. Schlecht modularisierte Systeme sind durch das verteilte Auftreten von Code eines Belangs über den Gesamtquellcode, sog. Zersplitterung von Code (engl. scattering of code), und durch die Kapselung von Code unterschiedlicher Belange, sog. Verwirrung von Code (engl. tangling of code), gekennzeichnet [LLO03, TOHS99, KLM+97]. Die Belange, deren Implementierung das verteilte Auftreten und die Verwirrung von Code verursachen, bezeichnet man als querschneidende Belange (engl. crosscutting concerns) [KLM+97, Ost03, LLO03]. Die Modularisierung von Quellcode, der einen Belang beeinflusst, erlaubt seine zusammenhängende Speicherung in einer unabhängigen Programmeinheit.[7]
Die Integration von Code querschneidender Belange in ein Modul erzeugt eine enge Kopplung der querschneidenden Belange an den Belang des Moduls. Das erzeugt Codereplikation, wenn einer der eng gekoppelten Belange unabhängig vom anderen variiert.
Aufgrund der verschiedenen Möglichkeiten der Separation des Quellcodes in Module erzeugt die Modularisierung des einen Belangs in einigen Programmiertechniken das Querschneiden eines anderen [BLS03]. Dieses Problem wird als Tyrannei der dominanten Aufteilung (engl. tyranny of the dominant decomposition) bezeichnet [TOHS99].
Software kann meist unter verschiedenen Gesichtspunkten, d. h. in unterschiedlichen Dimensionen, betrachtet werden. Veränderungen einer anderen als der aufteilenden Dimension verursachen folglich das Problem des verteilten und verwirrten Auftretens von Code. Die Aufteilung der Software nach Belangen entlang mehrerer Dimensionen (engl. multi dimensional separation of concerns; MDSOC) versucht, dieses Problem bei der adäquaten Modularisierung unabhängiger Belange zu lösen [TOHS99, OT00]. Hierbei wird die Software als möglicherweise überlappende Menge von Bestandteilen verschiedener Belange gesehen. Diese können nicht unabhängig modularisiert werden, ohne sie entlang verschiedener Dimensionen einzuordnen. Ziel des MDSOC ist es, allen Möglichkeiten der Modularisierung gleichsam Bedeutung beizumessen [TOHS99, OT00].

Modularisierung ermöglicht und erfordert die Definition von Schnittstellen zwischen

[7] Basierend auf Modulen kann Crosscutting als überschneidender Einfluss mehrerer Module der Programmiersprache auf Elemente des ausführbaren Programms beschrieben werden [MK03].

den einzelnen, isolierten Programmteilen.[8] Die Schnittstellen geben im Folgenden die Wirkungsgrenzen für Änderungen an der Implementierung von Modulen vor. Explizite Schnittstellen erlauben die Veränderung von Modulen nur unter Beachtung der Implementierung des manipulierten Moduls, seiner Schnittstelle und der Schnittstellen der im Modul referenzierten Module, sog. modulare Schlussfolgerung (engl. modular reasoning) [KM05]. Die Schnittstelle des veränderten Moduls sollte bei den Veränderungen konstant bleiben, um Seiteneffekte auf andere Module zu vermeiden. So kann ein Software-Projekt effizient und parallel durch ein Entwickler-Team bearbeitet werden [LLO03, Par72, Mey97]. Das parallele Entwickeln von Software ermöglicht eine Verringerung der Entwicklungsdauer.

```
1  BL:class LinkedList{
2  BL: Node* head;
3  BL: LinkedList(AbstractElement* elem){
4  BL:   head = new Node(elem->clone());
5  TR:   cout<<"new␣List:␣"<<head->getElement()->toString();}
6  BL: ~LinkedList(){
7  SY:   Scheduler::lock(this);
8  BL:   .../*delete all Elements*/
9  SY:   Scheduler::unlock(this);  }};
10 SY:class Scheduler{...};
```

Abbildung 2.1: Beispiel von querschneidenden Belangen in einem Modul.

Ein Beispiel für das verwirrte und verteilte Auftreten von Code querschneidender Belange ist in Abb. 2.1 dargestellt. Das Modul in Form der Klasse "`LinkedList`" (Zeile 1) implementiert den abstrakten Datentyp[9] (ADT) einer Liste[10]. In der dargestellten Implementierung enthält die Klasse "`LinkedList`" zusätzlich zur Kernlogik, sog. Business Logic (BL), den Code fremder Belange, wie die Ausgabe der ausgeführten Schritte, sog. Tracing (TR), oder die Fehlervermeidung bei parallelen Prozessen, sog. Synchronisation (SY).[11]
Durch die enge Kopplung der Belange und die dennoch gewünschte, unabhängige Wiederverwendung und Variation von Software erfordert die Veränderung eines Belangs die Replikation des Codes der anderen verwirrten Belange. Demnach erfordert eine Modulvariante ohne den Belang TR im Beispiel eine neue Klasse. Hierhin wird der Code der BL und SY repliziert. Die Codefragmente der querschneidenden Belange (SY, TR) sind zudem über mehrere unabhängige Module verteilt und somit schlecht zu pflegen. Nachfolgende Änderungen dieser Belange müssen in mehreren Software-Einheiten getätigt werden und bieten die Gefahr von Inkonsistenzen.

[8] Die Schnittstelle eines Objekts wird durch sein Protokoll bestimmt, d. h. die Menge von Nachrichten, die fehlerlos an das Objekt gerichtet werden kann [JF88].

[9] Menge von Funktionen, die Zugriff auf eine Menge von Elementen erlauben [Mey97]

[10] Menge von Elementen, die in einer linearen Datenstruktur verwaltet werden.

[11] Die Zuordnung der Codezeilen zu den Belangen ist durch die Annotation des Belanges am jeweiligen Zeilenanfang dargestellt.

2.1.4 Wiederverwendbarkeit

Wiederverwendung bedeutet die Benutzung eines Software-Fragments in mehr als einem Kontext [AC96]. Die Funktionalität des Fragments muss nicht für jeden speziellen Kontext neu entwickelt werden. Wiederverwendung unterstützt die Umsetzbarkeit der Forderung nach geringen Entwicklungskosten und kurzen Entwicklungsdauern für die Entwicklung von Software mit ähnlichen Fähigkeiten [Par76]. Der wiederholte Aufwand von Planung, Entwicklung und Testen kann vermieden werden. Die Korrektheit der Software kann durch die bei der wiederholten Entwicklung vermiedenen Fehler erhöht werden.

Die Grundlagen der Wiederverwendung wurden von Parnas im Konzept der Produktfamilien gelegt [Par76, Par78]. Das Ziel dieses Ansatzes ist es, eine Software in Module aufzuteilen, welche in mehreren "Familienmitgliedern", d. h. in mehreren Konfigurationen der Software-Familie, wiederverwendet werden können. Durch variables Hinzufügen der Module zu einer Basis sollen einzelne Familienmitglieder erzeugt werden. Familienmitglieder ihrerseits können die Basis für weitere Produktfamilien bilden, um selbst wiederverwendet zu werden [Par76].

Die Software-Fragmente sollen eine geringe Kontextbindung aufweisen, um unverändert in vielen Kontexten wiederverwendbar zu sein. Diese Anforderung wird durch feingranulare Komponenten erfüllt, da der wenige Code einer Komponente potentiell weniger Annahmen über den Kontext enthält, z. B. Klassennamen oder Schnittstellen im Kontext vorliegender Module. Die undifferenzierte Wiederverwendung komplexer Module kann Konflikte mit dem Kontext infolge der vielen inhärenten Entwurfsentscheidungen des Moduls verursachen, z. B. Mehrdeutigkeiten bei Modulbezeichnungen oder nicht vorhandene referenzierte Module.
Eine feingranulare Einheit besitzt einen geringen Wert und erzeugt mehr Verwaltungsaufwand im Verhältnis zu ihrem Nutzen als grobe Komponenten [Big98]. Die Verwaltung und Reorganisation einer Vielzahl kleiner Komponenten wird in Hinblick auf den Aufwand bei der Wiederverwendung als nicht wünschenswert angesehen. Die Grenze der Effizienz der Modularisierung ist anwendungsspezifisch und muss entsprechend den Gegebenheiten für jede Entwicklung neu festgelegt werden. Die Bestimmung der optimalen Aufteilung bezüglich der Effizienz stellt ein Optimierungsproblem dar.
Ziel der Wiederverwendung ist es, eine geringe Anzahl von Komponenten zu entwickeln, die einen hohen Wert besitzen und in vielen Szenarien eingesetzt werden können. Ausdruck findet dieser Widerspruch nach Biggerstaff in einer zweidimensionalen Darstellung, der sog. Skalierungsfläche (engl. scaling plane) [Big98]. Die horizontale Dimension der Skalierungsfläche stellt eine Bewertung der Kontextabhängigkeit dar, während die vertikale Dimension den Wert[12] eines Moduls bewertet. Biggerstaff argumentiert, dass Module entweder einen hohen Wert besitzen oder vielseitig einsetzbar sind. Starre Module führen nach Meyer zum äquivalenten Wiederverwendungs-Wiedererstellungs-Dilemma (engl. reuse-redo dilemma), d. h. die Komponente kann mit *allen* ihren inhärenten Merkmalen wiederverwendet werden oder muss vollständig neu codiert werden. Als Lösung werden *adaptierbare* Komponenten propagiert [Mey97].

Beispielhaft für die Wiederverwendung von Software sind Bibliotheken zu nennen, da sie durch verschiedene Applikationen benutzt werden können [Weg90, PH00]. Die

[12] Der Wert wird hier gleichgesetzt mit der Größe des Moduls [Big98].

Funktionalität von Bibliotheken ist meist sehr grundlegend und vielseitig einsetzbar. Die Standard-Template-Bibliothek (engl. Standard Template Library; STL) von C++ bietet dem Programmierer beispielsweise Standardimplementierungen für häufig gebrauchte ADT, z. B. Listen, und Navigationsstrategien auf den ADT [SL94].
Ein weiteres Beispiel für Wiederverwendung ist das Prinzip, Frameworks zu spezialisieren. Frameworks geben durch abstrakte Modulbeschreibungen, sog. abstrakte Klassen, die Architektur und den Programmablauf einer Software vor [JF88, LK94, Joh97]. Die abstrakten Klassen, d. h. hauptsächlich die bewährte Architektur der Software, werden wiederverwendet [LK94, Joh97].
Im Gegensatz zu Bibliotheken geben Frameworks bzw. deren Gestalter die Struktur und den Ablauf der Schritte einer Software vor und nicht der Programmierer. Der Programmierer implementiert lediglich die einzelnen Schritte des vorgegebenen Algorithmus, sog. Umkehrung der Kontrolle (engl. inversion of control) [JF88, Joh97]. Die in den einzelnen Produkten unterschiedlich implementierten Programmschritte grenzen im Folgenden die Varianten der Software von einander ab.
Entwurfsmuster (bewährte Herangehensweisen zum Aufbau und zur Verknüpfung von Modulen) bieten die Möglichkeit, die Funktionalität von Frameworks nachzubilden, z. B. durch das Entwurfsmuster "Template-Methode" [CE00, Joh97, GHJV95]. Weiterhin enthalten die Architekturbeschreibungen von Frameworks viele Entwurfsmuster, d. h. ein Framework besteht i. A. aus einer höheren Anzahl von Klassen als ein Entwurfsmuster [Joh97]. Frameworks tendieren daher dazu, anwendungsspezifisch zu sein [JF88]. Entwurfsmuster geben hingegen meist nur wenige in Modulen zu implementierende Konzepte vor [GHJV95].
Für die Evaluierung von Programmiertechniken wird die Verwendung allgemein bekannter, kontextunabhängiger Objektkompositionen als sinnvoll erachtet. In dieser Arbeit werden daher Entwurfsmuster verstärkt betrachtet.

2.1.5 Erweiterbarkeit

In vielen Software-Projekten reicht eine einzige, initiale Entwurfsphase nicht aus, die Software-Entwicklung wird zum Software-Lebens*zyklus* [Dum01, Mey97]. Die Gründe hierfür liegen in der Erfahrung, dass Nutzeranforderungen sich während und nach der Software-Entwicklung ändern. Diese Änderungen können für schlecht erweiterbare Software eine vollständige Neuentwicklung der Software erfordern und sich so negativ auf die Entwicklungskosten und Entwicklungsdauer auswirken.

Entspricht eine vorhandene Software-Lösung nicht exakt den Anforderungen eines Anwenders, so soll es durch Erweiterung möglich sein, den Großteil der benötigten, bereits implementierten Funktionalität wiederzuverwenden. Die Differenz zwischen den Anforderungen des Anwenders und den Gegebenheiten der Software soll bei guter Erweiterbarkeit ohne großen Aufwand hinzugefügt werden können.
Erweiterbarkeit ist im Zusammenhang der Software-Entwicklung die Möglichkeit, vorhandene Komponenten und Codefragmente um zusätzliche Funktionalität zu bereichern. Das kann z. B. durch das Hinzufügen von Methoden, welche die zusätzliche Funktionalität bereitstellen, oder durch das Hinzufügen von Modulen, die einen zusätzlichen Belang kapseln, erfolgen.

Variable Erweiterbarkeit kann nur durch additive (hinzufügende), modulare Erweiterungen erreicht werden [TOHS99]. Im Gegensatz zu invasiven Erweiterungen, d. h. der Veränderung von bestehendem Quelltext, bleibt so der gesamte wiederverwendete Code unberührt.[13] Invasive Änderungen sind notwendig, wenn die Änderungen und Erweiterungen, z. B. durch den Entwurf der Software, nicht additiv hinzugefügt werden können. Invasive Veränderungen des Quelltextes verursachen Codereplikationen, wenn beide Varianten des Quelltextes für die Konfigurierung zur Verfügung stehen sollen. Dies widerspricht der Forderung nach Modularität der Software und sollte vermieden werden (vgl. Abschnitt 2.1.3). Nichtinvasive Veränderungen bieten den Vorteil der höheren Flexibilität [Ern00].

Das Konzept der Programmfamilien (vgl. Abschnitt 2.1.4) ermöglicht die sukzessive Erweiterung der Familienmitglieder durch das Hinzufügen neuer Komponenten [Par78]. Gen-Voca und seine Erweiterung AHEAD sind formale Spezifikationen dieses Konzepts der sog. schrittweisen Erweiterung (engl. stepwise refinement) [BO92, BSR04]. Zu erweiternde Programmbausteine werden hier als Konstanten (engl. constants) und ihre Erweiterungen als Verfeinerungen (engl. refinements) bezeichnet [BSR04]. Verfeinerungen werden als Funktionen gesehen, welche die Konstanten, auf die sie angewendet werden, um ein abstraktes Merkmal *erweitern*.

2.1.6 Ressourcenverbrauch

Mit den Ressourcen des Arbeits- und Festplattenspeichers muss sparsam umgegangen werden. Gegenwärtig vollzieht sich im Bereich der Personal-Computer und Server ein rapider Preisverfall für Speicherbaugruppen. Im Bereich der eingebetteten Systeme stellt Speicher noch immer eine begrenzte und damit teure Ressource dar [AB04, SXG+04].
Der Speicherbedarf der ausführbaren, übersetzten Einheit, ohne die von ihr manipulierten Daten, hängt maßgeblich von der Komplexität der auszuführenden Programmlogik ab. Um speichereffiziente Software zu entwickeln, sollte diese speziell an den Anwendungszweck angepasst entwickelt werden. Nicht benötigte Programmlogik sollte nicht in der übersetzten Einheit enthalten sein, sowohl die Anzahl der für den Algorithmus abzuarbeitenden Befehle als auch die Logik zur Verwaltung von Modulen im übersetzten Programmcode müssen dafür minimiert werden [CE00].

2.1.7 Wechselwirkungen zwischen den Anforderungen

Die in den Abschnitten 2.1.1 bis 2.1.6 vorgestellten Anforderungen an qualitativ hochwertige Software können nicht gleichzeitig erfüllt werden. Im folgenden Abschnitt werden die Konflikte und gegenseitigen Abhängigkeiten zwischen ihnen betrachtet.

Performance und Ressourcenverbrauch – Entwicklungskosten. Ein elementarer Widerspruch liegt in den Anforderungen, zum einen kostengünstige Software zu entwickeln und zum anderen für jedes Anforderungsprofil eine ressourcenoptimale und

[13] Die Einwirkung eines Moduls auf ein anderes Modul ist invasiv, wenn sie im Zielmodul z. B. durch Methodenaufrufe sichtbar ist. Das Zielmodul hängt importierend von einem anderen Modul ab [Ern00].

performante Lösung zu präsentieren. Kostengünstige Software kann durch Standard-Software erreicht werden, die vielen möglichen Anwendungsprofilen genügt. Diese ist für den einzelnen Kunden nicht optimal bezüglich des Ressourcenbedarfs und der Performance.
Spezialentwicklungen im anderen Extrem werden optimal an die Anforderungen des einzelnen Kunden angepasst. Die beschränkte Absatzfähigkeit der speziell auf einen Abnehmer zugeschnittenen Software erhöht den Anteil der Beteiligung des einzelnen Kunden an den Entwicklungskosten. Das Produkt wird teurer.

Ein Mittelweg wird mit konfigurierbarer Software gegangen. Durch die Konfigurierung der Software, d. h. das Auswählen von Systemmerkmalen, kann die Software besser an Anforderungsprofile angepasst werden. Das erlaubt es, mehr Kundenanforderungen zu erfüllen. Die Software wird damit für den einzelnen Kunden kostengünstiger als eine Spezialentwicklung. Konfigurierbare Software kann dabei eine effiziente Ressourcennutzung und hohe Performance ermöglichen. Im Vergleich zu Spezial-Software erhöhen sich die Entwicklungskosten für die einzelne Produktvariante durch den Entwurf und die Implementierung von Variabilität. Durch die Vermeidung von stetigen Neuentwicklungen und die Möglichkeit der Wiederverwendung fallen diese jedoch insgesamt und für den einzelnen Kunden deutlich geringer aus als mehrere sequentiell entwickelte Speziallösungen [Par76].

Adaptierungszeitpunkt – Ressourcenverbrauch. Die Verschiebung der Konfigurationsentscheidungen an das Ende des Software-Entwicklungsprozesses spart Entwicklungsaufwand [SvGB05]. Eine Konfigurierung während der Laufzeit benötigt die Anweisungen sämtlicher Varianten der Software in der ausführbaren Einheit. Das erhöht den Speicherbedarf für Konfigurationen, die nur eine Variante verwenden, unnötig.
Für die dynamische Anpassbarkeit des Programmverhaltens wird durch den Compiler Logik erzeugt, die den Speicherbedarf der Software weiter erhöht, z. B. Zähler oder Funktionszeigertabellen [Str00, DH96, LST+06].
Virtuelle Funktionen bieten eine Möglichkeit, modulare, während der Laufzeit adaptierbare Software zu entwickeln [Str00]. Die Implementierung einer virtuellen Funktion wird dafür dynamisch ausgewählt. Für Klassen, die virtuelle Funktionen enthalten, müssen Funktionszeigertabellen gepflegt und gespeichert werden, welche der Auswahl des konkreten Verhaltens dienen [Str00, DH96, LST+06]. Weiterhin wird für jedes Objekt einer solchen Klasse ein Zeiger auf die entsprechende Tabelle gespeichert [Str00, DH96]. Der Mehrbedarf für zu speichernde Befehle, sog. direkte Kosten, wurde im Zusammenhang mit virtuellen Funktionen mit durchschnittlich 3,7% gemessen bzw. mit durchschnittlich 13,7% in Systemen, deren sämtliche Methoden als virtuell deklariert waren.[14] Virtuelle Funktionen können Speicherkosten auch dadurch verursachen, dass Compiler-Optimierungen nicht angewendet werden können, sog. indirekte Kosten virtueller Funktionen [DH96].

Adaptierungszeitpunkt – Performance. Laufzeitkonfigurierbarkeit ist erstrebenswert, da sie die Veränderung von Konfigurationsentscheidungen spät, u. U. nach Abschluss des Entwicklungsprozesses, ermöglicht. Die verschiedenen Implementierungen müssen dafür dynamisch während der Laufzeit ausgewählt werden, sog. Binden einer

[14] Dieser Fall wird in Abschnitt 3.6.1 betrachtet.

Implementierung.
Programmierter Code zur Auswahl einer Implementierung einer Methode ist querschneidend zum Belang des ihn umgebenden Codes, da er den Belang eines anderen Moduls betrifft. Er sollte vermieden werden. Für die Auswahl der Implementierung einer virtuellen Funktion während der Laufzeit ist ähnliche zusätzliche Verwaltungslogik wie der programmierte Code dennoch notwendig und wird durch den Compiler erzeugt.
Driesen und Hölzle zeigen, dass virtuelle Funktionen als Mechanismus der variablen Bindung von Implementierungen während der Laufzeit Verschlechterungen (hier: Kosten) in der Performance von im Mittel 5,2% verursachen [DH96]. Dieser Wert ist bei ihren Messungen bis maximal 29% gestiegen. Waren alle Methoden dynamisch gebunden, lagen die Messwerte der Verschlechterungen bei durchschnittlich 13,7% bzw. maximal 47%. Diese Kosten werden als direkte Kosten virtueller Funktionen bezeichnet [DH96]. Weiterhin verursacht späte Methodenbindung, dass performance-optimierende Codetransformationen nicht angewendet werden können, sog. indirekte Kosten [DH96].
Statische Methodenbindung, d. h. die Auswahl der Implementierung zur Übersetzungszeit, ist performanter als dynamische Bindung, reduziert jedoch die Adaptierbarkeit [CE00].

Performance – Ressourcenverbrauch und Modularität. Manuelle oder automatische Optimierungen wie Inlining oder Loop-Unrolling erzeugen schnellere Applikationen. Diese Optimierungen können speicherintensivere Software durch das Replizieren von Programmlogik verursachen [CE00]. Diese replizierte Programmlogik widerspricht zudem der Forderung nach Modularität.
Die Verbesserung der Performance erfordert die Vermeidung nicht notwendiger Berechnungsschritte oder Zugriffszeiten. Das kann durch das Speichern bereits berechneter Ergebnisse oder empfangener Daten umgesetzt werden. Spezielle Speicherstrukturen, welche als Cache arbeiten, ermöglichen die Bereitstellung dieser Ergebnisse ohne erneute Anfragestellung an die Peripherie oder Neuberechnung eines Ergebnisses. Dieser Geschwindigkeitsvorteil wird mit erhöhtem Speicherverbrauch infolge der zusätzlich zwischengespeicherten Daten erkauft.

Modularität – Performance und Ressourcenverbrauch. Speicherverbrauch und Performance können durch eine Modularisierung von diesbezüglich relevantem Programmcode beeinflusst werden. Durch die Auswahl von Modulen kann folglich Einfluss auf die Erfüllung der Kriterien genommen werden, z. B. durch die Auswahl einer performanten Implementierung.
Die Kapselung jeder Entscheidung in einem Modul, d. h. mindestens in einer Methode, erzeugt selbst Performanceeinbußen durch die Notwendigkeit vieler Sprunganweisungen [Big98, Par72].

Wiederverwendbarkeit – Modularität. Variabel anpassbare Software beruht bei Berücksichtigung entwicklerseitiger Anforderungen wie Modularisierbarkeit und Wiederverwendbarkeit auf der Zusammenstellung unabhängiger Komponenten. Ein Modul ist wiederverwendbar, wenn es keine Entwurfsentscheidungen enthält, welche zum wiederverwendenden Kontext in Konflikt stehen. Dies ist nur gegeben, wenn die wiederzuverwendende Einheit wenige Entwurfsentscheidungen beinhaltet, d. h. wenn das Modul

einfach aufgebaut ist. Je elementarer die modulare wiederverwendbare Einheit ist, umso weniger Konflikte kann sie in einem neuen Kontext erzeugen.
Diese minimale Aufteilung der Software steht im Widerspruch zur semantischen Vollständigkeit eines Moduls. Da die semantisch vollständige Lösung sich fortan auf mehrere Software-Einheiten aufteilt, ist eine Software-Einheit nicht mehr kapselnd für dieses Merkmal. Ergebnis ist eine wenig wünschenswerte Konfiguration auf Programmiersprachenebene, z. B. die Auswahl einzelner Funktionen.
Das Herauslösen von Objektverhalten aus einem konzeptionellen Objekt in ein assoziiertes Objekt, sog. Strategie [GHJV95], kann als Beispiel gesehen werden: Die Auslagerung des Verhaltens bricht hier die Forderung nach semantischer Vollständigkeit des Strategiemoduls. Das Strategiemodul ist nun eng an den manipulierten Kontext gebunden, der Wert der einzelnen wiederverwendbaren Einheit sinkt.

2.2 Untersuchte Programmiertechniken

In diesem Abschnitt erfolgt die Vorstellung der untersuchten Programmiertechniken. Die Techniken wurden ausgewählt, da sie die Umsetzung der Anforderungen durch unterschiedliche Methodiken und Herangehensweisen zum Inhalt haben.

Die Objektorientierte Programmierung ist eine anerkannte und verbreitete Programmiertechnik, welche die beschriebenen Qualitätsanforderungen durch bestimmte Aufbau- und Verknüpfungsvorgaben von Modulen erfüllen soll, sog. Entwurfsmuster (engl. design patterns).
Generische Programmierung fügt zusätzliche Schnittstellen zu Modulen hinzu, um eine variable Kombinierbarkeit der Module zu ermöglichen.
Aspektorientierte Programmierung hat zum Ziel, querschneidende Belange bei der Entwicklung modularer Software zu vermeiden.
Merkmalorientierte Programmierung wurde für die Analyse ausgewählt, da sie eine flexible Zusammenstellung von Modulen verspricht, welche in direkter Weise Nutzeranforderungen entsprechen.
Die vorliegende Arbeit betrachtet die untersuchten Programmiertechniken als weiterführende Konzepte des Klassenprinzips der Objektorientierung. Die Techniken sind konzeptionell ebenfalls auf Strukturierte Programmierung anwendbar. Die Kapselung jeder Methode in einer eigenen Klasse bietet für das Klassenprinzip jedoch die gleiche Aussagekraft.

Der folgende Abschnitt gliedert sich wie folgt:
Abschnitt 2.2.1 betrachtet die implementierungsnahen Mechanismen der Vererbung, der Delegation und Konsultation, der frühen und späten Bindung sowie Mixins. Die auf diesen Mechanismen aufbauenden und zu untersuchenden Programmiertechniken – Objektorientierte Programmierung, Generische Programmierung, Aspektorientierte Programmierung und Merkmalorientierte Programmierung – werden in den Abschnitten 2.2.2 bis 2.2.5 vorgestellt.

2.2.1 Implementierungsnahe Mechanismen

Vererbung

Vererbung stellt einen Mechanismus für die Wiederverwendung und inkrementelle Erweiterung von gekapselten Software-Modulen, sog. Klassen, dar. Sie unterstützt somit die Erfüllung der Forderungen nach preiswerter und korrekter Software [Bal99, Zam98, Weg90]. Vererbung ermöglicht das Übertragen von Status- und Funktionsbeschreibungen, sogenannte Member, einer vererbenden Klasse (auch: Oberklasse, Basisklasse, Elternklasse, Superklasse) auf die Beschreibung erbender Klassen (auch: abgeleitete Klasse, Unterklasse, Kindklasse, Subklasse) [Sny86, Str00, PH00].
Die nach der Beschreibung der erbenden Klasse erstellten Individuen, sog. Instanzen oder Objekte der Klasse, besitzen die gleichen Methoden und Member-Variablen wie die Instanzen der vererbenden Klasse [Mey97, AC96, WZ88]. Das Einfügen von Methoden in die Unterklasse *erweitert* ihre Klassenbeschreibung relativ zur Oberklasse.
Die Vererbung ist Ausdruck einer Teilmengenbeziehung: Die Menge der Instanzen der erbenden Klasse ist Teilmenge der Instanzen der vererbenden Klasse. Die Instanzen der als Unterklasse deklarierten Einheit können fortan anstelle von Instanzen der Oberklassen verwendet werden, sog. Subtyp-Polymorphismus[15] (engl. subtype polymorphism; auch: inclusion polymorphism, runtime-polymorphism) [Str00, Bal99, PH00, CW85]. Subtyp-Polymorphismus ist begründet durch das Vorhandensein der Schnittstelle der Oberklasse in der Schnittstelle der Unterklasse. Durch diese Interpretation werden Konzepte wie Generalisierung und Spezialisierung ermöglicht, welche das Verständnis des Quellcodes durch Abstraktion erhöhen [Zam98, Bal99, Weg90].
Vererbungsbeziehungen müssen zum Übersetzungszeitpunkt feststehen. Es gibt i. A. keine Beschränkung, wie viele Klassen als Unterklassen für eine Oberklasse gelten. Nach der Anzahl der möglichen direkten Oberklassen einer Unterklasse wird unterschieden in Einfach- und Mehrfachvererbung [Str00, SDNB03].
"Überschreiben" von Methoden ermöglicht das additive Redefinieren von ererbten Methoden für die Unterklasse und wird unter dem Gesichtspunkt der Delegation näher betrachtet (Abschnitt 2.2.1: Delegation und Konsultation).

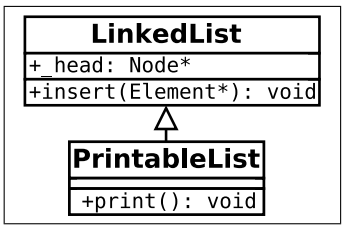

Abbildung 2.2: UML-Diagramm der Vererbung zwischen Klassen.

```
1  PrintableList* pList = new PrintableList();
2  // Methode ''insert'' wurde ererbt
3  pList->insert(new Element());
4
5  // stat. Typ: ''LinkedList'',
6  // dyn. Typ: ''PrintableList''
7  LinkedList* list = new PrintableList();
8
9  // Fehler, für LinkedList nicht definiert
10 list->print();
```

Abbildung 2.3: Beispiel von Subtyp-Polymorphie durch Vererbung.

[15] Ein Typ ist eine Menge von Werten, die bestimmte Anforderungen erfüllen, und verfügt über festgelegte Operationen. Ein Typ basiert in OOP auf einer Klasse [CW85, Boo97, Mey97]. Subtyp-Polymorphie ist die Manipulation von Objekten eines Subtyps anstelle von Objekten eines Supertyps [CW85].

Abbildung 2.2 zeigt ein Klassendiagramm für ein Beispiel einfacher Vererbung. Die Unterklasse "`PrintableList`" erbt die Member-Variablen (hier: "`_head`") und Member-Funktionen ("`insert`") der Oberklasse "`LinkedList`". Die nicht in der Klasse "`PrintableList`" definierte Methode "`insert`" wird ererbt und ist im Folgenden auf Objekte der Klasse anwendbar. Aus der Teilmengenbeziehung der Vererbung folgt, dass jede Instanz der Klasse "`PrintableList`" auch als Instanz der Klasse "`LinkedList`" behandelt werden kann, jedoch nicht umgekehrt.
Der Typ der erzeugten Objektinstanz (vgl. Abb. 2.3; hier: "`PrintableList`", Zeile 7) wird als dynamischer Typ dieses Objekts bezeichnet. Die zugewiesene, deklarierte Schnittstelle zur Manipulation des Objekts durch den nachfolgenden Code (hier "`LinkedList`"; Zeile 7) wird statischer Typ dieses Objekts genannt [Mey97]. Für Objekte des dynamischen Typs "`PrintableList`", die statisch als Instanz der Klasse "`LinkedList`" behandelt werden, ist die Methode "`print`" nicht definiert (Zeilen 9 – 10). Die explizite Umwandlung des statischen Typs eines Objekts durch den Programmierer wird als Casting bezeichnet. Casting kann zu Performanceeinbußen führen und Fehler während der Laufzeit des Programms verursachen, wenn der neue statische Typ inkompatibel zum dynamischen Typ des umgewandelten Objekts ist [Boo97, Mey97, AFM97]. Casting sollte daher vermieden werden.

Delegation und Konsultation

> In a nutshell, delegation involves redirecting some messages received by a delegating object to another object, called the delegate object, which the delegating object holds a reference to [CE00].[16]

Delegation bedeutet also das Übertragen von Aufgaben an assoziierte Objekte anstatt sie selbst auszuführen [Zam98, Boo97]. In der von Lieberman propagierten Prototyp-basierten Programmierung sollen Wiederverwendbarkeit und Minimierung des Enwicklungsaufwandes dadurch erreicht werden, dass die Definitionen von Objekten einzig ihre Unterschiede zu einer Vorlage enthalten [Lie86]. Zu beachten ist bei der Definition nach Lieberman, dass das Weitergeben der Aufgabe, d. h. des Methodenaufrufs, an das delegierte Objekt, nicht den Kontext der Methodenausführung, die sog. Selbstidentität ("`Self`"- oder "`this`"-Zeiger), verändert [Lie86, Kni99]. Sämtliche weiterführenden Nachrichten durch das delegierte Objekt, wie Methodenaufrufe, müssen zurück an das delegierende Objekt gerichtet sein [Lie86, Kni99]. Hier liegt der Unterschied zu Weiterleitungen von Anfragen (auch: Konsultation von Objekten, engl. consultation) in zusammengesetzten Datentypen an assoziierte Objekte [Kni99].
Die Assoziation der Module für eine Konsultation erfolgt durch die Kapselung einer Referenz auf eine Instanz der konsultierten Klasse. Anfragen anderer Module an das konsultierende Objekt werden an das konsultierte Objekt weitergeleitet, welches im Folgenden die Selbstidentität hält. Weiterführende Nachrichten, z. B. Methodenaufrufe des konsultierten Objekts, müssen explizit an eine übergebene Referenz des konsultierenden Objekts gerichtet sein, der Zugriff kann hier nicht über die Selbstidentität erfolgen. Die Aufgaben des konsultierten Objekts können die vollständige Öffnung der Schnittstelle des

[16] dt.: Zusammengefasst beinhaltet Delegation die Umleitung einiger Nachrichten, die durch das delegierende Objekt empfangen wurden, an ein weiteres Objekt, sog. delegiertes Objekt, auf welches das delegierende Objekt eine Referenz hält.

konsultierenden Objekts erfordern, d. h. die Ermöglichung des Zugriffs anderer Module auf *sämtliche* Funktionen des konsultierenden Objekts [Zam98, Kni99]. Dieses Vorgehen bricht die Kapselung[17] des konsultierenden Objekts und erzeugt eine enge Kopplung der beiden Modulimplementierungen [Zam98]. Das Problem, welches durch die Versetzung der Selbstidentität verursacht wird und durch die Manipulation des übergebenen, konsultierenden Objekts und die vollständige Veröffentlichung seiner Schnittstelle umgangen wird, nennt man "SELF"-Problem oder unterbrochene Delegation (engl. broken delegation) [Lie86, CE00].[18]
Virtuelle Funktionen erlauben ihre additive Redefinition in einer Unterklasse. Sie bietet in C++ einen Lösungsansatz für das SELF-Problem mit der Einschränkung, dass das delegierte Objekt statisch in Form der Oberklasse festgelegt werden muss [Ros05]. Virtuelle Funktionen erlauben ihre additive Redefinition in Unterklassen unter der Schnittstelle der Oberklasse [Boo97]. Die Vererbung mittels virtueller Funktionen ermöglicht der Unterklasse ihre eigene Beschreibung relativ zur Oberklasse zu definieren. Dafür wird das durch die Oberklasse vorgegebene Verhalten für die eigene Beschreibung *verändert* [MMP89].

Frühe und Späte Bindung

Späte Bindung (engl. late binding; auch: dynamische Bindung, engl. dynamic binding) ist die Auswahl von Methodenimplementierungen nach der Übersetzungszeit und dem Programmstart, d. h. während der Laufzeit der Software [DH96, Zam98, PS91]. Späte Bindung ermöglicht, durch Subtyp-Polymorphie, entsprechend dem dynamischen Typ eines Objekts, variables Verhalten unter einer einheitlichen Schnittstelle zu verbergen [Str00, CE00, Mey97]. Es folgt, dass der dynamische Typ eines Objekts von dessen statischem Typ verschieden sein kann.
Eine Umsetzung der dynamischen Bindung durch den Compiler ist die Verwendung einer Funktionszeigertabelle (engl. virtual function table) [DH96, Str00]. Diese Logik muss während der Laufzeit ausgewertet werden, um die korrekte Implementierung auszuwählen und zu binden. Die späte Bindung ermöglicht eine Konfigurierung der Software während der Laufzeit. Im Gegenzug verursacht späte Bindung Nachteile hinsichtlich der Laufzeit und des Speicherbedarfs [DH96]. In Java und Smalltalk werden Methodenimplementierungen im Gegensatz zu C++ *immer* dynamisch gebunden [CE00].

