Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Merkmalsorientierte Programmierung in C++

Verfasser:

Marko Rosenmüller

14. August 2005

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

Rosenmüller, Marko:
Merkmalsorientierte Programmierung in C++
Diplomarbeit
Otto-von-Guericke-Universität Magdeburg, 2005.

Zusammenfassung

Die *merkmalsorientierte Programmierung (Feature-Oriented Programming – FOP)* ist ein Paradigma der Softwareentwicklung, welches das Erstellen von Software durch das Zusammensetzen modular vorliegender Merkmale ermöglicht. Mit *aspektorientierter Programmierung (Aspect-Oriented Programming – AOP)* wird das Ziel verfolgt Eigenschaften von Software, die über große Teile des Programmcodes verteilt sind, zu kapseln und von anderen Eigenschaften zu trennen. Beiden Ansätzen ist eines gemein: Die Separate Behandlung der Merkmale einer Software (*Separation of Concerns*). Dennoch sind die Ansätze in ihrer Umsetzung von Grund auf verschieden und haben sowohl Vor- als auch Nachteile.

In dieser Arbeit wird FeatureC++ vorgestellt, eine Umsetzung der FOP für C++, die Elemente der AOP integriert. Es werden zwei Dinge gezeigt: (1) Es ist eine Anwendung der AOP auf merkmalsorientierten Quelltext möglich, und (2) die FOP lässt sich auf Aspekte erweitern. Dabei wird festgestellt, dass beide Ansätze von dieser Kombination profitieren.

Stichworte: Aspektorientiertes Programmieren, AHEAD, C++, Merkmalsorientiertes Programmieren, Produktlinien

Abstract

Feature-Oriented Programming (FOP) attempts to describe and assemble software from feature modules. *Aspect-Oriented Programming (AOP)* focuses on the separation and encapsulation of features that crosscut large parts of the program code. Both concepts follow the well known principles *Separation of Concerns*. Nevertheless they use completely different techniques for implementation, each having their pros and cons.

This work presents FeatureC++, a feature-oriented extension to C++ which incorporates aspect-oriented language elements. Two conclusions are made: (1) AOP concepts can be applied to feature-oriented designs, and (2) the FOP approach can be extended to aspects. Additionally, it is shown that both, FOP and AOP, profit from this combination.

Keywords: Aspect-Oriented Programming, AHEAD, C++, Feature-Oriented Programming, Product Lines

Danksagung

An dieser Stelle möchte ich all denen meinen Dank denen aussprechen, die mich bei der Erstellung dieser Arbeit unterstützt haben. Dieser Dank gilt vor allem meinen Betreuern Prof. Dr. rer. nat. habil. Gunter Saake, Dipl.-Inf. Sven Apel and Dipl.-Wirtsch.-Inf. Thomas Leich. Intensive Diskussion haben wesentlich zum Erfolg der Arbeit beigetragen und darüber hinaus gemeinsame Veröffentlichungen ermöglicht.

Des Weiteren möchte ich besonders Mandy Küsel, Falko Löbner, Jan Reidemeister und Robert Vernunft für umfangreiche Hinweise zu Inhalt und Form danken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AHEAD	Algebraic Hierarchical Equations for Application Design
AOP	Aspect-Oriented Programming
AST	Abstract Syntax Tree
ATS	AHEAD Tool Suite
DRC	Design Rule Check
FOA	Feature-Oriented Analysis
FODA	Feature-Oriented Domain Analysis
FOP	Feature-Oriented Programming
FOSE	Feature-Oriented Softwareengineering
GP	Generative Programming
JL	Java Layers
JTS	Jakarta Tool Suite
MDSOC	Multi-Dimensional Separation of Concerns
OOP	Object-Oriented Programming
OOSE	Object-Oriented Softwareengineering
PUMA	Pure Manipulator
SQL	Structured Query Language
UML	Unified Modeling Language

Kapitel 1

Einleitung

1.1 Motivation

Die Softwareentwicklung muss heutzutage sehr hohen Ansprüchen genügen. Immer komplexere Software muss mit hoher Qualität und Produktivität entwickelt werden. Außerdem wird eine einfache Wartung und Erweiterbarkeit der entwickelten Software verlangt. Dies kann nur mit einem hohen Maß an Wiederverwendung erreicht werden. Die heutige Softwareentwicklung beruht auf *objektorientierter Programmierung (Object-Oriented Programming – OOP)*, die diesen Ansprüchen nicht genügt. Bereits seit Jahrzehnten stellt die fehlende Skalierung bei der Wiederverwendung ein wesentliches Problem dar, das als Softwarekrise bekannt ist [Dij72]. Ursachen sind fehlende Rücksicht auf Wiederverwendbarkeit bei der Softwareentwicklung und die fehlende Möglichkeit wieder verwendbare Komponenten auf einfache Weise zusammenzusetzen. Die Automatisierung des Konfigurationsprozesses ist deshalb Gegenstand aktueller Forschung. Das Erstellen von Software soll zukünftig durch das Zusammenstellen der benötigten Eigenschaften erfolgen. Auf diese Weise lassen sich ähnliche Softwareprodukte generieren, die unterschiedlichen Ansprüchen genügen. Diese *Produktlinien* sind bereits aus anderen Industriezweigen bekannt und dort außerordentlich erfolgreich.

Mit *merkmalsorientierter Programmierung* (*Feature-Oriented Programming – FOP*) wird eine einfache Entwicklung von Produktlinien erreicht [BT97, CBML02]. Ein Schichtenaufbau, wie in der FOP, trägt dabei wesentlich zur Wiederverwendbarkeit, Erweiterbarkeit und Konfigurierbarkeit von Software bei [Par79]. Dies wurde in Beispielen aus dem Bereich Datenbanken [BCGS95, LAS01], Middleware [AB05], Avionics [BCGS95] und Netzwerkprotokollen [BO92] bestätigt. Eine Erfolg versprechende Umsetzung des FOP Ansatzes für Java ist die *AHEAD Tool Suite (ATS)*[1]. Mit *C++ Mixin Layers* [SB00] und *P++* [SB93] existieren auch Ansätze für C++. Diese verwenden allerdings eine

[1] http://www.cs.utexas.edu/users/schwartz/Hello.html

umständliche Syntax bei der Programmierung und eine unübersichtliche Konfiguration. Ihre Anwendbarkeit für die Entwicklung komplexer Software mit C++ ist daher nicht gegeben.

Dennoch ist C++ im industriellen Einsatz sehr weit verbreitet und wird es sicher auch in näherer Zukunft bleiben. In bestimmten Umgebungen (z. B. systemspezifische Anwendungen wie Betriebssysteme oder eingebettete Systeme) ist C++ derzeit nicht wegzudenken. Mit dem ANSI Standard existiert weiterhin eine Grundlage für die systemunabhängige Softwareentwicklung mit C++. Die Unterstützung der Entwicklung komplexer wieder verwendbarer Software mit C++ ist daher Gegenstand dieser Arbeit. Dazu wird FeatureC++ entwickelt, eine Erweiterung von C++ zum merkmalsorientierten Programmieren. Im Vergleich zu C++ Mixin Layers und P++ wird eine einfache, leicht verständliche Programmierung und Konfigurierung von Software erreicht. Außerdem werden Besonderheiten von C++ wie z. B. Templates einbezogen.

Aspektorientierte Programmierung (Aspect-Oriented Programming – AOP) [KLM+97] versucht Eigenschaften von Software zu kapseln, die sich über große Teile des Programmcodes erstrecken (*Crosscutting Modularity*). Diese werden getrennt vom übrigen Quelltext abgelegt, was eine bessere Strukturierung und Verständlichkeit des Programmcodes ermöglicht. Des Weiteren kann dadurch Redundanz in der Implementierung verhindert werden. Mit *AspectC++*[2] [SGSP02] existiert auch eine Umsetzung der AOP für die Sprache C++.

Beide Ansätze, FOP und AOP, sind viel versprechend im Hinblick auf die Entwicklung wieder verwendbarer Software. Es existieren aber in beiden Ansätzen sowohl Vorals auch Nachteile, so dass die Kombination von AOP und FOP in der aktuellen Forschung untersucht wird [MO04, LLO03, MFH01]. Eine vollständige Integration beider Ansätze existiert derzeit nicht. Mit FeatureC++ wird eine solche Integration vorgestellt. Es wird gezeigt, dass (1) durch die Verwendung aspektorientierter Elemente Probleme der FOP gelöst werden können und (2) dass sich der Ansatz der FOP auf *Aspekte*, die grundlegenden Elemente der AOP, erweitern lässt. Beide Ansätze können dabei von der Kombination profitieren. In Ansätzen, die Elemente der FOP und AOP kombinieren, verhindern unter anderem das *Konstruktorproblem*, das *Feature Optionality Problem* und das *Self Problem* praktische Anwendbarkeit [LHBC05]. Mit FeatureC++ werden diese Probleme zum Teil gelöst, und in einer Analyse wird aufzeigt dass weitere Lösungsmöglichkeiten bestehen.

Aufbauend auf der Spracherweiterung wird deren prototypische Umsetzung in einem Codetransformationssystem vorgestellt. Mit Hilfe des Prototyps wird gezeigt, dass die vorgestellte Erweiterung auch unter praktischen Gesichtspunkten für die Entwicklung

[2] http://aspectc.org

von Produktlinien geeignet ist und nicht nur ein theoretisches Konzept darstellt.

1.2 Gliederung

Für das Verständnis der in dieser Arbeit vorgestellten Erweiterung von C++ sind Kenntnisse der Softwareentwicklung notwendig, die in Kapitel 2 vermittelt werden. Dabei werden einige wichtige Konzepte der OOP und des generischen Programmierens besprochen. Außerdem werden Grundlagen der FOP und AOP vermittelt. Abschluss findet das Kapitel mit einer Einführung in die Codetransformation.

Für die Unterstützung bei der Produktlinienentwicklung existieren bereits Methoden, die in Kapitel 3 betrachtet werden. Dabei werden insbesondere die Umsetzungen der FOP und AOP sowie kombinierter Ansätze untersucht und deren Eigenschaften dargestellt. In einer Zusammenfassung erfolgt die Beurteilung der Verwendbarkeit der Ansätze für die Entwicklung von Produktlinien.

Kapitel 4 befasst sich mit dem Entwurf von FeatureC++. Dabei wird untersucht, welche Syntax für die Spracherweiterung notwendig ist und wie Elemente der AOP integriert werden können. Außerdem wird die prototypische Umsetzung mit einem Codetransformationssystem vorgestellt. Abschließend wird die entwickelte Spracherweiterung auf ihre Eignung zur Produktlinienentwicklung untersucht. Dazu werden an einem Beispiel Vor- und Nachteil sowie bestehende Problem betrachtet. Nach einem Vergleich mit anderen Ansätzen folgt eine Untersuchung möglicher Erweiterungen. Die Arbeit schließt mit einer Zusammenfassung in Kapitel 5.

Kapitel 2

Grundlagen

Die *objektorientierte Programmierung* (*Object-Oriented Programming – OOP*) ist Grundlage der derzeitigen Softwareentwicklung. Sie bildet den Ausgangspunkt für Spracherweiterungen, wie sie auch in dieser Arbeit vorgestellt werden. Der erste Teil dieses Kapitels befasst sich daher mit einigen Grundlagen der OOP, die für das weitere Verständnis notwendig sind. Im zweiten Abschnitt werden Grundlagen zum generischen Programmieren vermittelt, welches wesentlich für die Wiederverwendbarkeit von Software ist und oft in Zusammenhang mit OOP verwendet wird. In den darauf folgenden Abschnitten werden Grundlagen der Entwicklung von Produktlinien dargestellt. Abschließend wird in die Thematik der *Codetransformation* eingeführt. Diese spielt eine wesentliche Rolle bei der Erweiterung von Programmiersprachen und findet häufig Anwendung bei der Implementierung von Ansätzen zur Produktlinienentwicklung.