Das Gegenteil zur späten Bindung ist die *frühe* oder statische Auswahl und Bindung von Implementierungen. Den Namen, z. B. von Variablen, werden zum Zeitpunkt der Übersetzung die implementierenden Klassen zugeordnet [Boo97]. Statische Bindung ist nur dann möglich, wenn die Menge der möglichen Typen, welche eine Nachricht empfangen können, zum Übersetzungszeitpunkt auf Eins gesunken ist. Das erlaubt dem Compiler die Anwendung unterschiedlicher Codeoptimierungen [PS91, DLGD01, CE00].

Im Beispiel für späte Bindung der Abbildung 2.4 können Instanzen der Klasse "`SortedList`" (Zeilen 7 – 8) durch die bereits behandelte Teilmengenbeziehung ebenso

[17] Beschränkung der Interaktionen zwischen Modulen auf festgelegte Schnittstellen [Sny86, Zam98].
[18] "*SELF*" stellt in Smalltalk eine Referenz auf das Objekt der aktuell ausgeführten Methode dar. "*SELF*" korrespondiert zu "*me*" oder "*this*" anderer Sprachen [Sny86, Str00].

```
1  class LinkedList {
2    virtual void insert(Element* newElem){...}
3    virtual void modify(Element* elem){
4      remove(elem)
5      insert(elem); } };
6
7  class SortedList : public LinkedList {
8    virtual void insert(Element* newElem){...} };
9  ...
10 LinkedList* sub = new SortedList;
11 sub->insert(new Element());
```

Abbildung 2.4: Beispiel der Vererbung mittels virtueller Methoden.

als Instanzen der Klasse "`LinkedList`" (Zeilen 1 – 5) behandelt werden (Zeile 10). Bei der Anwendung der als dynamisch gebunden (virtuell) deklarierten Methode "`insert`" (Zeile 2 bzw. 11) wird deren Implementierung aus der Klasse "`SortedList`" (Zeile 8) bei Bedarf während der Laufzeit ermittelt und angewendet.

Mixins

Mixins können durch Vererbung variabel mehreren, unabhängigen Oberklassen zugeordnet werden, sog. konfigurierbare oder anonyme Vererbung [BC90, SB98, SDNB03]. Sie werden deshalb auch als abstrakte Unterklassen bezeichnet. Zum Zeitpunkt des Entwurfs und der Implementierung einer Mixin-Klasse steht ihre Oberklasse nicht fest [VN96, SB98, SB00].

> A mixin is a abstract subclass that may be used to specialize the behavior of a variety of parent classes [BC90].[19]

Mixins ermöglichen additive Erweiterungen unterschiedlicher Klassen ohne die Replikation des erweiternden Codes [SB00]. Im Gegensatz zur Einfachvererbung und Mehrfachvererbung stellt die mixin-basierte Vererbung eine Modularisierung des Software-Systems nach mehreren, *unabhängigen* Belangen dar. Die Teilmengenbeziehung der Mixin-Klasse gilt nicht nur bezüglich einer festen oder mehrerer fester, sondern beliebig vieler variabler Vererbungshierarchien.
Mixins erlauben, den intensionalen Aspekt einer Klasse, d. h. ihren Aufbau und ihre Schnittstelle, variabel zu gestalten [VN96].

In C++ können Mixins durch Templates[20] umgesetzt werden, deren zusätzliche Typ-Parameter auch den Supertyp der Klasse spezifizieren [CE00, VN96, SB98, SB00]. Die Konfigurierung des aus Mixin-Templates zusammengesetzten Typs erfolgt im Quellcode in der objekterzeugenden Methode durch die Komposition von Klassen. An dieser Stelle ist Implementierungswissen über die Mixins notwendig, um Interaktionen und Seiteneffekte zwischen den Mixins zu beachten.

[19] dt.: Ein Mixin ist eine abstrakte Unterklasse, die verwendet werden kann, um vielfältige Elternklassen zu spezialisieren.

[20] Templates sind Klassen mit konfigurierbaren Variablentypen. Die Bindung der variablen Typen erfolgt durch eine zusätzliche Schnittstelle. (genauer in Abschnitt 2.2.3)

```
1  class SimpleLinkedList{
2    Node* head;
3    void insert( AbstractElement* element){...}
4    ...};
5  template <class SUPER>
6  class BoundedLinkedList: public SUPER {
7    int _bound;
8    void insert(AbstractElement* element){...} };
9  ...
10 BoundedLinkedList<SimpleLinkedList> list;
```

Abbildung 2.5: Codebeispiel für parameterbasierte Vererbung durch Mixins.

Ein Beispiel für mixin-basierte Vererbung durch C++ -Templates ist in Abbildung 2.5 abgebildet. Eine einfache Liste wird um das Merkmal der Größenbeschränkung erweitert. Das Mixin "`BoundedLinkedList`" (Zeilen 5 – 8) erweitert die Klasse "`SimpleLinkedList`" (Zeilen 1 – 4) um neue Member-Variablen und verfeinert die Methode "`insert`". Die Konfigurierung erfolgt durch die Template-Instantiierung bei der Objekterzeugung (Zeile 10). Zu diesem Zeitpunkt wird der Supertyp des Mixins und die Teilemengenbeziehung explizit festgelegt. Das Mixin kann prinzipiell auch jeder anderen Oberklasse als "`LinkedList`" zugeordnet werden.

Eine weitere Möglichkeit, Mixins zu implementieren, ist die Vorverarbeitung der Mixins in einem Schritt der Übersetzung. Ein (Pre)Compiler kann die variable in eine feste Vererbungshierarchie umwandeln [Ros05].

2.2.2 Objektorientierte Programmierung und Entwurfsmuster

Objektorientierte Programmierung (OOP) ist die Zusammenfassung von Daten und Funktionen in modularen Einheiten wie sie auch in der Realität auftreten [Weg90, Zam98, MMP89, EFB01]. Klassen bilden in diesem Zusammenhang die abstrakte Bildungsvorschrift für Objekte (Instanzen) eines Typs, die während der Laufzeit erzeugt werden [Sny86, Zam98, AC96, WZ88].
Modularität wird in OOP durch Kapselung (engl. encapsulation; auch information hiding) der Klassenmodule erreicht. Für die Verwendung von Klassen ist im Folgenden einzig die durch sie bereitgestellte, öffentliche Schnittstelle notwendig. Die interne Umsetzung der bereitgestellten Merkmale und Funktionen bleibt dem Anwender verborgen und kann transparent variieren [Sny86, Zam98].
Wiederverwendung und Erweiterungen werden in OOP durch Vererbung von Klassen entlang einer Hierarchie sowie Objektkomposition umgesetzt [GHJV95, Weg90]. Entwurfsmuster sind diesbezüglich für nützlich befundene allgemeine Lösungen für wiederkehrende Probleme des Objektorientierten Software-Entwurfs [GHJV95, MRB97, BFVY96]. Sie geben vorgefertigte Vererbungs- und Assoziationsbeziehungen zwischen Klassen vor. Ziel von Entwurfsmustern ist die Entwicklung einer variablen, wiederverwendbaren und erweiterbaren Software durch Entkopplung von Systemmerkmalen und Verhaltensweisen sowie deren Modularisierung in assoziierte Klassen [GHJV95].
Die genannten Merkmale der "Musterlösungen" werden durch Modularisierung und Zuordnung von Verhaltensweisen und Aufgaben zu polymorphen Objekten erreicht. Diese Objekte werden durch Member-Variablen aggregierender oder assoziierender

ADT verwaltet und angesprochen. Die Polymorphie, genauer Subtyp-Polymorphie, der komponierten Objekte eines ADT erlaubt im Folgenden deren transparenten Austausch gegen Objekte anderer Implementierungen derselben Schnittstelle. Der übrige Code muss diese Variabilität nicht beachten [GHJV95].
Entwurfsmuster bieten neben Frameworks die wichtigste Möglichkeit, mittels OOP Programmfamilien (vgl. Abschnitt 2.1.4) zu entwickeln [CE99b].

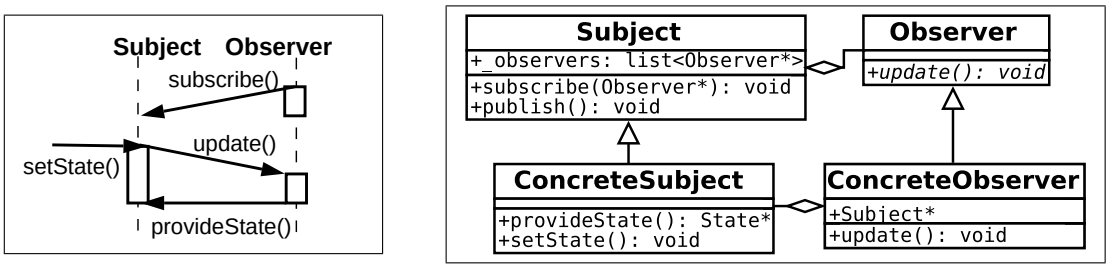

Abbildung 2.6: Flussdiagramm (links) und Klassendiagramm (rechts) für das Entwurfsmuster "Beobachter" nach [GHJV95].

Ein bekanntes Beispiel eines Entwurfsmusters ist die von Gamma et. al. beschriebene Klassenkonstellation des Beobachters (engl. observer), welche in Abbildung 2.6 dargestellt ist [GHJV95]. Dieses Entwurfsmuster erlaubt die Überwachung eines Objektes durch ein anderes. Ein Beobachterobjekt kann sein "Interesse" an den Veränderungen eines als Subjekt dienenden Objekts bekannt geben. Das beobachtende Objekt trägt sich dafür mittels der "`subscribe`"-Methode des Subjekts als dessen Beobachter in eine Liste (hier: "`_observers`") ein. Die Elemente dieser Liste werden im Folgenden durch den Aufruf ihrer "`update`"-Methode über Änderungen am Subjekt informiert. Diese Methode kann nun den Status des Beobachters aktualisieren (vgl. Flussdiagramm der Abb. 2.6).
Es ist zu bemerken, dass die Klassen der Subjekte und Beobachter lose gekoppelt sind. Die Austauschbarkeit der Klassen "`ConcreteSubject`" und "`ConcreteObserver`" gegen jeweils andere Implementierungen, die zur losen Bindung der Klassen führt, ist durch die Polymorphie der Unterklassen bezüglich ihrer Oberklassen möglich. Im Beispiel können u. a. Beobachter unterschiedlicher dynamischer Typen durch die statische Schnittstelle eines abstrakten Beobachters (hier: "`Observer`") vom Subjekt verwaltet werden.

2.2.3 Generische Programmierung – Templates

Generische Programmierung (GP) ermöglicht die Parametrisierung von Komponenten [CE00, DLGD01]. Die Umsetzung dieses Konzepts kann durch Templates erfolgen [Str00, SB00, DLGD01]. Zusätzliche Parameter eines Template-Moduls repräsentieren die durch den Algorithmus zu verwendenden Variablentypen. Der Algorithmus des generischen Moduls kann somit abstrakt gegen diese zusätzliche "Typschnittstelle" entwickelt werden, d. h. die konkreten Typen werden abstrahiert [Bra04]. Die zur Entwicklungszeit des Templates nicht festgelegten Variablentypen werden zum Übersetzungszeitpunkt durch die aufrufende oder objekterzeugende Methode mittels Template-Parametern konkretisiert. Dabei wird für jede Konfiguration der übergebenen Template-Parameter eine entsprechend getypte Methode bzw. Klasse ausgewählt oder durch den

Compiler generiert [Str00, CE00]. Dieses Vorgehen vermeidet Quellcodereplikation in statisch getypten Sprachen aufgrund von veränderlichen Variablentypen [CE00].
Das Ersetzen der Template-Parameter durch konkrete Typen durch den Compiler ermöglicht zum Zeitpunkt der Entwicklung des Templates in dessen Code die einheitliche Behandlung unterschiedlicher Typen. Die Kapselung des Konfigurationswissens über Parametertypen kann in einem sog. Konfigurationsdepot (engl. configuration repository; auch: trait class) erfolgen [Mye95, CE99b, CE00]. Die in Konfigurationsdepots modular definierten Typen erlauben die einheitliche Parametrisierung und Konfigurierung vieler Template-Klassen und somit eine Trennung des Konfigurationswissens vom Komponentenquellcode [CE99a, EBC00].
Die statisch einheitliche Behandlung von Objekten variabler Typen wird bei Templates als "parameterbasierte Polymorphie" bezeichnet [CE00, KFF98, Str00]. Virtuelle Funktionen werden dabei vermieden [DLGD01]. Im Vergleich zu Subtyp-Polymorphie, d. h. der Vererbung virtueller Funktionen, ist parameterbasierte Polymorphie auf statische Adaptierbarkeit beschränkt [CE00]. Verschieden parametrisierte Template-Klassen, sog. Template-Instantiierungen, stehen demnach während der Laufzeit trotz ihrer Generierung aus dem gleichen Template in keiner Beziehung [Str00, LBS04].
Im Unterschied zur OOP müssen GP-Module nicht die Typen der von ihnen unterstützten Parameter und Klassenvariablen vorgeben. Diese können angepasst an den konkreten Kontext von der aufrufenden (bei Template-Methoden) bzw. erzeugenden (bei Klassen-Templates) Methode festgelegt werden [CE00]. Die Konfigurierung durch Typen muss nicht explizit erfolgen, sondern kann bei Template-Methoden auch automatisiert anhand der Typen der Variablen in den Signaturen bestimmt werden [Str00].

In weiterführenden Ansätzen kann in Sprachen wie Java, Ada oder Eiffel die Variabilität von Template-Parametern durch zusätzliche Notationen eingeschränkt werden. Sogenannter gebundener oder beschränkter parameterbasierter Polymorphismus (engl. bounded parametric polymorphism; auch: constrained genericity) verlangt von den parametrisierenden Typen die Implementierung einer Schnittstelle [DLGD01, Gho04, Bra04, OW97]. C++ -Templates ermöglichen diese Einschränkung der Parameter nicht [DLGD01, Gho04].
Durch die Definition von sog. Template-Spezialisierungen kann eine semantische Variabilität der aufgerufenen Methode oder erzeugten Klasse, z. B. durch unterschiedliche Implementierungen in den Spezialisierungen, erreicht werden [Str00]. Die Auswahl der konkreten Implementierung erfolgt dabei anhand der durch den verwendenden Kontext übergebenen Parametertypen [Str00, CE00, DLGD01].[21]
Durch die Einschränkung auf statische Variabilität können Vorteile der statischen Bindung, wie verschiedene Compiler-Optimierungen, angewendet werden.

Im Beispiel in Abbildung 2.7 kann eine Liste durch Parameter angepasst werden (Zeile 5). Die Liste kann für die Verwaltung verschiedener Elementtypen konfiguriert werden. Der zu verwaltende Objekttyp "`ToHandle`" ist zum Entwicklungszeitpunkt der Liste variabel (Zeilen 4 und 6). Sämtliche Implementierungen der Klasse beziehen sich auf diesen Platzhalter (Zeile 8). Der Parameter und weitere Optionen der Konfigurati-

[21] Template-Spezialisierungen sind nicht in allen Sprachen mit Template-Unterstützung, wie Ada, Eiffel oder Generic Java, verfügbar. Die auf Template-Spezialisierungen aufbauende Template-Metaprogrammierung ist somit als C++ -spezifisch anzusehen und kann nicht stellvertretend für das Template-Konzept betrachtet werden [LBS04].

```
1  class Config1{
2    typedef ElementA ToHandle; };
3
4  template<class Config_>
5  class SimpleLinkedList{
6    typedef typename Config_::ToHandle ToHandle;
7    Node<Config_>* head;
8    void insert(ToHandle* element){
9      Node<Config_>* newFirst =
10         new Node<Config_>(element->clone());
11     newFirst->setNext(head->getNext());
12     head->setNext(newFirst);  } };
13 ...
14 SimpleLinkedList<Config1> list;
```

Abbildung 2.7: Codebeispiel für Template-Klasse und zugehöriges Konfigurationsdepot.

```
1  public class ForwardIterator
2  <LTYPE extends List<ITYPE>,ITYPE> {
3    private LTYPE _toIter;
4    private int _pos;
5    Iterator(LTYPE toIter){
6      _toIter = toIter;
7      _pos = 0; }
8    ITYPE getAct(){
9      return _toIter.getPos(_pos); }}
```

Abbildung 2.8: Beispiel für gebundene parameterbasierte Polymorphie.

on müssen durch die aufrufende oder objekterzeugende Methode konkretisiert werden (Zeile 14). Im Beispiel erfolgt die Parametrisierung unterschiedlicher Typen zusammengefasst mittels eines Konfigurationsdepots, d. h. einer Klasse, welche die Definitionen der für die Konfiguration anzuwendenden Typen kapselt (Zeilen 1 – 2).
Ein Beispiel für gebundene parameterbasierte Polymorphie in Java ist in Abb. 2.8 dargestellt. Die Java-Klasse "`ForwardIterator`" (Zeilen 1 – 9) kann nur durch Typen parametrisiert werden, die für eine Parametrisierung des Iterators vorgesehen sind. Der iterierte ADT muss dafür die Schnittstelle der Template-Klasse "`List`" implementieren (Zeile 2). Ein ADT "Baum" kann nicht als Parameter verwendet werden, da dieser nicht die Schnittstelle der Liste implementiert. Diese Einschränkung ist hier explizit erkennbar. Der Parameter der Liste für den Elementtyp (hier: "`ITYPE`"; Zeile 2) ist auch für den Iterator bindend (Zeile 8).

2.2.4 Aspektorientierte Programmierung

Aspektorientierte Programmierung (AOP) ermöglicht die Kapselung von Programmcode querschneidender Belange der OOP (vgl. Abschnitt 2.1.3) [KLM+97, SPLS+06, EFB01]. Das wird durch eine variable Verfeinerung von vorliegendem Programmcode ermöglicht [KLM+97]. Der Gesamtquelltext wird dazu in eine Menge von Komponenten und eine Menge von variabel kombinierbaren und optional anwendbaren Aspekten geteilt, welche die Komponenten erweitern. Für den im Aspekt gekapselten, methodenähnlichen Quellcode, sog. Advice, wird ein deklarativer Ausdruck seiner Wirkungsstellen im Programmablauf gepflegt, sog. Pointcut-Ausdruck (engl. pointcut expression) [KHH+01, LBS04, WL05].
Ein Pointcut-Ausdruck ist die Deklaration einer Menge von Aktionen im Ablauf des ausgeführten Programms. Die Aktionen selbst werden als Join-Points bezeichnet [MKD03, SGSP02]. Durch generische Platzhalter (in AspectC++: "%" und "...") und logische Verknüpfungen kann ein einfacher Pointcut-Ausdruck eine Menge von einzelnen Join-Points referenzieren [SGSP02, GSPS01]. Das Vorgehen, deklarativ die Menge der zu erweiternden Programmpunkte zu definieren, wird als Mengenbestimmung (auch: Quantifizierung; engl. quantification) bezeichnet [FF05].
Die statische Repräsentation der Join-Points im Komponentencode wird als lexikalischer

Schatten (engl. lexical shadow; auch: lexical join point, join point shadow) bezeichnet [MKD03, WL05, GB03]. Lexikalische Schatten können somit Klassenbezeichnungen sein, in die Methoden eingefügt werden sollen. Weitere mögliche lexikalische Schatten sind Methoden, Konstruktoren und Destruktoren, deren Aufruf oder Abarbeitung um Programmlogik erweitert werden sollen [MKD03]. Erweiternder Code kann vor oder nach der Ausführung des erweiterten lexikalischen Schattens angewendet werden (engl. before; after) bzw. ihn umschließen (engl. around). Verfeinerte Pointcut-Ausdrücke ermöglichen u. a. die Anwendung von Advice-Code in Abhängigkeit des Zustands des Programmablauf-Stacks während der Laufzeit [SGSP02, GSPS01].
Die Auswertung der Pointcut-Ausdrücke und die Verteilung und Integration des Advice-Codes in die Komponenten übernimmt ein Aspektweber[22] [KLM+97]. Der zu verändernde *Komponentencode* muss dabei nicht für die Manipulationen des Webers vorbereitet werden, sog. Vergesslichkeit (engl. obliviousness) des veränderten Codes [FF05, EFB01, LLO03].
In Abhängigkeit davon, wie mit der einzufügenden Logik, d. h. dem Advice-Code, umzugehen ist, wird die Implementierung unterschieden in statisches und dynamisches Crosscutting [KHH+01]. Statisches Crosscutting bereichert im Komponentencode vorliegende Klassen um gänzlich neue Methoden, sogenanntes Einschieben (engl. introductions; auch: inter-type declaration), und verändert statisch die Implementierung von Methoden oder die Programmstruktur, z. B. Vererbungsbeziehungen. Dynamisches Crosscutting bezeichnet das Einfügen von Advice-Code an bestimmten Stellen des Programmablaufs, sog. Code-Advice, z. B. vor dem Aufruf einer Methode [KHH+01, LST+06, SLU05]. Die Erweiterung bestehender Klassen um Funktionen, die außerhalb der Klasse definiert werden, bezeichnet man als Mechanismus offener Objekte, offener Module oder offener Klassen [MC99, Mey97, CLCM00].
Der eingewebte Advice-Code kann auf die Member-Variablen der ihn umgebenden Klasse Einfluss nehmen oder durch eine Join-Point-Schnittstelle Zugriff auf die Variablen der Signatur der erweiterten Methode[23] erlangen.
Das Hinzufügen von Komponenten umfasst in AOP – neben dem bekannten Hinzufügen neuer Klassen – die Umwandlung eines angewendeten Aspektes in eine Klasse, sog. Aspektinstantiierung, oder die Erweiterung um eine innere Klasse des Aspekts [SLU05, KHH+01].
Aspekte können in einer eigenen Vererbungshierarchie unabhängig von der Vererbungshierarchie der von ihnen beeinflussten Komponenten organisiert sein. Die Aspektvererbung ermöglicht eine Veränderung von namensbehafteten Pointcut-Ausdrücken[24] oder das Hinzufügen neuer Advice-Definitionen zu bestehenden Pointcut-Ausdrücken [SLU05].
Die Festlegung der relativen Reihenfolge der Anwendung von Aspekten erfolgt durch aspektordnende Advice-Definitionen (engl. order advice) [SLU05, LHBL06]. Ohne die Verwendung einer Order-Advice-Definition ist die Reihenfolge der Anwendung von Aspekten undefiniert [LHBL06].

Im Beispiel der Abbildung 2.9 wird gezeigt, wie eine einfache Liste um eine Größenbeschränkung erweitert wird. Um dies zu erreichen, muss die Liste ihre Beschränkung ken-

[22] z. B. AspectJ (http://www.eclipse.org/aspectj/) oder AspectC++ (http://www.aspectc.org/)
[23] Methodensignatur spezifiziert: Rückgabetyp; Anzahl und Typ der Argumente [Boo97, Mey97]
[24] Nicht jeder Pointcut-Ausdruck besitzt einen eigenen Namen.

```
1  aspect BoundedLL{
2    pointcut LL() = classes("LinkedList");
3
4    advice LL(): int _bound;
5
6    pointcut decoration(LinkedList* decoratedObj) =
7      construction(LL()) && that (decoratedObj);
8
9    advice decoration(decoratedObj): after(LinkedList* decoratedObj){
10     decoratedObj->_bound =3;  }
11
12   pointcut LLinsert(AbstractElement* element) =
13     execution("void␣LinkedList::insert(...)") && args(element);
14
15   advice LLinsert(element): around(AbstractElement* element){
16     if (tjp->that()->size() < tjp->that()->_bound){tjp->proceed();}
17      else {cout<<"bound␣wounded:␣inserting␣prohibited."<<endl;
18 } } };
```

Abbildung 2.9: Beispiel für die Erweiterung einer Liste durch einen Aspekt.

nen und jedem Einfügevorgang eine Prüfung voranstellen, ob die maximale Länge bereits erreicht wurde. Der Pointcut-Ausdruck "`LL()`" referenziert im Beispiel die zu erweiternde Klasse "`LinkedList`" (Zeile 2). Die zur Speicherung der Größenbeschränkung benötigte Variable "`_bound`" wird in die durch den Pointcut-Ausdruck referenzierten Klassen eingefügt (Zeile 4). Ein weiterer Pointcut-Ausdruck (Zeile 6) liefert alle Codestellen der Erzeugung eines Objekts vom Typ "`LinkedList`" und veranlasst im zugehörigen Code-Advice (Zeilen 9 – 10) die Initialisierung der neuen Member-Variablen. Nachfolgend wird die Unterbrechung der Ausführung der "`insert`"-Methode veranlasst. Dazu wird sowohl der Join-Point-Kontext beeinflusst (Die Klasse "`LinkedList`" wird angesprochen durch die Join-Point-Schnittstelle "`tjp->that()`"; Zeile 16), und nach erfolgreicher Prüfung wird die erweiterte Methode mittels "`proceed()`" ausgeführt (Zeile 16).

2.2.5 Merkmalorientierte Programmierung

Merkmalorientierte Programmierung (engl. feature-orientiented programming; FOP) ermöglicht die Konfigurierung eines Software-Systems durch die variable Gestaltung des Aufbaus einzelner Klassen sowie der Interaktionsmuster zwischen ihnen. Dies wird durch die flexible Zusammensetzung der Klassendefinitionen aus einer Menge von Merkmalen erreicht [Pre97, BSR04].
Merkmale (engl. Features) sind dafür wie folgt definiert:

> a feature is a product characteristic that is used in distinguishing programs within a family of related programs [BSR04].[25]

Die Konfiguration soll in FOP nicht auf Typebene der Programmierung erfolgen, sondern durch die Auswahl abstrakter, für den Kunden unterscheidbarer Systemmerkmale [LBN05, BLHM02, Pre97].
Die Implementierung eines Merkmals erfordert meist die Veränderung mehrerer Klassen. Die Menge dieser Codefragmente zur Implementierung eines abstrakten Merkmals

[25] dt.: Ein Merkmal ist eine Produkteigenschaft, die benutzt wird, um Programme in einer Familie verbundener Programme zu unterscheiden.

wird als Kollaboration bezeichnet [Boo97, SB98, BLHM02]. Kollaborationenentwürfe stellen somit die praktische Grundlage der Merkmalorientierten Programmierung dar. Die Menge der Fragmente einer Klasse, die an einer Kollaboration beteiligt sind, werden als Rolle der Klasse bezeichnet [VN96, SB98]. Kollaborationen werden als konzeptionelle *Schichten* der Software gesehen, die konsistent angewendet oder vernachlässigt werden [ALS06, BSR04].
Die sukzessive Erweiterung der Software wird durch das Hinzufügen von Schichten, sog. Refinements, im Prozess der schrittweisen Erweiterung (vgl. Abschnitt 2.1.5) ermöglicht [BSR04]. Für jede Klasse ergibt sich somit eine Verfeinerungskette (engl. refinement chain), deren Elemente Fragmente zu Klassen hinzufügen, d. h. FOP ist eine Umsetzung des Mechanismus offener Klassen (vgl. Abschnitt 2.2.4). Die unterschiedlichen Verfeinerungen einer Klasse können Aufgaben untereinander delegieren, da die beiden Module während der Laufzeit die gleiche Selbstidentität besitzen und direkt auf die Elemente des jeweils anderen Moduls zugreifen können.
FOP ermöglicht die Konfiguration der Software durch die Auswahl von abstrakten Systemmerkmalen bzw. den sie implementierenden Schichten. Um eine hohe Adaptierbarkeit zu erreichen, sollte jede Kollaboration kapselnd und unabhängig sein, d. h. keine Seiteneffekte über ihre Grenzen hinweg zulassen [SB98]. Das kann durch eine mögliche Schachtelung von Schichten unterstützt werden. Einheiten können selbst zusammengestellte Sammlungen von Einheiten (engl. collective) darstellen [BSR04].

Entsprechend der Gen-Voca- und AHEAD-Notation (vgl. Abschnitt 2.1.5) kann ein vollständiges Programm in diesen Kollaborationenmodellen als Gleichung betrachtet werden, in welcher der resultierende Programmcode als festgelegte Folge von Funktionen dargestellt wird, die auf Basiselemente, sog. Konstanten, angewendet werden. Jede Funktion erweitert das jeweilige Basisprogramm um eine abstrakte Eigenschaft [LBN05].

```
1 Conf1 = Sort(List)         // Conf1 ist sortierte Liste
2 Conf2 = Bound(Vector)      // Conf2 ist beschränkter Vector
3 Conf3 = Sort(Bound(List)) // Conf3 ist sortierte, beschränkte Liste
```

Abbildung 2.10: Darstellung der Verfeinerung von AHEAD-Konstanten.

Das Beispiel in Abbildung 2.10 soll den AHEAD-Gedanken illustrieren. Auf die Applikationen (Konstanten) "List" und "Vector" werden in einer Gleichung für die einzelnen Konfigurationen "Conf1" - "Conf3" bestimmte Merkmalfunktionen, "Sort" und "Bound", angewendet. Die sog. Gleichungsdatei (engl. equation file) definiert folglich die Reihenfolge von Funktionen, d. h. der Kollaborationen, welche sukzessive auf eine Basis angewendet werden, um das Zielsystem zu erzeugen [ALRS05b].

Mixin-Schichten (engl. mixin layers) sind eine Umsetzung des kollaborationenbasierten, merkmalorientierten Entwurfs [SB00]. Dafür werden die zu einer Kollaboration gehörenden Mixins als innere Klassen in umschließenden Mixin-Klassen, den konzeptionellen Schichten, gekapselt, um deren konsistente Anwendung sicherzustellen. Die durch Mixins eingeführte anonyme Vererbung erlaubt die Erweiterung unterschiedlicher Basisschichten und somit auch das vollständige Auslassen einer Schicht und dem in ihr implementierten Merkmal.
Die Umsetzung des merkmalorientierten Sprachentwurfs kann auch durch die Vor-

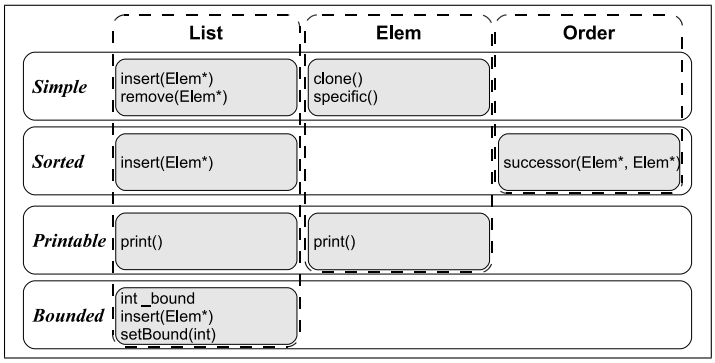

Abbildung 2.11: Kollaborationenmodell einer konfigurierbaren Liste.

```
1  refines class List{
2    int _bound;
3    LinkedList(){_bound = 0;}
4    void setBound(int newBound){
5      _bound = newBound; }
6    void insert(Elem* element){
7      if (size() < _bound) {
8        super::insert(element);
9      } else {
10       cout<<"bound wounded";
11     } } };
```

Abbildung 2.12: Code der Verfeinerung einer Klasse in FOP.

verarbeitung der Software-Einheiten durch den Compiler[26] und die eventuelle Verschmelzung der Verfeinerungshierarchie in eine einzige Klasse, das sog. Jampack, erfolgen [BSR04, Ros05].

Im Unterschied zu Jampacks können hinzugefügte Methoden in Mixin-Schichten nicht auf private Klassenelemente zurückliegender Entwicklungsstufen zugreifen, da auf diese Elemente nur innerhalb desselben Moduls, d. h. derselben Klasse bzw. desselben Mixins, zugegriffen werden kann.

Beispielhaft ist in Abb. 2.11 ein Kollaborationenmodell dargestellt. Dieses ermöglicht die Darstellung der Konfiguration einer Programmfamilie von Listen. Die zur Umsetzung verwendeten Klassen "`List`", "`Elem`" und "`Order`" bilden die Spalten dieser Matrix. Die für die Kapselung der abstrakten Merkmale verwendeten Kollaborationen werden durch die Zeilen repräsentiert. Die Matrixelemente stellen die Rollen der einzelnen Klassen in den Kollaborationen dar.
Die Konfiguration "`Simple`" kann im vorliegenden Beispiel um Merkmale wie Größenbeschränkung, sortierten Zugriff oder eine Ausgabefunktion ("`Bounded`","`Sorted`" bzw. "`Printable`") erweitert werden. Das Hinzufügen der Schicht "`Sorted`" verfeinert die Klasse "`List`" und fügt eine neue Klasse "`Order`" hinzu.
In Abb. 2.12 ist der Code für die FOP-Verfeinerung "`Bounded`" der Klasse "`List`" dargestellt. Die neu eingeführte Member-Variable "`_bound`" (Zeile 2) speichert die Längenbeschränkung der Liste. Der zusätzliche Code wird definiert (Zeilen 4 – 10) und die verfeinerte Methode "`insert`" unter der Bedingung ausgeführt, dass die Länge der Liste den Wert von "`_bound`" noch nicht erreicht hat (Zeile 8).
Durch die Vernachlässigung der Schicht "`Bounded`" wird die in Abb. 2.12 gelistete Programmlogik nicht übersetzt.

[26] z. B. AHEAD Tool Suite (http://www.cs.utexas.edu/users/schwartz/ATS.html) oder FeatureC++ (http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/)

Kapitel 3

Evaluierung der Programmiertechniken

Die Evaluierung der Programmiertechniken soll anhand mehrerer geeigneter Fallstudien erfolgen. Die Fallstudien werden unter dem Gesichtspunkt eines zu untersuchenden Qualitätskriteriums ausgewählt. Die Kapitel beinhalten diese Fallstudien, die auf Entwurfsmustern der OOP basieren, und ihre Umsetzung in den zu untersuchenden Programmiertechniken. Auf diese Weise ist eine zusammenhängende Betrachtung der Fallstudien und der durchgeführten Messungen möglich. Das führt zu einer besseren Vergleichbarkeit der Techniken bezüglich der Erfüllung des gewählten Qualitätskriteriums. Jedes Kapitel behandelt also die Erfüllbarkeit *eines* Qualitätskriteriums durch die Programmiertechniken[1]:
Abschnitt 3.1 stellt das für diese Arbeit gewählte durchgehende Beispiel vor. Die Abschnitte 3.2 bis 3.7 untersuchen die Techniken zusammenhängend bezüglich des Zeitpunktes der Adaptierung, der Performance der resultierenden Software, der Modularisierbarkeit von Belangen, der Wiederverwendbarkeit, Erweiterbarkeit und des Ressourcenverbrauches der durch die Verwendung der Techniken resultierenden Software. Die theoretisch erarbeiteten Ergebnisse der Abschnitte 3.3 und 3.7 hinsichtlich Performance und Ressourcenverbrauch werden in Abschnitt 3.8 experimentell überprüft.

3.1 Einheitliches Beispiel für die Evaluierung

Für die Evaluierung der Programmiertechniken sollen verschiedene Szenarien mit Beispielen belegt werden. Dafür wurde der ADT der verketteten Liste als Referenzbeispiel ausgewählt. Verkettete Listen dienen aufgrund ihrer allgemeinen Anwendbarkeit, ihrer einfachen Verständlichkeit und variabler Eigenschaften auch in der Literatur als Demonstrationsobjekte [CE00, AKL06].
Eine verkettete Liste verwaltet eine Menge von Elementen in einer linearen Struktur. Diese Struktur ist i. A. von dynamischer Größe, d. h. nicht größenbeschränkt. Verkettete Listen können die Grundlage für weitere ADT, wie Stacks[2] oder Queues[3], bilden und

[1] Möglich ist auch, eine Fallstudie und ein Kapitel für jede Programmiertechnik zu wählen. Diese Vorgehensweise beeinträchtigt jedoch die Vergleichbarkeit der Techniken.

[2] Das zuletzt in den ADT eingefügte Element ist manipulierbar [CE00].

[3] Das am längsten im ADT befindliche Element ist manipulierbar [CE00].