2.1 Objektorientiertes Programmieren

Im Folgenden sollen einige Konzepte der objektorientierten Programmierung betrachtet werden, die für die weitere Arbeit von besonderem Interesse sind. Auf Grundlagen der OOP kann im Rahmen dieser Arbeit nicht eingegangen werden. Hierzu sei auf entsprechende Literatur verwiesen [Zam99, Bru02].

Zur Verdeutlichung verschiedener Konzepte der OOP wird im Weiteren die Programmiersprache C++ verwendet. Eine Einführung und ein Überblick über C++ und andere objektorientierte Sprachen finden sich z. B. in [Str86, ES90, Zam99].

Wichtigste Eigenschaft der OOP ist die Kapselung von Daten und Methoden gleichartiger *Objekte* in einer *Klasse*. Eine Klasse bildet damit eine Vorlage für Objekte und enthält Definitionen von Variablen, die die Speicherung eines Zustandes erlauben. Neben Variablendefinitionen enthält eine Klasse Methoden, die das Manipulieren des Zustands eines Objektes und eine Interaktion mit anderen Objekten ermöglichen. Das Erstellen

eines Objektes aus einer Klasse wird als *Instanziierung* bezeichnet. Während dieses Vorgangs werden den Variablen des Objektes initiale Werte zugewiesen.

Durch die Verwendung von Klassen kann ein hoher Grad an Übersichtlichkeit geschaffen werden. Zusätzlich bildet dies einen wichtigen Ansatz zur Abstraktion von zugrunde liegender Funktionalität. Ein Nutzer einer Klasse muss nichts über die Implementierung der Funktionalität wissen. Sie ist in der Klasse verborgen und kann ausgetauscht werden, ohne dass dies nach außen sichtbar wird.

2.1.1 Wiederverwendung

Für die Wiederverwendung bereits entwickelter Funktionalität stehen im Wesentlichen zwei Verfahren zur Auswahl. Das Zusammenfassen von Klassen zu neuen Klassen wird als *Aggregation* (*object composition – black-box reuse*) bezeichnet. Dadurch werden Klasse zu komplexeren Strukturen zusammengefasst. Weiterhin können Klassen durch *Vererbung* (*inheritance – white-box reuse*) erweitert und verändert werden [GHJV95]. Beides sind grundlegende Möglichkeiten, um schrittweise von Implementierungsdetails zu abstrahieren und ermöglichen so erst die Entwicklung komplexer Softwarestrukturen.

```
1  class A {
2    int i;
3  };
4  class B {
5    int j;
6  };
7  class C {
8    int k;
9    A a;  //Verwendung der Klasse A
10   B b;  //Verwendung der Klasse B
11 };
```

Abbildung 2.1: OOP – Aggregation

In Abbildung 2.1 ist die Aggregation von Klassen dargestellt. Klasse `C` verwendet die Klassen `A` und `B` und bildet somit eine komplexere zusammengesetzte Klasse.

Abbildung 2.2 zeigt die Vererbung. Klasse `Derived` erbt dabei von der Klasse `Base` und wird als *Subklasse* (engl. *subclass*) bezeichnet. Die Klasse, von der geerbt wird (`Base`), wird *Basisklasse* (engl. *baseclass, superclass*) genannt. Klasse `Derived` ist eine Erweiterung der Klasse `Base`. Eine Instanz der Klasse `Derived` enthält damit sowohl die Variable `i` als auch die Variable `j`.

```
1 //Basisklasse
2 class Base {
3   int i;
4 };
5
6 //Subklasse
7 class Derived : public Base {
8   int j;
9 };
```

Abbildung 2.2: OOP – Vererbung

2.1.2 Polymorphismus

Ein weiterer grundlegender Mechanismus der OOP ist der *Polymorphismus*[1]. Dieser findet verschiedene Ausprägungen, wobei im Folgenden nur der Polymorphismus von Objekten näher betrachtet wird. Dabei kann ein Objekt als Instanz einer Klasse sowohl als Objekt dieser Klasse agieren, als auch als Objekt einer seiner Basisklassen. Auf diese Weise können spezialisierte Objekte verwendet werden, ohne dass etwas über die eigentliche Spezialisierung bekannt ist. Der Aufruf von Funktionen dieses Objektes erfolgt dann entsprechend der Verwendung des Objektes im jeweiligen Kontext des Aufrufs.

```
1  //einfaches Array
2  class Array {
3  public:
4    //nicht-virtuelle Funktion
5    void Add(Element* e){..}
6  };
7
8  //sortiertes Array
9  class SortedArray : public Array {
10 public:
11   void Add(Element* e){..}
12 };
13
14 //Verwendung des Array
15 SortedArray* sa = new SortedArray;
16 sa->Add(new Element);
17 Array* a = sa;
18 a->Add(new Element);
```

Abbildung 2.3: Polymorphismus in OOP

```
1  //einfaches Array
2  class Array {
3  public:
4    //virtuelle Funktion
5    virtual void Add(Element* e){..}
6  };
7
8  //sortiertes Array
9  class SortedArray : public Array {
10 public:
11   void Add(Element* e){..}
12 };
13
14 //Verwendung des Array
15 SortedArray* sa = new SortedArray;
16 sa->Add(new Element);
17 Array* a = sa;
18 a->Add(new Element);
```

Abbildung 2.4: Verwendung virtueller Funktionen in OOP

In Abbildung 2.3 ist die Klasse `Array` und die von ihr erbende Klasse `SortedArray` dargestellt. Die Methode `Add` ist in beiden Klassen definiert und implementiert unter-

[1] Polymorphismus – Vielgestaltigkeit

schiedliche Funktionalität. Mit dem Aufruf `sa->Add(...)` (Abbildung 2.3, Zeile 16) wird die Funktion `Add(...)` der Klasse `SortedArray` ausgeführt. Der Aufruf `a->Add(...)` (Zeile 18) hingegen führt die Funktion `Array::Add(...)` aus. In diesem Fall ist aber die Ausführung der Funktion der Klasse `SortedArray` notwendig, da das Element sortiert eingefügt werden muss. Dies kann durch die Verwendung *virtueller* Funktionen erfolgen: Eine virtuelle Funktion kann in einer erbenden Klasse *überschrieben* werden, so dass bei jedem Aufruf der Funktion der Basisklasse die Funktion der Subklasse ausgeführt wird.

Im Gegensatz zu Abbildung 2.3 wird in Abbildung 2.4 durch den Aufruf `a->Add(...)` die Funktion `SortedArray::Add(...)` ausgeführt. Ursache dessen ist die *virtuelle* Definition der Funktion `Array::Add`, die durch die Funktion `SortedArray::Add(...)` überschrieben wird. Auf diese Weise kann Funktionalität ausgeführt werden, ohne dass der Aufrufende die implementierende Klasse kennen muss. Der zum Aufruf solcher Funktionen verwendete Quellcode ist daher mit beliebigen anderen Implementierungen wieder verwendbar. Damit entsteht ein wichtiger Ansatz zur Erzeugung von Klassenvarianten. Jede Klasse implementiert dazu eine Variante der Lösung eines Problems, aber alle Varianten können auf gleiche Weise verwendet werden. Im Beispiel der Abbildung 2.4 kann dadurch eine Funktion der Klasse `SortedArray` ausgeführt werden, ohne dass die Klasse bekannt ist (Zeile 18).

2.1.3 Verschachtelte Klassen

Bisher wurden nur einfache Klassen der OOP betrachtet. Neben diesen können Klassen aber auch innerhalb anderer Klassen existieren (*verschachtelte Klassen* – engl. *nested classes*). Da verschachtelte Klassen für Konzepte die in dieser Arbeit vorgestellt werden von Interesse sind, werden diese im Folgenden kurz betrachtet.

```
1 class Outer {
2   class Inner {};
3 };
```

Abbildung 2.5: Klasse mit innerer Klasse

Verschachtelte Klassen erlauben es, Klassen zusammenzufassen (Abbildung 2.5) und auf diese Weise eine größere Komponente übersichtlicher zu gestalteten. Dabei werden innere und äußere Klassen unterschieden.

Ist eine Vererbungshierarchie äußerer Klassen vorhanden, können auch die inneren Klassen von den inneren Klassen anderer äußerer Klassen erben. In Abbildungen 2.6 und 2.7 ist ein solches Beispiel dargestellt. Die Klasse `OuterDerived` erbt von der Klasse `OuterBase`. Die inneren Klassen `Inner1` und `Inner2` der Klasse `OuterDerived` erben

von den inneren Klassen der Klasse `OuterBase` (Zeilen 2 und 3 in Abbildung 2.7).

```
1 class OuterBase {
2   class Inner1 {};
3   class Inner2 {};
4 };
```

Abbildung 2.6: Innere Klassen

```
1 class OuterDerived : public OuterBase {
2   class Inner1 : public OuterBase::Inner1 { };
3   class Inner2 : public OuterBase::Inner2 { };
4 };
```

Abbildung 2.7: Vererbung innerer Klassen

Die äußeren Klassen lassen sich so als Namensraum (*namespace*) verwenden. Auf diese Weise werden Funktionalitäten, die sich über mehrere Klassen erstrecken, zusammengefasst. Dies ist nicht sehr praktikabel und in dieser Form eher theoretischer Natur. Im Hinblick auf die weitere Arbeit spielt dies allerdings eine besondere Rolle.

2.2 Generisches Programmieren

Die OOP ist sehr erfolgreich und vermag viele Probleme der Softwareentwicklung zu lösen. Dennoch existieren Anwendungsfälle, in denen die Entwicklung mit OOP umständlich ist. Einige dieser können unter Zuhilfenahme generischer Programmierung besser gelöst werden.

Zur Verbesserung der Wiederverwendbarkeit von Algorithmen und Entwurfsmustern ist eine allgemeingültige Implementierung von Interesse, die sich mit beliebigen Klassen verwenden lässt. Häufig werden dazu *Schnittstellen* (engl. *interfaces*) entwickelt. Sie bieten die Möglichkeit einheitlich auf unterschiedliche Klassen zuzugreifen oder diese zu manipulieren. Jede Klasse, die eine solche Schnittstelle implementiert, kann so mit einem beliebigen Algorithmus verwendet werden, der diese Schnittstelle nutzt.

Abbildung 2.8 zeigt die Schnittstelle `Element` (Zeile 1) und ein Array, welches diese verwendet (Zeilen 3-6). Die Schnittstelle wird genutzt, um beliebige Objekte im Array abzulegen. Soll eine Klasse mit diesem Array verwendet werden, muss sie diese Schnittstelle implementieren. Dies ist am Beispiel der Klasse `String` in Zeile 8 dargestellt.

Die dazu notwendige Vererbung ist ein wesentlicher Nachteil des objektorientierten Ansatzes. Einerseits ist ein gewisser Aufwand bei der Implementierung notwendig, andererseits ist in einigen Fällen eine zusätzliche Vererbung nicht möglich. In Abbildung 2.8 ist dies in Zeile 8 zu sehen. Besteht keine Möglichkeit die Vererbung der Klasse `String` von der Klasse `Element` einzuführen (z. B. wenn die Klasse `String` aus einer Bibliothek importiert wird), ist eine Verwendung mit dem Array nicht möglich[2].