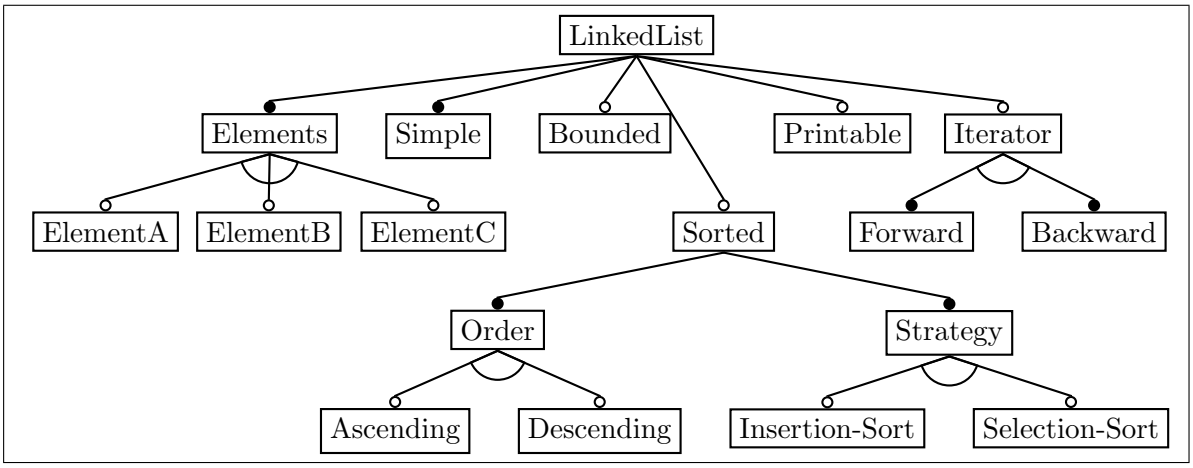

Abbildung 3.1: Merkmaldiagramm des durchgehenden Beispiels.

können auf diese Weise u. a. in Betriebssystemen, z. B. für die Speicherverwaltung, oder Datenbanken, z. B. Transaktionsverwaltung, zum Einsatz kommen.
Um der Forderung nach Wiederverwendbarkeit nachzukommen, wird die Liste als Produktfamilie betrachtet, welche sukzessive um verschiedene, unabhängige Merkmale erweitert wird. Die Erweiterung der Liste um Sortierung oder formatierte Ausgabe soll hier allgemein verständlich und repräsentativ Probleme der Software-Entwicklung aufzeigen. Die variablen Merkmale der Liste sollen vielfältiger Natur sein:

1. Variabilität soll zwischen konzeptionellen Elementen möglich sein.

2. Einzelne Elemente sollen variable Merkmale besitzen.

Die Variabilität zwischen konzeptionellen Elementen äußert sich in dieser Arbeit u. a. durch die Verwaltung von Objekten variabler Typen durch die Liste. Als Beispiel für die variablen Merkmale einzelner Elemente sei die Liste selbst genannt, die z. B. größenbeschränkt oder sortiert sein kann. Eine vollständige Darstellung der konfigurierbaren Merkmale ist in Abbildung 3.1 mithilfe eines Merkmaldiagramms zusammengefasst. Ein Merkmaldiagramm stellt die hierarchische Zergliederung der für den Kunden unterscheidbaren Merkmale einer Software-Familie dar. Die konfigurierbare Software (hier "LinkedList") bildet die Wurzel des Baumes, der optionale und obligatorische Merkmale (leerer bzw. gefüllter Kreis als Endpunkt einer Kante) zugeordnet werden. Die alternative Verknüpfung von Untermerkmalen wird durch einen leeren Halbbogen zwischen den benachbarten Kanten der Merkmalverbindungen dargestellt [KCH+90].
Die abstrakten Merkmale sollen der Liste, soweit in den entsprechenden Abschnitten nicht anders festgelegt, additiv, modular und unabhängig hinzugefügt werden. Das soll die variable Erweiterbarkeit der Software gewährleisten (vgl. Abschnitt 2.1.5).

3.2 Zeitpunkt der Adaptierung

Der Begriff der Adaptierbarkeit eines Software-Systems wurde im Abschnitt 2.1.1 behandelt. In den folgenden Abschnitten wird untersucht, welche Möglichkeiten die Paradigmen

für die Umsetzung von Adaptierbarkeit bieten und welche Folgen das in den einzelnen Programmiertechniken hat.

3.2.1 Adaptierung und Entwurfsmuster

Aufbauend auf später Bindung durch Subtyp-Polymorphie sowie Aggregation bzw. Assoziation von Objekten ermöglicht OOP die Auswahl der Varianten, d. h. die Adaptierung des Software-Systems, während der Laufzeit.
Eine Möglichkeit der Implementierung von Adaptierbarkeit in OOP ist die Vererbung. Unterklassen einer Oberklasse bilden im Folgenden die auswählbaren Varianten. Diese stellen durch Erweiterung und Redefinition der Beschreibung ihrer Oberklassen variable, auswählbare Verhaltensweisen für die Objekte des statischen Typs der Oberklasse dar.

Im durchgehenden Beispiel kann ein Listenobjekt verschiedene Sortierstrategien unterstützen, wie einfügebasierte Sortierung (engl. insertion sort) oder auswahlbasierte Sortierung (engl. selection sort).
Eine mögliche Umsetzung von Adaptierbarkeit besteht in der Vererbung der *abstrakten* Logik der Oberklasse an die Unterklasse. Die für die konkrete Sortierstrategie notwendigen variablen Methoden werden in der Oberklasse deklariert und in den Unterklassen definiert, sog. Template-Methode (engl. template method) [GHJV95]. Die Varianten der Liste sind im Folgenden variabel während der Laufzeit ohne Beeinflussung des Kontextes auswechselbar. Der Kontext verwaltet die unterschiedlichen Listen transparent unter der Schnittstelle der abstrakten Oberklasse[4].

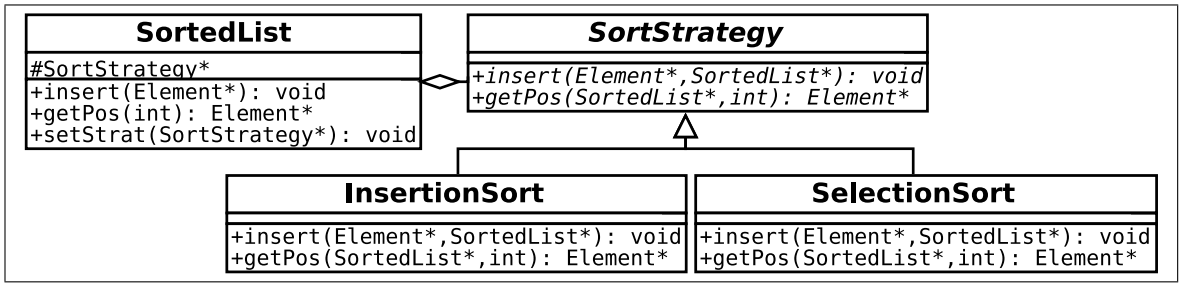

Abbildung 3.2: Darstellung des Entwurfsmusters "Strategie" nach [GHJV95].

Eine weitere Möglichkeit, Variabilität von Verhaltensweisen in OOP zu implementieren, ist die Modularisierung der Verhaltensweisen in ein assoziiertes, polymorphes Objekt, sog. Strategie (engl. strategy; vgl. Abb. 3.2) [GHJV95]. Durch die Methode "`setStrat`" des Listenobjekts kann sein Verhalten während der Systemlaufzeit angepasst werden. Das Kontextobjekt des Typs "`SortedList`" leitet im Folgenden Methodenaufrufe an das assoziierte Strategieobjekt vom statischen Typ "`SortStrategy`" weiter. Unter dieser abstrakten Schnittstelle können die Verhaltensweisen durch Objekte variabler dynamischer Typen, z. B. "`InsertionSort`" oder "`SelectionSort`", variieren.

[4]Eine abstrakte Klasse deklariert eine Schnittstelle für wiederverwendende erbende Klassen. Abstrakte Klassen können nicht zur Objekterzeugung verwendet werden [Joh97, Boo97, Mey97].

3.2.2 Adaptierung und Generische Programmierung

Generische Programmierung ermöglicht Adaptierbarkeit durch die Parametrisierung von Software-Modulen. Die Parameter legen dabei die Typen von Variablen fest, welche durch die Template-Methode oder die Template-Klasse manipuliert werden. Mittels Template-Spezialisierungen sind Varianten von Klassen- oder Methodenimplementierungen auswählbar. Die so erzeugten Varianten der Typen werden zum Zeitpunkt der Übersetzung vervollständigt und gebunden [CE00, Str00, DLGD01]. Im Gegensatz zu OOP erlaubt diese frühe Bindung der Varianten keine wechselnden Implementierungen eines Objekts während der Laufzeit. Um Laufzeitadaptierbarkeit zu erhalten, ist diese durch den Entwickler mittels entsprechender Programmlogik zu implementieren. Diese Logik bezieht sich auf den unabhängigen Belang eines anderen Moduls und widerspricht der Anforderung der Modularität. In OOP wird diese Auswahllogik für Varianten durch den Compiler erzeugt, z. B. Funktionszeigertabellen.

Beispielhaft wird wieder das Entwurfsmuster "Strategie" betrachtet. Durch die statische Bindung der GP müssen unterschiedliche Verhaltensweisen des Listenobjekts, die während der Laufzeit verfügbar sein sollen, in mehreren Strategie-Member-Variablen gehalten werden. Für die Auswahl der richtigen Strategie, d. h. der richtigen Member-Variablen, während der Laufzeit ist *programmierte* Auswahllogik notwendig.
Ist keine Laufzeitvariabilität erforderlich, so ist keine Auswahllogik notwendig, und es genügt eine Member-Variable. Die Auswahl der Strategie-Klasse erfolgt dann zum Zeitpunkt der Übersetzung durch die Parametrisierung der Kontextklasse mit dem Typ der anzuwendenden Strategie.

3.2.3 Adaptierung und Aspektorientierte Programmierung

AOP erlaubt das Verteilen von Advice-Code an deklarativ beschriebene Stellen des Ausgangsquellcodes. Die Verteilung des Codes zur Übersetzungszeit entscheidet über die Abfolge der Anweisungen während der Laufzeit. Die Adaptierung einer Software erfolgt durch das Verändern und Anpassen von Klassenbeschreibungen. Die Auswahl von Varianten der Klassen erfolgt durch das Auswählen der anzuwendenden Aspekte zum Zeitpunkt ihrer Übersetzung. Die Auswahl und Anwendung von Aspekten während der Laufzeit wird bisher nicht von allen AOP-Compilern unterstützt und kann nicht stellvertretend für das Konzept der AOP stehen, siehe AspectC++.
Die Ausführung der durch Aspekte hinzugefügten Implementierungen kann dennoch aufgrund dynamischer Gegebenheiten während der Laufzeit variabel sein. Pointcut-Ausdrücke, die Veränderungen am Basisprogramm in Abhängigkeit vom Programmablauf deklarieren, z. B. "`cflow`"[5], erlauben eine Anpassung des Systemverhaltens während der Laufzeit. Die Logik für die Evaluierung der Ausführung des zusätzlichen Codes wird durch den Compiler erzeugt und muss vom Entwickler nicht beachtet oder programmiert werden.[6] Das ist ähnlich wie bei virtuellen Funktionszeigertabellen in OOP. Somit stellt

[5] Anwendung von Advice-Code in Abhängigkeit des direkten oder indirekten Aufrufs der aktuellen Methode aus einer im Pointcut-Ausdruck referenzierten Menge von Methoden [GSPS01, SGSP02].

[6] Hinzugefügte Logik: Statusvariable und Verwaltungslogik in vorausgesetzter und aufgerufener Methode [LST+06, SPLS+06] oder die Speicherung und Auswertung des Programmablauf-Stacks [MKD03].

Aspektorientierte Programmierung ebenfalls eine Möglichkeit der modularen, dynamischen Konfigurierung dar.

```
1 aspect BoundedLL{
2   pointcut LLinsert(AbstractElement* element)
3     = execution("void␣LinkedList::insert(...)") &&
4       args(element)&& !cflow(execution("%␣Client::superuser()"));
5   advice LLinsert(element): around(AbstractElement* element){
6     ... }
7     ... };
```

Abbildung 3.3: Beispiel der Auswertung von Pointcut-Ausdrücken während der Laufzeit.

Das Beispiel variierenden Verhaltens von Listenobjekten kann in AOP durch konfiguriertes Einfügen entsprechender Methoden in die Listenklasse umgesetzt werden. Die Klasse besitzt während der Laufzeit die durch den Aspekt eingefügte Implementierung. Abbildung 3.3 zeigt ausschnittsweise ein AOP-Beispiel für Laufzeitvariabilität eines Methodenaufrufs in Form der Größenbeschränkung einer Liste. Der Pointcut-Ausdruck "`LLinsert`" (Zeilen 2 – 4) definiert die Anwendung des ihm zugewiesenen Advice-Codes (Zeile 5) in Abhängigkeit von den dynamischen Gegebenheiten des Programmablauf-Stacks während der Laufzeit. Ist der Aufruf der Methode "`insert`" direkt oder indirekt durch die Methode "`superuser`" der Klasse "`Client`" initiiert worden, so wird der zugehörige Advice-Code nicht ausgeführt. Andernfalls wird die Größenbeschränkung beachtet (Zeilen 5 – 6). Die Implementierung der Methode "`insert`" ist also während der Laufzeit variabel.

3.2.4 Adaptierung und Merkmalorientierte Programmierung

Durch die Bindung an Vererbungshierarchien (Mixin-Schichten) bzw. die Erzeugung einer Klasse (Jampack) müssen die FOP-Schichten spätestens zur Übersetzungszeit ausgewählt werden, damit objekterzeugende Software-Einheiten (Klassen) während der Laufzeit vollständig definiert sind. Durch diese Einschränkung sind keine variablen Änderungen während der Laufzeit auf Basis der FOP-Schichtenarchitektur möglich.

Im Beispiel der konfigurierbaren Sortierstrategie einer Liste können die variablen Verhaltensweisen jeweils gekapselt in FOP-Schichten implementiert werden. Die Schichten werden zum Zeitpunkt der Übersetzung ausgewählt und die spezifischen Methoden werden in die Schichtenarchitektur bzw. das Jampack der manipulierten Klasse eingefügt bzw. werden verfeinert.

3.3 Performance

Die Erfüllung der Anforderung nach performanter Software ist eng verbunden mit der Anzahl der programmierten und durch den Compiler erzeugten Prozessoranweisungen, die während der Laufzeit abgearbeitet werden müssen. Je weniger Logik abgearbeitet werden muss, desto performanter ist die Software.

Um Erweiterbarkeit und Adaptierbarkeit zu unterstützen, muss in einzelnen Programmiertechniken zusätzliche Verwaltungslogik abgearbeitet werden, welche u. U. für die aktuelle Konfiguration nicht benötigt wird. Es muss demnach untersucht werden, ob variable und erweiterte Programmlogik die Laufzeit des Programms negativ beeinflussen.

3.3.1 Performance und Entwurfsmuster

In OOP stehen für die modulare Implementierung von Adaptierbarkeit einzig Mechanismen der Laufzeitkonfigurierung zur Verfügung, z. B. Subtyp-Polymorphie.

> OO implementation mechanisms for implementing intra-application variability (e.g., dynamic polymorphism) are also used for inter-application variability [CE00].[7]

Die Einbußen bezüglich der Performance aufgrund von Mechanismen der späten Bindung sind somit in OOP *immer* zu tragen. Dies gilt selbst für statisch variable Merkmale, welche sich während der Laufzeit des Programms nicht ändern, d. h. keine zusätzliche Logik für späte Bindung benötigen (Inter-Applikationsvariabilität). Aufgrund der späten Bin-

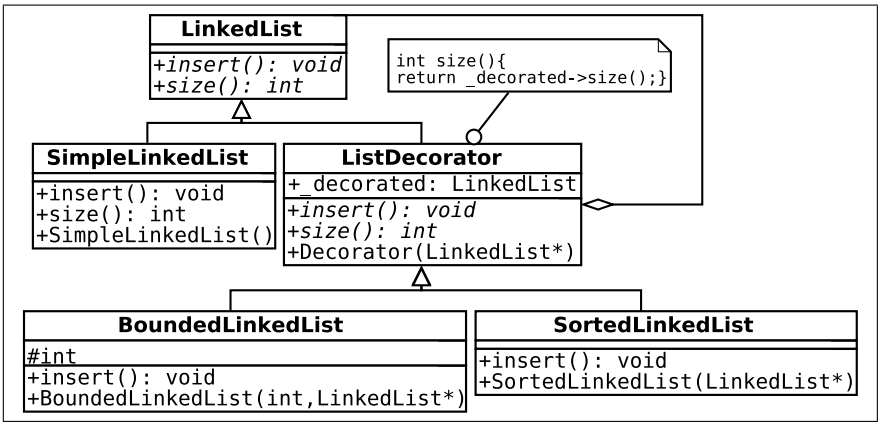

Abbildung 3.4: Darstellung der Dekoration einer einfachen Liste in OOP.

dung sind Compiler-Optimierungen, wie Vorberechnung von Methodenergebnissen oder Scheduling der Anweisungen[8], nicht anwendbar, da dynamische Bindung die Bestimmung des Programmflusses nicht eindeutig zulässt. Das kann sich negativ auf die Performance des Zielsystems auswirken.

In OOP sind häufig Indirektionen notwendig, welche den Sender einer Nachricht von ihrem Empfänger abkoppeln. Die lose Kopplung ermöglicht die unabhängige Variation der Typen des Senders und des Empfängers, fügt aber auch zusätzliche Programmschritte der Indirektionen hinzu.

Das Beispiel des Entwurfsmusters "Dekoration" soll dies veranschaulichen (vgl. Abbildung 3.4) [GHJV95]. Erweiterndes Objektverhalten zur Basisimplementierung

[7] dt.: OO-Implementierungsmechanismen für die Implementierung von Intra-Applikationsvariabilität (z. B. dynamischer Polymorphismus) werden ebenfalls für Inter-Applikationsvariabilität verwendet.

[8] Scheduling ist die äquivalente Umordnung von Maschinenbefehlen zur Verbesserung der Performance oder des Speicherbedarfs [CE00].

"`SimpleLinkedList`" wird hier in einer Klasse, der sog. Dekoration (engl. decorator), gekapselt und mit dem dekorierten Objekt durch eine Member-Variable verknüpft. Instanzen der Klasse "`ListDecorator`" dekorieren hier jeweils ein Objekt des statischen Typs "`LinkedList`". Durch diese Schnittstelle können sowohl einfache Listen als auch bereits dekorierte Listen erweitert werden. Mehrere Erweiterungen können dem dekorierten Objekt im Folgenden additiv und unabhängig hinzugefügt und genommen werden.
Der Aufruf einer für das dekorierte Komponentenobjekt definierten Funktion erfordert im Folgenden die Abarbeitung der zugehörigen Methoden der umschließenden Dekorationen. Mehrere Dekorationen erfordern hier mehrere Indirektionen. Methoden der Dekoration, die keine Funktionalität hinzufügen, führen dabei nur die Weiterleitung der Anfrage durch ohne selbst einen Mehrwert zu erzeugen.

In Abb. 3.4 leitet die Methode "`size`" einer umschließenden Dekoration "`SortedList`" die Anfragen an das dekorierte Objekt "`_decorated`" vom Typ "`LinkedList`" weiter und liefert das Ergebnis unverändert zurück. Diese Indirektionen stellen zusätzlich abzuarbeitende Programmlogik ohne Wert dar. Dynamische Bindung der dekorierenden und dekorierten Objekte verursacht in OOP zusätzliche Laufzeiteinbußen.
Wie dieses Entwurfsmuster stellvertretend zeigt, ist jegliche modulare Adaptierbarkeit und lose Kopplung in OOP mit dynamischer Bindung verbunden [GHJV95].

3.3.2 Performance und Generische Programmierung

Templates ermöglichen durch Parametrisierung die Entwicklung variabler Software ohne die Performancenachteile durch spätes Binden. Folglich sind diverse Compiler-Optimierungen anwendbar. Die Klassen und Methoden können direkt mit den finalen, zu manipulierenden Typen parametrisiert werden. Die Referenzierung und das Durchlaufen abstrakter Schnittstellen ist nicht notwendig.
Für die Erweiterbarkeit und Adaptierbarkeit der Software kann es für Templates wie für OOP notwendig werden, Indirektionen zu durchlaufen.

Ebenso wie bei OOP wird auch hier das Entwurfsmuster der Dekoration als Beispiel betrachtet. Die umschließende Dekoration fügt zusätzliche Funktionalität, aber auch Indirektionen hinzu. Wie bei OOP erfordert jegliche Manipulation des dekorierten Objekts die Abarbeitung zugehöriger, weiterleitender Methoden in den jeweilig dekorierenden Objekten. Die dekorierende Klasse wird in GP mit dem konkreten Typ der jeweils dekorierten Klasse parametrisiert. Späte Bindung ist nicht erforderlich.

3.3.3 Performance und Aspektorientierte Programmierung

Erweiterungen und Veränderungen von Komponenten durch einen vorverarbeitenden Aspektweber bedürfen i. A. *keiner* dynamisch gebundenen Methoden. Diese sind jedoch nicht gänzlich auszuschließen.[9] Durch die direkte Erweiterung von Methoden innerhalb einer Klasse können Indirektionen ohne Wert vermieden werden.
AOP-Konzepte können z. B. unter Verwendung zusätzlicher Template-Methoden umgesetzt werden. Diese führen im Folgenden zu Indirektionen einzig für die veränderte

[9] Der Zugriff des Advice-Codes auf Kontextvariablen führt in AspectC++ teilweise aufgrund von Typprüfungen zur Erzeugung virtueller Methoden durch den Compiler [LST+06].

Methode.
Zusätzliche Laufzeitkosten werden durch AOP aufgrund dynamischen Crosscuttings erzeugt. Die Ausführung des Advice-Codes kann dabei teilweise nicht statisch vorhergesagt werden und erfordert die Abarbeitung zusätzlicher Anweisungen während der Laufzeit [LST+06].[10] Auch der Zugriff auf Variablen des Join-Points benötigt die Abarbeitung der Logik der Join-Point-Programmierschnittstelle (engl. application programming interface; API).

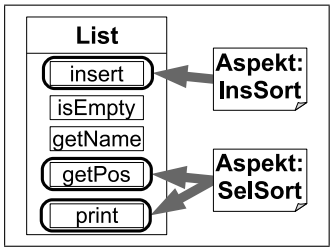

Abbildung 3.5: Schematische Darstellung einer Dekoration in AOP.

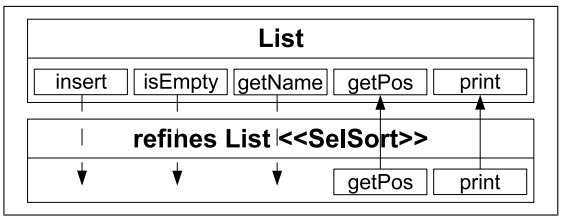

Abbildung 3.6: Abbildung der Dekoration in FOP.

Die Umsetzung des Entwurfsmusters "Dekoration" ist in AOP durch die direkte Dekorierung von Methoden möglich. Die Veränderung der Methode einer Klasse während der Übersetzung des Aspektes erfordert keine Indirektionen für nicht dekorierte Funktionen. Standardweiterleitungen werden vermieden.

Abbildung 3.5 zeigt die Dekoration einzelner Methoden der Liste durch unterschiedliche Aspekte. Es werden keine Indirektionen für die nicht dekorierten Methoden "`isEmpty`" oder "`getName`" benötigt. Die Anwendung der Aspekte ist dennoch unabhängig und optional.

3.3.4 Performance und Merkmalorientierte Programmierung

Ebenso wie GP ist auch FOP auf statische Variabilität begrenzt und ermöglicht so ein frühes Binden von Methodenimplementierungen durch den Compiler. Spät gebundene Methoden sind für gute Adaptierbarkeit und Erweiterbarkeit durch die nötige Vorverarbeitung nicht notwendig.
Methodenaufrufe können durch Vererbung zwischen den FOP-Schichten direkt an Oberklassen delegiert werden. Dieses Vorgehen ist ohne Indirektionen für nicht verfeinerte Methoden möglich.
Mixin-Schichten als eine Umsetzung des Kollaborationenentwurfs der FOP erfordern die Replikation nicht vererbbarer Elemente in jeder verfeinernden Klasse, z. B. Konstruktoren. Diese kopierten Elemente müssen während der Laufzeit abgearbeitet werden und stellen vom Compiler hinzugefügte Indirektionen dar. Die Umsetzung der FOP durch Jampacks vermeidet diese Indirektionen.

Ebenso wie AOP ermöglicht FOP die Umsetzung des Entwurfsmusters "Dekoration" durch die Verfeinerung jeweils relevanter Methoden. Nicht dekorierte Methoden werden unverändert, ohne Indirektionen und statisch gebunden in der verfeinernden Mixin-

[10] Die Ausführung von Advice-Code, der an "$cflow$"-Join-Points gebunden ist, kann Evaluierungslogik erfordern, die während der Laufzeit abgearbeitet werden muss [LST+06].

Schicht ererbt bzw. in der zusammengesetzten Klasse, dem Jampack, definiert.
In der Umsetzung der FOP mittels Mixin-Schichten treten Indirektionen nur für die dekorierten, verfeinerten Methoden auf, die ihr Äquivalent in der Oberklasse aufrufen und einen abstrakten Wert hinzufügen. Abbildung 3.6 illustriert die mixin-basierte Umsetzung, in der die dekorierten Methoden verfeinert werden (hier: "`getPos`" und "`print`"). Der solide Pfeil einer Verfeinerung demonstriert hier die Indirektion des Methodenaufrufs der verfeinerten Methode. Nicht dekorierte Methoden wie "`insert`" werden ererbt und können ohne Indirektion ausgeführt werden (unterbrochene Pfeile).

3.4 Modularisierbarkeit von Belangen

Die Programmiertechniken ermöglichen jeweils eine Aufteilung des Programmcodes in eigenständige Software-Einheiten. Die Separation der Belange ist die Zuordnung voneinander unabhängiger Systemmerkmale zu diesen Software-Einheiten. Jedes Merkmal, d. h. jede Konfiguration eines Belangs, betrifft dabei eine Menge von Stellen im Quellcode, die für die Ausprägung dieses Belangs spezifisch sind. Zur Veränderung der Konfiguration eines Belangs müssen die Codestellen manipuliert werden, die für die Merkmale vor und nach der Adaptierung spezifisch sind. Das Ziel der Modularisierung ist, die Anzahl dieser Stellen im Quellcode für *jeden* Belang auf Eins zu reduzieren und vom Restprogramm abzugrenzen. Das impliziert Merkmalkohäsion und erlaubt unabhängige Veränderungen der Belange durch den Austausch eines Moduls. Codezersplitterung und die Verwirrung von Code müssen also für die Modularisierung von Belangen primär vermieden werden. Da die Anzahl der variablen Belange eines konkreten Software-Produkts nicht vorhergesagt werden kann, sollen so viele Belange wie möglich zu unabhängigen Modulen zugeordnet werden können.
Eine weitere Anforderung an Module besteht in der Kapselung ihrer Implementierung, d. h. der Möglichkeit, explizite Schnittstellen zu definieren, und so die parallele Entwicklung durch unterschiedliche Entwicklerteams zu unterstützen [Sny86, LLO03].
Unabhängige Belange können beliebig genau spezifiziert werden. Durch Dekomposition und Komposition kann die Komplexität eines Großteils der Software vor dem Entwickler verborgen werden. Dieser bearbeitet dann lediglich einen Teil der Software oder Abstraktionen der Komponenten. (Aufgrund der besonderen Bedeutung für die Wiederverwendbarkeit von Software wird der Aspekt der semantischen Dekomposition bzw. Anpassbarkeit von Modulen an dieser Stelle vernachlässigt und in Abschnitt 3.5 näher betrachtet.)

3.4.1 Modularisierbarkeit und Entwurfsmuster

OOP ermöglicht in einer Vererbungshierarchie die Modularisierung unabhängiger Belange entlang *einer* Dimension in Klassen.
Die Schnittstelle einer Klasse identifiziert den Belang, den diese Klasse und die von ihr erbenden Klassen kapseln, und deklariert Funktionen, welche zur Erfüllung dieses Belangs benötigt werden. Die unterschiedlichen Unterklassen, welche diese Schnittstelle implementieren, stellen die Coderepräsentation der jeweiligen Konfiguration des Belangs

der Klasse dar.[11]
Die Zuordnung eines Merkmals, d. h. einer Implementierung, zu einem Belang, d. h. zu einer Schnittstelle, erfolgt durch Vererbung. Die Oberklasse ist hierbei fest vorgegeben. Durch die Vererbung werden sämtliche Entwurfsentscheidungen aus der Oberklasse übernommen, z. B. Typisierung und Benennung von Funktionen und Variablen. Variationen sind begrenzt auf die variable Implementierung der ererbten Schnittstelle. Das Modul der einzelnen Klasse stellt demnach die Modularisierung *eines* Belanges dar.[12] Die variable Zuordnung einer Klasse zu mehreren Belangen, d. h. zu mehreren Schnittstellen, ist nicht möglich.

Sämtliche Belange, die nicht auf die gewählte Klassenhierarchie abzubilden sind, sind querschneidend und können Codereplikation erzeugen (vgl. Abschnitt 2.1.3). Querschneidende Belange implizieren mehrere Codestellen zur Änderung des ausgeprägten Merkmals und widersprechen der Merkmalkohäsion. Viele Entwurfsmuster benötigen Programmlogik in mehreren Klassen und sind in diesen Fällen ebenso als querschneidende Belange zu sehen [Bos98, GHJV95].
Die Verknüpfung unabhängiger Belange, d. h. unabhängiger Vererbungshierarchien, ist in OOP einzig durch Assoziation möglich. Aufgrund der Unveränderlichkeit einer Klasse erzeugt diese Variabilität eine enge Kopplung der assoziierenden Klasse und ihrer Unterklassen an die assoziierten Klassen. Somit wird Codereplikation für die variable Assoziation von Vererbungshierarchien erzeugt.
Klassen, die Ausprägungen derselben Schnittstelle, d. h. desselben Belangs, sind und nicht in einer Vererbungsbeziehung stehen, müssen durch Programmlogik ausgewählt werden, z. B. "`switch-case`"-Anweisungen. Dieser Code ist querschneidend zum Belang des übrigen Codes der Klasse, da er sich auf einen unabhängigen Belang einer anderen Klasse bezieht.

Aufgrund der i. A. nicht ausreichenden Möglichkeit, querschneidende Belange zu vermeiden, folgt die Verteilung von Codefragmenten eines Belangs über mehrere Komponenten. Ein Modul in Form einer Klasse genügt für diese Belange nicht der Forderung nach Merkmalkohäsion.

Die Definition der Schnittstellen von Klassen, die Belange kapseln, ist explizit und sehr genau möglich.
Für bestimmte Entwurfsmuster der OOP muss jedoch die Intention von Schnittstellen, die Kapselung von Implementierungswissen, zurückstehen. Diese Kapselung ermöglicht die unabhängige Variation der Implementierung von Modulen. Entwurfsmuster, wie Strategie oder Besucher (engl. visitor), sehen Module vor, die vollständigen Zugriff auf die von ihnen manipulierten Klassen benötigen können [GHJV95]. Es kann zum Entwicklungszeitpunkt der Klasse i. A. nicht vorhergesagt werden, welche Klassenelemente durch diese zusätzlichen Module manipuliert werden müssen. Die Schnittstelle der manipulierten Klasse muss deshalb präventiv vollständig geöffnet werden, um Codereplikation oder invasives Eingreifen zu vermeiden [GHJV95]. Die folgende geringe Kapselung der Klasse kann eine enge Kopplung von Modulen verursachen.

[11] Die Methoden "$isSorted(List*)$" und "$sort(List*)$" identifizieren den Belang der Sortierung einer Liste. Deren unterschiedliche Implementierungen in Unterklassen stellen Ausprägungen des Belangs dar.

[12] Mehrfachvererbung erlaubt die Zuordnung einer Implementierung zu mehreren Belangen. In statisch getypten Sprachen, wie C++ oder Java, wird ein Objekt *einem* statischen Typ zugewiesen. Der Wechsel des statischen Typs zu einem anderen Supertyp der Klasse erfordert Typumwandlungen und kann somit Laufzeitfehler verursachen [Bra04].

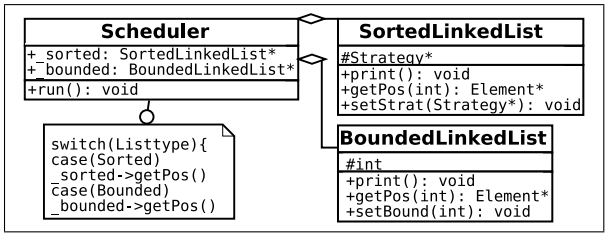

Abbildung 3.7: Darstellung der Modularisierung einer Liste mittels Assoziation.

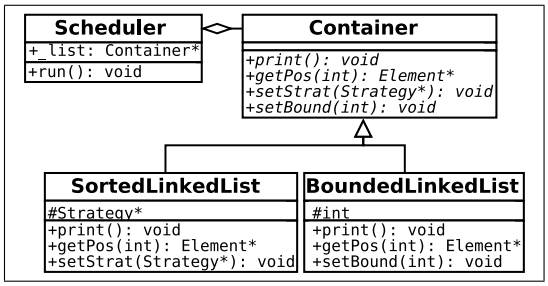

Abbildung 3.8: Bild der Modularisierung von Belangen mit Vererbung.

Ein Beispiel soll die behandelten Probleme veranschaulichen (vgl. Abb. 3.7). Die Klassen "`SortedLinkedList`" und "`BoundedLinkedList`" stehen in keiner Vererbungs- und Teilmengenbeziehung. Durch die Unabhängigkeit der Klassen können Entwurfsentscheidungen separat variieren. Im Folgenden sind die beiden Typen jedoch nicht polymorph und erfordern ebenso ihre separate Behandlung im Code der Klasse "`Scheduler`", z. B. mittels "`switch-case`"-Anweisungen. Dadurch wird eine enge Kopplung der "`Scheduler`"-Klasse an die Listen erzeugt. Eine Erweiterung um unabhängige Varianten der Listen ist nicht darstellbar. Die Auflösungslogik betrifft den Belang eines anderen Moduls als die Klasse "`Scheduler`". Weiterhin wird der Code einheitlicher Merkmale der Listen, z. B. die Methode "`print`", repliziert.
Eine Lösung dieser Probleme stellt eine abstrakte Schnittstelle für die assoziierten Klassen dar (vgl. Abb. 3.8). Der Scheduler ist nun lose mit den Typen seiner Member-Variablen "`_list`" gekoppelt und enthält keine Auflösungslogik mehr. Auch die Codereplikation zwischen den Typen der Member-Variablen kann vermieden werden, da gemeinsame Logik aus der Oberklasse geerbt werden kann. Um statische Typsicherheit[13] zu garantieren, muss die Oberklasse "`Container`" eine abstrakte Schnittstelle ihrer Unterklassen darstellen. Das macht die Definition von redefinierbarem Standardverhalten der Methoden "`setStrat`" und "`setBound`" in der Oberklasse notwendig. Die Kapselung unabhängiger Belange, d. h. die Verwirrung des Codes, ist in Form der Oberklasse anhand dieser Methoden sehr gut zu erkennen.

3.4.2 Modularisierbarkeit und Generische Programmierung

Die explizite Vererbung der GP ist wie für OOP als Dekomposition entlang einer Dimension zu sehen, d. h. eine Klasse kapselt einen unabhängigen Belang. Das führt wie bei OOP zu querschneidenden Belangen und Codereplikation.
Durch die statische Bindung von Templates sind die Implementierungen verschiedener Template-Klassen und Template-Klassenkonfigurationen nicht während der Laufzeit polymorph. Dies hat verschiedene Auswirkungen:

1. Die Notwendigkeit der Laufzeitadaptierbarkeit kann Codereplikationen in verwaltenden Methoden und Member-Variablen verursachen. Die statisch getypten Variablen können nur Elemente *eines* konkreten statischen Typs während der Laufzeit aufnehmen. Für Laufzeitvariabilität dieser Typen muss durch den Entwickler

[13] Der Compiler garantiert die Kompatibilität eines Objekts zu seinem statischen Typ [Boo97].

zusätzlicher Verwaltungscode zur Auswahl der richtigen Implementierung programmiert werden. Das erzeugt Verwirrung und Zersplitterung von Code eines Belangs.

2. Typen von Template-Klassen werden konkretisiert durch die Übergabe von Parametern im aufrufenden Code [LBS04]. Ein Modul, welches das Template nur verwaltet, kann somit nicht mehr dessen expliziten statischen Typ vorhersehen. Das verwendende Modul muss eventuell ebenso parametrisiert werden. Durch die Rekursion der Parametrisierung kann die Notwendigkeit bestehen, große Teile der Software feingranular zu parametrisieren. Die Konfigurierung von Template-Klassen führt somit zur Notwendigkeit, Entscheidungen über Adaptierbarkeit der Software *früh*, d. h. zum Zeitpunkt des Entwurfs aller Module, zu treffen. Die Variabilität dieser Entscheidungen ist folglich sehr lange zu beachten und zu pflegen, die Variabilität ist hier nicht transparent für unbeteiligten Code.
   Die Parametrisierung der Module erfolgt auf Typebene, was die Notwendigkeit von internem Wissen über große Teile der Software voraussetzt. Konfigurationsdepots erlauben die Kapselung von Konfigurationswissen und somit Abstraktionen bei der Konfigurierung. Ihre Verwendung vermeidet dennoch nicht die Parametrisierung vieler Module. Konfigurationsdepots erzeugen weiterhin explizites Verwirren von Code unterschiedlicher Belange in der Depotklasse. Diese kann nur invasiv angepasst werden.

Durch die Parametrisierung und Assoziation der Template-Klassen im aufrufenden Code ist in den Komponenten selbst kein Zusammenhang zu anderen Komponenten erkennbar. Daraus folgt, dass verschiedene Implementierungen eines Belangs keine syntaktische Verbindung aufweisen müssen. Sofern ein Merkmal nicht allein durch eine Template-Klasse gekapselt werden kann, ist zudem die semantische Beziehung zu assoziierten und aggregierten Klassen nicht erkennbar. Merkmalkohäsion ist in GP nicht generell gegeben.

Eine Klasse als Modul ist ebenso wie bei OOP die Implementierung einer Schnittstelle. Teile der Schnittstellen von Templates, z. B. Variablentypen in Methodensignaturen, können direkt durch die Übergabe von Parametern im aufrufenden Code beeinflusst werden. Im Allgemeinen sind Schnittstellen von Templates also nicht verlässlich explizit festgelegt.
Schnittstellen für Klassen können in GP auch durch Template-Spezialisierungen variabel sein.[14] Unterschiedliche Parametrisierungen eines Moduls können dadurch vollkommen unterschiedliche Schnittstellen erzeugen. Die parallele Entwicklung einer parametrisierten Klasse ist erschwert und erfordert in diesem Fall zusätzlich zur Definition der Schnittstelle eines Moduls eine Verständigung, für welche Parameterkonfiguration diese gelten soll. Gebundene parameterbasierte Polymorphie ermöglicht die Vermeidung derartiger Probleme, indem eine minimale Schnittstelle der Template-Parameterklasse vorgegeben wird – trotzdem finden dabei keine impliziten Typumwandlungen statt.
Die kapselnde Eigenschaft von Schnittstellen kann auch in GP nicht immer gewährleistet werden. Das kann eine enge Kopplung mehrerer Module erzeugen und die Variabilität bei der Anpassung einzelner Module mindern. Strategie-Objekte oder Besucher-Objekte

[14] Einige Programmiersprachen unterstützen Template-Spezialisierungen nicht umfassend, z. B. Java.

können eine Manipulation des internen Zustands erfordern und dafür wie bei OOP eine vollständig offene Schnittstelle der Template-Klasse verlangen [GHJV95].

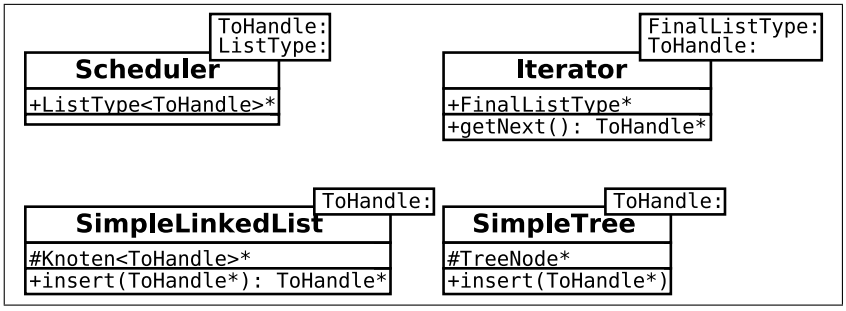

Abbildung 3.9: Darstellung der fehlenden Merkmalkohäsion von Templates.

Das Beispiel in Abb. 3.9 illustriert die behandelten Merkmale von GP-Modulen. Die Klasse "`Iterator`" implementiert eine Navigation über Listen. Trotz semantischer Verbindung ist keine syntaktische Verbindung zur Listenklasse vorhanden. Allein durch die Analyse des Quelltextes des Iterators kann ermittelt werden, dass dieser nicht für Bäume geeignet ist. Beschränkter parameterbasierter Polymorphismus bietet hier die Möglichkeit der Einschränkung des Template-Parameters "`FinalListType`" auf eine minimale Schnittstelle der verwalteten Klasse, z. B. "`SimpleLinkedList`". Baumtypen wie "`SimpleTree`" können nun nicht mehr fälschlicherweise als Template-Parameter verwendet werden.
Die Klasse "`SimpleLinkedList`" und "`SimpleTree`" implementieren die gleiche Schnittstelle, dennoch ist ihre Verbindung als Containerklasse[15] nicht explizit erkennbar.

3.4.3 Modularisierbarkeit und Aspektorientierte Programmierung

Der in Aspekten gekapselte Code ermöglicht die Verknüpfung und Erweiterung mehrerer Komponenten. Erweiterungen in AOP sind somit nicht an Klassenhierarchien gebunden und können Code querschneidender Belange der OOP und GP ohne Codereplikation in Aspekten kapseln.
Neben der Vermeidung von Replikation des Codes ist die Merkmalkohäsion von Bedeutung, d. h. die eineindeutige Zuordnung von Modulen zu Belangen. In dieser Hinsicht kann AOP Probleme erzeugen:

1. Kann ein querschneidender Belang nicht durch das alleinige Hinzufügen und Verfeinern von Methoden gekapselt werden, so müssen zusätzliche Klassen eingebunden werden. Diese liegen neben dem Aspekt-Code wie die erweiterten Klassen im Quellcode vor oder müssen durch interne Klassen des Aspekts realisiert werden. Das verteilte und undifferenzierte Vorliegen von Code eines Belangs im Gesamtquelltext widerspricht der Anforderung nach Merkmalkohäsion. Innere Klassen des Aspekts sind an die Anwendung des Aspekts gebunden und nur schlecht wiederverwendbar oder erweiterbar [LHBC05].

[15] Klasse, deren Instanzen eine Menge von Instanzen anderer Klassen verwalten [Mey97, Boo97].

Aspektinstantiierung, d. h. die Umwandlung eines Aspekts in eine objekterzeugende Einheit, ermöglicht die Vermeidung der Definition *einer* zusätzlichen Klasse. Die Verwendung eines Aspekts als Klasse kann die Unwissenheit des Komponentencodes über die Anwesenheit von Aspekten gefährden, wenn die Aspektklasse auch durch den Komponentencode instantiiert werden kann. Der Aspekt ist dann eng an die Komponente gebunden.

2. Der semantische Zusammenhang von Aspekten ist nicht immer syntaktisch darstellbar. Codereplikation zwischen Aspekten kann und sollte durch die Modularisierung des replizierten Codes in einen neuen Aspekt vermieden werden [MF05]. Im Folgenden liegen im Quellcode 2 Module undifferenziert zu den erweiterten Klassen und anderen Aspekten vor. Vererbung bietet hier die einzige Möglichkeit, die Abhängigkeit der übrigen Aspekte vom gemeinsam genutzten Aspekt im Quellcode zu verdeutlichen.
   Vererbung zwischen Aspekten bedeutet die gleichzeitige Anwendung des Superaspekts und des abgeleiteten Aspekts. Fügt der Superaspekt neue Elemente zu Klassen hinzu, muss das mehrfache Erweitern einer einzelnen Klasse vermieden werden, da sonst Übersetzungsfehler auftreten. Der Superaspekt muss demnach unabhängig sein oder paarweise disjunkt definierte Pointcut-Ausdrücke in den abgeleiteten Aspekten besitzen. Die Trennung der Pointcut-Ausdrücke vernachlässigt die Aspektquantifizierung.

Die fehlende Bindung zwischen semantisch zusammenhängenden Einheiten (Komponente-Aspekt; Aspekt-Aspekt) widerspricht der Merkmalkohäsion und erzeugt hier Codereplikation der Pointcut-Ausdrücke. Die einzelne kapselbare Einheit der AOP ist demnach i. A. nicht semantisch vollständig.
(Ein *unabhängiger* vorausgesetzter Aspekt kann durch einen Order-Advice referenziert werden (vgl. Abschnitt 2.2.4) [SLU05]. Diese Definition einer Ordnung dient jedoch nicht der Sicherstellung der Anwendung des Aspekts.)

Die Schnittstelle eines Aspekts ist variabel und hängt von dem zugrunde liegenden System erweiterter Komponenten und von bereits angewendeten Aspekten ab [KM05]. Aus Sicht der Komponente ist das nicht als Problem erkennbar, da sie auf die Interaktion mit Aspekten nicht vorbereitet werden muss. Eine inkonsistente oder unvollständige Anwendung von vorausgesetzten Aspekten oder der Austausch der Komponente können die Schnittstelle der erweiterten Komponente verändern und das undokumentierte Scheitern nachfolgender Aspekte sowie Laufzeitfehler verursachen.[16]
Die Intention von Entwurfsmustern, wie "Strategie" oder "Besucher", kann in AOP durch das Einfügen der variablen und zusätzlichen Programmlogik in die manipulierte Klasse, d. h. damit in ihren Namensraum, umgesetzt werden. Die Schnittstelle der erweiterten Klassen muss dafür nicht vollständig veröffentlicht werden, da nur der Zugriff, ausgehend von anderen Klassenmodulen, eine öffentliche Schnittstelle erfordert. Module, welche die durch Aspekte manipulierte Klasse als konfigurierbare Bibliothek betrachten, können weiterhin von den Details der Implementierung getrennt werden.

[16] Schnittstellen, die die Wirkung von Aspekten darstellen (engl. aspect-aware interface), können ohne Analyse des Aspektes erst zum Zeitpunkt seiner Anwendung erzeugt werden. Dynamisches Crosscutting stellt hier ein Problem dar und wird nicht abschließend betrachtet [KM05].

```
1 aspect PrepareVisitor{
2   pointcut elements1() =
3     //erweitere folgende Klassen:
4     classes("ElementA"||"ElementB");
5   advice elements1():
6     void accept(Visitor* v){
7       v->accept(this);  }};
```

```
1 aspect SelCountVisitor: public Visitor{
2   pointcut doSelCount()= ...;
3   advice doSelCount(): ...;
4   int _resultA;int _resultB;int _resultC;
5   SelCountVisitor():
6     _resultA(0), _resultB(0), _resultC(0){}
7   void accept(ElementA* elem){_resultA++;}
8   void accept(ElementB* elem){_resultB++;}};
```

Abbildung 3.10: Beispielcode für Merkmalkohäsion von Aspekten in einem Besucheraspekt.

Abb. 3.10 zeigt beispielhaft zwei Aspekte. Sie implementieren gemeinsam das Entwurfsmuster eines Besuchers (engl. visitor), d. h. die nichtinvasive Erweiterung mehrerer bestehender Klassen um Funktionalität [GHJV95].[17] Im Beispiel werden Elemente in Abhängigkeit ihres Typs gezählt (Aspekt: "`SelCountVisitor`"; Zeilen 7 – 8). Die globalen Informationen werden hier in der Aspektinstantiierung des abhängigen Aspekts gehalten und durch die "`accept`"-Methoden des vorausgesetzten Aspekts gepflegt. Die Extraktion des vorausgesetzten Advice-Codes in den Aspekt "`PrepareVisitor`" ist notwendig, um Codereplikationen und Übersetzungsfehler bei der Anwendung mehrerer Besucher-Aspekte zu vermeiden, die ebenfalls die "`accept`"-Methode in eine Klasse einfügen. Folglich muss auch die mehrfache Anwendung des vorausgesetzten Aspekts "`PrepareVisitor`" auf eine Klasse vermieden werden, in Abb. 3.10 wurde deshalb auf die Vererbungsbeziehung zwischen den Aspekten verzichtet. Die Aspekte sind unabhängig.

Die Problemstellung der nicht definierbaren Abhängigkeiten ergibt sich ebenfalls für einige Implementierungen des wiederverwendbaren Entwurfsmusters "Beobachter" (vgl. Abschnitt 2.2.2) [SLU05]. Die Rollen "Subject" und "Observer" werden in abgeleiteten Aspekten den entsprechenden Klassen zugeordnet. Die mehrfache Zuordnung einer Vererbungsbeziehung zu einer als Subjekt oder Beobachter deklarierten Klasse ist hier kritisch und muss vermieden werden. Aspekte, die diese Vererbung nicht veranlassen sind im Folgenden abhängig von dem Aspekt, der diese Beziehung deklariert.

In AOP besteht weiterhin die Möglichkeit, das Entwurfsmuster "Beobachter" mittels Hash-Tabellen und Aspektinstantiierung umzusetzen [HK02]. Die Hash-Tabelle speichert in diesem Fall die Beobachter eines Subjekts. Zur besseren Wiederverwendbarkeit der Programmlogik des Entwurfsmusters werden die Rollen "Beobachter" und "Subjekt" in abgeleiteten Aspekten entsprechenden Klassen durch leere Schnittstellen zugeordnet. Die Vererbungsbeziehung darf auch hier nicht mehrfach zugeordnet werden. Für diese Implementierung sind Typumwandlungen der mit einer leeren Schnittstelle versehenen Beobachter- und Subjektobjekte sowie Indirektionen der Hash-Tabelle die Folgen. *Unterschiedliche* beobachtende Typen müssen hier eine einheitliche Aktualisierungsschnittstelle bieten oder durch Laufzeittypinformationen ermittelt und differenziert aktualisiert werden. Diese Schnittstelle kann wiederum jeweils *einmalig* durch den abgeleiteten Aspekt typspezifisch eingefügt werden.

[17] Typabhängig werden auf Objekten durch den Besucher verschiedene Funktionen ausgeführt. Indirektionen ermöglichen den Besucher als Funktionsargument dieser Objekte [GHJV95].

3.4.4 Modularisierbarkeit und Merkmalorientierte Programmierung

FOP-Schichten kapseln Klassenfragmente, sog. Rollen[18], welche zur Bereitstellung eines abstrakten Merkmals der resultierenden Software kollaborieren. FOP basiert bei der Zusammenstellung der Kollaborationen auf der Zuordnung jeder Rolle zu *einer* anonymen Oberklasse.[19] Zugleich wird jede Methodenerweiterung auf *eine* anonyme Basismethode angewendet. FOP ist also im Gegensatz zu AOP auf statisches Crosscutting beschränkt, d. h. das Erweitern und Hinzufügen von Methoden und Modulen. Im Gegensatz zur AOP fehlen weiterhin Freiheitsgrade bei der Deklaration von Join-Points der Erweiterungen. Dies kann in FOP Codereplikation durch das Hinzufügen von identischen Codefragmenten zu verschiedenen Klassen und Methoden verursachen [ALRS05b]. Querschneidende Belange, welche derartige Codereplikation verursachen, werden als homogene querschneidende Belange bezeichnet. Belange, welche das Hinzufügen unterschiedlichen Codes zu verschiedenen Software-Einheiten erfordern, werden als heterogen bezeichnet und verursachen in FOP keine Replikation von Quellcode [CC04, ALS06].
Die Ursache für die Replikation von Code homogener querschneidender Belange in FOP wird in der statisch hierarchischen Struktur von FOP-Erweiterungen gesehen. Merkmale, deren Implementierungen nicht in einer hierarchischen Beziehung stehen, können folglich in FOP nur mit Codereplikation umgesetzt werden [MO04].
Durch die Beschränkung auf statisches Crosscutting bereitet FOP Probleme bei Merkmalen, welche den Kontrollfluss beeinflussen (dynamisches Crosscutting) oder Signaturerweiterungen von Methoden (statisches Crosscutting) erfordern.[20] Änderungen am Kontrollfluss oder an Methodensignaturen können einzig durch die Redefinition sämtlicher aufrufender Methoden erreicht werden [MO04]. Homogene querschneidende Belange erfordern die Vervielfältigung des erweiternden Codes für *jeden* Erweiterungspunkt. Klassische Beispiele für solche Belange sind Tracing oder Synchronisation [KHH+01, CC04, Spi05].

Ein Beispiel für die feingranulare Kapselungsmöglichkeit der FOP ist in Abb. 3.11 dargestellt. Die Abbildung zeigt ein Fragment, welches für die konfigurierbare Sortierung einer Liste verwendet wird. Dieser Code stellt für sich einen *Teil* einer semantisch vollständigen Kollaboration dar. Jedoch einzig durch die gemeinsame Anwendung mit weiteren Schichten wird die aufsteigende Sortierung der Liste erreicht. Eine variable Schachtelung von Modulen sichert hier die Definition von semantisch vollständigen Komponenten [BSR04].

```
1 refines class LinkedList{
2   //sort in ascending order
3   bool successor(Element* low, Element* high){
4     return !(*low<*high); }};
```

Abbildung 3.11: Code der FOP-Erweiterung für die Ordnung einer Liste.

[18] Elemente *einer* Klasse, die der Bereitstellung eines Systemmerkmals dienen (vgl. Abschnitt 2.2.5)
[19] Die Reihenfolge der Schichten zur Übersetzung legt die explizite Oberklasse fest.
[20] Signaturerweiterungen einer Methode erzeugen Codereplikation für deren weitere Verfeinerungen.

Durch die fehlende Quantifizierung[21] können in FOP Schnittstellen von Merkmalmodulen explizit definiert und geprüft werden. Diese Schnittstelle eines Moduls umfasst dabei bereitgestellte Klassen, eingeführte Member-Variablen und -Methoden sowie Methodenverfeinerungen.
FOP kann wie bei AOP für die Umsetzung von Entwurfsmustern, wie Strategie oder Besucher, ein vollständiges Offenlegen der Implementierung von Klassen vermeiden. Die in den Entwurfsmustern verfolgte Intention kann durch die Integration der entsprechenden Programmlogik in den Namensraum der Klasse umgesetzt werden und erfordert nicht die Veröffentlichung von Methoden, die durch die Erweiterung aufgerufen werden.[22] Zusammengesetzte FOP-Klassen können als Bestandteile einer Bibliothek ihre Implementierung gegenüber dem Kontext kapseln. Im Unterschied zu Jampacks können in der FOP-Umsetzung durch Mixin-Schichten die Verfeinerungen nicht auf private Elemente einer verfeinerten Schicht zugreifen. Die Vererbungshierarchie erzeugt FOP-Verfeinerungen in Unterklassen außerhalb des Namensraumes einer Klasse. Die Verkapselung von Methoden muss für die Umsetzung durch Mixin-Schichten demnach mindestens eingeschränkten Zugriff ermöglichen.[23]

3.5 Wiederverwendbarkeit

Wiederverwendbarkeit ist eine Anforderung an Software, deren Erfüllung es ermöglicht, programmierten Code in unterschiedlichen Kontexten anwenden zu können. Zusätzlich soll ein hoher Wert des einzelnen Moduls erreicht werden. Adaptierbare Komponenten bieten eine Lösung für gute Wiederverwendbarkeit [Mey97]. Bewertet werden muss demnach, ob Software-Module feingranular angepasst werden können und ob die einzelnen Module einen hohen Wert besitzen. Die Anpassung der Software-Einheiten muss in Bezug auf die Wiederverwendung auf syntaktischer und semantischer Ebene erfolgen. Syntaktische Anpassungen beinhalten die Unabhängigkeit von Gegebenheiten des Kontexts, z. B. von Typenbezeichnungen, die Anpassung auf semantischer Ebene umfasst die Veränderung der Funktionalität an die Bedürfnisse des wiederverwendenden Kontextes.
Das Vernachlässigen von Code bei der semantischen Anpassung impliziert den Umgang mit dem Fehlen von Modulen im restlichen Code.

3.5.1 Wiederverwendbarkeit und Entwurfsmuster

OOP ermöglicht Wiederverwendung durch Vererbung und Objektinstantiierung, d. h. Assoziation und Aggregation von Instanzen der Klassen [Weg90, Bos98, CE00]. Entwurfsmuster geben hierfür wiederverwendbare Konzepte vor. Die Klasse als Software-Modul der OOP stellt demnach die minimal wiederverwendbare Einheit dar. Klassen bieten einen hohen Wert, können jedoch additiv *weder syntaktisch noch semantisch* verändert, sondern durch Vererbung und Instantiierung nur gekapselt wiederverwendet werden. Klassen können zu Paketen zusammengestellt werden, die den Wert eines Moduls weiter erhöhen. Die im Paket enthaltenen Untermodule sind jedoch in ihrem Aufbau, ihrer

[21] deklarative, mengenbasierte Bestimmung von Erweiterungspunkten in AOP (vgl. Abschnitt 2.2.4)
[22] Die Implementierung der FOP durch Mixin-Schichten impliziert das Vererben von Methoden und Klassen in den Schichten. Jampacks setzen das Konzept durch Codeersetzung um.
[23] In Java und C++: "*protected*".

Interaktion und ihrer Funktion fixiert. Es findet keine Anpassung eines Moduls statt, sondern nur die Auswahl von Gruppen von Klassen bzw. von einzelnen Klassen, die interagieren müssen aber auch inkompatibel sein können.
Klassen enthalten i. A. viele Annahmen über den Kontext, z. B. Typbezeichnungen oder Methodensignaturen, welche die Klasse an den Kontext binden. Das verwirrte Auftreten von Belangen – auch von Entwurfsmustern – verstärkt diese Bindung. Meist ist nur das Konzept eines Entwurfsmusters wiederverwendbar [Bos98].
Durch die Notwendigkeit, Klassenbeziehungen, wie Vererbung, Assoziation oder Aggregation, zum Zeitpunkt des Entwurfs der Komponente festzulegen, kann Adaptierbarkeit in OOP einzig durch abstrakte Klassen ermöglicht werden. Die Klassen des neuen Kontextes müssen die von der Schnittstelle der wiederverwendeten Klasse deklarierten Typen implementieren. Dies ist nicht immer möglich, z. B. bei Bibliotheksklassen, oder verursacht Typkonvertierung.[24] Das Entwurfsmuster "Adapter" ermöglicht hier die Übersetzung der Schnittstellen der nicht anpassbaren Klassen und somit ihre Verwendung [GHJV95]. Diese Herangehensweise erfordert für *jede* Klasse einen eigenen Adapter, was die Komplexität der Software stark erhöht.

Die Wiederverwendung von Einheiten kann ihre Unabhängigkeit von optionalen Modulen erfordern. Diese Adaptierbarkeit ist in OOP ohne Replikation von Code nur auf der Seite der aufgerufenen Einheit (engl. called side) möglich. Die aufrufende Seite ist durch die Kapselung der Klassen nicht veränderbar. Die aufgerufene Seite, z. B. eine Methode, muss vorhanden sein, kann aber durch Polymorphie variieren. Die Methode muss, wenn ihre Funktionalität nicht benötigt wird, Standardwerte erzeugen, die den übrigen Code nicht beeinflussen [SvGB05]. Diese Anforderung ist nicht trivial oder allgemein lösbar und birgt Nachteile in Ressourcenbedarf und Performance. Die Vernachlässigung des Codes unabhängiger Belange ist in OOP demnach nicht möglich.

```
1 class ForwardIterator: public AbstractIterator{
2   ForwardIterator(LinkedList* toIter){...}
3   AbstractElement* getFirst(){...}
4   AbstractElement* getNext(){...}
5   bool isDone(){...}};
```

Abbildung 3.12: Codebeispiel für Adaptierbarkeit durch abstrakte Klassen.

Das Entwurfsmuster "Iterator" wird zur Verdeutlichung der betrachteten Merkmale herangezogen (vgl. Abb. 3.12) [GHJV95]. Iteratoren haben die Aufgabe, über Containerklassen, wie Listen oder Bäume, zu navigieren.
Für die Verwaltung der ADT muss die abstrakte Schnittstelle der Iteratorenklasse zum Zeitpunkt ihrer Erstellung festgelegt werden. Verschiedene Typen werden dazu im Quelltext fixiert, wie der Typ des verwalteten Containers (hier: "`LinkedList`"; Zeile 2) oder der Typ der iterierten Elemente (hier: "`AbstractElement`"; Zeilen 3 – 4).
Adaptierbarkeit und Wiederverwendbarkeit des Iterators sind im Folgenden einzig für Containerklassen möglich, welche die vom Iterator festgelegte Schnittstelle der "`LinkedList`" implementieren. Zudem müssen alle weiteren Typen, deren Objekte durch

[24] Typkonvertierungen können zu Fehlern während der Programmlaufzeit führen und sollten vermieden werden [Boo97]. Sie deuten auf Probleme der Programmiertechnik hin, variable Typen zu manipulieren.

den Iterator manipuliert werden, polymorph zu den Typen sein, die im Iterator spezifiziert wurden, hier z. B. zum Elementtyp "`AbstractElement`".
Die Klassenschnittstelle des Iterators gibt im Folgenden den statischen Typ der iterierten Elemente an, welcher nach der Iteration zur weiteren Manipulation der Elemente dient. Da, um variable Wiederverwendung zu erreichen, keine Annahmen über den Kontext der Wiederverwendung gemacht werden dürfen, und um die Zuweisung inkompatibler Elementtypen durch den Iterator zu vermeiden, wird die Schnittstelle der Elementklasse leer sein, d. h. keine anwendbaren Methoden deklarieren (z. B. "`void*`" in C++ oder "`Object`" in Java).[25] Im Anschluss haben iterierte Objekte den statischen Typ einer leeren Schnittstelle und sind kaum noch zu verwenden. Als Auswege verbleiben nur die Konvertierung des statischen Typs der Elemente, z. B. auf ihren dynamischen Typ, oder das invasive Verändern der Iteratorenklasse. Typumwandlungen können Laufzeitfehler verursachen. Invasive Änderungen erzeugen Codereplikation.

3.5.2 Wiederverwendbarkeit und Generische Programmierung

Klassen bieten in GP wie bei OOP einen hohen Wert. Durch ihre Komplexität sinkt ihre Einsetzbarkeit aufgrund der vielen inhärenten Entwurfsentscheidungen. Auch GP-Klassen können zu Paketen zusammengestellt werden und erlauben so Module mit einem größeren Wert als eine einzelne Klasse. Pakete dienen in der GP der Dekomposition. Die Pakete stellen auch hier nur Sammlungen von Klassen dar, die als Ganzes ausgewählt, aber nicht als Ganzes transparent angepasst werden können. Die Schnittstellen der Klassen können folglich inkompatibel sein.

Im Unterschied zu OOP sind GP-Klassen feingranular syntaktisch anpassbar. Die Konfiguration durch Typen des Kontextes wird im aufrufenden Code vorgenommen. Ihre Anwendung erfolgt durch Compiler-Erweiterungen des Template-Codes oder des Programmcodes zum Zeitpunkt der Programmübersetzung. Im Gegensatz zur OOP geben Templates für die manipulierten Objekte keine speziellen Schnittstellen vor, sondern prüfen die Schnittstelle der Parametertypen auf deren Verwendbarkeit. Dieses Vorgehen beschränkt die Typprüfung auf die Methoden, die selbst aufgerufen werden. Gebundene parameterbasierte Polymorphie wirkt an dieser Stelle hinderlich, da sie von den Parametertypen die Implementierung einer Schnittstelle erfordert. Das Voraussetzen einer Schnittstelle erzeugt ähnliche Nachteile, wie sie für OOP bereits betrachtet worden sind. Die wiederzuverwendende Einheit trifft Annahmen, die der Kontext erfüllen muss. Ist dies nicht möglich, sind zusätzliche Adapterklassen notwendig.

Der Template-Code wird mit finalen Typen instantiiert. Das hat zur Folge, dass die Schnittstellen des Templates keine implizite Typumwandlung von Argumenten in einen Supertyp verursachen. Dadurch sind sämtliche vor der Behandlung durch Templates auf das Objekt anwendbaren Operationen auch nach der Manipulation für das Objekt definiert. Die feingranulare, syntaktische Anpassbarkeit macht Typumwandlungen oder invasives Eingreifen nicht länger erforderlich. So kann der Compiler statisch die Korrektheit der Typzuordnungen überprüfen und es folgt die statische Typsicherheit [Bra04].
Die Kapselung der Klasse verhindert für GP die semantische Anpassung von Modulen und die Vernachlässigung von Code optionaler Merkmale. Auf semantischer Ebene

[25] Annahmen über verwaltete Module sind als Entwurfsentscheidungen zu sehen. Sie erzeugen Anforderungen an den Kontext, diese Annahmen zu erfüllen, was die Wiederverwendbarkeit mindert.

sind Templates i. A. *nicht* modular anpassbar. Sprachen, die Template-Spezialisierungen erlauben, bieten die Möglichkeit der Auswahl von semantisch variierenden Implementierungen. Spezialisierungen von Klassen-Templates erfordern Codereplikation nicht variierender Klassenelemente. Spezialisierungen von Template-Methoden erfordern Übergabeparameter für die Auswahl der Implementierung und unterschiedliche Signaturen der Varianten. Die variable Implementierung einer Methoden-Signatur ist nicht möglich.

```
1 template<class Conf>class ForwardIterator: public AbstractIterator<Conf>{
2   typedef typename Conf::FinalListType FinalListType;
3   typedef typename Conf::ToHandle ToHandle;
4   ForwardIterator(FinalListType* toIter): AbstractIterator<Conf>(...){}
5   ToHandle* getFirst(){...}
6   ToHandle* getNext(){...}
7   bool isDone(){...}};
```

Abbildung 3.13: Codebeispiel für Kontextunabhängigkeit von GP-Modulen.

Im Beispiel des Entwurfsmusters "Iterator" können die Typen der Listen und von iterierten Elementen statisch festgelegt werden (vgl. Abb. 3.13; Zeilen 1 – 3). Die Konfigurierung wird im Beispiel durch ein Konfigurationsdepot "`Conf`" gekapselt vorgenommen (Zeile 1). Die Methoden, z. B. "`getFirst`" (Zeile 5), geben im Folgenden Objekte des im Kontext spezifizierten Typs zurück. Es wird keine implizite Typumwandlung vorgenommen. Typkonformität der Template-Argumente hängt von den im Template verwendeten Methoden ab.

3.5.3 Wiederverwendbarkeit und Aspektorientierte Programmierung

Aspektorientierte Programmierung teilt die Software-Einheiten in Komponenten und Aspekte. In diesem Zusammenspiel können Aspekte die Komponenten semantisch an einen neuen Kontext anpassen. Das erhöht die Wiederverwendbarkeit dieser Komponenten stark, da sie keine variablen Entwurfsentscheidungen oder querschneidenden Belange kapseln müssen und feingranular angepasst werden können.
Optionale und variable Belange können in AOP auf der Seite eines Methodenaufrufs (engl. caller side) nichtinvasiv und modular beeinflusst werden. Das Vorgehen der Veränderung eines Moduls durch ein anderes benötigt jedoch mehrere Software-Einheiten (die Komponente und einen oder mehrere Aspekte), was als nicht erstrebenswert angesehen wird. Viele elementare Module müssen komponiert werden und erfordern somit einen erhöhten Aufwand bei der Komponentenverwaltung im Vergleich zum Wert des einzelnen Moduls. Einzelne Module sind in AOP nicht anpassbar.
Die Dekomposition des Komponentencodes in Pakete ist mit Einschränkungen verbunden. Probleme der Inkompatibilität von Klassen können in AOP durch Aspekte unterbunden werden. Sie dienen hier als Adapter zwischen den Komponenten und erhöhen, wie die Adapter-Klassen der OOP und GP, die Komplexität des Codes. Aspekte ordnen sich aufgrund ihrer Intention, *querschneidende* Belange zu kapseln, i. A. nicht in die Dekompositionshierarchie der Komponenten ein. Durch die mögliche Kapselung nichthierarchischer, querschneidender Belange gestaltet sich eine sinnvolle

Dekomposition von Aspekten häufig schwierig.

Die Anpassung von Aspekten ist in AOP nicht kontrolliert möglich. Schwierig gestaltet sich die deklarative Beschreibung von Join-Points der Aspekte. Aufgrund des Prinzips, Join-Points bezeichnungsbasiert in Pointcut-Ausdrücken zu bestimmen, folgt eine implizite Anforderung an den neuen Kontext, ebendiese Bezeichnungen wieder zu verwenden. Durch die deklarative Beschreibungsform bieten Aspekte eine bestimmte Form der Unabhängigkeit vom Kontext, welche die automatische Anpassung der Join-Points an das erweiterte Programm betrifft. Somit ist die Erweiterung unterschiedlicher Basisprogramme möglich. Dies ist jedoch nicht unproblematisch.
Das Prinzip der *deklarativen* Beschreibung von Join-Points sieht das Vernachlässigen nicht übereinstimmender Erweiterungspunkte vor. In einem neuen Kontext kann dieses Vorgehen zu Problemen bei der Evaluierung von Software führen. Es ist im Folgenden nur mühsam nachprüfbar, ob der Aspekt an den gewünschten Stellen gewirkt hat, sog. black-box-Wiederverwendung. Ebenso muss geprüft werden, ob der Aspekt an Stellen gewirkt hat, an denen er nicht vorgesehen war. Dieses Problem bei der Evaluierung der Wirkungspunkte von Aspekten wird als "Dilemma der Adaption" bezeichnet und ist aufgrund der großen Menge von potentiellen lexikalischen Join-Points[26] offensichtlich nicht zufriedenstellend lösbar [Lie04]. Die Alternative besteht in der vollständigen Analyse des Aspektes, sog. white-box-Wiederverwendung. Dabei müssen seine Pointcut-Ausdrücke analysiert und ihre lexikalischen Join-Points im Komponentencode ermittelt werden. Vollständiges Wissen über die erweiterte Komponentenstruktur sowie über die interne Umsetzung des Aspekts und anderer Aspekte sind notwendig. Generische Platzhalter in den Pointcut-Ausdrücken und verfeinerte Pointcut-Ausdrücke, z. B. "`cflow`", erschweren beide Vorgehensweisen zusätzlich, denn sie erfordern Wissen über den Programmablauf vieler Methoden.

Werden Aspekte nicht angewendet, so kann der Code optionaler Merkmale vollständig vernachlässigt werden. Diese *positive* Eigenschaft erfordert explizite Beachtung von Interaktionen zwischen den Modulen. Durch die mangelhafte Merkmalkohäsion können in AOP fehlende vorausgesetzte Aspekte zum Fehlschlagen oder der fehlerhaften Anwendung erweiternder Aspekte führen. Dieses Problem der Abhängigkeit von der Anwendung bzw. dem Erfolg vorausgesetzter Aspekte folgt aus dem Dilemma der Adaption. Es wird als Problem optionaler und korrelierender Systemmerkmale (engl. feature optionality problem, feature interaction problem; FIP) bezeichnet [LARS05, LBN05, Pre97].
Das FIP äußert sich in AOP durch das Fehlen von lexikalischen Join-Points, die von vorausgesetzten Aspekten hätten definiert werden müssen. Durch die deklarative Beschreibung der Erweiterungspunkte eines Aspekts im Pointcut-Ausdruck kann der Weber nicht zwischen der beabsichtigten und der fehlerhaften Abwesenheit eines lexikalischen Join-Points unterscheiden. Trotz korrekter Übersetzung ist ein fehlerhaftes Systemverhalten möglich, z. B. bei unvollständiger Anwendung von Synchronisation.

Einfache AOP vermag eine Software auf semantischer Ebene anzupassen. Die Anpassung an neue Typen in gegebenen Codefragmenten ist jedoch nicht modular möglich. Der Umgang mit neuen Typen erfordert invasives Eingreifen in vorhandenen Code oder Codereplikation. Durch OOP-Mechanismen kann neuen Typen in AOP auch durch Subtyp-

[26] statische Repräsentation von Ereignissen des Programmablaufs im Quellcode (vgl. Abschnitt 2.2.4)

Polymorphismus begegnet werden. In diesem Fall treten die gleichen Probleme auf, die für die OOP betrachtet worden sind, z. B. Typkonvertierungen (vgl. Abschnitt 3.5.1).