[2]Für die Implementierung eines solchen Arrays ließe sich in C++ auch ein Zeiger auf *void∗* als

```
1  class Element { };
2
3  class Array {
4    void Add(Element* elem) {..}
5    Element* Get(int i) const {..}
6  };
7
8  class String : public Element {
9    //Konstruktor
10   String() {..}
11   //Copy-Konstruktor
12   String(const String& src) {..}
13 };
14
15 Array a;
16 a.Add(new String("abc"));
17 String* s = (String*)a.Get(0);
```

Abbildung 2.8: Array Implementierung mit OOP

```
1
2  template <class TYPE>
3  class Array {
4    void Add(TYPE* elem) {..}
5    TYPE* Get(int i) const {..}
6  };
7
8  class String {
9    //Konstruktor
10   String() {..}
11   //Copy-Konstruktor
12   String(String* src) {..}
13 };
14
15 Array<String> a;
16 a.Add(new String("abc"));
17 String* s = a.Get(0);
```

Abbildung 2.9: Array Implementierung mit generischer Programmierung

Eine Möglichkeit dies zu umgehen, besteht in der Verwendung von *Typvorlagen* (*Templates*). Czarnecki und Eisenecker [CE00] definieren Typvorlagen als Vorlagen für Typen oder Werte von Parametern. Die Verwendung solcher Typvorlagen wird als *generisches Programmieren* bezeichnet. Sie ermöglichen das Festlegen der in einer Klasse oder Methode verwendeten Typen zu einem späteren Zeitpunkt.

In Abbildung 2.9 ist das Beispiel aus Abbildung 2.8 unter der Verwendung von Typvorlagen für die Programmiersprache C++ dargestellt. In Zeile 2 wird die Typvorlage mit dem Schlüsselwort `template` definiert. Die Typvorlage `TYPE` kann nach dieser Definition in der darauf folgenden Klassendefinition (Zeilen 3-6) verwendet werden. Die Ersetzung der Typvorlage durch den zu verwendenden Typ erfolgt während der *Instanziierung* des Templates (Zeile 11). Im Vergleich zum objektorientierten Ansatz sind die Unterschiede gering, jedoch tritt damit das angesprochene Problem der notwendigen Implementierung einer Schnittstelle nicht mehr auf.

Die angesprochene Instanziierung der Templates erfolgt im Falle von C++ während der Übersetzung des Programms. Dabei wird für die zu verwendenden Typen spezialisierter Quellcode generiert, der anschließend in Maschinencode übersetzt wird. Andere Implementierungen von Typvorlagen wie in Java erzeugen ähnlich C++ Quellcode, welcher der bereits dargestellten objektorientierten Implementierung entspricht und Typumwandlungen verwendet.

Element verwenden, jedoch ist dies spätestens bei der Erweiterung der Schnittstelle, z. B. für die Verwendung mit einem sortierenden Array, nicht mehr möglich.

Ein weiterer Vorteil der generischen Programmierung wird in Zeilen 17 der beiden Beispiele sichtbar. Während die Funktion `Get` der OOP einen Zeiger auf die Schnittstelle liefert, wird im generischen Ansatz ein Zeiger auf den eigentlich verwendeten Typen zurückgegeben. Im ersteren Fall führt dies zur Notwendigkeit einer Typumwandlung und damit zu einer nicht typsicheren Implementierung (Zeile 17 der Abbildung 2.8). Hier kann teilweise Abhilfe durch Verwendung einer neuen Klasse geschaffen werden. Diese erbt von der Klasse `Array` und nimmt alle notwendigen Typumwandlungen vor. So kann die Gefahr einer falschen Typumwandlung minimiert und bei häufiger Verwendung der Klasse auch Implementierungsarbeit verringert werden. Eine typsichere Implementierung lässt sich allerdings nur mit dem generischen Ansatz realisieren.

```
1  class Element {
2    //Abstrakte Methode zum Duplizieren
3    //eines Elements
4    virtual Element* Duplicate() const = 0;
5  };
6
7
8  class Array {
9    ..
10
11   //Array kopieren
12   void Copy(const Array& src) {
13     for (int i=0; i<src.Length(); i++)
14       Add(src.Get(i)->Duplicate());
15   }
16 };
17
18 class String : public Element {
19   //Implementierung zum Duplizieren
20   //eines Strings
21   virtual Element* Duplicate() const {
22     return new String(*this);
23   };
24 };
```

Abbildung 2.10: Schnittstellen in OOP

```
1
2
3
4
5
6
7  template <class TYPE>
8  class Array {
9    ..
10
11   //Array kopieren
12   void Copy(const Array& src) {
13     for (int i=0; i<src.Length(); i++)
14       Add(src.Get(i)->Duplicate());
15   }
16 };
17
18 class String {
19   //Implementierung zum Duplizieren
20   //eines Strings
21   virtual String* Duplicate() const {
22     return new String(*this);
23   };
24 };
```

Abbildung 2.11: Implizite Schnittstellen bei generischer Programmierung

Der generische Ansatz ist auch bei der Verwendung einer komplexeren Schnittstelle praktikabel. Im objektorientierten Ansatz wird die zu verwendende Schnittstelle explizit vorgegeben (siehe Abbildung 2.10, Zeilen 1-5), im generischen Ansatz hingegen wird sie implizit durch die Verwendung der Typvorlage definiert. In den Abbildungen 2.10 und 2.11 verlangen die Klassen `Array` jeweils die gleiche Schnittstelle (die Methode `Duplicate` eines Elementes). Während diese für den OOP-Ansatz explizit angegeben ist, wird sie in der generischen Implementierung durch die Verwendung der Methode

`Duplicate` (Abbildung 2.11 Zeile 14) impliziert. Im Vergleich zur OOP ist die zu implementierende Schnittstelle für den Anwender nicht direkt sichtbar, so dass er auf entsprechende Dokumentation oder Fehlermeldungen des Compilers angewiesen ist.

2.3 Merkmalsorientiertes Programmieren

Bisher wurden mit OOP und generischer Programmierung Grundlagen der Softwareentwicklung vorgestellt, die sich seit langer Zeit bewährt haben. Mit dem *merkmalsorientierten Programmieren* (*Feature-Oriented Programming – FOP*) soll im Folgenden ein wesentlich neuerer Ansatz vorgestellt werden, der als Weiterentwicklung der OOP betrachtet werden kann. Zum Teil findet eine Umsetzung der FOP mit Hilfe der OOP und generischem Programmieren statt. Dennoch ist die Herangehensweise bei Entwurf und Programmierung eine andere als bei der OOP und beruht auf den Merkmalen einer Software.

Ein komplexes Softwaresystem lässt sich auf Grund seiner Eigenschaften oder auch Merkmale (*Features*) beschreiben. Diese Merkmale sind in Komponenten [CE00, Szy97, HC91, Gri98, NAT94] verschiedener Größen implementiert und reichen von einzelnen Klassen bis zu Komponenten, die aus einer großen Zahl von Klassen oder auch wiederum aus kleineren Komponenten bestehen. Für den Begriff der Komponente existieren verschiedenen Definitionen. So bezeichnen Hooper et al. [HC91] alles als Komponente, was wieder verwendbar ist. In [NAT94] werden alle an der Softwareentwicklung beteiligten Artefakte als Komponente bezeichnet. Hier soll der Begriff wie in [Gri98] verwendet werden: Als Software, die in binärer Form oder als Quelltext vorliegt, eine bestimmte Funktionalität besitzt und über eine Schnittstelle mit anderen Komponenten zusammenarbeiten kann.

Die Erstellung einer Software ist im Idealfall das Zusammenfügen verschiedener Komponenten. Dieses Zusammenstellen von Teilen einer Software zu einem fertigen Produkt oder zu einer größeren Softwarekomponente wird als *Konfiguration* bezeichnet. Häufig lassen sich Merkmale einer Software nicht in Komponenten fassen, sondern sind Bestandteil vieler Komponenten. Daher ist ein Zusammenfügen von Komponenten meist nicht ausreichend für die Konfiguration einer Software. Hier ist eine Kombination von Merkmalen notwendig, welche nicht als Komponenten vorliegen.

Das Zusammenstellen einer Software mit Hilfe ihrer Merkmale ist Gegenstand der *Produktlinienentwicklung*. Wie auch in anderen Industriezweigen üblich, werden dabei aus einer Menge von Merkmalen die benötigten ausgewählt. So lässt sich eine Reihe ähnlicher Programme, eine *Produktlinie*, entwickeln. Jedes enthaltene Produkt erfüllt dabei verschiedene Bedürfnisse der Anwender. Im eigentlichen Sinne steht der Begriff

Produktlinie für Programme, die aus einer bestimmten Sichtweise zusammengehörig sind [BCS00]. Withey bezeichnet Programme als Produktlinie, die teilweise gleiche Eigenschaften besitzen, welche auf einem bestimmten Markt einen Zweck erfüllen [Wit96].

Neben den Produktlinien wird in der Softwareentwicklung von *Programmfamilien* gesprochen. Der Begriff wurde zuerst von Parnas verwendet, um Programme zusammenzufassen, die eine Menge gleicher Eigenschaften besitzen [Par76]. Withey beschreibt eine Programmfamilie als Menge von Programmen, die aus einer Menge von Elementen erstellt werden können [Wit96]. Die Definition von Produktlinien folgt daher einer Marketing Strategie, wohingegen sich Programmfamilien über eine ihnen gemeinsame Menge technischer Eigenschaften definieren[3] [CE00]. Es können demnach ein oder mehrere Programmfamilien notwendig sein, um eine Produktlinie zu erstellen und eine Programmfamilie kann in mehreren Produktlinien Anwendung finden [CE00].

Bei der *merkmalsorientierten Softwareentwicklung* (*Feature-Oriented Softwareengineering – FOSE)* werden die Merkmale einer Software während des gesamten Entwicklungsprozesses als wesentliche Bestandteile betrachtet und getrennt von anderen Artefakten behandelt. Dies betrifft unter anderem die Domänenanalyse (*Feature-Oriented Domain Analysis – FODA* [KCH+90]), den Entwurf, die Implementierung mit FOP, Dokumentation, Konfiguration und Test. Im Weiteren werden nur Entwurf, Implementierung und Konfiguration betrachtet, wobei das Hauptaugenmerk der FOP gilt.

2.3.1 Separation of Concerns

Sowohl die Merkmale einer Software als auch alle weiteren Artefakte der Softwareentwicklung werden als *Belange* (*concerns*, *points of interest*) bezeichnet [Par72]. Sutton et al. [SR03] unterteilen die Belange der Softwareentwicklung in logische (Konzepte, Eigenschaften, Probleme) und physische (Einheiten der Software bzw. Elemente ihrer Implementierung wie Komponenten und Klassen). Zu den physischen Belangen gehören dabei z. B. UML-Diagramme, Klassen der OOP, Dokumentationselemente und Testkonzepte. Häufig wird der Begriff Concern nur im Sinne der logischen Belange und unabhängig von den physischen Belangen gebraucht.

Der oft gebrauchte Vergleich zur Automobilindustrie lässt sich sehr gut zur Verdeutlichung verwenden. Ein Fahrzeugmerkmal wie „Automatikgetriebe“, kann nicht ohne weiteres in eine Klasse gefasst. Eine Klasse `Automatikgetriebe` könnte zwar existieren, das Merkmal „Automatikgetriebe“ betrifft aber nicht nur diese Komponente. Die Pedale (fehlendes Kupplungspedal, breiteres Bremspedal) oder der Motor sind ebenfalls abhängig von diesem Merkmal. Ein Merkmal überschneidet sich mit anderen Merkmalen.

[3] Die Begriffe Produktlinie und Programmfamilie werden in der Literatur nicht einheitlich verwendet und zum Teil auch vertauscht.

Man spricht von *Crosscuts* und *crosscutting Concerns* [BLS03].

Die Merkmale einer Software sind für den Anwender besonders wichtig, da sie intuitiv verständlich und meist einfach formulierbar sind. Sie bilden deshalb die Grundlage bei der Entwicklungsplanung. Die Elemente der Implementierung hingegen sind häufig abstrakter Art, aber für den Entwicklungsprozess zwingend notwendig. Daraus ergibt sich eines der zurzeit wesentlichsten Probleme der Softwareentwicklung: In der Implementierung ist nur eine dieser Sichtweisen möglich. Daher muss eine Entscheidung für eine Sicht getroffen werden. Diese fällt bei der Programmierung folglich auf die Elemente der Implementierung [OT00]. Die zur Beschreibung verwendeten Merkmale sind damit über viele Klassen, oder auch die gesamte Software verteilt und lassen sich ohne weiteres nicht mehr erkennen. Es entsteht Programmcode der schwer verständlich und schlecht zu warten oder erweitern ist. Dabei werden insbesondere zwei Probleme betrachtet:

- *Code tangling* wird verwendet, um herauszustellen, dass Programmcode mit anderem Programmcode verflochten ist, obwohl er in Bezug auf die Funktion nur begrenzt mit diesem in Verbindung steht.