Der im Beispiel wiederzuverwendende Iterator kann durch das Einfügen entsprechender Methoden- und Advice-Definitionen semantisch angepasst werden, um z. B. unterschiedliche Iterationsreihenfolgen zu unterstützen. Die Verwaltung von Elementen und Listen unterschiedlicher Typen erfordert die Replikation der Iterationslogik.

3.5.4 Wiederverwendbarkeit und Merkmalorientierte Programmierung

FOP-Merkmalmodule können semantisch und feingranular auf Methodenebene angepasst werden, ohne auf zusätzliche Module zurückgreifen zu müssen. Durch die mögliche Schachtelung von FOP-Modulen kann ein Modul mehrere kleinere Module zusammenfassen und so selbst einen hohen Wert besitzen. Zusätzlich können Klassen ähnlich zu Paketen der OOP und GP modularisiert werden. Die Software wird jedoch primär anhand von abstrakten Systemeigenschaften dekomponiert bzw. die einzelnen Module werden dahingehend zusammengefasst. Im Gegensatz zu OOP und GP sollen die Elemente, d. h. die Merkmalmodule, *unabhängig* komponiert werden und keine Interaktionen, d. h. auch keine Inkompatibilitäten, besitzen.[27] Die Komposition der FOP-Module erzeugt ein angepasstes Modul.

Die Join-Points von Merkmalmodulen sind im Gegensatz zur AOP nicht deklarativ und generisch, sondern explizit festgelegt. Dadurch können die lexikalischen Schatten, also die Wirkungspunkte im Ausgangsquelltext, sehr gut vorhergesagt werden. Die fehlerhafte Anwendung von Merkmalmodulen ist auf syntaktischer Ebene simpel erkennbar. Sind z. B. verfeinerte Methoden oder Klassen nicht definiert, liegt ein Fehler vor.[28] Das Vorgehen schränkt durch diese explizite Abhängigkeit von Elementen die Wiederverwendbarkeit der Module auf eine Software-Familie ein. Aspekte der AOP können teilweise auch auf völlig andere Programmfamilien angewendet werden.
Die Möglichkeit der Vernachlässigung von Code optionaler Belange ist *positiv*. Sie verlangt wie bei AOP die besondere Beachtung von Interaktionen zwischen den Merkmalmodulen. Durch das Vernachlässigen von optionalen Merkmalen bzw. Schichten können referenzierte Elemente fehlen und zum FIP führen. Diese Abhängigkeit von Systemmerkmalen bzw. deren Implementierungen mindert die Wiederverwendbarkeit einzelner Module. Das FIP kann sowohl syntaktische als auch semantische Fehler verursachen.
Syntaktische Probleme können in FOP verursacht werden durch das Fehlen von:

1. Typen von Variablen oder statische Referenzen,
2. Typen aus Methodensignaturen,
3. Oberklassen,
4. Gerufenen Methoden,
5. Verfeinerten Klassen oder

[27] Im Gegensatz zur OOP und GP sollen in Paketen ausgewählte Module nicht direkt zusammenarbeiten, da es durch den Sprachentwurf möglich ist, querschneidende Belange zu vermeiden.

[28] AOP nimmt hier das kalkulierte Fehlschlagen der Aspektquantifizierung an und ignoriert den Fehler.

6. Verfeinerten Methoden.

Semantische Fehler entstehen durch unerwartetes Verhalten von aufgerufenen Methoden und können nur schwer automatisch erkannt werden. Dies kann auf das Fehlen von Schichten zurückgeführt werden, die keine neuen Konstanten, sondern einzig zusätzliches Verhalten zu bestehenden Konstanten hinzufügen.

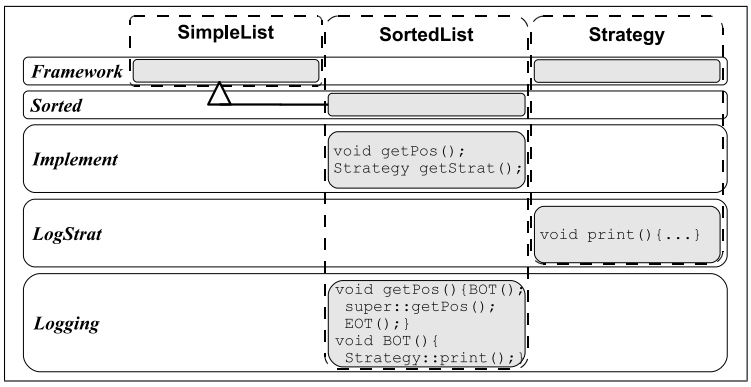

Abbildung 3.14: Beispiel für das Problem korrelierender Merkmale in FOP.

```
1 //Schicht ElementA
2 refines class Iterator{
3   ElementA* getFirst(){...}
4   ElementA* getNext(){...}};
5 //Schicht ElementB
6 refines class Iterator{
7   ElementB* getFirst(){...}
8   ElementB* getNext(){...}};
```

Abbildung 3.15: Beispiel für FOP-Codereplikation.

Zur Demonstration wird ein komprimierendes Beispiel mit allen FIP-Ausprägungen betrachtet (vgl. Abb. 3.14), in realen Anwendungen treten diese Fehler seltener auf. Die Abbildung zeigt ein Kollaborationenmodell zur Darstellung merkmalorientierter Schichtenentwürfe. Die sortierte Liste wird hier aufbauend auf OOP durch das Entwurfsmuster "Strategie" realisiert und erbt von einer einfachen Liste (Schicht "`Sorted`") [GHJV95]. Fehlt diese Schicht, so versucht die Schicht "`Implement`", die nicht vorhandene FOP-Konstante "`SortedList`" zu verfeinern (Ursache 5). Ist die Kollaboration "`Implement`" nicht Teil einer Konfiguration, so versucht die Erweiterung "`Logging`" die nicht definierte Methode "`getPos`" zu erweitern (Ursache 6). Fehlt die Schicht "`LogStrat`", so referenziert die Klasse "`SortedList`" in der Schicht "`Logging`" eine nicht vorhandene Methode der Klasse "`Strategy`" (Ursache 4). Fehlen die Schichten "`Framework`" und "`LogStrat`", so wird, neben den bereits genannten Fehlern, gar eine nicht definierte Klasse "`Strategy`" referenziert (Schicht "`Logging`"; Ursache 1). Die Klasse "`SortedList`" versucht in diesem Fall auch, von einer nicht vorhandenen Oberklasse zu erben (Ursache 3). Weiterhin referenziert die Signatur der Methode "`getStrat`" (Schicht "`Implement`") einen nicht vorhandenen Typ (Ursache 2).

Die Ursache für das FIP wird in der Kapselung verschiedener Merkmalinteraktionen in einer Software-Einheit gesehen [LBN05]. Eine weitere mögliche Ursache kann in der Veränderlichkeit der Reihenfolge der FOP-Schichten gesehen werden, ein stets nur erweiterter FOP-Stack vermeidet das FIP. Dieses Vorgehen ist jedoch sehr unflexibel und erzeugt Codereplikation. Die zur Überwindung des FIP vorgeschlagenen Ansätze bieten bisher noch keine befriedigende Lösung:

- Die Aufteilung der Merkmalmodule in Teile, die von anderen Modulen abhängig bzw. unabhängig sind, widerspricht der Merkmalkohäsion [LBN05, Pre97, LARS05].

- Ein teilweises Ignorieren von Fehlern oder das Einführen von Aspekten lässt keine Unterscheidung zwischen versehentlichem und absichtlichem Fehlen von Code zu [Ros05, LARS05].

- Zusätzliche Schlüsselworte, z. B. "`optional`", verlieren durch rekursive Abhängigkeiten von optionalen Bestandteilen ihre Aussagekraft in tiefen Schichten [LARS05]. (Elemente, die optionale Elemente direkt oder indirekt referenzieren, müssen selbst optional deklariert werden. Die Wahrscheinlichkeit hierfür wächst in tiefen Verfeinerungsschichten.) Weiterhin wird die Annotation des Quellcodes als problematisch empfunden, da veränderliche Anforderungen ständige Veränderungen der Beschreibungen erfordern könnten.

Ebenso wie OOP und AOP sind auch FOP-Module nicht robust gegenüber Veränderungen manipulierter Typen. Die Typen müssen zum Zeitpunkt der Erstellung der Merkmalmodule festgelegt werden. Das erfordert Typumwandlungen oder Codereplikation für den Umgang mit unbekannten Typen.

Eine Gruppe entsprechender Module ist für das Entwurfsmuster "Iterator" in Abbildung 3.15 dargestellt. Die Klasse wird im Beispiel entsprechend der im Kontext verwendeten Typen erweitert. Die Zeilen 2 – 3 bzw. 7 – 8 zeigen die Codereplikation, die durch Typvarianten der iterierten Elemente verursacht wird. Durch die Möglichkeit der feingranularen Erweiterung auf Methodenebene muss nicht die gesamte Klasse wie bei OOP repliziert werden, vermeiden lässt sich Codereplikation aber nicht.

3.6 Erweiterbarkeit

Die zentrale Frage der Erweiterbarkeit ist die, inwieweit der Software-Entwurf verändert und vervollständigt werden kann. Entscheidungen in dieser Entwurfsphase einer Software-Entwicklung betreffen u. a. die Modularisierung, d. h. welche Module erstellt werden sollen, deren Aufbau und wie sie interagieren [Dum01]. Weiterhin werden die zu verwendenden Programmiertechniken gewählt.
Erweiterbarkeit erfordert die Möglichkeit der Konkretisierung und Redefinition dieser Entscheidungen. Zur Bewertung der Erweiterbarkeit wird demnach die Möglichkeit einer Programmiertechnik untersucht, den Software-Entwurf inkrementell verfeinern zu können, sog. Software-Evolution [Boo97]. Erweiterbarkeit steht in engem Zusammenhang mit Modularisierbarkeit, da nur additive, modulare Erweiterungen in einer flexiblen Software resultieren (vgl. Abschnitt 2.1.5).

3.6.1 Erweiterbarkeit und Entwurfsmuster

OOP fasst Daten und Funktionen zu Objekten zusammen und beschränkt durch Kapselung den Zugriff auf diese. Diese Kapselung verhindert die Veränderung von Klassen nach ihrer Erstellung. Querschneidende Belange erfordern das verteilte Hinzufügen von Erweiterungen zu mehreren Klassen durch explizite Vererbung. Das erzeugt Codereplikation ebendieser Erweiterung.
Erweiterungen können durch Vererbung inkrementell vorgenommen werden, jedoch nur an finalen Klassen der Vererbungshierarchie. Die Erweiterung einer Oberklasse durch

Vererbung erzeugt die Replikation ihrer Unterklassen, da ihnen die Erweiterung nicht vererbt wird. Weiterhin sind die Erweiterungen entlang eines Vererbungspfades nicht variabel kombinierbar, da eine Oberklasse explizit und statisch im Quellcode angegeben werden muss. Unabhängige Erweiterbarkeit von Typen lässt sich nur durch Assoziation umsetzen, z. B. durch das Entwurfsmuster "Dekoration". Das führt zu Indirektionen und vielen Standardmethoden ohne Wert (vgl. Abschnitt 3.3).
Die Einschränkung der linearen Erweiterung von der finalen Klasse verlangt schon im Software-Entwurf die Planung und Vorbereitung der Erweiterungen durch Polymorphie und Assoziation. Varianten, welche in der Entwurfsphase einer Klasse nicht derart vorbereitet wurden, verursachen im Folgenden die Codereplikation von Erweiterungen. Der Code ist mit dem Code anderer Belange verwirrt.

Die modulare Erweiterbarkeit von Klassen beinhaltet die Redefinition von Methoden. Um einer Methode zusätzliche Programmschritte hinzuzufügen oder um variable und erweiterbare Implementierungen von Methoden unter einer einheitlichen Schnittstelle zu unterstützen, müssen diese Methoden in der OOP als virtuell deklariert werden. Da zum Entwicklungszeitpunkt einer Klasse später notwendige Redefinitionen nicht vorhergesagt werden können, müssen präventiv sämtliche Methoden als virtuell deklariert werden. Erfolgt dies nicht, können invasive Änderungen der Klasse notwendig werden, welche Codereplikation erzeugen. Die generelle Verwendung virtueller Methoden kann erhebliche Nachteile für die Performance und den Speicherbedarf der Software erzeugen (vgl. Abschnitt 2.1.7).

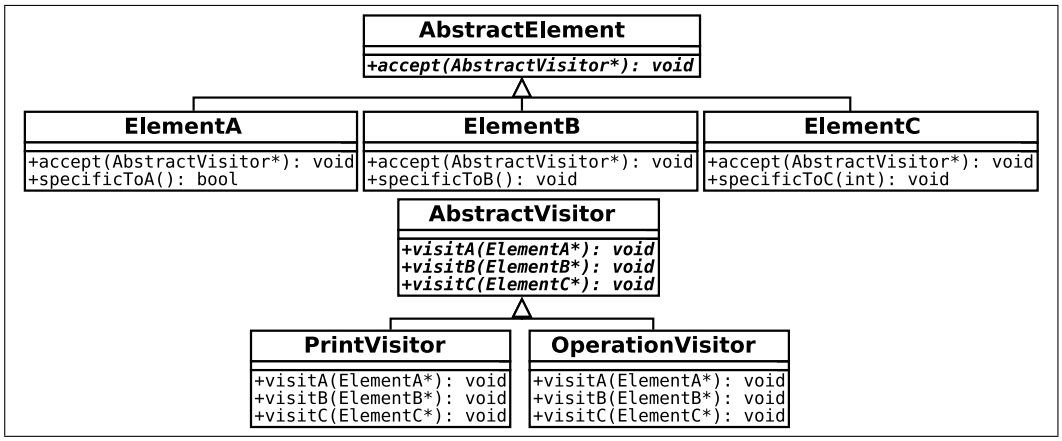

Abbildung 3.16: Darstellung des Entwurfsmusters "Besucher".

Durch spezielle Entwurfsmuster, wie "Besucher" (engl. visitor; vgl. Abb. 3.16), ist es in OOP möglich, bestehende Klassen additiv ohne Codereplikation oder Einführung einer Unterklasse um neue Funktionen zu erweitern [GHJV95].[29] Das Hinzufügen von Methoden zu Klassen erfolgt danach durch die Definition einer Besucherklasse (hier: "`AbstractVisitor`"), welche die zusätzlichen Funktionen für die zu erweiternden Klassen beinhaltet. Dazu wird in jede Klasse eine Indirektion eingeführt, welche je nach dynamischem Typ des Objekts eine Methode im Besucherobjekt initiiert.
Die Besucherobjekte können durch Subtyp-Polymorphie einheitlich durch die "`accept(...)`"-Methode der erweiterten Klasse behandelt werden (Die polymorphen Be-

[29] Die Besucherklasse stellt eine sog. operationengetriebene Modularisierung dar, im Gegensatz zur sog. datengetriebenen Modularisierung durch Klassen und Vererbung [Tor04].

sucherklassen "`PrintVisitor`" und "`OperationVisitor`" parametrisieren die "`accept`"-Methode der erweiterten Klassen "`ElementA`", "`ElementB`" und "`ElementC`". Diese Methode initiiert Aktionen des Besucherobjekts spezifisch zum eigenen Objekttyp.).

Durch Vererbung können Datentypen modular hinzugefügt werden, jedoch ist die Erweiterung um Funktionen nicht unabhängig und modular möglich. Diese müssen u. U. verteilt über den Quellcode in die verschiedenen Klassen integriert werden. Die Umsetzung von Erweiterbarkeit ist durch Vererbung additiv, aber nicht immer modular möglich. Das Entwurfsmuster "Besucher" ermöglicht die modulare Erweiterung von Klassen um Funktionen, erschwert aber andererseits die modulare Erweiterung um neue Datentypen. Das Hinzufügen eines neuen Datentyps erfordert in diesem Entwurfsmuster die verteilte Anpassung aller Besucherklassen. Das Problem bei der gleichzeitigen Erweiterbarkeit um Typen und Funktionen wird als Ausdrucksproblem (engl. expression problem) bezeichnet [Tor04, Coo90, KFF98].

3.6.2 Erweiterbarkeit und Generische Programmierung

Templates bauen wie OOP auf dem Klassenprinzip auf. Der im Software-Entwurf festgelegte Aufbau einer Klasse und die Vererbungsbeziehungen zwischen den Klassen sind wie bei OOP gekapselt und nicht veränderbar. Templates verursachen demnach ebenso das Ausdrucksproblem (vgl. Abschnitt 3.6.1). Die Planung der Adaptierbarkeit in einer vorgelagerten Entwurfsphase ist notwendig.
Die statische Bindung der Templates wirkt sich negativ auf deren Erweiterbarkeit aus. In der Folge statischer Bindung mit dem variablen, finalen Typ ist der dynamische Typ eines Objekts stets gleich seinem statischen Typ. Typumwandlungen können daher fatale Folgen haben und machen es erforderlich, auch Erweiterungen der Software durch dieselben Parameter variabel zu gestalten wie die erweiterte Software. Auf diese Weise der rekursiven Konfigurierung schleppen sich Implementierungsdetails der erweiterten Software durch ihre Erweiterungen und referenzierenden Klassen. Die Konfigurierung dieser Details erfordert bei der Entwicklung von Erweiterungen implementierungsnahes Wissen über große Software-Teile.
Beispielsweise macht die Erweiterung um einen externen Iterator dessen Parametrisierung mit dem Elementtyp und dem Typ der parametrisierten Liste notwendig. Der Iterator kann im Folgenden bei ungebundener parameterbasierter Polymorphie überhaupt keine Annahmen über die von ihm manipulierten Typen treffen. Jede Festlegung auf eine minimale Schnittstelle bindet diesen Typ statisch und verhindert nachfolgende Erweiterungen des manipulierten Typs.

```
1 template<class Config_>class List{
2   typedef Config_::FinalListType FinalListType;
3   FinalListType* clone(){
4     FinalListType* temp = new FinalListType();
5     ((FinalListType*)this)->copyTo(temp);
6     return temp;} } ...};
```

Abbildung 3.17: Beispiel für Casting zur Erweiterung von GP-Modulen.

Durch die statische Bindung und fehlende Subtyp-Polymorphie kann es notwendig werden, dass einige Funktionen den finalen Typ der eigenen Klasse kennen müssen, um

die korrekten Methoden aufzurufen. Eine Lösung des Problems besteht darin, den finalen Typ den Oberklassen durch einen Template-Parameter bekannt zu geben (vgl. Abbildung 3.17; Zeilen 1 – 2). Im Folgenden kann vor dem Aufruf kritischer Methoden der "`this`"-Zeiger auf den übergebenen, finalen Typ verändert werden (Zeile 5). Diese Lösung erfordert stets eine Unterklasse, da die Oberklasse nicht mehr autark instantiierbar ist. Ebenso wie bei OOP ist auch für die Erweiterung von Methoden, die den finalen Typ benötigen, die Verwendung eines allgemeinsten Supertyps, wie "`Object`" oder "`void*`", möglich. Dies verursacht jedoch den Verlust der Typinformationen der Objekte und erfordert die Umwandlung der statischen Typen vor der weiteren Verwendung der Objekte.

Auch für die GP ist das Entwurfsmuster "Besucher" notwendig, um bestehende Klassen additiv um Funktionen zu erweitern. Indirektionen und eine schlechte Erweiterbarkeit um neue Typen sind wie bei OOP die Folge. Variable Besucher und Elementtypen sind jedoch ohne dynamisch gebundene Methoden, z. B. durch Template-Methoden, manipulierbar, was sich positiv auf den Ressourcenbedarf und die Performance der Software auswirkt.

3.6.3 Erweiterbarkeit und Aspektorientierte Programmierung

Komponenten können in AOP durch Aspekte additiv, modular und unabhängig mittels statischem Crosscutting um neue Methoden und Funktionalität erweitert werden. Das Hinzufügen von Komponenten ist ebenfalls möglich. Das Ausdrucksproblem der OOP und GP (vgl. Abschnitt 3.6.1) wird vermieden. AOP erfordert demnach *keine* vorgelagerte Entwurfsphase, sondern ermöglicht die Erweiterung des Software-Entwurfs.

Anhand einer deklarativen Beschreibung lexikalischer Merkmale kann ein kleiner Pointcut-Ausdruck in einem Aspekt eine große Menge von Join-Points für eine Erweiterung vorsehen. Die Beschreibung lexikalischer Gegebenheiten erzeugt eine enge Kopplung des Aspekts an das von ihm erweiterte Programm [TBG03, GB03]. Durch die enge Kopplung kann das Hinzufügen von Komponenten ungewollte Seiteneffekte und Fehler durch bereits vorhandene Aspekte erzeugen.
Die "falsche" Benennung von hinzugefügten Methoden oder Klassen kann deren Erweiterung durch den Aspekt ungewollt verhindern oder verursachen. Die Ungebundenheit der Aspektquantifizierung kann so unvorhersehbare Seiteneffekte auf Programmerweiterungen haben [TBG03, LHB05, ALS06]. Der Pointcut-Ausdruck muss u. U. bei jeder iterativen Erweiterung um Komponenten und Methoden neu definiert, d. h. teilweise repliziert, werden. Die statisch im Quelltext fixierte Vererbungsbeziehung zwischen Aspekten erlaubt hier keine unabhängige, iterative Erweiterung von angepassten Pointcut-Ausdrücken. Die explizite und unabhängige Definition eines jeden Join-Points in einem eigenen abgeleiteten Aspekt führt das Grundprinzip der Quantifizierung ad absurdum.
Weiterhin ist die vorsätzliche Einbettung eines für den Aspektweber anhand des Pointcut-Ausdrucks unterscheidbaren Merkmals in die Komponentenstruktur möglich. So kann mit nachträglich hinzugefügtem Code umgegangen werden. Diese Umsetzung erfordert jedoch u. U. die Veränderung bestehenden Komponentencodes, die Anpassung weiterer Aspekte und widerspricht dem Prinzip der Unwissenheit des Komponentencodes über die Existenz von Aspekten [GB03].
Die notwendige Einhaltung bestimmter Muster durch die Komponenten für die kor-

rekte Arbeitsweise von Aspekten wird Problem der abgesprochenen Muster (engl. arranged pattern problem) genannt [GB03]. Der Widerspruch zwischen der mangelhaften iterativen Erweiterbarkeit und der additiven, modularen Anwendbarkeit von Aspekten wird als Evolutionsparadoxon der Aspektorientierten Software-Entwicklung (engl. AOSD-Evolution paradox) bezeichnet [TBG03]. Iterative Erweiterungen sind in AOP nur bedingt möglich.
Das Problem bei iterativen Erweiterungen kann durch Beachtung der Anwendungsreihenfolge der Aspekte gelöst werden. Die Änderungen durch Aspekte sollen dabei an Programmausprägungen vorhergehender Entwicklungsstufen gebunden werden [LHB05]. Erweiterungen dürfen dafür nur in Aspekten umgesetzt sein, da hinzugefügte Komponenten in dieser zur Bindung der Aspektquantifizierung verwendeten Reihenfolge nicht erfasst sind. Das ist nicht immer möglich (vgl. Abschnitt 3.4).

Erweiterungen von Aspekten können Aspektvererbung erfordern, um Pointcut-Ausdrücke und Advice-Definitionen wiederverwenden oder redefinieren zu können. In AspectC++ und AspectJ können nur als abstrakt deklarierte Aspekte durch Vererbung erweitert werden [KHH+01, Spi05, ALS05b]. Die Erweiterbarkeit eines Aspekts muss damit vorbereitet werden. Abstrakte Aspekte sind im Folgenden nur noch *zusammen* mit konkreten Subaspekten anwendbar, welche die abstrakten Elemente definieren.
Eine Erweiterung von ererbten Advice-Definitionen ist in AspectC++ und AspectJ nicht direkt möglich [KHH+01, SGSP02, SLU05]. Das widerspricht dem Prinzip der Einheitlichkeit (engl. principle of uniformity), was sämtliche Produktbestandteile als erweiterbare Elemente vorsieht [BLS03, BSR04]. Named Advice bezeichnet den Vorschlag, Advice-Definitionen explizit, methodenähnlich zu benennen. In Subaspekten können diese Definitionen explizit verändert und erweitert werden [ALS05b]. Die Benennung von Advice-Definitionen wird nicht von allen AOP-Compilern unterstützt.

```
1 aspect Printer{
2   pointcut ElemA() = "ElementA";
3   advice ElemA(): void toCout(){...}
4   pointcut ElemB() = "ElementB";
5   advice ElemB(): void toCout(){...}};
```

Abbildung 3.18: Codebeispiel für Besucher mittels AOP.

Die für diesen Abschnitt gewählte Fallstudie des Entwurfsmusters "Besucher" kann in AOP meist allein durch das Einfügen von Methoden in die erweiterte Klasse realisiert werden. Zusammenfassendes, übergreifendes Wissen über besuchte Elemente kann zusätzlich Aspektinstantiierung oder die Verwendung einer neuen Klasse erfordern. Die Erweiterung durch Aspekte bedarf keiner Vorbereitung in den Komponenten. Unterklassen müssen ebenfalls nicht repliziert werden, wenn ihre Oberklasse additiv verändert wird. Die in eine Klasse durch Aspekte eingefügten Methoden müssen jedoch unterschiedliche Signaturen haben, um Mehrdeutigkeiten in der erweiterten Klasse und Übersetzungsfehler zu vermeiden.

Abbildung 3.18 zeigt eine Implementierung eines Ausgabebesuchers für die Klassen "`ElementA`" und "`ElementB`". In jede Klasse wird eine typspezifische, zusätzliche Ausgabefunktion eingefügt. Durch den Austausch äquivalenter Ausgabeaspekte können ohne Codereplikation Varianten der erweiterten Klassen erzeugt werden.

3.6.4 Erweiterbarkeit und Merkmalorientierte Programmierung

Durch anonyme Vererbung[30] von Fragmenten und deren Zusammensetzung zu einer gleichnamigen finalen Klasse ist die Anwendung von Kollaborationen für den übrigen Programmcode transparent. Die anonyme Vererbung erlaubt die unabhängige Erweiterung um statische Varianten ohne invasives Eingreifen in bestehenden Code. Die additive Veränderung einer Oberklasse erfordert daher in FOP keine Replikation der Unterklassen.
Das Hinzufügen von Schichten, d. h. von Klassen, Methoden, Methodenverfeinerungen und Member-Variablen, erlaubt die iterative und additive Erweiterung von Modulen *und* die Erweiterung um Module. Das Ausdrucksproblem und das Evolutionsparadoxon werden vermieden. Entscheidungen bezüglich des Entwurfs der Software sind somit nicht auf eine vorgelagerte, unteilbare Entwurfsphase beschränkt.

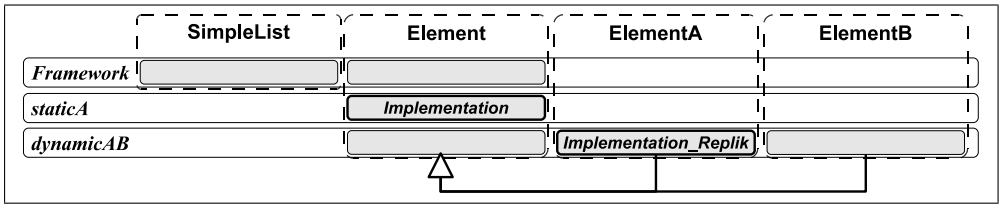

Abbildung 3.19: Beispiel für konfigurierbare Variantenbindung in FOP.

Die Erweiterung um Typen, die von Klassen des FOP-Stacks verwendet werden, ist ohne Typumwandlungen möglich. Dafür müssen jedoch die bereits verwendeten Typen des Stacks erweitert werden. Die neue Typbeschreibung muss also in den Stack integriert werden und tritt im Folgenden unter der im FOP-Stack erweiterten Typenbezeichnung auf.
Die Erweiterung um Typen, die während der Laufzeit polymorph zu den bestehenden sind, ist auch bei ihrer Integration in den FOP-Stack mit Codereplikation verbunden. Abbildung 3.19 veranschaulicht dies. Der vorhandene und durch die Liste manipulierte Typ "`Element`" kann statisch gebunden verwendet werden. Durch eine veränderte Bindung des Typs der durch die Liste manipulierten Elemente sollen nun unterschiedliche Typen während der Laufzeit verwaltet werden können ("`ElementA`" und "`ElementB`"). Der Elementtyp "`ElementA`" muss folglich ein Subtyp des durch die Liste verwalteten Typs "`Element`" werden und unter einem anderen Namen auftreten. Das erzeugt Codereplikation, wie die Hervorhebungen der identischen Implementierungen in Abb. 3.19 zeigen, da Erweiterungen explizit einer Klasse zugeordnet werden müssen.

Im Gegensatz zur AOP sind in FOP sämtliche Codefragmente redefinierbar. FOP erfüllt somit das Prinzip der Einheitlichkeit.

Die Umsetzung des Entwurfsmusters "Besucher" ist in FOP ohne zusätzliche Logik und Indirektionen durch eine direkte Manipulation der zu erweiternden Klassen möglich. Die entsprechenden Methoden werden durch Verfeinerungen hinzugefügt. Globales Wissen kann in Klassen gehalten werden, die als Bestandteil der Kollaboration gut modularisiert sind und folglich gut wiederverwendet und erweitert werden können.

[30] Deklaration des abstrakten, aber nicht des konkreten Typs der Oberklasse

3.7 Ressourcenverbrauch

Ressourcen sind ein knappes Gut und müssen sparsam verwendet werden. Eine dieser knappen Ressourcen ist der Speicherplatz. Um der Forderung nach geringem Speicherverbrauch gerecht zu werden, muss das Programm so einfach wie möglich gestaltet werden. In diesem Zusammenhang korreliert das Merkmal des Ressourcenverbrauchs stark mit der Möglichkeit zur Modularisierung unabhängiger Belange. Die Einfachheit der Software bezüglich des Speicherverbrauchs misst sich in der Zahl der abzuarbeitenden Programmschritte, deren zugehörige Prozessorinstruktionen im ausführbaren Programm hinterlegt sein müssen.
Adaptierbarkeit kann zusätzliche Programmlogik erfordern und Compiler-Optimierungen verhindern. Somit müssen die Programmiertechniken auch hinsichtlich ihrer *Abhängigkeit* von zusätzlicher Auflösungslogik zur Bereitstellung von Adaptierbarkeit und Erweiterbarkeit bewertet werden. Weiteren Einfluss hat die Fähigkeit der Programmiertechniken, nicht benötigte Programmlogik statisch von der Übersetzung auszuschließen und so die Zahl der in der übersetzten Einheit enthaltenen Instruktionen zu verringern.

3.7.1 Ressourcenverbrauch und Entwurfsmuster

Durch die Kapselung und die begrenzten Möglichkeiten der unabhängigen Erweiterung von OOP-Modulen (Vererbung und Assoziation) ergeben sich in OOP für viele Entwurfsmuster Nachteile infolge der Notwendigkeit von Indirektionen bzw. Standardmethoden zur Weiterleitung, z. B. Weiterleiten von Methodenaufrufen an ein Strategieobjekt [Bos98]. Des Weiteren ermöglicht OOP sowohl Adaptierbarkeit unter Beachtung der Modularität als auch Erweiterbarkeit einzig durch dynamische Bindung. Dynamische Bindung erhöht den Speicherbedarf durch zusätzliche Auflösungslogik und verhindert Compiler-Optimierungen bezüglich des Ressourcenbedarfs.
Die Kapselung von Klassen verhindert zusätzlich die Vernachlässigung nicht benötigter Programmlogik.

3.7.2 Ressourcenverbrauch und Generische Programmierung

Template-Instantiierung ermöglicht das Adaptieren der Software ohne die Auflösungslogik virtueller Methoden. So sind Compiler-Optimierungen möglich. Die statische Bindung der Template-Typen kann auf verschiedene Arten durch den Compiler umgesetzt werden:

1. Bei homogener Übersetzung wird ein Template-Modul durch den Compiler mit *einem* universellen Typ, wie "`void*`" oder "`Object`", instantiiert, sog. Löschung (engl. erasure) [BOSW98, Gho04, AFM97]. Des Weiteren werden durch den Compiler Typumwandlungen an den Stellen, wo das Template verwendet wird, eingefügt. Diese Umwandlungen erzeugen den kontextspezifischen Typ der Template-Klasse [CE00, BOSW98, OW97]. Diese Vorgehensweise wird auch als generische Ausdrucksweise (engl. generic idiom) bezeichnet [BOSW98].

2. Heterogene Übersetzung von Templates bedeutet das Replizieren der Template-Einheit. Die variablen Typenbezeichnungen werden hierbei für jede Klassenvariante durch die übergebenen Template-Parameter ersetzt [CE00, BOSW98, OW97].

Heterogene Übersetzung tendiert demnach dazu, größere Codemengen zu erzeugen, wenn viele unterschiedlich parametrisierte Objekte instantiiert werden. Das massenhafte Replizieren des Codes durch heterogene Übersetzung wird als Aufblähen von Code (engl. code bloat) bezeichnet [CE00]. Nicht parametrisierter Code muss wiederum nicht instantiiert werden [Str00, EBC00]. Dies mindert den Ressourcenverbrauch.[31]
Die Vernachlässigung des Codes nicht benötigter und nicht gekapselter Merkmale ist wie bei OOP durch das Kapselungsprinzip einer Klasse nicht möglich.

3.7.3 Ressourcenverbrauch und Aspektorientierte Programmierung

Durch die mögliche Vorverarbeitung des Codes in AOP können Adaptierbarkeit realisiert sowie unnötige Logik von der Übersetzung ausgeschlossen werden. Programmlogik zur späten Bindung von Methodenimplementierungen ist so in Fällen vermeidbar, in denen während der Laufzeit nur eine Variante benötigt wird. Zusätzlich werden speicheroptimierende Vorverarbeitungsschritte des Compilers möglich.
Der in Aspekten gekapselte Code wird durch den Compiler pseudoinvasiv[32] in den Komponentencode eingefügt. So können wertlose Indirektionen, wie für die OOP-Implementierung des Entwurfsmusters "Dekoration", vermieden werden. In der AOP ist daher auch die Vernachlässigung der Programmlogik dieser Weiterleitungen möglich. Die Umsetzung des Webens erfordert u. U. das Einfügen zusätzlicher Programmlogik, welche den Zugriff des Advice-Codes auf den Kontext des Join-Points ermöglicht [LST+06]. In Konfigurationen, die Laufzeitvariabilität erfordern, sind in AOP eventuell zusätzliche Ressourcen nötig.[33]

3.7.4 Ressourcenverbrauch und Merkmalorientierte Programmierung

Die FOP ermöglicht wie AOP das statische Vernachlässigen von Programmcode. Durch die feingranulare Anpassbarkeit der FOP auf Methodenebene kann eine differenzierte Auswahl das Übersetzen nicht benötigten Codes vermeiden. Die Kollaborationen, welche nicht benötigte Systemmerkmale implementieren, werden vollständig vernachlässigt.
Durch eine mögliche Vorübersetzung des Codes ist statische Adaptierbarkeit modular und ohne die Notwendigkeit virtueller Funktionen möglich. Auch FOP ermöglicht somit zusätzliche Compiler-Optimierungen, welche den Ressourcenbedarf senken.

[31] Beide Herangehensweisen finden Anwendung: C++ setzt heterogene Übersetzung von Templates um [Str00, BOSW98, Gho04]. Java wendet homogene Template-Übersetzung an [Bra04].
[32] pseudoinvasiv bezeichnet im Folgenden invasive Veränderungen von Modulen durch den Compiler
[33] z. B. AspectC++: Ein "$cflow$"-Pointcut-Ausdruck kann einen Zähler und verwaltende Logik des Zählers erfordern [LST+06].

3.8 Experimentelle Evaluierung

Die in den vorangegangenen Kapiteln ausgeführten theoretischen Aussagen bezüglich der Performance und des Ressourcenbedarfs sollen in diesem Kapitel experimentell evaluiert werden. Im Abschnitt 3.8.1 werden die Rahmenbedingungen der Experimente erläutert. Abschnitt 3.8.2 betrachtet Messungen bezüglich des Einflusses der Programmiertechniken auf den Ressourcenbedarf der Software. Die experimentelle Untersuchung bezüglich der Performance erfolgt in Abschnitt 3.8.3.