- *Code scattering* bezeichnet das über große Bereiche des Quelltextes verstreute Vorliegen von Programmcode. Der Zusammenhalt dieses Codes und damit des Merkmals das er implementiert, geht dabei verloren.

Zur Lösung dieser Probleme wird in der FOP eine *Trennung der Belange* der Softwareentwicklung (*Separation of Concerns*) vorgenommen. Dabei wird unter anderem versucht Merkmale einer Software gegeneinander abzugrenzen und unabhängig voneinander zu behandeln. In diesem Zusammenhang wird auch von der *Dekomposition* der Software (*software decomposition*) gesprochen [Par72]. Für die Entwicklung komplexer Software und von Produktlinien ist dies grundlegende Voraussetzung.

2.3.2 Domänenanalyse

Die Betrachtung der Merkmale einer Software oder Softwarekomponente ist bereits während der Analyse wesentlich. In FODA werden dazu Merkmalsdiagramme verwendet. Sie stellen neben den Merkmalen selbst auch die Beziehungen zwischen diesen dar. In Abbildung 2.12 ist ein Merkmalsdiagramm für eine Komponente dargestellt, die eine Liste implementiert. Das Diagramm ist ein Baum, dessen Knoten die einzelnen Merkmale darstellen. Die Liste selbst, als Wurzel des Baums, hat demnach zwei Merkmale: Synchronisation (`Sync`) und Sortierung (`Sort`). Wird ein Merkmal nicht weiter in Teilmerkmale unterteilt, so stellt es ein Blatt des Baumes dar. Bei weiterer Unterteilung werden alle Teilmerkmale als Kinder des Merkmals abgebildet.

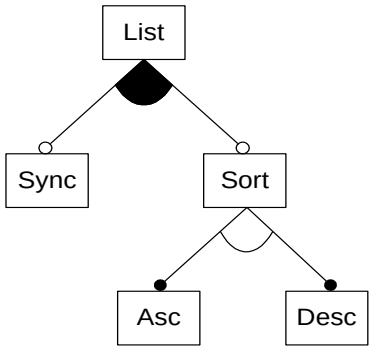

Abbildung 2.12: Merkmalsdiagramm einer Liste

Zur Kennzeichnung der Beziehungen der Merkmale untereinander werden Teilmerkmale mit einem Kreisbogen verbunden. Ist dieser ausgefüllt, sind die Merkmale gleichzeitig auswählbar. Ein nicht ausgefüllter Kreisbogen wird zur Darstellung einander ausschließender Merkmale verwendet. Von diesen Merkmalen findet immer nur eines gleichzeitig Anwendung in einer konkreten Software. Das Merkmal Synchronisation der Abbildung 2.12 ist daher zusammen mit dem Merkmal Sortierung anwendbar. Hingegen schließen sich die Teilmerkmale `Asc` (aufsteigende Sortierung) und `Desc` (absteigende Sortierung) aus.

Des Weiteren beschreiben die Endpunkte der Verbindungen, ob ein Merkmal optional ist (nicht ausgefüllter Kreis) oder zwingend erforderlich (ausgefüllter Kreis). So sind die Synchronisation und die Sortierung im Beispiel optional. Ist jedoch eine Sortierung vorhanden, muss ausgewählt werden, ob auf- oder absteigend sortiert werden soll (ausgefüllte Kreise an `Asc` und `Desc`). In Kombination mit dem nicht ausgefüllten Kreisbogen ergibt sich, dass bei Benutzung einer Sortierung entweder aufsteigende oder absteigende Sortierung verwendet werden muss. Weitere mögliche Kombinationen in Merkmalsdiagrammen werden in [CE00] diskutiert.

2.3.3 Kollaborationenentwurf

Die in der Analyse entwickelten Merkmale finden sich auch im Entwurf wieder. Eine Trennung dieser Merkmale wird im *Kollaborationenentwurf* vorgenommen. Dabei werden die Klassen entsprechend der Merkmale an denen sie teilhaben zerlegt. Auf Grund der Zusammenarbeit der Klassen innerhalb eines Merkmals, werden die Merkmale als *Kollaborationen* bezeichnet.

Die Zerlegung erfolgt dabei mit Hilfe einer Matrix, dargestellt in einem *Kollaborationendiagramm* (Abbildung 2.13). In den Zeilen der Matrix werden die Merkmale oder auch Kollaborationen einer Software oder Komponente aufgeführt, in den Spalten die

Elemente der Implementierung, die Klassen. Die Schnittpunkte beider und damit die Elemente der Matrix werden als *Rollen* bezeichnet. Sie stellen die jeweilige Aufgabe und Funktion dar, die eine Klasse in einer bestimmten Kollaboration einnimmt. Die Klasse `A` der Abbildung 2.13 nimmt eine Rolle in allen Kollaborationen ein, es ist also für die Implementierung aller Merkmale eine Veränderung der Klasse notwendig. `A2` ist die Rolle, die Klasse `A` in Kollaboration `2` einnimmt und enthält daher alle für dieses Merkmal wesentlichen Funktionen und Daten. Die Klassen `B` und `C` sind hingegen nicht von allen Merkmalen betroffen.

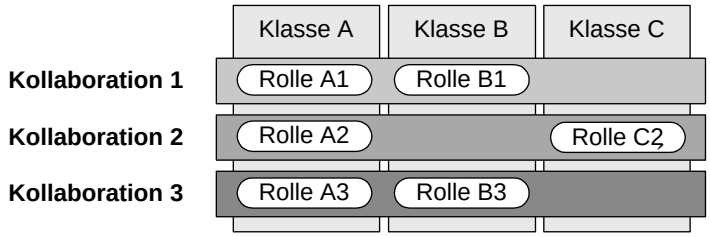

Abbildung 2.13: Kollaborationendiagramm

Das Entfernen oder Hinzufügen eines Merkmals zu einer Komponente entspricht somit dem Hinzufügen oder Entfernen der einzelnen Rollen. Während des Entwurfs werden im Kollaborationendiagramm alle existierenden Merkmale einer Software aufgelistet, auch wenn eine Kombination mit anderen Merkmalen nicht möglich ist. Die zur Erstellung einer bestimmten Software verwendeten Merkmale werden während der *Konfigurierung* festgelegt. Zu diesem Zeitpunkt muss ebenfalls entschieden werden, ob die verwendeten Merkmale miteinander kombinierbar sind, eine bestimmte Konfiguration also gültig ist.

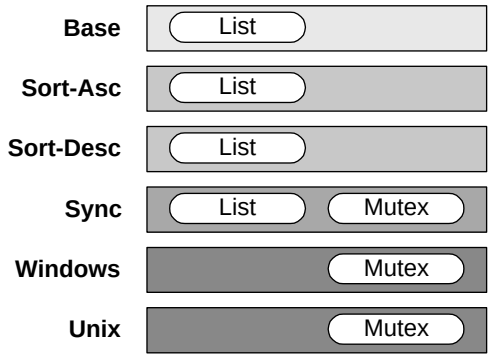

Abbildung 2.14: Kollaborationendiagramm einer Liste und eines Synchronisationsobjektes

Für das Beispiel der Liste aus Abbildung 2.12 ergibt sich ein Kollaborationenentwurf

wie in Abbildung 2.14. Zusätzlich wurde die Klasse `Mutex` hinzugefügt, die eine Klasse zur Synchronisation darstellt. Hinzugekommen sind auch die Merkmale `Unix` und `Windows`, welche systemabhängige Eigenschaften darstellen und sich gegenseitig ausschließen. Die Klasse `List` nimmt Rollen in der Kollaboration `Base` (eine solche wird häufig definiert, um grundlegende Funktionalität zu implementieren), den Sortierungsmerkmalen (`Sort-Asc` und `Sort-Desc` – zur auf- bzw. absteigenden Sortierung) sowie der Synchronisation (Kollaboration `Sync`) ein. Von den Kollaborationen `Windows` und `Unix` ist die Klasse nicht betroffen.

Die Klasse `Mutex` nimmt Rollen in `Sync`, `Windows` und `Unix` ein. Sie wird von der Klasse `List` zur Synchronisation verwendet, die damit unabhängig von der jeweils zugrunde liegenden systemspezifischen Implementierung ist.

2.3.4 GenVoca

Die Implementierung des Entwurfs stellt ein Problem dar, da für die Programmierung nur die Implementierungselemente (Klassen bzw. Komponenten) von Interesse sind. Aufbauend auf den Softwaregeneratoren *Genesis* [Bat88] und *Avoca* [BCGS95] stellt *GenVoca* [BO92] eine Theorie zur Umsetzung des Kollaborationenentwurfs und gleichzeitig eine mathematische Beschreibung dieser Umsetzung bereit. Dabei wird eine Klasse aus *Konstanten* und deren *Verfeinerungen* (*Refinements*) erzeugt. Diese entsprechen den Rollen des Kollaborationenentwurfs. Ausgehend von der Konstanten wird dieser durch eine Verfeinerung minimale Funktionalität hinzugefügt. Die Verfeinerungen selbst werden als *Funktionen* bezeichnet und können auch auf bereits verfeinerte Konstanten angewandt werden. Viele aufeinander aufbauende Verfeinerungen (*progressive refinements*, *step-wise refinements*) werden verwendet, um die endgültige Klasse zu erzeugen. Sie entsprechen damit den Rollen des Kollaborationenentwurfs.

Die Verfeinerungen haben Ähnlichkeit mit der Vererbung der OOP. Im Gegensatz zur Vererbung ist dieser Prozess der sukzessiven Erweiterung aber linearer Natur. Es besteht daher nicht die Möglichkeit in Mitten einer solchen Kette von Verfeinerungen eine Verzweigung zu erzeugen, wie es bei der Vererbung möglich ist.

Die Funktionen des GenVoca Ansatzes können auf bereits verfeinerte Klassen angewandt werden. Auf diese Weise lässt sich jede Klasse C als Gleichung darstellen:

$$C = C_2(C_1). \tag{2.1}$$

Funktion C_2 ist Verfeinerung der Konstanten C_1. In GenVoca wird hierzu auch die folgende Operatorschreibweise verwendet:

$$C = C_2 \bullet C_1. \tag{2.2}$$

Mehrere aufeinanderfolgende Verfeinerungen lassen sich als Verkettungen der Funktionen darstellen:

$$\begin{aligned} C &= C4(C3(C2(C1))) \\ &= C4 \bullet C3 \bullet C2 \bullet C1. \end{aligned} \tag{2.3}$$

Der verwendete Verfeinerungsoperator ist assoziativ: Sind beide Operanden eine Verfeinerung, so ist auch das Ergebnis eine Verfeinerung, die sich wiederum auf weitere Verfeinerungen und Klassen anwenden lässt. Ist der zweite Operand eine Klasse, so ist das Ergebnis ebenfalls eine Klasse. Kommutativität des Operators ist hingegen nicht gegeben, da bereits angewandte Verfeinerungen von neuen überschrieben werden. Die Reihenfolge, in der die Verfeinerungen angewandt werden, ist daher von besonderem Interesse.

```
1  class C1 { .. };
2  class C2 : public C1 { .. };
3  class C3 : public C2 { .. };
4  class C4 : public C3 { .. };
5  typedef C4 C;
```

Abbildung 2.15: Implementierung von Verfeinerungen mit OOP

Eine Verkettung wie in Gleichung 2.3 lässt sich mit einer Vererbung der OOP wie in Abbildung 2.15 wiedergeben. Die Verfeinerungen $C2$ bis $C4$ entsprechen dabei den Veränderungen, die durch die Klassen `C2` bis `C4` der Klasse `C1` hinzugefügt werden[4]. Die letzte Vererbung ergibt damit die vollständig verfeinerte Klasse `C`, im Beispiel durch die Typdefinition in Zeile 5 wiedergegeben wird.