3.8.1 Experimentelle Rahmenbedingungen

In diesem Abschnitt werden die Bedingungen erläutert, unter denen die Experimente zur Evaluierung von Merkmalen der Programmierparadigmen durchgeführt wurden. Dazu gehören die Vorstellung der untersuchten Implementierung und der Ablauf der Experimente.

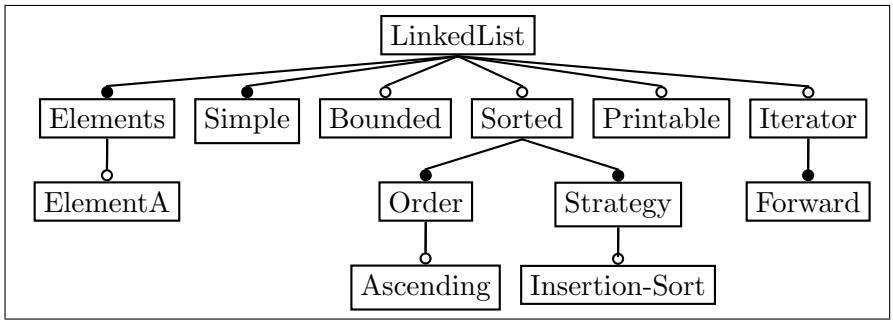

Abbildung 3.20: Darstellung der Konfiguration der experimentell untersuchten Liste.

Fallstudie. Für die im Folgenden analysierten Versuche wurde das durchgängige Beispiel der verketteten Liste in den Programmierparadigmen OOP, GP, AOP und FOP implementiert (vgl. Abschnitt 3.1). Die Implementierungen der Listen sollen die in Abschnitt 3.1 deklarierten Anforderungen erfüllen, d. h. additiv und unabhängig anwendbare Module für variable Merkmale besitzen. Eine Darstellung des Merkmalmodells der variablen und homogenen Merkmale der Programmfamilie wurde bereits in Abb. 3.1 dargestellt. Die für die Untersuchung festgelegte Konfiguration wird in Abb. 3.20 gezeigt.
Die Bewertung der Programmiertechniken nach ihrer Unterstützung von Laufzeitvariabilität wurde in Abschnitt 3.3 durchgeführt. Laufzeitvariabilität ist jedoch nicht immer erforderlich und sei an dieser Stelle nicht verlangt. In diesen Versuchen sollen die Kosten der Programmiertechniken evaluiert werden, die notwendig sind, um adaptierbare Software im Allgemeinen zu erstellen. Die folgenden Messungen beschränken sich daher auf statische Adaptierbarkeit der Applikationen.

Die Umsetzung der Listenimplementierungen in den einzelnen Programmiertechniken erfolgt entsprechend den schematischen Darstellungen in den Abbildungen 3.21 bis 3.23.

Die unterschiedlichen Listenkonfigurationen sollen es ermöglichen, unterschiedliche Elementtypen (hier: ElementA, ElementB und ElementC) ohne Codereplikation variabel zu verwalten. Die Programmiertechniken müssen dazu eine polymorphe Behandlung der verwalteten Elemente unterstützen. In OOP ist dies durch Subtyp-Polymorphie einer abstrakten Oberklasse möglich. In GP ist derartige Unabhängigkeit von Typen durch parameterbasierte Polymorphie bis zum Zeitpunkt der Übersetzung möglich. AOP und FOP gewährleisten die Variabilität eines Typs durch die entsprechende Verfeinerung des Elementtyps. Die dabei angewendeten pseudoinvasiven Änderungen der Elementklasse können als "induzierte Polymorphie" bezeichnet werden.
In OOP und GP wurden die Kernmerkmale der Liste (Sortierung und Größenbeschränkung) mithilfe des Entwurfsmusters "Dekoration" umgesetzt (vgl. Abschnitt 3.3.1). Dieses Muster erlaubt, Veränderungen von Klassen additiv, unabhängig und für die Kernlogik modular in einer konzeptionell umschließenden Klasse vorzunehmen. Durch die Notwendigkeit der Konsultation entstehen für jede Dekoration zusätzliche Speicherkosten für virtuelle Methoden und für die zu speichernden und zu durchlaufenden Standardimplementierungen der Methoden.[34] In GP entfallen für diese Implementierung die Kosten durch virtuelle Methoden. Die Logik einer Dekoration für Standardweiterleitungen kann jedoch aufgrund der Umsetzung des Klassenprinzips und der Vererbung nicht umgangen werden. In FOP und AOP wurden die Veränderungen durch Verfeinerungen der Methoden bzw. durch erweiternde Code-Advice-Definitionen erzeugt. Indirektionen und Standardimplementierungen für *alle* Methoden des dekorierten Objekts sind hier nicht notwendig. Indirektionen können, z. B. in FOP-Mixin-Schichten, für erweiterte Methoden und für Konstruktoren entstehen.

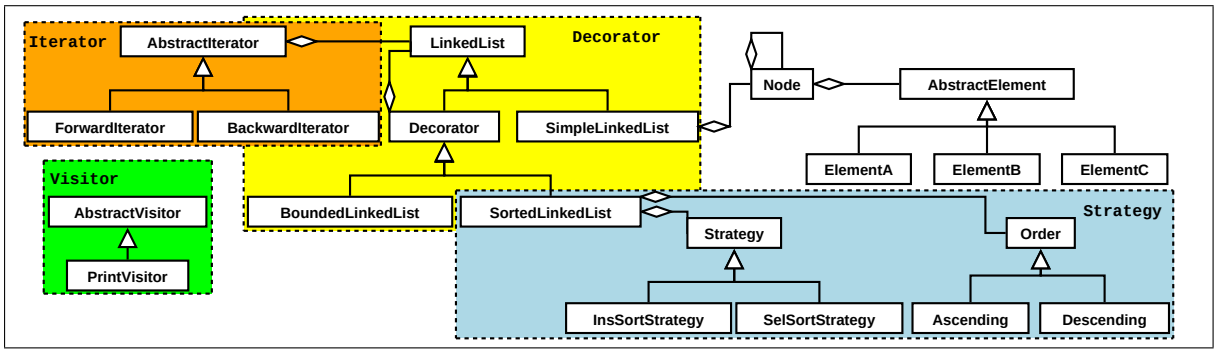

Abbildung 3.21: Darstellung der untersuchten OOP-Implementierung.

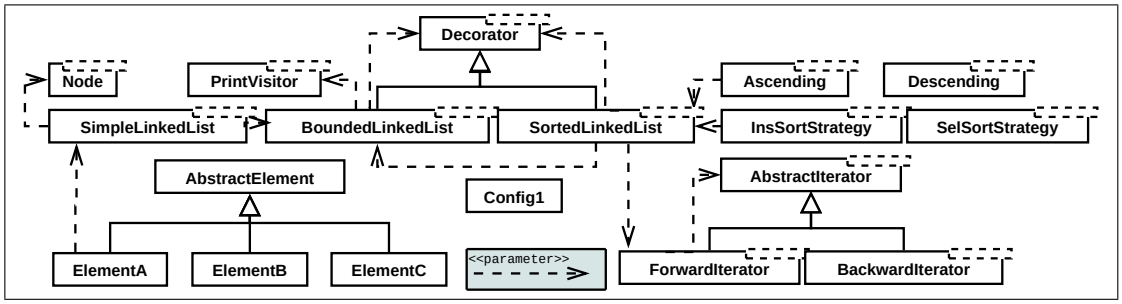

Abbildung 3.22: Darstellung der untersuchten GP-Implementierung.

[34] Jede Methode des dekorierten Objekts benötigt eine Weiterleitung in der umhüllenden Dekoration.

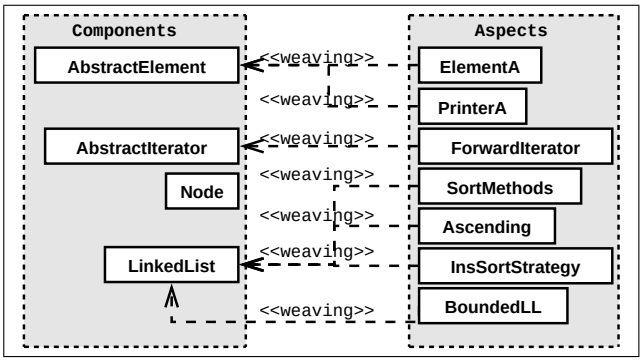

Abbildung 3.23: Darstellung der untersuchten AOP-Implementierung.

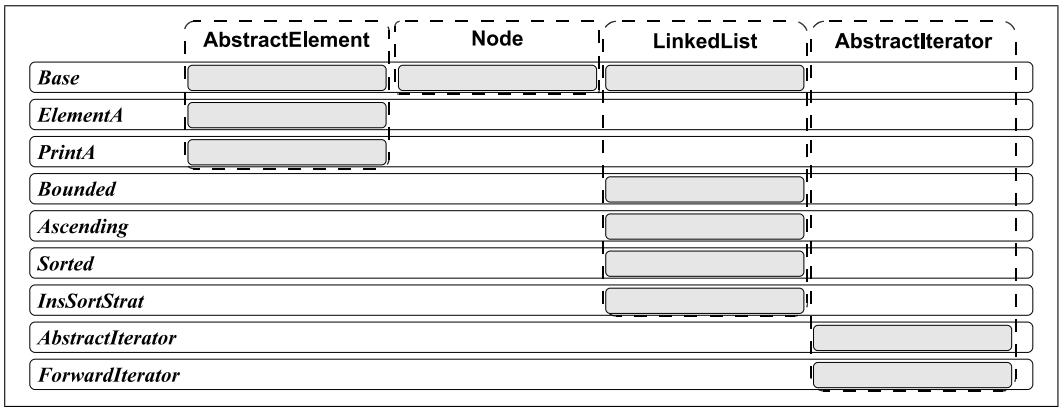

Abbildung 3.24: Darstellung der untersuchten FOP-Implementierung.

Die Untermerkmale des Produktmerkmals "`Sorted`" – "`Order`" und "`Strategy`" – stellen variable Eigenschaften der Dekoration "`SortedList`" dar und wurden aufgrund der guten Erweiterbarkeit in OOP und GP durch das Entwurfsmuster "Strategie" implementiert. In AOP und FOP bestehen die Möglichkeiten, derartige Adaptierungen durch Erweiterungen einzelner zugehöriger Methoden auszudrücken.
Das Hinzufügen optionaler Methoden zu bestehenden Klassen, z. B. eine Ausgabefunktion, ist in OOP und GP durch das Entwurfsmuster "Besucher" umgesetzt worden. Dieses erzeugt in OOP zusätzliche Logik durch Indirektionen und abstrakte Klassen (vgl. Abschnitt 3.6). In GP können die abstrakten Besucherklassen infolge der Verwendung von Template-Methoden in den besuchten Klassen "`ElementA`" bis "`ElementC`" vernachlässigt werden, die Template-Methode wird dafür mit dem finalen Typ des übergebenen Besuchers parametrisiert. Indirektionen bei der Konsultation des Besucherobjekts müssen dennoch in Kauf genommen werden. Für AOP und FOP stellen derartige Erweiterungen eine grundlegende Intention der Programmiertechniken dar (vgl. Mechanismus offener Klassen; Abschnitt 2.2.4). Entsprechende Methoden können additiv und ohne Indirektionen pseudoinvasiv in die zu erweiternden Klassen eingefügt werden.
Eine Navigation über die erzeugte Liste ist mittels des Entwurfsmusters "Iterator" möglich [GHJV95]. Die verschiedenen Element- und Listentypen sowie variable Iterationsreihenfolgen erfordern in der OOP abstrakte Schnittstellen der entsprechenden Klassen. GP vermeidet *abstrakte* Klassen durch die explizite Deklaration der zu verwaltenden, finalen Elementtypen im aufrufenden Code. Die Oberklasse "`AbstractIterator`" kapselt homogene Codefragmente ihrer Unterklassen. AOP und FOP ermöglichen durch pseu-

doinvasives Eingreifen die Veränderung des Elementtyps. In der Iteratorenklasse sind bezüglich der verwalteten Typen keine Unterscheidungen notwendig, da nur der interne Aufbau des Elementtyps verändert wird, die Typenbezeichnung bleibt erhalten. Dieses Vorgehen vermeidet abstrakte Klassen, ermöglicht jedoch nur einen Typ zur Verwaltung durch die Containerklasse "`LinkedList`" während der Laufzeit (vgl. Abschnitt 3.6.4).

Es ist zu anzumerken, dass die GP-Implementierung keinerlei explizite Assoziationen mehr beinhaltet, d. h. dass jede Klasse vollständig gegen eine abstrakte Template-Schnittstelle implementiert wurde, um Erweiterbarkeit und Variabilität der Software sicherzustellen. (Jede Bindung an einen expliziten Typ innerhalb einer Klasse erzeugt Probleme bei der variablen Zusammenstellung von Klassen.) Sämtliche Klassen außer den Elementklassen mussten in der GP-Implementierung parametrisiert werden. Ein Großteil dieser Parametrisierung ist auf die unterschiedlichen, zu verwaltenden Elementtypen zurückzuführen. Diese Parametrisierung schleppt sich durch die Erweiterungen. Die entsprechend benötigten Typen werden im Beispiel aus der neu eingefügten Klasse "`Config1`" entnommen, welche als Konfigurationsdepot sämtliche Template-Klassen parametrisiert.

Die Komplexität der OOP-Implementierung konnte in AOP und FOP stark reduziert werden. Die komplexen Klassenkonstellationen der verschiedenen Entwurfsmuster konnten in FOP beispielsweise linearisiert werden. (Ähnliche Erfahrungen berichtet Bosch in seinem Vorschlag einer schichtenbasierten Umsetzung ("LayOM") für verschiedene OOP-Entwurfsmuster [Bos98].)

Ablauf der Experimente. Der Code wurde übersetzt durch den Microsoft™C/C++ Compiler (Version 13.10.3077 for 80x86). Aspekt- und merkmalorientierte Erweiterungen wurden mittels des AspectC++ -Compilers (Version 0.9) bzw. des FeatureC++ -Compilers (Version 0.3) umgesetzt. Ein Intel Pentium M 1.5 GHz, 512 MB RAM mit dem Betriebssystem Windows XP wurde als Messplattform verwendet.
Die Software wurde entsprechend dem Experiment mit verschiedenen Optimierungsstufen des C++ -Compilers übersetzt:
Das Compiler-Argument "`/Od`" unterbindet Codeoptimierungen.
Die Software wird durch die Option "`/O1`" bezüglich geringen Speicherbedarf optimiert.
Die Performance einer Implementierung wird mit der Compiler-Option "`/O2`" optimiert.
Weiterhin wurden die verschiedenen Implementierungen mit der höchsten Optimierungsstufe "`/Ox`" übersetzt.
Im Folgenden werden besondere Rahmenbedingungen der einzelnen Versuche betrachtet:

Performance. Die Untersuchung der Software hinsichtlich der Performance erfolgte mit den Möglichkeiten der Standard-C-Bibliothek "time.h". Um Seiteneffekte, z. B. durch andere Prozesse, auszuschließen, wurden für jede Konfiguration 10 Messungen durchgeführt.
Eine einfache Anwendung verwendet die Liste. Die Anwendung fügt mehrere Elemente in die Liste ein. Ein Iterator wird verwendet, um eine formatierte Ausgabe der Liste zu simulieren.[35]

Ressourcenverbrauch. Um Wrapper- und Ladecode von der Betrachtung des Ressourcenverbrauchs auszuschließen, wurden Objektdateien statt ausführbarer Programme

[35] Um die Messergebnisse nicht zu verfälschen, wurden der eigentliche Ausgabebefehl und die dazugehörigen eingebundenen Header-Dateien vernachlässigt.

erzeugt. Im Anschluss wurden die Symboltabellen der übersetzten Einheiten mittels des "`strip`"-Befehls entfernt. Die Größe der Einheiten wurde durch das Programm "`size`" ermittelt (GNU strip/size 2.13.90 20030111).

3.8.2 Experimentelle Untersuchung des Ressourcenbedarfs

Das Ergebnis der experimentellen Untersuchung des Speicherverbrauches der in den einzelnen Programmiertechniken implementierten Software ist in Tabelle 3.1 dargestellt. Die Tabelle zeigt den Speicherverbrauch der unterschiedlich implementierten Software für die einzelnen Optimierungsstufen des C++ -Compilers.

	/Od	***/O1***	***/Ox***
OOP	6696	2932	3944
GP	4584	1824	3608
AOP	4404	1685	2804
FOP	4312	1921	2840

Tabelle 3.1: Messdaten des Speicherverbrauchs in Byte.

Die Messungen zeigen, dass die OOP-Implementierung in sämtlichen Optimierungsstufen den höchsten Speicherbedarf hat. GP kann durch die Verringerung der Klassenzahl und die Vermeidung von dynamischer Bindung, den Speicherbedarf deutlich senken. Für die nicht optimierte Konfiguration ("`/Od`") hatte die GP-Implementierung einen um 31% geringeren Speicherbedarf als die OOP-Variante. Dieser Vorteil steigt auf 37% gegenüber OOP für die auf Speicherplatz optimierte Konfiguration ("`/O1`"). Die in AOP und FOP umgesetzte Software vermag zusätzlich die Logik zu vernachlässigen, welche in der aktuellen Konfiguration nicht benötigt wird. Zudem sind durch pseudoinvasive Änderungen und Erweiterungen der AOP und FOP unabhängige Varianten von Klassen möglich. Dabei sind keine wertlosen Indirektionen notwendig. Das betrifft im Speziellen die Entwurfsmuster "Dekoration" und "Besucher". Für die betrachtete Implementierung wurde eine Verbesserung gegenüber OOP von 42% für die Konfiguration "`AOP-/O1`" erreicht. Die positiven Effekte lassen sich bereits in der nicht optimierten Konfiguration ("`/Od`") erkennen. Hier zeigt FOP den geringsten Speicherbedarf.
Die merkmalorientierte Spracherweiterung wird auf der Basis von Mixin-Schichten vorgenommen. Für diese Umsetzung ist u. a. zusätzliche Logik für die Propagation von nicht vererbbaren Klassenelementen notwendig.[36] Das erhöht den Speicherbedarf. Eine Umsetzung der FOP-Spracherweiterung durch Jampacks (vgl. Abschnitt 2.2.5) verspricht, diesen Nachteil zu vermeiden.
AOP erfordert zusätzliche Ressourcen für die Logik zur Umsetzung der Join-Point-Schnittstellen, um im Advice-Code auf den Kontext eines Join-Points zugreifen zu können [LST+06]. Diese Schnittstellen werden in AspectC++ durch Templates realisiert. Wird die Priorität auf den Speicherbedarf gelegt ("`/O1`"), so können neben den GP-Modulen auch diese Templates zusätzliche Vorteile aus C++ - und compilerspezifischen Optimierungen ziehen. Diese Optimierungen basieren teilweise nicht auf

[36] Java und C++ erlauben keine Vererbung von Konstruktoren. Die mixin-basierte Umsetzung der FOP erfordert vom Compiler das Kopieren der für ein Mixin deklarierten Konstruktoren in die jeweiligen Unterklassen. Der Konstruktor wird propagiert [CBML02, SB00].

dem Sprachentwurf von Templates und sind aufgrund ihrer Komplexität und Vielfalt nicht eindeutig zuzuordnen. Sie werden daher nicht für die Bewertung herangezogen.

Die experimentelle Untersuchung des Ressourcenverbrauches hat ergeben, dass ohne Compiler-Optimierungen die Umsetzung der Software in der Programmiertechnik FOP den geringsten Speicherbedarf hat. Die durch die Join-Point-API verursachten zusätzlichen Speicherkosten sind für die untersuchte AOP-Konfiguration höher als die zusätzlichen Speicherkosten durch Konstruktorpropagation der FOP. GP erlaubt die Vermeidung von virtuellen Methoden in variabler Software. Somit ermöglicht GP eine bessere Umsetzung der Software hinsichtlich des Ressourcenbedarfs als OOP. Die stets notwendigen Kosten der OOP aufgrund der zwingend notwendigen Umsetzung dynamischer Bindung sowie Kosten aufgrund der mangelnden Modularisierbarkeit können in Tabelle 3.1 nachvollzogen werden. OOP erzielt in jeder Optimierungsstufe das schlechteste Ergebnis hinsichtlich des Ressourcenbedarfs.

3.8.3 Experimentelle Untersuchung der Performance

In diesem Abschnitt sollen die bereits theoretisch analysierten Merkmale der Programmiertechniken bezüglich der Performance (vgl. Abschnitt 3.3) experimentell evaluiert werden. Die Tabelle 3.2 stellt dazu die mittleren Durchlaufzeiten der in den unterschiedlichen Programmiertechniken implementierten Programme dar. Verbesserungen in der Laufzeit, d. h. der Verkürzung der Abarbeitungsdauern, bedeuten auch Verbesserungen der Performance. Die Techniken werden in der Tabelle anhand der gemittelten Durchlaufzeiten den Optimierungsgraden ("`/Od`", "`/O2`", "`/Ox`") gegenübergestellt. Eine detaillierte Darstellung des Messergebnisses der Optimierungsstufe "`/Od`" ist in Abbildung 3.25 dargestellt. Die Säulen stellen die Durchlaufzeiten der Software je Programmiertechnik und je Messwiederholung dar.

	/Od	***/O2***	***/Ox***
OOP	120112	67137	70063
GP	112464	55239	55434
AOP	112964	51079	51304
FOP	107744	66606	67055

Tabelle 3.2: Messdaten der mittleren Laufzeit in Millisekunden.

Die Implementierung der OOP liefert in sämtlichen Optimierungsstufen den schlechtesten Wert der Messergebnisse, was für eine mangelhafte Umsetzbarkeit der Performance mittels der Programmiertechnik im Vergleich zu sämtlichen anderen untersuchten Programmiertechniken spricht. Diese Tatsache wird auf die direkten und indirekten Kosten dynamisch gebundener, virtueller Funktionen[37] sowie auf Indirektionen durch unterschiedliche Entwurfsmuster zurückgeführt.
GP ermöglicht durch die Vernachlässigung von virtuellen Funktionen eine Verkürzung der gemittelten Laufzeiten gegenüber OOP um 6% ohne Optimierung und um bis zu 20% mithilfe von Optimierungen ("`/Ox`"). Durch die Verwendung von AOP wurden mittlere

[37] Direkte Kosten virtueller Funktionen: Ermitteln der Implementierung während der Laufzeit; Indirekte Kosten: Unterbindung von Compiler-Optimierungen durch virtuelle Funktionen (vgl. Abschnitt 2.1.7)

Laufzeitverbesserungen gegenüber OOP von bis zu 26% ("`/Ox`") erreicht. Die nicht optimierte Konfiguration der Software erreichte eine Verkürzung der Laufzeit um 5%. FOP in der Konfiguration ohne Optimierung verkürzt die Laufzeit um 10%. Dieser prozentuale Wert der Verbesserung gegenüber OOP wird in den optimierten Konfigurationen nicht erneut erreicht.

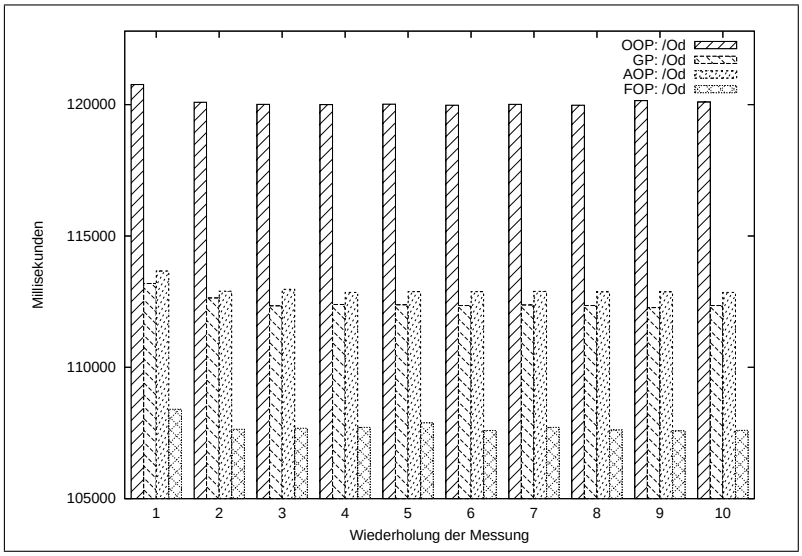

Abbildung 3.25: Detaillierte Ergebnisse der Laufzeitmessung ohne Optimierung (Compiler-Option "/Od").

In Tabelle 3.2 und Abbildung 3.25 sind starke Schwankungen der Messwerte bezüglich unterschiedlicher Optimierungsstufen und Durchläufe zu beobachten. Diese werden im Folgenden betrachtet.
Die sinkenden Messwerte der Laufzeiten mit steigender Anzahl der Durchläufe der Messungen könnten auf Caching-Effekte oder Speicherzugriffsstrategien der zugrunde liegenden Plattform zurückzuführen sein. Die Laufzeit nähert sich nach 1 – 3 Messdurchläufen einem Wert an und bleibt im Folgenden annähernd stabil.

Die Schwankungen der GP-Implementierungen bezüglich der unterschiedlichen Optimierungsstufen können auf C++- und compiler-spezifische Optimierungen zurückgeführt werden, wie z. B. heuristisches Inlining oder Vorberechnung von Ergebnissen.
AOP verursacht die Abarbeitung zusätzlichen Codes zur Manipulation eines Join-Points. Diese Join-Point-API wird in AspectC++ durch Templates umgesetzt und profitiert in den Optimierungsstufen "`/O2`" und "`/Ox`" ebenso wie GP von C++- und compiler-spezifischen Optimierungen. Unterschiede zwischen den beiden Programmiertechniken sind daher durch Indirektionen begründet. Diese erlauben in der GP unabhängige Variabilität und Erweiterbarkeit der Sender und Empfänger eines Methodenaufrufs. AOP fügt hingegen Indirektionen durch die Join-Point-API hinzu. In der durchgeführten Messung stellten sich die beiden zugehörigen Implementierungen ohne Optimierung als annähernd gleichwertig dar.
FOP kann wie AOP Indirektionen von Entwurfsmustern der OOP und GP vermeiden. Die Umsetzung der FOP durch Mixin-Schichten erfordert dennoch Indirektionen bei nicht vererbbaren Elementen, z. B. Konstruktoren. Die Kopie dieser Elemente in jede Verfeinerung einer Klasse erzeugt zusätzlich abzuarbeitende Indirektionen. Jampack-

Komposition verspricht, diesen Nachteil zu minimieren, da die verschiedenen Erweiterungen pseudoinvasiv auf eine Klasse angewendet werden und keine Zusatzlogik oder Indirektionen notwendig sind. Jampacks werden von FOP-Compilern bisher nicht ausreichend unterstützt, siehe FeatureC++. Die Umsetzung der FOP mittels Mixin-Schichten baut nach der Codetransformation durch den Precompiler FeatureC++ auf OOP und expliziter Vererbung auf. Diese Tatsache begründet die geringen Schwankungen der Vorteile der FOP-Konfiguration im Vergleich zur OOP-Konfiguration - die Laufzeitvorteile betragen hier 0,7 – 10%.

3.9 Fazit aus den einzelnen Programmiertechniken

In diesem Kapitel wurde der Einfluss der einzelnen Programmiertechniken OOP, GP, AOP und FOP auf die Erfüllbarkeit der Anforderungen an Software untersucht.
Die einzelnen Kapitel werden im Folgenden kurz zusammengefasst:

Zeitpunkt der Adaptierung. Die Analyse der Programmiertechniken bezüglich des Zeitpunktes der Adaptierung hat gezeigt, dass einzig die Subtyp-Polymorphie der OOP und bestimmte Pointcut-Ausdrücke der AOP modular eine Adaptierung während der Laufzeit ermöglichen. Die Adaptierung von Modulen der Programmiertechniken GP und FOP wird zum Zeitpunkt der Übersetzung gebunden. Durch den Compiler wird hier keine zusätzliche Logik erzeugt, welche eine Auswahl von Varianten während der Laufzeit ermöglichen könnte. Tabelle 3.3 bietet eine Übersicht.

OOP	**GP**	**AOP**	**FOP**
Laufzeit	Übersetzungszeit	Laufzeit	Übersetzungszeit

Tabelle 3.3: Zusammenfassung der Evaluierung des Konfigurationszeitpunktes.

Performance. Die Bewertung der Performance erfolgt aufgrund der in den Abschnitten 3.3 und 3.8.3 (theoretische bzw. experimentelle Evaluierung) untersuchten Merkmale der Programmiertechniken. Diese beziehen sich auf die Notwendigkeit zusätzlicher Logik, um konfigurierbare Software zu erstellen. Des Weiteren fließt in die Bewertung die Fähigkeit der Programmiertechniken ein, die Abarbeitung von Code ohne Wert zu vermeiden (vgl. Tab. 3.4).

	OOP	**GP**	**AOP**	**FOP**
Statische Methodenbindung	-	+	+	+
Indirektionen	-	-	+	+
Spezifische Kosten	+	+	-	-
Gemessen/Gesamt	-	0	0	+

Tabelle 3.4: Zusammenfassung der Evaluierung der Performance.[38]

[38] "+" = gute Erfüllbarkeit der Anforderung; "0" = teilweise Erfüllbarkeit; "-" = mangelhafte Erfüllbarkeit der Anforderung

Die OOP zeigt bezüglich der Performance deutliche Schwächen, da sie auf späte Bindung angewiesen ist und die Abarbeitung von Indirektionen benötigt. GP erlaubt die Vermeidung von später Bindung, Indirektionen zur Erstellung adaptierbarer Software sind auch hier notwendig. Die Umsetzung der AOP erzeugt Indirektionen, die den Zugriff auf den Join-Point ermöglichen. Äquivalent dazu erzeugt FOP spezifische Performancenachteile durch Konstruktorpropagation.
In der experimentellen Evaluierung verursachten für die nicht optimierte Konfiguration die Indirektionen der AOP ähnliche Messwerte für die Laufzeit wie die GP-Implementierung. Die zusätzlich eingefügten Indirektionen der FOP zeigten sich in der Messung als geringfügiger Nachteil. FOP unterstützt demnach die Forderung nach hoher Performance am besten.

Modularisierbarkeit von Belangen. OOP ermöglicht durch Kapselung und explizite Vererbung von Klassen die Modularisierung *eines* unabhängigen Belangs in einer Vererbungshierarchie. Sämtliche weiteren Belange, welche nicht auf die Vererbungshierarchie abgebildet werden können, sind querschneidend und erzeugen Replikation von Programmcode der veränderlichen Klasse sowie der jeweiligen Unterklassen. Aufgrund der Komplexität der Klasse als modulare Einheit und der expliziten Vererbung erzeugen auch kleine Variationen einer Klasse massive Codereplikation.
Templates trennen die Konfiguration der Module von ihrer Semantik. Die Klasse als Modularisierungseinheit und objekterzeugende Einheit und die Vererbung als Technik der Erweiterung bleiben jedoch erhalten. Das verursacht querschneidende Belange und somit Codereplikation.
FOP vermag die Kapselung minimaler Einheiten und bietet eine Möglichkeit zur unabhängigen Bestimmung des Aufbaus einer zugrunde liegenden OOP-Klasse. Viele unabhängige Belange können somit modularisiert werden. Dennoch erzeugt auch FOP Replikation von Code bei der Implementierung nichthierarchischer, querschneidender Belange (z. B. homogene querschneidende Belange).
AOP kann ebenso wie FOP feingranular Komponenten zusammensetzen. Im Gegensatz zu FOP sind durch AOP neben heterogenen auch homogene querschneidende Belange ohne Codereplikation umsetzbar. Auch Änderungen des Kontrollflusses erzeugen in AOP keine Codereplikation. Die Unterstützung von Merkmalkohäsion und Schnittstellendefinition ist dennoch mangelhaft.

	OOP	**GP**	**AOP**	**FOP**
Crosscutting concerns	-	-	+	0
Merkmalkohäsion	-	-	0	+
Verlässliche Schnittstellen	+	-	0	+
Gesamt	-	- -	+	+

Tabelle 3.5: Zusammenfassung der Evaluierung der Modularisierbarkeit.

Die Vermeidung von Codereplikation bzw. die Vermeidung querschneidender Belange sind als eine Voraussetzung für Merkmalkohäsion zu sehen, da repliziert vorliegender Code eines Belangs von vornherein die Voraussetzungen kohärenter Merkmalmodule verletzt. Auch Schnittstellen sind abhängig von der Modularisierung, da querschneidende Belange keine einzelnen Module bzw. Schnittstellen besitzen können. Aus diesem Grund

wird die Vermeidung von repliziertem Code bei der Bewertung der Modularisierbarkeit als wichtiger eingeschätzt als die Unterstützung von Merkmalkohäsion und Schnittstellendefinition.

Eine Übersicht bietet Tabelle 3.5. Es muss festgehalten werden, dass sämtliche Paradigmen die Forderung nach Modularisierbarkeit von Software nicht allumfassend erfüllen.

Wiederverwendbarkeit. Software-Einheiten sollen in vielen Kontexten verwendet werden können. Gleichzeitig soll die einzelne wiederverwendete Einheit einen großen Wert besitzen. Es wird als sinnvoll angesehen, dass Software-Module feingranular angepasst werden sollen und somit einen hohen Wert besitzen können.
Die syntaktische Anpassung an unbekannte Typen ist einzig in GP kontrolliert möglich. Das wird durch die Trennung der Typenkonfiguration von der Logik der GP-Komponenten erreicht. OOP, AOP und FOP erlauben keine kontrollierte syntaktische Anpassung der wiederzuverwendenden Einheiten.
FOP ermöglicht die feingranulare semantische Anpassung eines Merkmalmoduls an die Anforderungen des neuen Kontexts. In AOP sind Komponenten feingranular anpassbar, jedoch einzig durch Aspekte, d. h. zusätzlich zu verwaltende Module. OOP- und GP-Module sind i. A. nicht feingranular semantisch anpassbar.
Module der OOP, GP und FOP können einen hohen Wert besitzen.
Eine Übersicht ist in Tabelle 3.6 zu sehen.

	OOP	**GP**	**AOP**	**FOP**
semantische Anpassbarkeit von Modulen	-	-	+	+
syntaktische Anpassbarkeit von Modulen	-	+	-	-
Module mit hohem Wert	+	+	0	+
Gesamt	-	+	0	+

Tabelle 3.6: Zusammenfassung der Evaluierung der Wiederverwendbarkeit.

Die Erfüllung der Forderung nach Wiederverwendbarkeit wird für sämtliche Programmiertechniken als nicht zufriedenstellend erfüllt angesehen. Keine der Techniken vermochte, die feingranulare Anpassbarkeit einer Komponente mit der Kontextunabhängigkeit durch variable Typen zu verbinden. Eine Abwägung zwischen syntaktischer und semantischer Anpassbarkeit erscheint nicht sinnvoll.[39]

Erweiterbarkeit. Sowohl das Hinzufügen neuer Module als auch die Erweiterung bestehender wird durch FOP sehr gut unterstützt (vgl. Tab. 3.7). Die AOP ermöglicht eine effiziente Erweiterung von bestehenden Komponenten. Die Erweiterung von Software, welche bereits Aspekte beinhaltet, gestaltet sich für das Hinzufügen von Komponenten und Aspekten schwierig. OOP ermöglicht lediglich modulare Erweiterungen von Komponenten *oder* um Komponenten. Die statische Bindung der GP mindert die Erweiterbarkeit der Software, da Klassen für ihre Erweiterungen vorbereitet werden müssen (Parameter der finalen Klasse). GP-Implementierungen können wie OOP modular nur

[39]Mit den zwei Dimensionen der Skalierungsfläche impliziert Biggerstaff die Gleichwertigkeit beider Möglichkeiten zur Bewertung der Wiederverwendbarkeit (vgl. Abschnitt 2.1.4) [Big98].

	OOP	GP	AOP	FOP
Iterative Erweiterbarkeit	-	-	0	+
Unabhängige Erweiterbarkeit	-	-	+	+
Einheitliche Erweiterbarkeit	+	-	-	+
Gesamt	0	- -	+	++

Tabelle 3.7: Zusammenfassung der Evaluierung der Erweiterbarkeit.

um neue Komponenten *oder* um zusätzliche Funktionen bestehender Komponenten erweitert werden.