Um mehrere Klassen analog dem Kollaborationenentwurf zu verändern, wird dieser Ansatz auf das gleichzeitige Verfeinern mehrerer Klassen erweitert (*large scale refinements*). Eine Konstante P ist damit die Menge aller Konstanten der Klassen und eine Verfeinerung R die Menge aller Verfeinerungen der für eine Kollaboration notwendigen Klassen:

$$\begin{aligned} P &= \{X_P, Y_P, Z_P\}, \\ R &= \{X_R, Z_R\}. \end{aligned} \tag{2.4}$$

X_P in Gleichung 2.4 ist somit Konstante der Klasse X. X_R ist Verfeinerung dieser Klasse, die für die gesamte Verfeinerung R notwendig ist. P und R entsprechen damit Kollaborationen. Die Konstanten und Verfeinerungen der Klassen entsprechen den Rollen des

[4]Die Klassen sind nicht den Verfeinerungen gleichzusetzen, da sie bereits Funktionalität der zuvor implementierten Verfeinerungen enthalten.

Kollaborationenentwurfs. Auf diese Weise lässt sich eine Software mit allen beteiligten Klassen durch eine Gleichung beschreiben. Die Komposition $R \bullet P$ einer Software wird dann aus der Menge der einzelnen Kompositionen gebildet:

$$R \bullet P = \{X_R \bullet X_P, Y_P, Z_R \bullet Z_P\}. \tag{2.5}$$

Die einzelnen Verfeinerungen, wie $X_R \bullet X_P$, sind Verfeinerungen einzelner Klassen wie auch in Gleichung 2.3.

Die Beschreibung einer Software durch Gleichungen hat einen weiteren Vorteil, der mit GenVoca in zukünftigen Erweiterungen umgesetzt werden soll: Es können Optimierungen einer Software vorgenommen werden. Diese ähneln den Optimierungen der SQL (Structured Query Language), welche ebenfalls auf algebraischen Ausdrücken basieren. Anhand vorgegebener Zielfunktionen lassen sich die verwendeten Gleichungen optimieren, um eine Software analytisch hinsichtlich Laufzeit oder Ressourcenverbrauch anzupassen.

Häufig wird in Verbindung mit dem Kollaborationenentwurf der Begriff *Schicht* (*Layer*) verwendet, der einer Kollaboration entspricht. Die in einer Konfiguration verwendeten Schichten werden auch als *Stapel* (engl. *stack*) bezeichnet. Wie bereits erwähnt, ist der Verfeinerungsoperator nicht kommutativ. Aus diesem Grund ist in den meisten Fällen eine bestimmte Sortierung der verwendeten Schichten notwendig, um eine gültige Konfiguration zu erhalten. Diese Reihenfolge der Verfeinerungen, die *Verfeinerungshierarchie*, wird oft mit der Vererbungshierarchie der OOP verglichen. Darin werden erbende Klassen meist unterhalb ihrer Basisklasse dargestellt. So ist mit der Bezeichnung *oberste Schicht* die am wenigsten spezialisierte (in GenVoca die Konstante) Schicht gemeint. Hingegen lässt die Bezeichnung als Stack auch die umgekehrte Verwendung zu: die oberste Schicht ist dabei die zuletzt auf dem Stack platzierte und damit am weitesten verfeinerte Schicht. Im Weiteren soll erstere Bezeichnung verwendet werden, da so Übereinstimmung mit der Vererbung der OOP gegeben ist.

Entwurfsregeln Mit GenVoca-Gleichungen lassen sich beliebige Kombinationen von Merkmalen darstellen. Syntaktische und semantische Korrektheit einer Konfiguration ist jedoch nicht zwangsläufig gegeben. So könnte ein Merkmal M die Verwendung eines Merkmals N ausschließen. Es lässt sich dennoch eine Gleichung angeben, die beide Merkmale verwendet. Beschränkungen der möglichen Gleichungen sind daher notwendig, um ungültige auszuschließen. Diese Beschränkungen werden als *Entwurfsregeln* (engl. *design rules*) bezeichnet. Sie definieren für jedes Merkmal notwendige Vorbedingungen (*preconditions*) und Nachbedingungen (*postconditions*). Durch das Überprüfen dieser Regeln (*Design Rule Check – DRC*) kann festgestellt werden, ob eine ungültige Konfigurationen vorliegt. Ob eine gültige Konfiguration vorliegt kann hingegen nicht immer entschieden

werden, da es für komplexe Systeme eine kaum zu bewältigende Aufgabe wäre, entsprechende Regeln zu entwerfen. Seiteneffekte durch die Kombination zweier Verfeinerungen einer Komponente sind nicht auszuschließen. Umgekehrt besteht so auch die Möglichkeit aus den vorhandenen Regeln mögliche Kombinationen zu ermitteln, um so interaktiv eine Konfiguration einer Software zu erstellen.

AHEAD

Algebraic Hierarchical Equations for Application Design (AHEAD) wurde als Erweiterung von GenVoca entworfen. Mit diesem Ansatz wird versucht, auch andere Teile des Softwareentwicklungsprozesses in die algebraische Beschreibung von GenVoca einzubeziehen. Da auch zwischen anderen Elementen dieses Prozesses und den Merkmalen einer Software Crosscuts entstehen, ist eine Zerlegung dieser notwendig. So werden neben den Implementierungselementen einer Software, auch Artefakte wie Dokumentation, UML-Diagramme oder Testszenarien berücksichtigt. Diese werden, wie auch die Klassen, im Kollaborationendiagramm dargestellt und können ebenso konfiguriert werden. So ist Zusammenstellen eines Softwaresystems mit all seinen Elementen auf sehr einfache Weise möglich.

Mit AHEAD ist es somit möglich Gleichungen wie 2.5 auf nicht Code-Artefakte anzuwenden. So könnte die Konstante X_P auch ein Dokument der Spezifikation sein. Die Funktion X_R würde damit die zur Verfeinerung R gehörige Dokumentation implementieren. AHEAD-Funktionen führen demnach je nach Artefakt andere Operationen aus, da etwa ein UML-Diagramm anders behandelt werden muss als ein Teil der Dokumentation oder eine Klasse.

Die Komposition verschiedener Merkmale kann dadurch unabhängig von allen Elementen der Softwareentwicklung erfolgen. Es ist lediglich eine Gleichung zur Angabe der gewünschten Konfiguration notwendig.

2.3.5 Mixin Layers

Eine Umsetzung des Kollaborationenentwurfs sind *Mixin Layers*. Grundlage für die Implementierung der Rollen bilden *Mixins*, die im Folgenden beschrieben werden. Anschließend wird auf die Implementierung der Mixin Layers näher eingegangen.

Mixins

Als *Mixin* citemixins1, oder *abstract subclass*, wird eine Klasse bezeichnet, deren Basisklassen zum Zeitpunkt der Implementierung nicht bekannt ist [BC90]. Die in Abschnitt 2.2 vorgestellten Typvorlagen bilden die Grundlage zur Implementierung von Mixins in

C++. Dabei wird die Vererbung der OOP in Verbindung mit Typvorlagen verwendet, um die Basisklasse der Vererbung später festzulegen.

```
1 template <class SUPER>
2 class mixin : public SUPER { .. };
```

Abbildung 2.16: Mixin in C++

Dies Parametrisierung der Basisklassen erfolgt im Falle von C++ als Parameter eines Templates. Im Beispiel der Abbildung 2.16 wird über den Parameter `SUPER` die Basisklasse angegeben. Die Festlegung der eigentlichen Klasse erfolgt erst zum Zeitpunkt der Instanziierung des Templates.

Mixins lassen sich wiederum auf andere Mixins anwenden, so dass sich durch Verkettung und Kombination mit einer Basisklasse eine beliebige lineare Vererbungshierarchie konfigurieren lässt. In Abbildung 2.17 ist die Verwendung eines Arrays als Basisklasse dargestellt. Zeilen 4 bis 5 zeigen die Definition eines Mixins zur Sortierung und Zeilen 7 bis 13 ein Mixin zur Synchronisation. Ab Zeile 10 ist die Verfeinerung der Methode `Add` dargestellt, die Elemente synchronisiert einem Array hinzufügt. Der Zugriff auf die Basisklasse erfolgt über den Template Parameter `SUPER`. Im Beispiel erfolgt so in Zeile 12 der Zugriff auf die Funktion `void Add(..);` der verfeinerten Klasse.

```
1  class Array { .. };
2  class List { .. };
3
4  template <class SUPER>
5  class Sort : public SUPER { .. };
6
7  template <class SUPER>
8  class Synchronize : public SUPER {
9    ..
10   void Add(const Element& elem) {
11     Lock();           //Synchronisation
12     SUPER::Add(elem); //Speichern des Elementes
13   }
14 };
15
16 class SyncSortedArray : public Synchronize< Sort <Array> > {};
17 class SortedList : public Sort <List> {};
```

Abbildung 2.17: Mixin Definition und Verwendung in C++

Die Eigenschaften `Sort` und `Synchronize` sind nun beliebig austauschbar und lassen sich auch auf eine andere Basisklasse wie z. B. eine Liste anwenden. In Zeilen 16 und

17 ist die Konfiguration in zwei Beispielen angegeben. Zeile 16 zeigt ein Array, welches synchronisiert und sortiert ist und Zeile 17 eine sortierte Liste.

Mit der Hilfe von Mixins kann daher auch ein Problem der OOP gelöst werden, das als *Library Scalability Problem* oder *Library Scaling Problem* [Big94] bekannt ist: Durch die Vererbungshierarchie der OOP, ist eine Skalierung bei der Implementierung von Klassenvarianten nicht gegeben. Soll eine Klasse mit mehreren unterschiedlichen Eigenschaften entwickelt werden, so muss für jede Kombination dieser Eigenschaften eine Implementierung erfolgen. Die Anzahl bereitzustellender Varianten nimmt exponentiell zu. Um beispielsweise eine Container-Klasse mit den Eigenschaften Synchronisation und Sortierung zu versehen, muss ausgehend vom einfachen Container zusätzlich eine Implementierung der Synchronisation mit Sortierung und ohne Sortierung sowie Sortierung ohne Synchronisation erfolgen. Es ergeben sich demnach $2^2 = 4$ Varianten.

Die Anzahl zu implementierender Varianten verdoppelt sich mit jedem hinzugefügten Merkmal. Sollen für eine Klasse n austauschbare Eigenschaften entwickelt werden, so müssen 2^n Klassen implementiert werden, um alle Kombinationen zu erzeugen[5]. Ist eine größere Anzahl unabhängiger Merkmale zu implementieren, ist eine effektive Entwicklung nicht möglich. Mit Mixins hingegen können die benötigten Varianten durch einfache Kombination der einzelnen Eigenschaften zum Zeitpunkt ihrer Verwendung erstellt werden. Es müssen daher nur die Merkmale selbst implementiert werden. Der Entwicklungsaufwand ist daher linear von der Anzahl der Merkmale abhängig, und die Wartung der entwickelten Komponenten vereinfacht sich entsprechend.

Mixins bieten die Möglichkeit Funktionalität zu kapseln, bleiben aber dennoch beliebig mit anderen Klassen kombinierbar. Im Falle von C++ erfolgt die Erzeugung der eigentlichen Klassen zum Zeitpunkt der Instanziierung der Templates und damit während der Übersetzung des Programms. Aus diesem Grund wird dies als statische Konfiguration bezeichnet.

Voraussetzung für die Verwendung solcher Mixins mit beliebigen Klassen ist das Vorhandensein einer bestimmten Schnittstelle der Basisklasse. Jedes Mixin setzt durch den Aufruf bestimmter Methoden der Basisklasse eine solche Schnittstelle voraus[6]: Es definiert implizit ein Interface, welches die Basisklasse implementieren muss. Der Aufruf einer nicht vorhandenen Funktion wird dabei durch den Compiler mit einer Fehlermeldung beanstandet. Auf diese Weise ist ebenfalls eine Überprüfung des Vorhandenseins der benötigten Schnittstelle möglich. Dies erfolgt zum Zeitpunkt der Instanziierung des Mixins und kann daher nicht für beliebige Konfigurationen im Voraus überprüft werden.

[5]Dies trifft lediglich für die Implementierung optionaler Merkmale zu, die in nur einer Variante vorliegen. Existieren zu jedem Merkmal mehrere mögliche Implementierungen m, so ergeben sich m^n Varianten [Bat98].

[6]vgl. Abschnitt 2.2

Die semantische Korrektheit einer Instanziierung eines Mixins ist nicht zwangsläufig gegeben. Sie ist nur im Zusammenhang mit der gesamten Konfiguration zu sehen. Deshalb bedarf es neben der Überprüfung der Schnittstelle entsprechender Dokumentation der Vorraussetzungen zur Kombination mit anderen Mixins und Klassen.

Ein weiteres Problem entsteht bei der Verwendung von nicht-Standard-Konstruktoren in Mixins: Ist bei der Instanziierung eines Mixins in der Basisklasse kein Standard-Konstruktor vorhanden, so besteht keine Möglichkeit zur Initialisierung der Basisklasse. Ähnliches trifft auf die Verwendung mehrerer Konstruktoren zu, da diese bei der Implementierung des Mixins nicht bekannt sind. Diese als *Konstruktorproblem* bekannte Schwierigkeit wurde in verschiedenen Ansätzen untersucht. Ein Lösung für C++ Mixins wird in [EBC00] gegeben. Die Umsetzung erfolgt mit Template-Metaprogrammierung und führt zu sehr komplexem Programmcode.

Eine weitere Schwierigkeit besteht bei der Instanziierung der Templates. Mit zunehmender Anzahl verwendeter Eigenschaften nimmt die Konfiguration sehr schnell unübersichtliche Formen an. In [EBC00] wird auch dieses Problem mit Template-Metaprogrammen gelöst. Wie bei der angesprochenen Lösung des Konstruktorproblems ist ebenfalls die praktische Anwendbarkeit auf Grund sehr komplexer Metaprogramme begrenzt.

Mixin Layers in C++

Der Ansatz der Mixins, die das Problem der Skalierbarkeit für die Kombination von Merkmalen für einzelne Klassen lösen, kann auf mehrere Klassen und damit auf ganze Komponenten angewandt werden. Dies bildet die Grundlage der Umsetzung des Kollaborationenentwurfs in Mixin Layers.

Eine Implementierung dieses Ansatzes sind *C++ Mixin Layers* [SB00], deren Grundlage Mixins bilden. Für die Programmiersprache Java steht mit den *Java Layers* ein ähnlicher Ansatz (siehe Abschnitt 3.3.2) zur Verfügung. Zur Einführung in die Thematik wird zunächst die Umsetzung mit Hilfe von C++ Templates betrachtet. Als Umsetzung der FOP für C++ sind sie im Rahmen dieser Arbeit von besonderem Interesse.

Für die Implementierung der Mixin Layers sind verschachtelte Klassen[7] notwendig, wie sie in Zeilen 1-5 der Abbildung 2.18 dargestellt sind. Die äußere Klasse bildet den Namensraum, der einer Kollaboration entspricht. Mixins mit inneren Klassen erben wie einfache Mixins von einer nicht bekannten Basisklasse (`Collaboration1` in Zeile 8 und 9). Die inneren Klassen der Mixin Layers (Klassen `Collaboration1::Role1` und `Collaboration1::Role2`) erben von den inneren Klassen der Basisklasse. In Abbildung 2.18 erfolgt dies durch die Verwendung des Parameters `SUPER` (Zeilen 11-12 und 19-20).

[7] vgl. Abschnitt 2.1.3

```
1  //Basislayer
2  class Base {
3    class Role1 {..};
4    class Role2 {..};
5  };
6
7  //Kollaboration mit 2 Rollen
8  template <class SUPER>
9  class Collaboration1 : public SUPER {
10 public:
11   class Role1 : public SUPER::Role1 {..};
12   class Role2 : public SUPER::Role2 {..};
13 };
14
15 //Kollaboration mit 2 Rollen
16 template <class SUPER>
17 class Collaboration2 : public SUPER {
18 public:
19   class Role1 : public SUPER::Role1 {..};
20   class Role2 : public SUPER::Role2 {..};
21 };
22
23 //Konfiguration
24 class Program : public Collaboration2 <
25                       Collaboration1 <
26                       Base> > {};
```

Abbildung 2.18: Mixin Layers in C++

Durch die Konfiguration wie bei herkömmlichen Mixins (Zeile 24-26) wird die Festlegung einer Vererbungshierarchie für eine ganze Reihe von Klassen auf einen späteren Zeitpunkt verschoben. So kann im verwendeten Beispiel durch weglassen der Klasse `Collaboration1` in Zeile 25 auch eine Klasse erstellt werden, die lediglich den Basislayer und `Collaboration2` enthält. Alle enthaltenen Klassen der Kollaboration würden dementsprechend auch entfallen. Auf diese Weise ist eine Implementierung des Kollaborationenentwurfs möglich. Die äußeren Klassen entsprechen demnach den Kollaborationen und die inneren Klassen den Rollen.

Das Festlegen der verwendeten Schichten erfolgt wie bei den Mixins zum Zeitpunkt der Instanziierung des Templates und ist damit wiederum eine statische Konfiguration.

2.3.6 Erweiterung der OOP

Neben der Möglichkeit, die generische Programmierung für die Umsetzung des Kollaborationenentwurfs zu verwenden, ist auch eine Erweiterung der objektorientierten Programmierung möglich. Unter Verwendung einer angepassten Syntax wird eine einfache Repräsentation der benötigten Strukturen ermöglicht.

Eine Kollaboration und damit ein Merkmal wird auch hier durch die gleichzeitige Verfeinerung mehrerer Klassen wiedergegeben. Die Verfeinerungen einer Kollaboration finden sich analog den Mixin Layers alle in einer Schicht wieder. Die Implementierung der Rollen erfolgt in Verfeinerungen. Sie Stellen einen Teil einer Klasse dar, sind aber ähnlich den Klassen der OOP aufgebaut. Im Folgenden wird dies anhand der Jakarta Tool Suite beschrieben.

Jakarta Tool Suite

Mit der *Jakarta Tool Suite (JTS)* und ihrem Nachfolger der *AHEAD Tool Suite (ATS)* ist eine Sammlung von Werkzeugen zur Unterstützung der FOP vorhanden. Grundlage bildet eine Spracherweiterung für Java. Verfeinerungen folgen bis auf einige Ausnahmen dem Aufbau der Klassen der OOP. Sie werden durch das Schlüsselwort `refines` gekennzeichnet. Die GenVoca Gleichung

$$\begin{aligned} A &= R3(R2(R1(C))) \\ &= R3 \bullet R2 \bullet R1 \bullet C \end{aligned} \tag{2.6}$$

lässt sich als Verfeinerungskette wie in Abbildung 2.19 darstellen. Die Konstante C entspricht dabei der Klasse `C` in Zeilen 1-3 der Abbildung. Die Zeilen 5-13 zeigen die Verfeinerungen $R1-R3$ der Gleichung 2.6. Durch die einheitliche Verwendung des Namens `C` ist eine eindeutige Zuordnung der Verfeinerungen zur verfeinerten Klasse möglich. Die Unterscheidung der einzelnen Verfeinerungen erfolgt über ihre Schicht: Alle zu einer Schicht gehörenden Klassen und Verfeinerungen werden in einem Ordner aufbewahrt, so dass die Verzeichnisstruktur den Kollaborationen des Entwurfs entspricht. Für den Zugriff auf die verfeinerte Klasse wird das Schlüsselwort `super` verwendet (siehe Abbildung 2.19 Zeile 8, und Abbildung 2.20 Zeile 14).

Die erweiterte Syntax der Programmiersprache Java kann auf verschiedene Weisen in herkömmlichen Java Code transformiert werden, um daraufhin vom Java Compiler in Java Bytecode umgewandelt zu werden[8]. Hierzu stehen zwei Möglichkeiten zur Auswahl: (1) Die Umsetzung in eine Klassenhierarchie, bei der jede Verfeinerung auf eine erbende Klasse abgebildet wird und (2) die Zusammenfassung aller Verfeinerungen in einer Klasse, die damit alle Daten und Methoden enthält (als *Jampack* bezeichnet, Abbildung 2.22).

Abbildung 2.21 zeigt den aus Abbildung 2.20 generierten Code bei der Umwandlung in eine Klassenhierarchie. Für jede Konstante einer Klasse und jede Verfeinerung wird eine eigene Klasse erstellt, deren Name den Namen der Schicht enthält aus dem die Verfeinerung stammt (Zeile 2). Der Name der Klasse wird durch die Zeichenfolge `"$$"` vom

[8]Näheres zu Codetransformationen findet sich in Abschnitt 2.6

```
1  class C {                //C
2    void foo() { .. }
3  };
4
5  refines class C {        //R1
6    void foo() {
7      ..
8      super.foo();
9    }
10 };
11
12 refines class C { .. }; //R2
13 refines class C { .. }; //R3
```

Abbildung 2.19: Verfeinerungen in JTS

```
1  //Layer Base
2  class Array {
3    //Element hinzufügen
4    void Add(Element e) {
5      m_Elements[m_Length++]=e;   //Element speichern
6    }
7  }
8
9  //Layer Sync
10 refines class Array {
11   //Verfeinerung der Methode Add
12   void Add(Element e) {
13     Lock();       //Code zur Synchronisation
14     Super.Add(e); //Aufruf der Methode des Basislayers
15   }
16 }
```

Abbildung 2.20: Verfeinerung eines Arrays

Namen der Schicht getrennt. Die letzte Verfeinerung erhält den Namen der eigentlichen Klasse (Zeile 9), was für die spätere Verwendung notwendig ist. Um zu verhindern, dass einer nicht vollständig verfeinerten Klasse erstellt wird (z. B. Klasse `Array$$Base` des Beispiels), wird zusätzlich das Schlüsselwort `abstract` verwendet (Zeile 2).

Im Vergleich dazu ist in Abbildung 2.22 die *Jampack* Ausgabe dargestellt. In diesem Fall werden die Konstante und alle Verfeinerungen dieser in eine Klasse transformiert. Hierdurch entsteht kein semantischer Unterschied, dennoch unterscheiden sich beide Varianten aus technischer Sicht. Beim Jampack sind die zu einem Merkmal gehörenden Membervariablen nicht auf den ersten Blick erkennbar, da die Member aller Verfeinerungen in einer Klasse dargestellt werden. Dies kann bei großen Klassen zu Unübersichtlichkeit während des Debuggens führen, da diese auch im Debugger in dieser Form dargestellt werden. Bei kleineren Klassen hingegen kann dies von Vorteil sein, da alle

```
1  //Code erzeugt aus Layer Base
2  abstract class Array$$Base {
3
4    void Add(Element e) {
5      m_Elements[m_Length++]=e;
6    }
7  }
8
9  //Code erzeugt aus Layer Sync
10 class Array extends Array$$Base {
11   void Add(Element e) {
12     Lock();
13     Super.Add(e);
14   }
15 }
```

Abbildung 2.21: Generierter Code – Vererbungshierarchie

```
1  //Klasse erzeugt aus beiden Layern
2  class Array {
3    //Code erzeugt aus Layer Base
4    void Add$$Base(Element e) {
5      m_Elements[m_Length++]=e;
6    }
7
8
9
10   //Code erzeugt aus Layer Sync
11   void Add(Element e) {
12     Lock();
13     Add$$Base(e);
14   }
15 }
```

Abbildung 2.22: Generierter Code – Jampack

Elemente sofort sichtbar sind und nicht in den Basisklassen gesucht werden müssen.