Ressourcenverbrauch. Das Vernachlässigen von Programmlogik und später Bindung vermindert den Ressourcenbedarf der Software. OOP benötigt späte Bindung, um Adaptierbarkeit und Erweiterbarkeit modular umzusetzen. Das Vernachlässigen von Programmlogik ist durch die Kapselung von Klassen nicht möglich. GP erlaubt die Vermeidung von Auflösungslogik virtueller Methoden. FOP und AOP erlauben das variable Zusammensetzen von Klassen und vermeiden so virtuelle Funktionen *und* nicht benötigte Programmlogik. AOP benötigt spezifische Logik zur Manipulation und ggf. Evaluierung des Join-Point-Kontextes. Für FOP-Mixin-Schichten ergeben sich zusätzliche Kosten durch die Propagation nicht vererbbarer Elemente.

	OOP	GP	AOP	FOP
Statische Methodenbindung	-	+	+	+
Standardmethoden	-	-	+	+
Spezifische Kosten	+	+	-	-
Vernachlässigung von Code	-	-	+	+
Gemessen/Gesamt	- -	-	0	+

Tabelle 3.8: Zusammenfassung der Evaluierung des Ressourcenbedarfs.

In der experimentellen Evaluierung (vgl. Abschnitt 3.8) erwies sich die stets erzeugte, zusätzliche Logik von AOP und FOP als weniger nachteilig als die zusätzlichen Klassen und Indirektionen der OOP und GP. Die Messwerte erfordern auch eine differenzierte Betrachtung der AOP und FOP.
Die Evaluierung bezüglich des Ressourcenverbrauchs ist in Tabelle 3.8 zusammengefasst.

Die Erfüllbarkeit der Nutzeranforderungen "geringe Entwicklungskosten", "kurze Entwicklungsdauern" und "Korrektheit" wurde indirekt durch die entwicklerseitigen Voraussetzungen ihrer Umsetzung bewertet. Die Möglichkeit der Modularisierung, der Wiederverwendung und der Erweiterung von Software sind Voraussetzungen der indirekt behandelten Anforderungen oder wirken sich förderlich auf sie aus.

Eine Gesamtübersicht der Resultate der einzelnen Untersuchungen ist in Tabelle 3.9 dargestellt. Diese Tabelle ermöglicht auch einen zusammenfassenden Blick auf die Fähigkeiten der Programmiertechniken:

- OOP bietet die Möglichkeit, den Zeitpunkt für Konfigurationsentscheidungen auf die Laufzeit zu verschieben. Dies führt im Zusammenhang mit mangelhafter Mo-

	OOP	GP	AOP	FOP
Zeitpunkt der Adaptierung	Laufzeit	Übersetzung	Laufzeit	Übersetzung
Performance	-	0	0	+
Modularisierbarkeit	-	- -	+	+
Wiederverwendbarkeit	-	+	0	+
Erweiterbarkeit	0	- -	+	++
Ressourcenverbrauch	- -	-	0	+

Tabelle 3.9: Zusammenfassung der Analyse.

dularisierbarkeit der Software zu Nachteilen hinsichtlich des Ressourcenverbrauchs und der Performance der Software. Die Unveränderbarkeit von komplexen Modulen der OOP wirkt sich nachteilig auf die Wiederverwendbarkeit der Software-Module aus. Die Erweiterbarkeit der Software ist aufgrund des eindimensionalen Erweiterungsmechanismus der Vererbung nur eingeschränkt möglich.

- GP erlaubt die Konfigurierung der Software bis zum Zeitpunkt der Übersetzung. Generische Module sind unabhängig vom Kontext und daher gut wiederverwendbar. Aufgrund der statischen Bindung und der Komposition in einer Klassenhierarchie fallen Erweiterungen von generischen Modulen schwer. Generische Programmierung erfüllt die Anforderung nach performanter und ressourcensparender Software nur ausreichend bis mangelhaft. Das Prinzip gekapselter Klassen und der Mechanismus der expliziten Vererbung bieten nur eine unzureichende Möglichkeit zur Modularisierung von Software.

- Ausgehend von einer guten Modularisierbarkeit der Software bietet die AOP die Möglichkeit, ressourcensparende und gut erweiterbare Software zu entwickeln. Indirektionen der Join-Point-API mindern die Performance der Software. Die Konfigurierung der Software ist während der Laufzeit möglich. In AOP ist jedoch nur ein Teil der Module gut wiederverwendbar.

- FOP ermöglicht eine gute Modularisierbarkeit der Software, was sich positiv auf die Performance und den Ressourcenverbrauch der Software auswirkt. Die Auswahl einer Konfiguration erfolgt zum Zeitpunkt der Übersetzung. Merkmalmodule sind gut wiederverwendbar und bieten basierend auf anonymer Vererbung eine sehr gute Erweiterbarkeit.

Es kann festgestellt werden, dass die in jüngster Zeit viel diskutierten Programmiertechniken AOP und FOP hinsichtlich der untersuchten Anforderungen gute Lösungsansätze für Probleme konventioneller Techniken wie OOP und GP bieten.
Weiterhin ist festzustellen, dass für einige Anforderungen, wie Modularisierbarkeit von Belangen oder Wiederverwendbarkeit, keine optimale Lösung erzielt werden konnte.
Wie die obige Aufstellung zeigt, besitzt keine der einzelnen Programmiertechniken genügend Sprachmittel, um andere Programmiertechniken vollständig zu übertreffen. Oft erfüllen die Techniken nur disjunkte Aspekte einer Anforderung und werden in der Zusammenfassung als gleichwertig bezüglich dieser Anforderung bewertet.

Auch in dieser Analyse konnte beobachtet werden, dass die Modularisierung von Software nach mehreren Belangen großen Einfluss auf sämtliche der hier betrachteten

Anforderungen hat. Wiederverwendbarkeit hängt von der Menge der Entwurfsentscheidungen ab, die potentiell in Konflikt stehen können. Entwurfsentscheidungen können als Merkmal eines Belangs gesehen werden und sind somit abhängig von der Möglichkeit der Modularisierung der Software nach vielen Belangen. Erweiterbarkeit beruht ebenso auf additiver modularer Redefinition von Belangen ohne Codereplikation.
Die Untersuchung unterstrich die enge Kopplung der Nutzeranforderungen an diverse entwicklerseitigen Voraussetzungen und damit die Bedeutung der entwicklerseitigen Anforderungen an qualitativ hochwertigen Code.

Kapitel 4

Kombinationen von Programmiertechniken

Die Analyse zeigt, dass die einzelnen Programmiertechniken die Qualität der Software im Vergleich zur OOP nur teilweise verbessern können. Die Techniken vernachlässigen dafür andere Qualitätsanforderungen. Einzelne Anforderungen wie Modularisierbarkeit von Belangen können in sämtlichen Programmiertechniken nicht zufriedenstellend erfüllt werden.
Verschiedene Kombinationen der Programmiertechniken wurden vorgeschlagen, um deren Vorteile zu vereinen und ihnen inhärente Nachteile zu vermeiden [LBS04, ALS06, AKL06].
Das folgende Kapitel behandelt die Kombinationen Aspekt-Mixin-Schicht, Generisches Merkmalmodul und Generischer Advice. Dabei werden besonders die markanten Unterschiede zwischen den Kombinationen und den ihnen jeweilig zugrunde liegenden Einzeltechniken betrachtet. Eigenschaften, die sich einzig aus den Eigenschaften *einer* Technik ableiten lassen, werden nur kurz betrachtet. Weiterhin werden Hinweise für die zukünftige Betrachtung und Umsetzung der Kombinationen gegeben.

4.1 Aspekt-Mixin-Schichten

Aspekt-Mixin-Schichten (engl. aspectual mixin layers; AML) stellen eine Kombination der Aspektorientierten und Merkmalorientierten Programmierung dar [ALS06, ALRS05a, ALRS05b]. Mixin-Schichten sind eine Umsetzung der Merkmalorientierten Programmierung. In diesem Abschnitt werden zusätzlich auch Jampacks für die Umsetzung des FOP-Ansatzes betrachtet.
Aspekte werden neben Klassenerweiterungen als mögliche Elemente von FOP-Schichten in AML eingeführt. Die Wirkungsweise von Aspekten soll in AML auf zurückliegende Schichten begrenzt sein, um Seiteneffekte auf später hinzugefügte Komponenten zu vermeiden, sog. Bindung der Aspektquantifizierung (engl. bounding aspect quantification) [ALRS05a, ALS05a].

AOP überträgt ihre Eigenschaften hinsichtlich des Zeitpunktes der Adaptierung auf diese Kombination. Die Wiederverwendbarkeit wird durch FOP-Mechanismen bestimmt. Der Ressourcenbedarf wird durch die spezifischen Nachteile beider Techniken (Propagation von Klassenelementen der FOP; Join-Point-API der AOP) festgelegt.

Bezüglich der einzelnen Techniken sind folgende Unterschiede zu beobachten und Anmerkungen zu treffen:

Performance. AOP und FOP ermöglichen die Vernachlässigung von Programmlogik nicht benötigter Belange. Auch Indirektionen im Kontrollfluss, verursacht durch feste Modulbeziehungen oder virtuelle Funktionen, können vermieden werden. Die Umsetzung der Spracherweiterungen erfordert die Abarbeitung zusätzlicher Logik. AOP benötigt u. U. die Abarbeitung der Join-Point-API. Die Umsetzung von FOP-Erweiterungen durch Mixin-Schichten erfordert die Propagation nicht vererbbarer Klassenelemente, z. B. Konstruktoren. Diese stellen potentiell eine zusätzliche Indirektion je Klassenverfeinerung dar. Jampacks vermeiden diesen Nachteil durch das Verschmelzen einer FOP-Vererbungshierarchie in einer Klasse.
Jampacks erlauben durch Inlining für einfache FOP auch die Vermeidung von Sprunganweisungen zwischen den Verfeinerungen einer Methode. Für AML ist dies nicht ohne weiteres möglich. Beachtet werden muss, dass die Methodensignaturen der Verfeinerungen als Join-Points für die eingebetteten und *gebundenen* Aspekte benötigt werden. Inlining sämtlicher Verfeinerungen vor der Anwendung von Aspekten ist für AML *kontraproduktiv*. Eine Trennung der Aspekt- und Mixin-Übersetzung, wie sie in FeatureC++ durchgeführt wird, ist nicht mehr umsetzbar. Die durch Inlining zusammengesetzten Funktionen bieten nur noch den Join-Point der finalen verfeinerten Methode. Die Bindung der Aspektquantifizierung an *bestimmte* Schichten ist nicht mehr möglich. Jampacks können hier durch iterative Übersetzung der AML oder durch die Nachbehandlung der durch Aspekte veränderten Mixins umgesetzt werden.

Die Umsetzung der AML durch Jampacks ohne eine iterative Übersetzung erfordert eine erweiterte Pointcut-Deklarationssprache. Diese Sprache muss *bestimmte* einzelne Elemente in den zusammengesetzten Funktionen des Jampacks referenzieren können, um die Bindung der Aspektquantifizierung umzusetzen. Das erhöht die Komplexität der Pointcut-Deklarationssprache stark, da die Elemente innerhalb einer Funktion, wie Funktionsaufrufe oder Schleifen, keine eindeutige Bezeichnung haben müssen. Die modulare Einheit der einzelnen Anweisung erzeugt einen sehr hohen Verwaltungsaufwand für die feingranularen Module und eine enge Kopplung zwischen Funktionsmodulen. LogicAJ 2 ist ein Vorschlag, der für einfache AOP die Referenzierung einzelner Anweisungen erlaubt [TR06].

Modularisierbarkeit von Belangen. AML ermöglichen die FOP-typische, *kohärente* Speicherung verschiedener Fragmente, die für ein abstraktes Merkmal benötigt werden. Interne Klassen der Aspekte sind nicht mehr notwendig und können innerhalb des AML-Moduls veräußert werden. Die Merkmalkohäsion bleibt erhalten.
Codereplikation aufgrund homogener querschneidender Belange und aufgrund dynamischen Crosscuttings kann durch AOP-Mechanismen vermieden werden.
Die Definition von Schnittstellen gestaltet sich aufgrund der möglichen Seiteneffekte der Aspekte auf unterschiedliche Basisschichten auch für AML schwierig.

Bisher manipulieren die Aspekte eines AML stets sämtliche vorhergehenden Schichten. Die Aspektquantifizierung könnte weiterhin noch enger auf *bestimmte* übergeordnete Schichten begrenzt werden, um eine Menge von Schichten oder geschachtelten Merkmalmodulen inklusive AML ohne Seiteneffekte in einer anderen Schichten-Konfiguration

verwenden zu können.

Erweiterbarkeit. Die Möglichkeit der Erweiterung von Aspekten ist ein weiteres hervorzuhebendes Merkmal dieser Kombination, sog. Aspektverfeinerung (engl. aspect refinement) [ALS05a]. Eine Erweiterung von Superaspekten durch die anonyme Vererbung der FOP ermöglicht die modulare und *unabhängige* Erweiterung oder Redefinition von Pointcut-Ausdrücken sowie das Hinzufügen weiterer Advice-Definitionen. Für die Aspekterweiterung sind keine abstrakten, autark nicht instantiierbaren Aspekte notwendig.[1] Im AML stellt ein Aspekt ein einfaches Mixin dar, das unabhängig von bereits bestehenden Verfeinerungen durch zusätzliche Verfeinerungen erweitert werden kann [ALS05a]. Die unabhängige, mehrfache Erweiterung eines Aspekts erfordert in AML keine Replikation des Pointcut-Ausdrucks.

Durch die nun unabhängige, iterative Erweiterbarkeit von Aspekten, setzen auch AML mithilfe von anonymer Vererbung der FOP das Prinzip der Einheitlichkeit um. Dieses sieht die iterative Erweiterbarkeit sämtlicher Programmteile vor.

Neben der Erweiterung von Aspekten ist in AOP die Erweiterung um Komponenten problematisch. Neue Komponenten können unerwartet Einfluss auf bereits integrierte Aspekte haben und ihnen neue lexikalische Join-Points bieten. Die Bindung der Aspektquantifizierung beschränkt die lexikalischen Join-Points auf Code zurückliegender Entwicklungsstufen, d. h. vorhergehender Schichten [LHB05, LHBL06]. Nachfolgend eingefügte Komponenten werden durch den Aspekt nicht manipuliert. Unvorhersehbares Verhalten durch das Hinzufügen von Komponenten wird vermieden [ALRS05a, ALS05a]. Das Evolutionsparadoxon der AOSD kann vermieden werden.
Für die Beschränkung der Aspektquantifizierung wird in der Literatur eine Umstrukturierung der Pointcut-Ausdrücke vorgeschlagen [ALS05a, ALS05b].

```
1 refines aspect sync {
2   pointcut lockPoint() = execution("% LinkedList::size(...)")
3     || super:: lockPoint();};
```

Abbildung 4.1: Codebeispiel einer Aspekt-Verfeinerung in AML.

Ein Beispiel für die Verfeinerung eines Aspekts ist in Abb. 4.1 dargestellt. Die Darstellung zeigt die Erweiterung des Pointcut-Ausdrucks "`lockPoint`" um zusätzliche Join-Points (Zeilen 2 – 3). Die Vererbungsbeziehung des dargestellten erweiternden Aspekts ist anonym deklariert (Zeile 1), d. h. das Mixin kann mehreren expliziten Superaspekten zugeordnet werden. Die Erweiterung des anonymen Superaspekts "`sync`" durch die dargestellte Verfeinerung ist unabhängig von anderen Aspektverfeinerungen möglich und erzeugt keine Codereplikation.
Die für die Bindung der Aspektquantifizierung beispielhafte schematische Darstellung in Abbildung 4.2 stellt die Verfeinerung der Schichten "`Simple`" und "`Sorted`" durch den AML "`Synchronize`" dar. Nachfolgend hinzugefügte Komponenten und Verfeinerungen, wie durch die Kollaboration "`Printable`", werden vom Aspekt nicht verändert.
Ein Beispiel für die Restrukturierung eines Pointcut-Ausdrucks für Mixin-Schichten ist in Abb. 4.3 dargestellt. Das obere Codefragment zeigt den vom Programmierer im AML definierten Pointcut-Ausdruck, während das untere Fragment den gebundenen, vom Com-

[1] Durch Vererbung direkt erweiterbare Aspekte mussten in AOP als abstrakt deklariert werden.

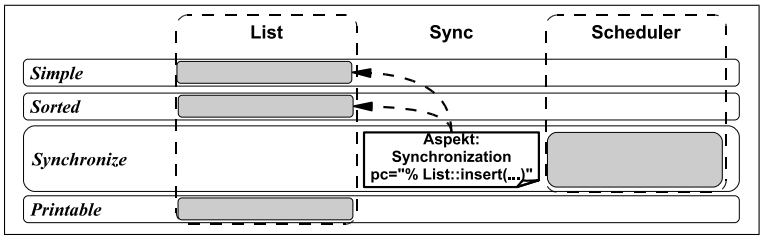

Abbildung 4.2: Schematische Darstellung eines AML.

```
1  "% List::insert(...)"
```

⇓

```
1  "% List_Simple::insert(...)"
2  ||
3  "% List_Sorted::insert(...)"
```

Abbildung 4.3: Beispiel für Pointcut-Restrukturierung.

piler erweiterten Ausdruck darstellt. Mixins der Schicht "`Printable`" werden von der Aspektquantifizierung nicht mehr erfasst.

Zusätzliche Anmerkungen zu AML. In Bezug auf die Erweiterbarkeit der Software muss die Umsetzung des AML-Konzepts durch den Compiler weiterhin beachtet werden, insbesondere die Interaktion zwischen Aspekten und Klassenverfeinerungen. Eine in der Literatur propagierte Lösung für die Bindung der Aspektquantifizierung ist die Pointcut-Restrukturierung [ALS05a, ALS05b].
Je nach Umsetzung der Restrukturierung des Pointcut-Ausdrucks wird im Compiler eine Methodenerweiterung bei der Anwendung des restrukturierten Pointcut-Ausdrucks als *eigenständige* Methode betrachtet und für sich erweitert[2] oder als einzelnes, zusammengesetztes Fragment betrachtet und *einmalig* in der jeweils finalen Ausprägung erweitert. Die Restrukturierung von Pointcut-Ausdrücken kann zusätzlich in den erweiterten Mixin-Schichten zu einer neuen Form des FIP führen. Wird durch den Advice ein lexikalischer Join-Point einer vorhergehenden Schicht erweitert, so sind evtl. Methoden, welche durch den Advice am Join-Point aufgerufen werden, noch nicht definiert, da sie erst in nachfolgenden Mixin-Schichten des lexikalischen Join-Points, d. h. in Unterklassen des Mixins z. B. im AML, definiert werden. Das erzeugt in der mixin-basierten Umsetzung der FOP einen Übersetzungsfehler. Für die Umsetzung der FOP durch Jampacks tritt das Problem nicht auf, da nachträglich eingefügte Methoden im selben Namensraum der zusammengesetzten, finalen Klasse definiert sind wie die bereits vorhandenen Methoden. Mixin-Schichten und Jampacks sind demnach bezüglich AML nicht vollkommen äquivalent.

Die Umsetzung der Einschränkung von Pointcut-Ausdrücken kann neben der Pointcut-Restrukturierung auch durch iterative Übersetzung der AML erfolgen. Dies erhöht die Dauer der Übersetzung und erlaubt für Aspektverfeinerungen nur die Verallgemeinerung des im verfeinerten Aspekt definierten Pointcut-Ausdrucks. Dessen Einschränkung ist so nicht möglich, da der Weber für verschiedene Verfeinerungen eine einzige Funktion hinterlassen kann. Diese müsste aufgespalten werden, um bereits gewebten Advice-Code zu entfernen.
Ein in der Literatur noch nicht betrachteter Ansatz, um Aspekte in FOP-Mixin-Schichten einzubetten, könnte auf der Einordnung des Crosscuttings basieren.
Der Ansatz für statisches Crosscutting[3] beinhaltet die Propagation der lexikalischen Join-Points des eingebetteten Aspektes aus übergeordneten Schichten in den AML. Im

[2]Die vollständig erweiterte Methode wird also mehrfach verändert.
[3]Hinzufügen von Klassenelementen oder Erweiterung bestehender Funktionen (vgl. Abschnitt 2.2.4)

Folgenden ist eine Transformation des Pointcut-Ausdrucks notwendig, sodass lexikalische Join-Points einzig *im* AML selbst auftreten können. Durch die Identität einer Methodenbezeichnung in einer Schicht würde die Anwendung des Aspektes *einmalig* je Methode erfolgen. Die Aspektquantifizierung würde dennoch umgesetzt. Die Methoden vorhergehender Schichten werden ererbt und sind für den Advice am Join-Point vollständig definiert.
Die Erzeugung zusätzlicher Join-Points durch den Compiler verursacht Nachteile für die Performance und den Speicherbedarf, die durch eine Nachbehandlung vermieden werden können.
In Abbildung 4.2 würden sämtliche lexikalischen Schatten aus den Schichten "`Simple`" und "`Sorted`" in die Schicht "`Synchronize`" propagiert, d. h. im AML würde eine Methode erzeugt werden, die ihre jeweilige Basismethode aufruft. Der Aspekt würde dann derart angepasst, dass nur die Propagation einer Methode im AML "`Synchronize`" erweitert wird. Undefinierte Funktionen am Join-Point und die mehrfache Anwendung der Advice-Definition je Methode können vermieden werden.
Dynamisches Crosscutting[4] könnte für die Erweiterung eines Methodenaufrufes, sog. call-Advice[5], durch die redefinierbare Deklaration sämtlicher Klassen und Methoden in der Basisschicht umgesetzt werden. Die Join-Points dieser Erweiterungen sind für Mixin-Schichten nicht mittels Propagation einer weiterleitenden Methode in den AML erfassbar. Im Folgenden sind die am Join-Point aufgerufenen Methoden deklariert und können in nachfolgenden Schichten überschrieben werden, in der instantiierten finalen Klasse sind die Methoden definiert. Die Notwendigkeit zur Redefinition der Methoden in verfeinernden Mixin-Schichten, d. h. in Unterklassen, erfordert hier die dynamische Bindung dieser Methoden. Dynamisch gebundene Methoden erzeugen Nachteile in Performance und Ressourcenverbrauch.

Änderungen an bestehendem Quellcode werden ebenso wie die Veränderung von festgelegten Schichtenreihenfolgen als nachteilig erachtet: Die Bindung der Aspektquantifizierung bietet Vorteile durch die Vermeidung von ungewollten Seiteneffekten auf zusätzlich hinzugefügte Komponenten. Es wird dennoch als sinnvoll angesehen, dem Programmierer freizustellen, diese Möglichkeit der Bindung zu nutzen. Ungebundene Aspekte eines AML können querschneidend zur gesamten Programmstruktur sein. Diese Aspekte bzw. deren umschließende AML müssen durch die Bindung der Aspektquantifizierung derzeit stets an die Stelle der finalen Erweiterung verschoben werden.

Hinsichtlich der Wirkungsweise verfeinerter Aspekte ist u. a. das Anwenden von Aspekten auf Schichten zu untersuchen, welche den Aspekt von seiner Verfeinerung trennen. Die sie trennenden Schichten wurden nach dem Aspekt hinzugefügt und sollten daher nicht erweitert werden. Die dazwischen liegenden Schichten wurden jedoch vor der Aspektverfeinerung hinzugefügt und sollten daher von dieser erweitert werden. Zusätzlich stellt sich aufgrund der gebundenen Aspektquantifizierung die Frage, welche Advice-Definitionen, die des originalen oder die des erweiternden Aspekts, an den durch die zwischenliegenden Schichten hinzugekommenen, lexikalischen Join-Points wirken sollen. Wirken die Advice-Definitionen des erweiterten Aspekts, ist die Intention der Verfeinerung die Umgestaltung des Pointcut-Ausdrucks. Die Bindung der Aspektquantifizierung des erweiterten Aspekts wird *aufgehoben*. Wirken die Advice-Definitionen des

[4] Veränderung des Kontrollflusses innerhalb von bestehenden Funktionen
[5] Unterbrechung des Aufrufs von Methoden für die Anwendung rücksprungspezifischer Erweiterungen.

erweiternden Aspekts ist die konsistente Wiederverwendung der Pointcut-Ausdrücke das Ziel, die Aspektquantifizierung des erweiterten Aspekts bleibt gebunden.

4.2 Generische Merkmalmodule

Eine weitere Möglichkeit, die Erfüllung der Anforderungen zu verbessern, liegt in der Kombination von GP und FOP, den sog. Generischen Merkmalmodulen (engl. generic feature modules; GFM) [AKL06, KLA06]. Der Ansatz sieht die Elemente der FOP-Merkmalmodule als parametrisierbare Mixins vor. Die Klassen und Verfeinerungen werden abstrakt gegen eine variable Typschnittstelle entwickelt.
FOP bestimmt in dieser Kombination die Umsetzbarkeit der Kriterien der Performance, der Erweiterbarkeit und des Ressourcenverbrauchs einer Software. Der Zeitpunkt der Adaptierung entspricht dem der beiden Einzeltechniken, d. h. dem Zeitpunkt der Übersetzung der Software.

Die Kombination GFM erzeugt gegenüber den Einzeltechniken GP und FOP folgende Unterschiede in Bezug auf die Umsetzbarkeit der Anforderungen an qualitativ hochwertige Software.

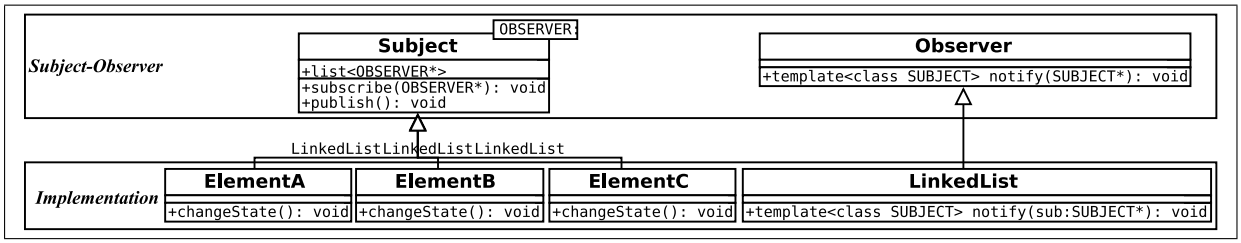

Abbildung 4.4: Beispiel der Umsetzung homogener querschneidende Belange durch GFM.

Modularisierbarkeit von Belangen. GFM ermöglichen durch FOP-Techniken die Kapselung heterogener, querschneidender Belange. Durch einen Synergieeffekt kann die Kombination beider Programmiertechniken den Code von statischen homogenen querschneidenden Belangen besser modular umsetzen als es jede einzelne vermag [KLA06].
Code homogener Belange, d. h. die Verteilung identischen Codes an unterschiedliche Stellen des erweiterten Komponentencodes, war in reiner FOP stets mit Codereplikation verbunden. Die Erweiterung unterschiedlicher Klassen um ein Konzept kann in FOP mehrere, angepasste Replikate der Erweiterung und auch Typkonvertierungen erfordern. In GP hingegen waren querschneidende Belange nicht *konfigurierbar* kapselbar.
Die Verteilung homogenen Codes auf unterschiedliche Klassen ist durch die Kombination von GP mit FOP möglich [KLA06]. Beispielhaft ist in Abbildung 4.4 die Realisierung des Entwurfsmusters "Beobachter" dargestellt (vgl. einführendes Beispiel, Abschnitt 2.2.2) [GHJV95].
Ziel des Beispiels ist es, bei der Veränderung von Listenelementen die Sortierung der Liste wiederherzustellen, d. h. die Liste beobachtet hier ihre Elemente. Die gemeinsame Logik des Entwurfsmusters wird dazu in der Schicht "`Subject-Observer`" abstrakt mittels GP-Modulen definiert. Die Zuordnung der Rollen zu den entsprechenden Klassen erfolgt durch Vererbung in nachfolgenden Schichten. Dabei kann jedes dieser neu definierten Subjekte den genauen Typ seines Beobachters bestimmen. Für die Verwaltung der

variablen Beobachter und Subjekte sind keine virtuellen Methoden oder Typumwandlungen erforderlich. Auch Codereplikation der Erweiterung kann vermieden werden.
Generische Merkmalmodule erlauben dennoch *keine* Verteilung von identischem Code innerhalb einer Klasse. Die verschiedenen Methoden, welche eine Benachrichtigung an die Beobachter erzeugen, müssen explizit und redundant im entsprechenden Merkmalmodul verfeinert werden.

Wiederverwendbarkeit. Generische Module aufbauend auf Klassenmodulen sind komplexe Software-Einheiten und nur auf syntaktischer Ebene anpassbar.
FOP ermöglicht die Umsetzung semantisch anpassbarer Module mit einem hohen Wert. Die Schwächen von FOP liegen im Umgang mit variablen Typen. Der in den FOP-Schichten gekapselte Code verwendet explizite Typen, was die Wiederverwendbarkeit mindert. Die Typdefinition kann ausgetauscht werden, jedoch verursacht dieses Vorgehen u. U. Replikation von Code. Sollen unterschiedliche Typen für den Kontext homogen behandelt werden können, bietet FOP zwei Wege:

1. Der variable Typ wird pseudoinvasiv verändert und tritt im Folgenden unter dem selben Namen auf. Nachteilig ist hier die exklusiv alternative Behandlung der Typen, d. h. dass stets nur ein Typ während der Laufzeit verfügbar ist, sowie die Notwendigkeit seiner Definition in den FOP-Schichten.

2. Variable Typen unter verschiedenen Namen sind in FOP aufbauend auf OOP-Mechanismen durch späte Bindung umsetzbar. Die Kosten der späten Bindung sind hier auch dann zu tragen, wenn nur eine Variante während der Laufzeit verfügbar sein soll.

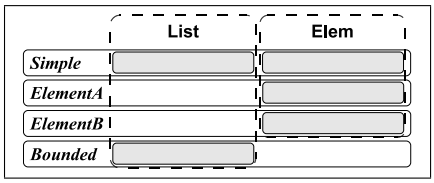

Abbildung 4.5: Beispiel alternativer FOP-Schichten für Typvariabilität.

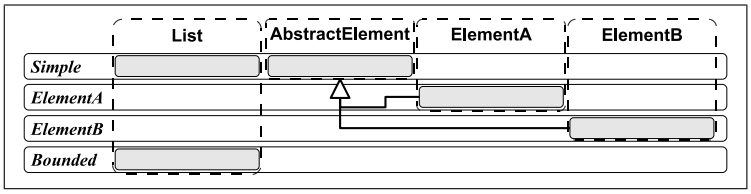

Abbildung 4.6: Beispiel für FOP-Typvariabilität durch spätes Binden.

Abbildung 4.5 zeigt die Umsetzung von Variabilität durch alternative Schichten. Virtuelle Funktionen werden vermieden. "`ElementA`" und "`ElementB`" können fortan nicht mehr gemeinsam in einer Konfiguration auftreten. Eventuell gleiche Schnittstellen würden sich gegenseitig überschreiben oder unerwartetes Verhalten erzeugen (Objekte wären von zwei konzeptionellen Typen). Abbildung 4.6 zeigt die Umsetzung von Typvariabilität durch spätes Binden. Beide Varianten sind in einer Konfiguration möglich, jedoch sind *permanente* Nachteile hinsichtlich der Performance und des Speicherverbrauchs festzustellen, die auch zu tragen sind, wenn nur ein Typ während der Laufzeit benötigt wird.

Geschachtelte GFM-Module können einen hohen Wert besitzen *und* feingranular semantisch angepasst werden. Die Einführung der Typ-Parametrisierung ermöglicht, ein Modul auch syntaktisch feingranular an Typen des wiederverwendenden Kontextes anzupassen.
Die parameterbasierte Polymorphie der Generischen Merkmalmodule ermöglicht die Festlegung von manipulierten Typen außerhalb der FOP-Schichten. Die Software kann so ohne Erweiterungen und Typkonvertierungen an neue Typen angepasst werden und selbst wiederverwendet werden.[6]
Der Wechsel zwischen statischer und dynamischer Variabilität sowie der Austausch von Typen ist in dieser Kombination durch die Parametrisierung des ADT mit einem subtyp-polymorphen Typ möglich. Die Vererbungsbeziehung der variabel statisch oder dynamisch gebundenen, speziellen Klassen zu ihrer Oberklasse kann in optionalen Schichten ohne Codereplikation implementiert werden. Virtuelle Funktionen sind nur für Laufzeitvariabilität notwendig und können konfiguriert erzeugt werden. (Diese Implementierung ist für Mixins eine Umsetzung der konfigurierbaren Bindung mittels FOP [ASB04, CE00].) Die Wiederverwendbarkeit eines Codefragments ist demnach auch unter Beachtung dieser Variationspunkte erhöht. Der Wechsel zwischen statischer und dynamischer Bindung muss nicht im Software-Entwurf vorbereitet werden, sondern geschieht durch die Parametrisierung mit entsprechend polymorphen oder finalen Typen zum Zeitpunkt der Übersetzung.

```
1 refines class AbstractElement{
2   AbstractElement(...):super(...){}
3   virtual AbstractElement* clone() =0; };
4
5 class myConfig{
6   //typedef ElementA ToHandle;
7   typedef AbstractElement ToHandle;
8   ... };
```

Abbildung 4.7: Codebeispiel der Erweiterung einer Klasse um Subtyp-Polymorphie.

Eine Kollaboration, welche die dynamische Bindung für zu verwaltende Elemente einführt, ist in Abbildung 4.7 dargestellt.[7] Zusätzlich zu den statisch homogenen Methoden verwalteter Elemente kapselt die Oberklasse "`AbstractElement`" durch die dargestellte Erweiterung auch eine Schnittstelle zu den variablen Methodenimplementierungen ihrer Unterklassen (Zeile 3). Durch die Parametrisierung der Liste mit dem nun polymorphen Elementtyp der Oberklasse (Zeile 7) ist die polymorphe Verwaltung mehrerer Elementtypen durch eine einheitliche Schnittstelle in der Liste während der Laufzeit möglich.

[6] In einfacher FOP sind hierfür abstrakte Klassen und u. U. Typkonvertierungen notwendig.
[7] Die Fragmente der Zeilen 1 – 3 bzw. 5 – 8 sind entgegen der Darstellung nicht Element einer Datei.

4.3 Generischer Advice

Auch die Kombination von GP und AOP, sog. Generischer Advice (engl. generic advice; GA), ist bemerkenswert [LBS04]. Aspekte werden hier mit Metavariablen versehen und erhalten so Zugriff auf das System statischer Typen der erweiterten Join-Points. Die Variablentypen müssen somit zum Zeitpunkt der Aspektentwicklung nicht festgelegt werden, sondern werden durch den Compiler zum Zeitpunkt der Übersetzung in Abhängigkeit des Join-Points bestimmt. Die Kombination ermöglicht die Verteilung von Advice-Code, dessen Implementierung abhängig vom konkreten Join-Point ist [LBS04]. Eine weitere Möglichkeit der Kombination liegt in der Veränderung von Template-Klassen und Template-Methoden durch AOP-Mechanismen [LBS04].
Die Eigenschaften hinsichtlich der Erfüllung der Kriterien des Zeitpunktes der Adaptierung, der Modularisierbarkeit und des Ressourcenverbrauchs werden von der AOP auf die Kombination übertragen.
Die Eigenschaft der Performance ergibt sich aus den Laufzeitkosten beider Techniken (Indirektionen; Join-Point-API). Diese Kombination hat folgende markante Unterschiede hinsichtlich der Erfüllbarkeit der Qualitätsanforderungen im Vergleich zu ihren zugrunde liegenden Einzeltechniken:

```
1  aspect Caching{
2    //Ergebnisspeichernde Klasse mit explizit festgelegten Typen
3    class myCache{
4      int _arg; AbstractElement* _result;
5      bool inside(int askedArg_){...}
6      AbstractElement* get(){...}
7      void set(int askedArg_, AbstractElement* newResult_){...} };
8
9    pointcut ToCache(AbstractElement* result_,int arg_) =
10     execution("% LinkedList::getPos(...)") && args(arg_) && result(result_);
11
12   advice ToCache(result_, arg_): around(AbstractElement* result_, int arg_){
13     static myCache* cache = new myCache();
14     if (cache->inside(arg_)){ //cache-hit
15       result_ = cache->get();
16     } else { //cache-miss
17       tjp->proceed();
18       cache->set(arg_, result_);  //Aktualisiere den Cache
19 }}};
```

Abbildung 4.8: Code des Zwischenspeicherns eines Ergebnisses mittels AOP.