Die Verfeinerung der FOP ist nicht mit der Vererbung der OOP zu verwechseln. Verfeinerungen werden dazu verwendet, Klassen entsprechend den Merkmalen zu zerlegen (Separation of Concerns). Abbildung 2.23 verdeutlicht dies: Während in der OOP Merkmale in Klassen enthalten sind (dargestellt als hell- und dunkelgraue Streifen innerhalb der Klassen), werden diese in der FOP in Verfeinerungen zusammengefasst. Die Klasse `Logfile` ist gesondert zu betrachten. Sie enthält Funktionalität, die nur im Layer `Logging` verwendet wird. Somit ist sie in der FOP-Implementierung auch nur in diesem wieder zu finden.

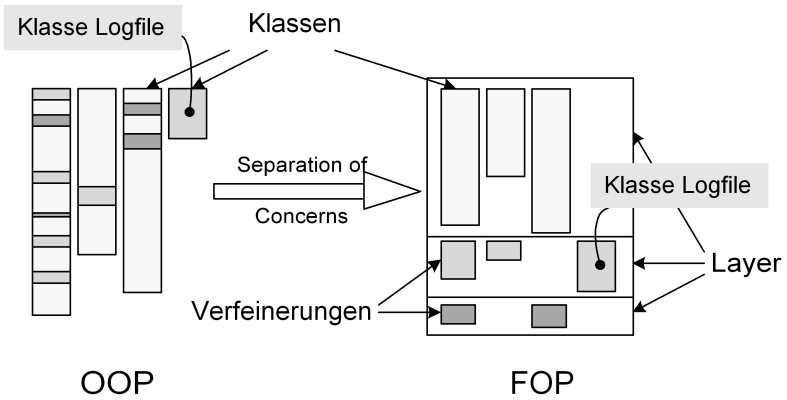

Abbildung 2.23: Separation of Concerns in FOP

Die Verwendung von Konzepten der OOP wie der Vererbung ist innerhalb der FOP

ebenfalls möglich. Dabei ist zu beachten, dass die erbende Klasse immer von der am weitesten verfeinerten Variante einer Klasse erbt. In Abbildung 2.24 ist ein solcher Fall dargestellt. Die Klasse `ExtArray` erbt von der Klasse `Array` (Zeile 3) und enthält damit auch sämtliche Verfeinerungen dieser Klasse. Im Layer `Sync` ist eine Verfeinerung der Klasse `Array` implementiert, die eine Synchronisation des Arrays ermöglicht. Bei der Verwendung dieses Layers enthält die Klasse `ExtArray` bereits alle Elemente der Verfeinerung der Klasse Array, in diesem Fall z. B. die Membervariable `m_Sync` (Zeile 7). Bei Verfeinerung der Klasse `ExtArray` im Layer `Sync` muss daher lediglich Funktionalität implementiert werden, die nicht in der Verfeinerung der Klasse `Array` enthalten ist. Abbildung 2.25 verdeutlicht dies anhand des resultierenden transformierten Quelltextes: Klasse `ExtArray$$Base` erbt von der finalen Klasse `Array` (Zeile 3).

```
1  //Layer Base
2  class Array {..}
3  class ExtArray extends Array {..}
4
5  //Layer Sync
6  refines class Array {
7    Mutex m_Sync;
8    ..
9  }
10 refines class ExtArray {..}
```

Abbildung 2.24: Vererbung in FOP

```
1  //Code erzeugt aus Layer Base
2  abstract class Array$$Base {..}
3  abstract class ExtArray$$Base extends Array {..}
4
5  //Code erzeugt aus Layer Sync
6  class Array extends Array$$Base {
7    Mutex m_Sync;
8    ..
9  }
10 class ExtArray extends ExtArray$$Base {..}
```

Abbildung 2.25: Vererbung in FOP – Generierter OOP Code

Ein weiterer wesentlicher Unterschied zu OOP ist die Verwendung der Konstruktoren. Während bei der OOP mit jeder Vererbung alle vorhandenen Konstruktoren erneut definiert werden müssen, um sie weiter verwenden zu können, werden bei einer Verfeinerung alle bereits definierten Konstruktoren *propagiert*. Sie werden im Falle der Erzeugung einer Klassenhierarchie für alle erbenden Klassen automatisch generiert.

Zur Konfiguration einer Software werden die in GenVoca / AHEAD beschriebenen Gleichungen in *Equation Files* festgehalten. Sie enthalten eine Auflistung der verwendeten Merkmale und damit die Schichten in der zu verwendenden Reihenfolge. Mehrere solcher Dateien lassen sich verwenden, um Merkmale in Teilmerkmale zu zerlegen und erlauben so eine skalierende Konfiguration bis hin zu komplexen Anwendungen.

Neben der Definition der Spracherweiterung für Java, steht in der ATS auch ein Werkzeug zum Überprüfen von Entwurfsregeln zur Verfügung (DRC-Tool). Die Entwurfsregeln verwenden eine spezielle Syntax zur Beschreibung der Vor- und Nachbedingungen einer Schicht und werden in Dateien in jeder Schicht hinterlegt. Nach Festlegung einer Konfiguration kann diese anhand der Regeln überprüft werden.

2.4 Aspektorientiertes Programmieren

Neben merkmalsorientierter Programmierung versucht auch die *aspektorientierte Programmierung* (*Aspect-Oriented Programming – AOP*) Merkmale zu modularisieren, die andernfalls über den Quelltext verteilt sind. Es wird versucht den bestimmte Merkmale betreffenden Programmcode von der übrigen Implementierung zu trennen (Separation of Concerns).

In diesem Abschnitt wird die Anwendung der AOP auf objektorientierter Programmierung dargestellt[9]. Crosscuts, die eine oder mehrere Klassen betreffen, werden aus diesen entfernt und in *Aspekten* zusammengefasst. Der Aufbau der Aspekte ähnelt dem Aufbau der Klassen der OOP.

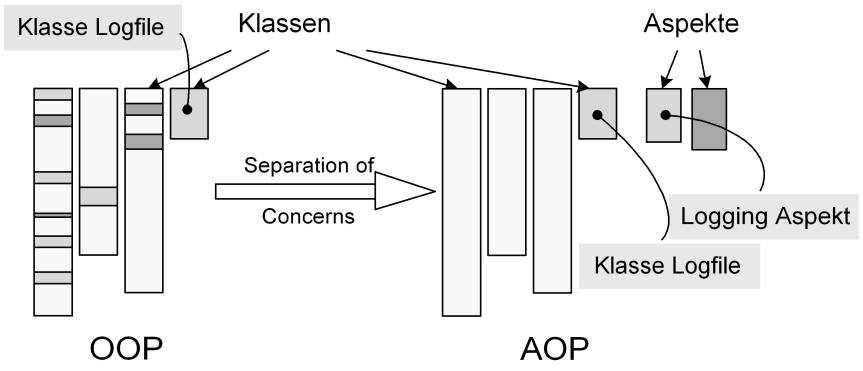

Abbildung 2.26: Separation of Concerns in AOP

Die Zerlegung der Klassen in der AOP ist in Abbildung 2.26 dargestellt (vgl. Abbildung 2.23 für FOP). Verfeinerungen einer Klasse werden in der FOP in einem Element je Merkmal zusammengefasst. Hingegen wird in der AOP eine Reihe ähnlicher, meist einem Merkmal zugehöriger, Verfeinerungen in einem Aspekt zusammengefasst. Ein Merkmal besteht damit aus einer Menge von Aspekten und / oder Klassen. In Abbildung 2.26 ist z. B. für das Merkmal Logging die Klasse `Logfile` und der Aspekt `Logging` notwendig. Der Aspekt enthält dabei alle Informationen, die für die Verfeinerungen der drei dargestellten Klassen notwendig sind. Ebenso sind Merkmale möglich, die aus mehreren Klassen und Aspekten bestehen.

Wie auch in FOP erfolgt in AOP eine Verfeinerung vorhandener Klassen. Die Stellen im Programmcode, an denen diese Verfeinerung erfolgt, werden als *Joinpoints* bezeichnet. Sie werden mit Hilfe von *Pointcuts (PC)* innerhalb der Aspekte definiert. Ihre Beschreibung wird durch die Verwendung von Suchausdrücken vereinfacht, welche auf eine oder

[9]Eine Verwendung in anderen Umgebungen, z. B. funktionalen Programmiersprachen, ist möglich, jedoch an dieser Stelle nicht von Interesse.

mehrere Klassen bzw. Methoden verweisen. Die verwendeten Suchausdrücke enthalten üblicherweise *Wildcards*, um gleichzeitig als Vorlage mehrerer Klassen- oder Methodennamen zu dienen. Auch Platzhalter für Parameter von Funktionen oder Rückgabewerte sind möglich.

In Kombination mit den Pointcuts wird in einem Aspekt *Advice-Code* definiert. Er stellt den Programmcode der Verfeinerung dar und wird an den Joinpoints ausgeführt. Im Gegensatz zur FOP wird dieser nicht innerhalb der Klassen definiert, sondern außerhalb, in den Aspekten. Neben dem Verfeinern von Methoden können auch Klassen erweitert werden: Es können sowohl neue Methoden definiert als auch neue Membervariablen eingeführt werden. Crosscuts die solche statischen Verfeinerungen benötigen werden als *statische Crosscuts* bezeichnet. Erfolgt zur Laufzeit auf Grund des aktuellen Zustandes eine Entscheidung ob und wie eine Verfeinerung erfolgt, spricht man von *dynamische Crosscuts*.

Außer dem Verfeinern der Klassen ist das Definieren von Membervariablen und Methoden in Aspekten möglich. Da während der Laufzeit Instanzen des Aspektes vorhanden sind[10], können im Aspekt Informationen gespeichert werden, auf die aus dem Aspekt-Code zugegriffen werden kann. So ist die Definition von Membervariablen möglich (wie z. B. Logfiles bei der Implementierung eines Logging Aspektes), auf die ähnlich wie auf globale Variablen zugegriffen werden kann.

Aspektorientierter Quelltext kann entweder mit objektorientierten Programmcode oder mit bereits kompiliertem Programmcode zusammengeführt werden. Dieser Vorgang wird als *Weben* bezeichnet. Der *Aspektweber* (engl. *aspect weaver*) ermittelt dazu alle Joinpoints, die über die Pointcuts definiert wurden und fügt an diesen Stellen den Advice-Code ein. Geschieht dies vor der Laufzeit des Programms, spricht man von statischem Weben. Beim dynamischen Weben wird der Aspekt-Code während der Laufzeit mit dem übrigen Programmcode verbunden. Abbildung 2.27 zeigt statisches Weben auf Basis objektorientierten Codes, wie es z. B. in AspectC++[11] stattfindet.

Abbildung 2.27: Statisches Weben

Durch die Definition von Pointcuts mit Hilfe von Suchausdrücken kann mit einem ein-

[10]In AspectC++ ist immer genau eine Instanz vorhanden. Die Implementierung entspricht dabei einem Singleton. In AspectJ hingegen können mehrere Instanzen eines Aspektes existieren.

[11]Näheres zu AspectC++ findet sich im folgenden Abschnitt.

zigen Pointcut der Zugriff auf eine Reihe von Funktionen gesteuert werden, die auch über mehrere Klassen verteilt sein können. Auf diese Weise kann gleicher Code an verschiedenen Stellen eingefügt werden und Redundanz bei der Implementierung kann verhindert werden. Daher ist die AOP insbesondere für Belange von Interesse, die sich über den gesamten Programmcode erstrecken. Dies sind beispielsweise Analyseaufgaben wie Logging, Optimierungen[12] oder Synchronisation. In diesen Fällen ist der gesamte Quelltext oder große Teile davon betroffen. Die grundlegende Funktionalität wird durch diese Eingriffe in der Regel nicht beeinflusst. Eine externe Verwaltung der Funktionalität in den Aspekten ist in diesen Fällen für das Verständnis des Quelltextes nicht hinderlich.