Wiederverwendbarkeit. Die Wiederverwendbarkeit der Template-Komponenten und Aspekte wird erhöht.
Durch die Auslagerung von Code variabler und querschneidender Belange in die Aspekte enthalten die erweiterten Template-Komponenten keinen Code, der in Konflikt zum Kontext stehen kann. Die Komponenten sind demnach gut wiederverwendbar und bedürfen keiner invasiven Änderungen.
In reiner AOP werden sämtliche Variablentypen für den Advice explizit lexikalisch deklariert. Das bindet den Aspekt eng an den Join-Point-Kontext. Ziel der Kombination GA ist es, durch Metavariablen den Advice-Code kontextspezifisch, d. h. join-point-

abhängig, anpassen zu können, ohne Codereplikation zu erzeugen [GK06]. Die join-point-spezifischen Gegebenheiten in Form von konkreten Variablentypen können im Generischen Advice durch eine statische Join-Point-API manipuliert und der Aspekt-Code syntaktisch angepasst werden. Das erhöht die Wiederverwendbarkeit der Aspekte. Durch die Extraktion von Funktionalität vermindert sich der Wert der einzelnen Komponente.

```
1  aspect Caching{
2    //Ergebnisspeichernde Template-Klasse mit statisch polymorphen Typen
3    template<class ARGTYPE, class RESULTTYPE> class myCache{
4      ARGTYPE _arg; RESULTTYPE _result;
5      bool inside(ARGTYPE askedArg_){...}
6      RESULTTYPE get(){...}
7      void set(ARGTYPE askedArg_, RESULTTYPE newResult_){...} };
8
9    pointcut ToCache() = execution("% LinkedList::getPos(...)");
10
11   advice ToCache(): around(){
12     //Ermitteln des Typs und des Wertes des Arguments
13     JoinPoint::Arg<0>::ReferredType arg_ =
14       *(JoinPoint::Arg<0>::ReferredType *) tjp ->arg(0);
15     //join-point-spezifische Instantiierung der Typen der Cache-Klasse
16     static myCache<JoinPoint::Arg<0>::Type, JoinPoint::Result >* cache =
17       new myCache<JoinPoint::Arg<0>::Type, JoinPoint::Result >();
18     if (cache->inside(arg_)){ //cache-hit
19       *tjp->result() = cache->get();
20     } else { //cache-miss
21       tjp->proceed();
22       cache->set(arg_, *(tjp->result())); //Aktualisiere den Cache
23 }};
```

Abbildung 4.9: Code des Zwischenspeicherns eines Ergebnisses durch GA.

Ein Beispiel soll dies illustrieren.
In Abbildung 4.8 ist die Umsetzung eines Caching-Aspektes in reinem AOP dargestellt. Der Aspekt ist im Folgenden einzig auf Methoden mit einem Argument vom Typ "`int`" und einem Rückgabewert vom Typ "`AbstractElement*`" anwendbar (Zeilen 9 – 10). Diese homogene Behandlung von unterschiedlichen Typen erfordert Subtyp-Polymorphie der verschiedenen Argument- und Rückgabetypen oder die invasive Veränderung des Aspekts, um Typkonvertierungen zu vermeiden. Auch die Verlagerung des Pointcut-Ausdrucks in einen abgeleiteten Aspekt ist hier nicht hilfreich, da die Elemente der zwischenspeichernden Klasse "`myCache`" (Zeilen 3 – 7) explizit getypt werden müssen.
Templates bieten den Vorteil des statischen Umgangs mit unterschiedlichen Typen. Das betrifft hier die Einführung zusätzlicher Variabilität durch die "Abfrage" der Typen des Kontextes mittels der statischen Join-Point-API. Ein entsprechendes Beispiel ist in Abb. 4.9 für den bereits in reinem AOP betrachteten Caching-Aspekt (aus Abb. 4.8) dargestellt. Die Klasse "`myCache`" ist als Klassen-Template implementiert und ermöglicht so die Speicherung von Variablen unterschiedlicher Typen. Die Instantiierung dieser Cache-Klasse erfolgt in Abhängigkeit von jedem konkreten Join-Point und den *dort* verwendeten Typen (Zeilen 16 – 17). Im Beispiel werden join-point-spezifisch der Rückgabe- und der Parametertyp des konkreten Advice-Kontexts von der Join-Point-API erfragt und zur Parametrisierung der Cache-Klasse verwendet. Durch die Auslagerung des Pointcuts in einen erbenden Aspekt ist die Wiederverwendbarkeit des Aspekts zusätzlich erhöht.

Erweiterbarkeit. Die parameterbasierte Polymorphie des Advice-Codes gegenüber den Typen der Join-Points hat keine Auswirkungen auf die Erweiterbarkeit. Die Erweiterung von Templates kann ebenso wie bei reiner AOP durch das Einfügen und Verfeinern von Methoden geschehen.
Für GA ist weiterhin die Erzeugung von Template-Spezialisierungen für Klassen und Methoden vorgesehen [LBS04]. Durch Template-Spezialisierungen können zum Zeitpunkt der Übersetzung unterschiedliche Objekte eines generischen Typs erzeugt werden. Im Komponentencode soll demnach ein einziges Template vorliegen, welches durch Aspekte mittels Template-Spezialisierungen erweitert wird.[8] Derartige Erweiterungen erfordern keine Replikation des umgebenden Quellcodes (die Template-Spezialisierung einer Klasse durch den Entwickler erfordert die Replikation sämtlicher Methoden der Template-Klasse im Quellcode; Template-Spezialisierungen für Methoden erfordern Unterschiede in deren Signaturen).
Beachtet werden muss die neue Qualität des möglichen FIP. Erweiterungen und Funktionen können nur für bestimmte Konfigurationen der Template-Klasse verfügbar sein. In anderen Konfigurationen können sie trotz der Anwendung entsprechender Aspekte für eine bestimmte Parameterkonfiguration undefiniert sein. In diesem Fall ist eine enge Zusammenarbeit zwischen den Entwicklern notwendig, was einer Intention der Modularisierung von Belangen in Aspekte entgegensteht.
Ein Hinzufügen von Komponenten und Aspekten zu einem System, welches bereits Aspekte enthält, verursacht die gleichen Probleme wie bei einfacher AOP.

4.4 Zusammenfassende Betrachtung der Kombinationen

Durch die Kombination der Programmiertechniken konnten ihre einzelnen Vorteile vereint und ihre inhärenten Nachteile teilweise überwunden werden.
Aspekt-Mixin-Schichten überwinden die mangelnde iterative Erweiterbarkeit von reiner AOP und erhöhen die Merkmalkohäsion. Somit wurden Schwachpunkte beider einzelnen Programmiertechniken (AOP und FOP) behoben.
Generische Merkmalmodule erlauben die Verbindung und Verbesserung von Wiederverwendbarkeit, Erweiterbarkeit und Modularisierbarkeit von Software. Die Unabhängigkeit vom Kontext kann hier verbunden werden mit der semantischen und modularen Anpassbarkeit der Software, die Wiederverwendbarkeit kann damit sehr gut unterstützt werden. Weiterhin erlauben FOP-Mechanismen auch in der Kombination mit GP eine sehr gute Erweiterbarkeit der Software. Die Modularisierbarkeit von Belangen stellt sich insgesamt bezüglich homogener querschneidender Belange und dynamischen Crosscuttings für GFM trotz kleiner Verbesserungen als mangelhaft dar.
Die Kombination Generischer Advice erlaubt die Verbindung wiederverwendbarer Software-Einheiten mit einem hohen Grad an Modularität. Die bisher grobgranulare Struktur von GP-Klassen kann durch Aspekte semantisch sehr feingranular angepasst werden. Auch Probleme der GP bezüglich Erweiterbarkeit können im Rahmen der

[8] In der Literatur wird hierfür u. a. eine Vervielfältigung des zu spezialisierenden Templates vorgeschlagen [LBS04]. In der spezialisierten Kopie kann die Erweiterung anschließend gewebt werden. Das Vorgehen ist nur für Sprachen möglich, die Template-Spezialisierungen unterstützen.

	OOP	**GP**	**AOP**	**FOP**	**AML**	**GFM**	**GA**
Zeitpunkt der Adaptierung	Laufzeit	Übersetzung	Laufzeit	Übersetzung	Laufzeit	Übersetzung	Laufzeit
Performance	-	0	0	+	+	+	0
Modularisierbarkeit	-	- -	+	+	++	+	+
Wiederverwendbarkeit	-	+	0	+	+	++	+
Erweiterbarkeit	0	- -	+	++	++	++	+
Ressourcenverbrauch	- -	-	0	+	+	+	0

Tabelle 4.1: Zusammenfassung der Wirkung der Kombinationen.

Möglichkeiten von AOP verbessert werden. Dennoch ist die Umsetzung der Erweiterbarkeit als mangelhaft zu bewerten, da auch GA dem Evolutionsparadoxon der AOSD unterliegt. Ein Zersplittern von Aspekten mindert in der Kombination weiterhin die Erfüllbarkeit der Modularisierung. Querschneidende Belange treten im Gegensatz zu GP aber nicht mehr auf.
Tabelle 4.1 bietet einen Überblick über das durch die Kombinationen Erreichbare.

Kapitel 5

Zusammenfassung und Ausblick

Die Analyse der Programmiertechniken im Kapitel 3 unterstrich die Bedeutung der Modularisierbarkeit von Software hinsichtlich der Nutzeranforderungen und der entwicklerseitigen Anforderungen. Viele Nutzeranforderungen, wie sparsamer Umgang mit Ressourcen oder hohe Performance, sowie die software-technischen Voraussetzungen können durch die Wahl der Modularisierungstechnik beeinflusst werden. Es wurde durch Beispiele und Messungen gezeigt, welchen Einfluss die Programmiertechniken auf die Erfüllung von Qualitätskriterien, wie Performance, Ressourcenverbrauch u. a., der resultierenden Software haben.

Die Bewertung der Programmiertechniken bezüglich der Qualitätsanforderungen erfolgte durch das Abwägen verschiedener Gesichtspunkte, die ein zu untersuchendes Kriterium beeinflussen. Die identische Bewertung von Programmiertechniken bedeutet nicht deren Substituierbarkeit, da die unterschiedliche Erfüllung von Teilanforderungen durch die Techniken in der Zusammenfassung als gleichwertig angesehen werden konnten. Dies ist u. a. für die Forderung nach hoher Wiederverwendbarkeit sehr gut nachvollziehbar. Während FOP die semantische Anpassung von Modulen ermöglicht, aber nicht die syntaktische, erlaubt GP die Anpassung der Syntax, aber nicht die Anpassung der Semantik von Modulen.
Die Untersuchungen zeigten, dass bestimmte Anforderungen, wie Modularisierbarkeit oder Wiederverwendbarkeit, durch *keine* der betrachteten Programmiertechniken umfassend erfüllt werden konnte.

Jede Programmiertechnik offenbarte neben Vorteilen auch Nachteile. Ein Beispiel ist die Vermeidung von Codereplikation durch die Aspektorientierte Programmierung. Die nichthierarchischen deklarativen Beschreibungen der zugrunde liegenden Pointcut-Deklarationssprache verursachten Probleme bezüglich der Wiederverwendbarkeit und der Erweiterbarkeit der Module.

Ausgehend von der Erkenntnis, dass einige Anforderungen nicht umfassend erfüllt werden konnten, wurden Kombinationen der Programmiertechniken untersucht. Durch gezielte Kombinationen der Techniken war es möglich, Nachteile der einzelnen Programmiertechniken zu vermeiden. Synergieeffekte sind u. a. in der Kombination Generische Merkmalmodule erkennbar. Hier ist unter bestimmten Voraussetzungen eine Verbesserung der Modularisierbarkeit über die Fähigkeiten der einzelnen Techniken hinaus möglich.

Die Stärken der einzelnen Programmiertechniken konnten in den Kombinationen erhalten werden. Eine umfassende Lösung zur Erfüllung aller Anforderungen bieten die Kombinationen bisher jedoch nicht. Die Kombination zweier Programmiertechniken kann nicht vollständig alle Mechanismen einer dritten Technik substituieren. Ein zukünftiger Schritt könnte demnach die weitere Kombination der Programmiertechniken sein. Mögliche Probleme dafür wurden für die Kombination Generischer Advice analysiert.
Derzeit besteht keine ausreichende Compiler-Unterstützung für eine Kombination von Objektorientierter, Generischer, Aspektorientierter und Merkmalorientierter Programmierung.

Aus der Analyse ergeben sich mögliche Leitsätze für die Software-Entwicklung unter Verwendung der Techniken.
Es wird als sinnvoll angesehen, sämtliche Komponenten einer Programmfamilie generisch zu implementieren. Dies umfasst sowohl Komponenten als auch Verfeinerungen und Aspekte. Auf diese Weise wird die Forderung nach hoher Kontextunabhängigkeit und Wiederverwendbarkeit gut umgesetzt.
Die Komponentenbildung sollte auf Generischer Programmierung aufbauend durch Aspekt-Mixin-Schichten realisiert werden. Die Programmiertechniken Aspektorientierte Programmierung und Merkmalorientierte Programmierung ergänzen sich hierbei hinsichtlich der Vermeidung von Codereplikation und der Bereitstellung von Merkmalkohäsion.
Aufgrund der besseren Unterstützung der von Modularität abhängigen Anforderungen, wie Erweiterbarkeit und Korrektheit der Anwendung, sollten Aspekte in Aspekt-Mixin-Schichten nur dann zum Einsatz kommen, wenn Konzepte der Merkmalorientierten Programmierung unangemessene Nachteile erzeugen. Diese Fälle umfassen das dynamische Crosscutting und nichthierarchisches statisches Crosscutting. Module der Merkmalorientierten Programmierung sollten für statisches, heterogenes Crosscutting, d. h. die Verteilung unterschiedlichen Codes auf unterschiedliche Module, verwendet werden. Interne Klassen von Aspekten sollten AML vermieden werden. Die generischen Komponenten und Aspekte können durch Merkmalorientierte Programmierung einheitlich erweitert werden.

Statische Konfigurierung kann sich positiv auf die Umsetzbarkeit der Forderungen nach hoher Performance und geringem Ressourcenbedarf auswirken. Falls statische Konfigurierung der dynamischen aus den genannten Gründen vorgezogen wird, so sollte die dynamische Methodenbindung und Subtyp-Polymorphie dennoch im Rahmen der Möglichkeiten angewendet werden, um Laufzeitvariabilität zur Verfügung zu stellen. Das Verlagern von Konfigurationsentscheidungen durch Techniken der statischen Bindung (z. B. "`switch-case`") an das Ende der Software-Entwicklung wird als unzureichend angesehen. Die Anwendung dieser Techniken in dem Zusammenhang widerspricht dem Ziel der Modularität von Software und kann die Wiederverwendbarkeit der Software durch enge Bindung negativ beeinflussen.
Die modulare Implementierung von Laufzeitvariabilität kann die Verwendung nachteiliger Mechanismen erfordern, die für statische Variabilität nicht notwendig sind. Die Entscheidung der Verwendung dieser Mechanismen sollte dem Entwickler überlassen werden und ohne Codereplikation veränderbar sein. Diese Anforderungen können durch Generische Merkmalmodule erfüllt werden, welche eine konfigurierbare Bindung ohne Codereplikation ermöglichen. Der Compiler muss daraufhin u. U. eigenständig die zusätzlich

benötigte Logik einfügen.

Die Kombination der Aspekt-Mixin-Schichten bietet weitere Forschungsaufgaben, die im Bereich ihrer Übersetzung, aber auch in Zusammenhang mit der Interpretation von Aspektverfeinerungen mit eingeschobenen Schichten, angesiedelt sein sollten. Diese betreffen die Umsetzung des Konzepts durch den Compiler und die Interaktionen zwischen Aspekten und Verfeinerungen.

Alternative Ansätze wie Hyper/J bieten weitere interessante Konzepte, welche die Modularisierbarkeit erhöhen [OT01, TO00]. Die für Hyper/J vorgeschlagene Modularisierung der Software durch Modulbeschreibungen, die außerhalb des Quelltextes vorliegen, erlaubt die Trennung vieler unabhängiger und *überlappender* Belange ohne Zersplitterung des Codes. Überlappende Belange benötigen gleiche Codefragmente und konnten in sämtlichen Techniken nur durch Ausgliederung der überlappenden Bestandteile in ein unabhängiges Modul umgesetzt werden. Das widerspricht der Merkmalkohäsion. Eine komponierende Einordnung dieser extrahierten Module gestaltet sich aufgrund der Zuordnung zu mehreren Belangen schwierig.

Die Vielzahl der Publikationen auf diesem Gebiet und ähnlichen Gebieten zeugt von einem wachsenden Interesse an qualitativ hochwertiger Software. So sind auch in nächster Zeit viele interessante Vorschläge und Herangehensweisen zu erwarten.

Literaturverzeichnis

[AB04] Apel, S.; Böhm, K.: Towards the development of ubiquitous middleware product lines. In Gschwind, T.; Mascolo, C. (Hrsg.): *SEM*, Lecture Notes in Computer Science, Band 3437, S. 137–153. Springer, 2004.

[AC96] Abadi, M.; Cardelli, L.: *A Theory of Objects*. Springer-Verlag, New York and Berlin and Heidelberg, 1996.

[AFM97] Agesen, O.; Freund, S. N.; Mitchell, J. C.: Adding type parameterization to the Java language. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, S. 49–65. Atlanta, GA, 1997.

[AKL06] Apel, S.; Kuhlemann, M.; Leich, T.: Generic feature modules: Two-staged program customization. In *Proceedings of International Conference on Software and Data Technologies (ICSOFT'06)*, September 2006. to appear.

[ALRS05a] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: Combining feature-oriented and aspect-oriented programming to support software evolution. In Cazzola, W.; Chiba, S.; Saake, G.; Tourwé, T. (Hrsg.): *RAM-SE*, S. 3–16. Fakultät für Informatik, Universität Magdeburg, 2005.

[ALRS05b] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: Feature-oriented and aspect-oriented programming in C++. Technischer Bericht Nr. 3, Fakultät für Informatik, Universität Magdeburg, 2005.

[ALS05a] Apel, S.; Leich, T.; Saake, G.: Aspect refinement and bounding quantification in incremental designs. *apsec*, Band 0, S. 796–804, 2005.

[ALS05b] Apel, S.; Leich, T.; Saake, G.: Mixin-based aspect inheritance. Technischer Bericht Nr. 10, Fakultät für Informatik, Universität Magdeburg, 2005.

[ALS06] Apel, S.; Leich, T.; Saake, G.: Aspectual mixin layers: Aspects and features in concert. In *Proceedings of IEEE and ACM SIGSOFT 28th International Conference on Software Engineering (ICSE'06)*, Mai 2006. to appear.

[ASB04] Apel, S.; Sichting, H.; Böhm, K.: Configurable Binding: How to Exploit Mixins and Design Patterns for Resource-Constrained Environments. Technischer Bericht Nr. 14, Fakultät für Informatik, Universität Magdeburg, 2004.

[Bal99] Balzert, H.: *Lehrbuch der Objektmodellierung — Analyse und Entwurf*. Spektrum Akademischer Verlag, Heidelberg*Berlin, 1999.

[BC90] Bracha, G.; Cook, W.: Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, S. 303–311. ACM Press, New York, NY, USA, 1990.

[BFVY96] Budinsky, F. J.; Finnie, M. A.; Vlissides, J. M.; Yu, P. S.: Automatic code generation from design patterns. *IBM Systems Journal*, Band 35, Nr. 2, S. 151–171, 1996.

[Big98] Biggerstaff, T. J.: A perspective of generative reuse. *Annals of Software Engineering*, Band 5, S. 169–226, 1998.

[BLHM02] Batory, D.; Lopez-Herrejon, R. E.; Martin, J.-P.: Generating product-lines of product-families. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, S. 81. IEEE Computer Society, Washington, DC, USA, 2002.

[BLS03] Batory, D.; Liu, J.; Sarvela, J. N.: Refinements and multi-dimensional separation of concerns. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, S. 48–57. ACM Press, New York, NY, USA, 2003.

[BO92] Batory, D.; O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Band 1, Nr. 4, S. 355–398, 1992.

[Boo97] Booch, G.: *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Object Technology Series. Addison-Wesley Publishing Company, Menlo Park, California, USA, 2. Auflage, 1997.

[Bos98] Bosch, J.: Design patterns as language constructs. *Journal of Object-Oriented Programming*, Band 11, Nr. 2, S. 18–32, 1998.

[BOSW98] Bracha, G.; Odersky, M.; Stoutamire, D.; Wadler, P.: Making the future safe for the past: Adding genericity to the Java programming language. In Chambers, C. (Hrsg.): *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, S. 183–200. Vancouver, BC, 1998.

[Bra04] Bracha, G.: Generics in the java programming language. Technical report, Sun Microsystems, Inc., 2004.

[BSR04] Batory, D.; Sarvela, J.; Rauschmayer, A.: Scaling stepwise refinement. *IEEE Transactions on Software Engineering*, Band 30, S. 1278–1295, 2004.

[CBML02] Cardone, R.; Brown, A.; McDirmid, S.; Lin, C.: Using mixins to build flexible widgets. In *AOSD*, S. 76–85, 2002.

[CC04] Colyer, A.; Clement, A.: Large-scale aosd for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, S. 56–65. ACM Press, New York, NY, USA, 2004.

[CE99a] Czarnecki, K.; Eisenecker, U. W.: Components and generative programming (invited paper). In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, S. 2–19. Springer-Verlag, London, UK, 1999.

[CE99b] Czarnecki, K.; Eisenecker, U. W.: Synthesizing objects. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, S. 18–42. Springer-Verlag, London, UK, 1999.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[CK94] Chidamber, S. R.; Kemerer, C. F.: A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, Band 20, Nr. 6, S. 476–493, 1994.

[CLCM00] Clifton, C.; Leavens, G. T.; Chambers, C.; Millstein, T.: MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, S. 130–145, 2000.

[Coo90] Cook, W. R.: Object-oriented programming versus abstract data types. In Bakker, J. W. d.; Roever, W. P. d.; Rozenberg, G. (Hrsg.): *REX Workshop*, Lecture Notes in Computer Science, Band 489, S. 151–178. Springer, 1990.

[CW85] Cardelli, L.; Wegner, P.: On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, Band 17, Nr. 4, S. 471–522, 1985.

[DH96] Driesen, K.; Hölzle, U.: 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*, S. 306–323. ACM Press, New York, NY, USA, 1996.

[DIN98] DIN: DIN 66270 : 1998-01: Bewerten von Software-Dokumenten, Januar 1998.

[DLGD01] Duret-Lutz, A.; Géraud, T.; Demaille, A.: Design patterns for generic programming in C++. In *In the Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, S. 189–202. USENIX Association, San Antonio, Texas, USA, January-February 2001.

[Dum01] Dumke, R.: *Software Engineering*. Vieweg Verlag, Brauschweig and Wiesbaden, 3. Auflage, 2001.

[EBC00] Eisenecker, U. W.; Blinn, F.; Czarnecki, K.: A solution to the constructor-problem of mixin-based programming in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, 10 2000.

[EFB01] Elrad, T.; Filman, R. E.; Bader, A.: Aspect-oriented programming: Introduction. *Commun. ACM*, Band 44, Nr. 10, S. 29–32, 2001.

[Ern00] Ernst, E.: Syntax based modularization: Invasive or not? Department of Computer Science, University of Twente, The Netherlands, 2000.

[FF05] Filman, R. E.; Friedman, D. P.: Aspect-oriented programming is quantification and obliviousness. S. 21–35. Addison-Wesley, Boston, 2005.

[GB03] Gybels, K.; Brichau, J.: Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, S. 60–69. ACM Press, New York, NY, USA, 2003.

[GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gho04] Ghosh, D.: Generics in Java and C++ - A comparative model. *SIGPLAN Not.*, Band 39, Nr. 5, S. 40–47, 2004.

[GK06] Günter Kniesel, T. R.: A definition, overview and taxonomy of generic aspect languages. *L'Objet*, Band 11, Nr. 3, 2006. to appear.

[GSPS01] Gal, A.; Schröder-Preikschat, W.; Spinczyk, O.: AspectC++: Language proposal and prototype implementation. In De Volder, K.; Glandrup, M.; Clarke, S.; Filman, R. (Hrsg.): *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, Oktober 2001.

[HK02] Hannemann, J.; Kiczales, G.: Design pattern implementation in java and aspectj. In *OOPSLA*, S. 161–173, 2002.

[JF88] Johnson, R. E.; Foote, B.: Designing reusable classes. *Journal of Object-Oriented Programming*, Band 1, Nr. 2, S. 22–35, June/July 1988.

[Joh97] Johnson, R. E.: Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, S. 10–17, 1997.

[KCH+90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report Nr. CMU/SEI-90-TR-021, Software Eng. Inst., Carnegie Mellon Univ., Pittsburgh, November 1990.

[KFF98] Krishnamurthi, S.; Felleisen, M.; Friedman, D. P.: Synthesizing object-oriented and functional design to promote re-use. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, S. 91–113. Springer-Verlag, London, UK, 1998.

[KHH+01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, S. 327–353. Springer-Verlag, London, UK, 2001.

[KLA06] Kuhlemann, M.; Leich, T.; Apel, S.: Einfluss erweiterter Programmier-Paradigmen auf die Entwicklung eingebetteter DBMS. In *Grundlagen von Datenbanken*, S. to–appear, 2006.

[KLM+97] Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J.: Aspect-oriented programming. In Akşit, M.; Matsuoka, S. (Hrsg.): *Proceedings European Conference on Object-Oriented Programming*, S. 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KM05] Kiczales, G.; Mezini, M.: Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, S. 49–58. ACM Press, New York, NY, USA, 2005.

[Kni99] Kniesel, G.: Type-safe delegation for run-time component adaptation. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, S. 351–366. Springer-Verlag, London, UK, 1999.

[LARS05] Leich, T.; Apel, S.; Rosenmueller, M.; Saake, G.: Handling optional features in software product lines. In *Proceedings of OOPSLA Workshop on Managing Variabilities consistently in Design and Code*. San Diego, USA, Oktober 2005.

[LBN05] Liu, J.; Batory, D. S.; Nedunuri, S.: Modeling interactions in feature oriented software designs. In Reiff-Marganiec, S.; Ryan, M. (Hrsg.): *FIW*, S. 178–197. IOS Press, 2005.

[LBS04] Lohmann, D.; Blaschke, G.; Spinczyk, O.: Generic advice: On the combination of AOP with generative programming in AspectC++. In Karsai, G.; Visser, E. (Hrsg.): *GPCE*, Lecture Notes in Computer Science, Band 3286, S. 55–74. Springer, 2004.

[LH89] Lieberherr, K. J.; Holland, I. M.: Assuring good style for object-oriented programs. *IEEE Software*, Band 6, Nr. 5, S. 38–48, 1989.

[LHB05] Lopez-Herrejon, R. E.; Batory, D.: Improving incremental development in aspectj by bounding quantification. In Bergmans, L.; Gybels, K.; Tarr, P.; Ernst, E. (Hrsg.): *Software Engineering Properties of Languages and Aspect Technologies*, März 2005.

[LHBC05] Lopez-Herrejon, R. E.; Batory, D. S.; Cook, W. R.: Evaluating support for features in advanced modularization technologies. In Black, A. P. (Hrsg.): *ECOOP*, Lecture Notes in Computer Science, Band 3586, S. 169–194. Springer, 2005.

[LHBL06] Lopez-Herrejon, R.; Batory, D.; Lengauer, C.: A disciplined approach to aspect composition. In *Proc. ACM SIGPLAN 2006 Workshop on Partial Evaluation and Program Manipulation (PEPM 2006)*, S. 68–77. ACM Press, 2006.

[Lie86] Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In Meyrowitz, N. (Hrsg.): *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, S. 214–223. ACM Press, New York, NY, 1986.

[Lie04] Lieberherr, K. J.: Controlling the complexity of software designs. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, S. 2–11. IEEE Computer Society, Washington, DC, USA, 2004.

[LK94] Lajoie, R.; Keller, R. K.: Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert. In *Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS)*. Montreal, Canada, Mai 1994. Colloquium on Object Orientation in Databases and Software Engineering; to appear.

[LLO03] Lieberherr, K. J.; Lorenz, D. H.; Ovlinger, J.: Aspectual collaborations: Combining modules and aspects. *Comput. J.*, Band 46, Nr. 5, S. 542–565, 2003.

[LST+06] Lohmann, D.; Scheler, F.; Tartler, R.; Spinczyk, O.; Schröder-Preikschat, W.: A quantitative analysis of aspects in the eCOS kernel. In *European Chapter of ACM SIGOPS: Proceedings of the First EuroSys Conference*, 2006.

[MC99] Millstein, T.; Chambers, C.: Modular statically typed multimethods. *Lecture Notes in Computer Science*, Band 1628, S. 279–??, 1999.

[Mey97] Meyer, B.: *Object-oriented software construction*. Prentice Hall PTR, Upper Saddle River, NJ 07458, USA, 2. Auflage, 1997.

[MF05] Monteiro, M. P.; Fernandes, J. M.: Towards a catalog of aspect-oriented refactorings. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, S. 111–122. ACM Press, New York, NY, USA, 2005.

[Mil88] Mills, E. E.: Software metrics. Technischer Bericht Nr. SEI-CM-12-1.1, Seattle University, 1988.

[MK03] Masuhara, H.; Kiczales, G.: Modeling crosscutting in aspect-oriented mechanisms. In Cardelli, L. (Hrsg.): *ECOOP*, Lecture Notes in Computer Science, Band 2743, S. 2–28. Springer, 2003.

[MKD03] Masuhara, H.; Kiczales, G.; Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In *Compiler Construction (CC2003)*, 2003.

[MMP89] Madsen, O. L.; Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, S. 397–406. ACM Press, New York, NY, USA, 1989.

[MO04] Mezini, M.; Ostermann, K.: Variability management with feature-oriented programming and aspects. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, S. 127–136. ACM Press, New York, NY, USA, 2004.

[MRB97] Martin, R. C.; Riehle, D.; Buschmann, F. (Hrsg.): *Pattern languages of program design 3*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Mye95] Myers, N. C.: Traits: a new and useful template technique. June 1995.

[Ost03] Ostermann, K.: *Modules for hierarchical and crosscutting models*. Dissertation, April 2003.

[OT00] Ossher, H.; Tarr, P.: Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[OT01] Ossher, H.; Tarr, P.: Hyper/j: multi-dimensional separation of concerns for java. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, S. 821–822. IEEE Computer Society, Washington, DC, USA, 2001.

[OW97] Odersky, M.; Wadler, P.: Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97), Paris, France*, S. 146–159. ACM Press, New York (NY), USA, 1997.

[Par72] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM*, Band 15, Nr. 12, S. 1053–1058, 1972.

[Par76] Parnas, D. L.: On the design and development of program families. *IEEE Trans. Software Eng.*, Band 2, Nr. 1, S. 1–9, 1976.

[Par78] Parnas, D. L.: Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, S. 264–277. IEEE Press, Piscataway, NJ, USA, 1978.

[PH00] Poetzsch-Heffter, A.: *Konzepte objektorientierter Programmierung*. Springer, Berlin and Heidelberg and New York, 2000.

[Pin05] Pinzger, M.: *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. dissertation, Mai 2005.

[Pre97] Prehofer, C.: Feature-oriented programming: A fresh look at objects. In *ECOOP*, S. 419–443, 1997.

[PS91] Palsberg, J.; Schwartzbach, M. I.: Object-oriented type inference. In Meyrowitz, N. (Hrsg.): *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, New York, NY, 1991.

[RDL04] Rieger, M.; Ducasse, S.; Lanza, M.: Insights into system-wide code duplication. In *WCRE*, S. 100–109, 2004.

[RM95] Rinne, H.; Mittag, H.-J.: *Statistische Methoden der Qualitätssicherung*. Carl Hanser Verlag, München, 3. Auflage, 1995.

[Ros05] Rosenmüller, M.: Merkmalsorientierte Programmierung in C++. diploma thesis, Otto-von-Guericke-Universität Magdeburg,Fakultät für Informatik, Institut für Technische und Betriebliche Informationssysteme, August 2005.

[SB98] Smaragdakis, Y.; Batory, D.: Implementing layered designs with mixin layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, S. 550–570. Springer-Verlag LNCS 1445, 1998.

[SB00] Smaragdakis, Y.; Batory, D.: Mixin-based programming in C++. In *Conf. on Generative and Component-Based Software Engineering (GCSE)*, S. 163–177. Springer-Verlag LNCS 2177, 2000.

[SDNB03] Schärli, N.; Ducasse, S.; Nierstrasz, O.; Black, A.: Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, LNCS, Band 2743, S. 248–274. Springer Verlag, Juli 2003.

[SGSP02] Spinczyk, O.; Gal, A.; Schröder-Preikschat, W.: AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Conference on Tools Pacific*, S. 53–60. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2002.

[SL94] Stepanov, A. A.; Lee, M.: The Standard Template Library. Technischer Bericht Nr. X3J16/94-0095, WG21/N0482, 1994.

[SLU05] Spinczyk, O.; Lohmann, D.; Urban, M.: Advances in AOP with AspectC++. In *In Proceedings of SoMeT '05: New Trends in Software Methodologies, Tools and Techniques (Volume 129)*, S. pp. 33–53. IOS Press, Tokyo, Japan, 2005.

[Sny86] Snyder, A.: Encapsulation and inheritance in object-oriented programming languages. In Meyrowitz, N. (Hrsg.): *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, S. 38–45. ACM Press, New York, NY, 1986.

[Spi05] Spinczyk, O.: AspectC++ - A language overview. Technical report, Computer Science 4, Friedrich-Alexander University Erlangen-Nuremberg, Mai 2005.

[SPLS+06] Schröder-Preikschat, W.; Lohmann, D.; Scheler, F.; Gilani, W.; Spinczyk, O.: Static and dynamic weaving in system software with AspectC++. In *HICSS*. IEEE Computer Society, 2006.

[Str00] Stroustrup, B.: *Die C++ Programmiersprache.* Addison-Wesley Verlag, München, 4. Auflage, 2000.

[SvGB05] Svahnberg, M.; Gurp, J. v.; Bosch, J.: A taxonomy of variability realization techniques: Research articles. *Softw. Pract. Exper.*, Band 35, Nr. 8, S. 705–754, 2005.

[SXG+04] Subramonian, V.; Xing, G.; Gill, C.; Lu, C.; Cytron, R.: Middleware specialization for memory-constrained networked embedded systems. In *RTAS '04: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, S. 306. IEEE Computer Society, Washington, DC, USA, 2004.

[TBG03] Tourwé, T.; Brichau, J.; Gybels, K.: On the existence of the AOSD-evolution paradox. In *AOSD Workshop on Software-engineering Properties of Languages for Aspect Technologies*. Boston, USA, 2003.

[TO00] Tarr, P.; Ossher, H.: Hyper/J user and installation manual. Technischer Bericht, IBM T. J. Watson Research Center, 2000.

[TOHS99] Tarr, P. L.; Ossher, H.; Harrison, W. H.; Sutton, S. M. J.: N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, S. 107–119, 1999.

[Tor04] Torgersen, M.: The expression problem revisited. In *ECOOP*, S. 123–143, 2004.

[TR06] Tobias Rho, M. A., G. K.: Fine-grained generic aspects, workshop on foundations of aspect-oriented languages (foal'06), aosd 2006. Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), in conjunction with Fifth International Conference on Aspect-Oriented Software Development (AOSD.06), March 20-24, 2006, Bonn, Germany, Mar 2006.

[VN96] VanHilst, M.; Notkin, D.: Using C++ templates to implement role-based designs. In *ISOTAS '96: Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software*, S. 22–37. Springer-Verlag, London, UK, 1996.

[Weg90] Wegner, P.: Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, Band 1, Nr. 1, S. 7–87, 1990.

[WL05] Wu, P.; Lieberherr, K.: Shadow programming: Reasoning about programs using lexical join point information. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*, 2005.

[WZ88] Wegner, P.; Zdonik, S. B.: Inheritance as an incremental modification mechanism or what like is and isn't like. In Gjessing, S.; Nygaard, K. (Hrsg.): *ECOOP*, Lecture Notes in Computer Science, Band 322, S. 55–77. Springer, 1988.

[Zam98] Zamir, S.: *Handbook of Object Technology.* CRC Press, Inc., Boca Raton, FL, USA, 1998.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 25. August 2006

Martin Kuhlemann