Liegt ein Fehler auf Grund der Zusammenarbeit zwischen Advice-Code und ursprünglichem Programmcode vor, gestaltet sich die Fehlersuche und -behebung schwierig. Dabei besteht das Problem, dass der Quelltext nicht in Zusammenhang mit dem Aspekt-Code betrachtet werden kann. Die durch den Aspekt-Code entstehenden Effekte und Fehlerquellen können so nur schwer beurteilt werden. Hier ist das globale Zusammenspiel der Aspekte und des übrigen Programmcodes von Interesse[13]. Der Quellcode muss dazu im Zusammenhang mit dem Aspekt-Code untersucht werden. Erweiterungen existierender Entwicklungsumgebungen können dabei Abhilfe schaffen. Derzeit ist dies jedoch nicht möglich.

Neben der beschriebenen Verwendung der Aspekte, kann wie in OOP eine Vererbungshierarchie von Aspekten erstellt werden. Die Syntax entspricht dabei der Vererbung der OOP. Auf diese Weise können Abstraktionsebenen eingeführt und die Wiederverwendbarkeit von Aspekten verbessert werden. Die Verwendung des Schlüsselwortes `virtual` entspricht dabei nicht vollständig ihrer Verwendung in der OOP. Hier ist vielmehr ein statisches Überschreiben des als virtuell deklarierten Pointcuts gemeint.

2.4.1 AOP in C++ und Java

Für die aspektorientierte Programmierung stehen derzeit einige Erweiterungen von OO Programmiersprachen zur Verfügung. Im Folgenden sollen AspectJ[14] [KHH+01] und AspectC++[15] [SGSP02] betrachtet werden, da sie für Java und C++ die wichtigsten Ansätze darstellen. Dabei wird insbesondere auf die Implementierung für C++ Wert gelegt, da sie Grundlage für die weitere Arbeit ist.

[12] Hier ist beispielsweise die Verwendung eines Caches für Berechnungen möglich, so dass Ergebnisse sich wiederholender Berechnungen gespeichert werden.

[13] Insbesondere bei der Synchronisation besteht das Problem, dass der Aspekt-Code oft im Kontext mit dem übrigen Quelltext betrachtet werden muss, da Fehler (etwa Deadlocks) große Programmteile betreffen.

[14] http://aspectj.org

[15] http://aspectc.org

Beide Ansätze haben eine ähnliche Syntax: Aspekte werden mit dem Schlüsselwort `aspect` definiert und Pointcuts mit dem Schlüsselwort `pointcut`. Die in Pointcuts angegebenen Ausdrücke unterscheiden sich in ihrer Syntax. Ihnen ist gemein, dass Platzhalter innerhalb der Suchausdrücke verwendet werden können. In AspectC++ wird das Zeichen `%` als Platzhalter für Namen oder Namensteile von Klassen und Typen verwendet. Um einen Suchausdruck für eine Methode mit einer beliebigen Anzahl von Parametern zu definieren findet die Ellipse (`...`) Anwendung. Sie wird ebenfalls für beliebige Namensräume verwendet. Der AspectC++ Suchausdruck `"void %::Print(...)"` trifft daher auf alle Methoden `Print` beliebiger Klassen im globalen Namensraum zu. Die Signatur der betreffenden Funktionen muss dem Suchausdruck entsprechen. So muss der Rückgabewert `void` sein, die Parameterliste ist beliebig.

Der in beiden Sprachen vordefinierte Pointcut `call` wird verwendet, um Joinpoints bei Funktionsaufrufen zu setzen. Neben diesem existieren weitere vordefinierte Pointcuts[16], die grundlegende Funktionalität zur Unterbrechung bereitstellen. Durch eine Kombination von Pointcuts, z. B. durch Verknüpfung mit logischen Operatoren, können beliebige Stellen im Quelltext angesteuert werden. Der Pointcut `cflow` findet Verwendung zur Unterbrechung des Kontrollflusses abhängig vom Kontext während des Aufrufs. So kann je nach bereits aufgerufenen Methoden entschieden werden, ob Advice-Code ausgeführt werden soll oder nicht. Dies erlaubt es Aspekte nur für Aufrufe bestimmten Ursprungs zu „aktivieren“.

Pointcut	Betroffene Codestellen
call	Funktionsaufrufe.
construction	Erzeugung einer Instanz einer Klasse.
destruction	Zerstören einer Instanz einer Klasse.
cflow	Dynamischer Kontext während der Ausführung.

Tabelle 2.1: Beispiele für Pointcuts in AspectC++

In beiden Sprachen existieren vordefinierte Ausdrücke, die Beschreiben wie mit dem Advice-Code zu verfahren ist (siehe Tabelle 2.2). So ist es beispielsweise durch die Definition eines Advice mit `before` möglich, Advice-Code vor dem Aufruf bestimmter Funktionen auszuführen. Zu beachten ist, dass in AspectC++ Advice-Code gesondert mit `advice` angegeben wird. Dies ist in AspectJ nicht der Fall.

In Abbildung 2.28 ist ein einfacher Aspekt zur Synchronisation einer Liste in AspectC++ dargestellt. Er führt für alle Operationen auf einer Liste Synchronisation

[16] Beispiele für AspectC++ finden sich in Abbildung 2.1. Die Pointcuts beider Sprachen unterscheiden sich zum Teil. Für eine genauere Betrachtung sei auf die jeweilige Sprachreferenz verwiesen [pur04].

Advice	Ausführung des Advice-Codes
before	Ausführung vor dem Pointcut.
after	Ausführung nach dem Pointcut.
around	Ausführung an Stelle des Pointcuts. Ein Aufruf der ursprünglichen Funktion muss explizit angegeben werden.

Tabelle 2.2: Advice Deklarationen in AspectC++

ein: Es wird ein Synchronisationsobjekt `_sync` zur Klasse `List` hinzugefügt (Zeile 5) und entsprechend des Pointcut `sync()` (Zeile 8) wird der Aufruf aller Funktionen der Klasse unterbrochen. Vor dem weiteren Ausführen des Programmcodes wird der Code des Advice ausgeführt (Zeilen 11-15). Dieser sperrt die Liste mit Hilfe der Klasse `LockObject` (Zeile 14). Ebenso ist eine Erweiterung des Aspekts auf eine ganze Reihe von Klassen möglich. Unter Verwendung von Namensräumen können die Pointcuts `list()` und `sync()` so verändert werden, dass alle Klassen eines Namensraums `Container` betroffen sind. Dazu muss lediglich der Suchausdruck für die Klassen angepasst werden. Für den Pointcut `sync()` ergibt sich damit z. B. `"% Container::%::%(...)"`, der für die Synchronisation einer Bibliothek von Containerobjekten verwendet werden kann.

```
1  aspect Sync {
2  public:
3    // Synchronisationsobjekt in Liste einfügen
4    pointcut list() = "List";
5    advice list() : Semaphore _sync;
6
7    // Pointcut für Funktionen der Klasse "List"
8    pointcut sync() = call("% List::%(...)");
9
10   // Unterbrechen vor der Ausführung
11   advice sync() : before() {
12     // Sperren der Liste
13     List* l = tjp->that();
14     LockObject lock(l->_sync);
15   }
16 };
```

Abbildung 2.28: Synchronisationsaspekt in AspectC++

2.4.2 Vergleich zu FOP

Nachdem grundlegende Eigenschaften der AOP und FOP beschrieben wurden, werden in diesem Abschnitt Gemeinsamkeiten und Unterschiede beider Konzepte aufgezeigt. Gemeinsames Merkmal beider Ansätze ist das angestrebte Ziel: *Separation of Concerns*, dabei sind klare Unterschiede zwischen beiden Umsetzungen zu erkennen.

Die auf Kollaborationen basierenden Ansätze zerlegen die Implementierungselemente (die Klassen der OOP) entlang der Merkmalen einer Software wie sie für die Beschreibung notwendig sind. Die Orthogonalität zwischen diesen findet sich mit den Klassen und Kollaborationen auch im Entwurf wieder.

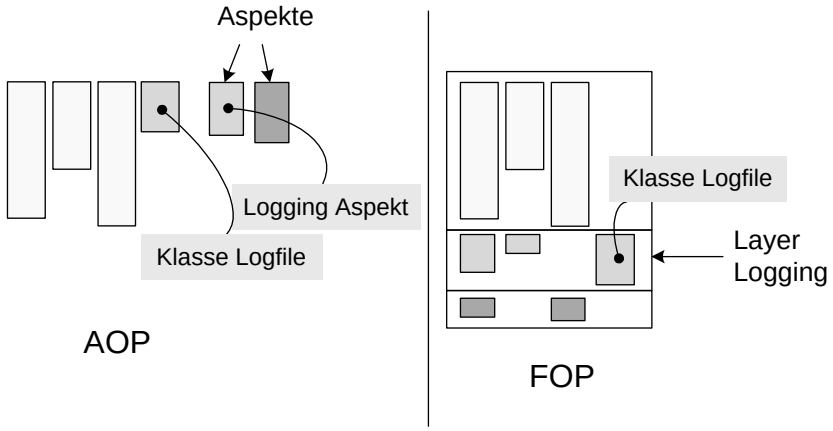

Abbildung 2.29: Separation of Concerns – Vergleich AOP und FOP

In der AOP bilden die Aspekte in Verbindung mit weiteren Klassen die Einheiten für die Implementierung der Merkmale. Im Gegensatz zur FOP ist der Zusammenhang zwischen den Merkmalen und Implementierungselementen nicht offensichtlich und auch nicht ohne weiteres zugänglich. Ursache ist die Auslagerung der Verfeinerungen in Aspekte. Abbildung 2.29 zeigt, dass dies im Falle der FOP leicht möglich ist. Ebenfalls ist ein Zusammenhalt des Merkmals `Logging` bestehend aus der Klasse `Logfile` und dem Aspekt `Logging` im Falle der AOP nicht erkennbar.

Während eine Rolle der FOP genau den Teil einer Klasse beschreibt, der für ein Merkmal zuständig ist, beschreibt ein Advice einen Teil eines Merkmals, der für ein oder mehrere Klassen von Interesse ist. Grund für die Bildung eines Advice ist ein technischer: Es lässt sich gleicher Programmcode für verschiedene Methoden zusammenfassen. Ist ein solcher crosscutting Concern vorhanden, spricht man auch von einem *homogenen Crosscuts* [CC04]. Werden homogenen Belange mittels FOP implement, zeigt sich, dass sich wiederholender Code für die betroffenen Klassen implementiert werden muss. Damit kann Redundanz nicht verhindert werden. Die FOP ist besser für Belange geeignet, bei

denen sich der zu implementierende Programmcode unterscheidet, so genannte *heterogene Crosscuts*. Die AOP ist für diese Belange auf Grund genannter Probleme weniger geeignet.

2.5 Multi-Dimensional Separation of Concerns

Merkmale einer Software beeinflussen sich häufig gegenseitig. Im Kollaborationenentwurf erfolgt eine Trennung der Merkmale. Diese ist aber nicht ausreichend, um eine möglichst einfache Beschreibung von Software zu gewährleisten. Die Trennung von Merkmalen entlang mehrerer Dimensionen (*Multi-Dimensional Separation of Concerns – MDSOC*) wurde in von Tarr et al. vorgestellt [TOHS99]. Dabei werden Arten von Merkmalen als Dimensionen eines n-dimensionalen Raums aufgefasst. Entlang einer Dimension finden sich nur unabhängige Merkmale, die sich nicht gegenseitig beeinflussen. Die Anzahl vorhandener Dimensionen `n` ergibt sich aus den verschiedenen Arten von Merkmalen. Merkmale verschiedener Dimensionen können sich überlappen und beeinflussen sich gegenseitig. Im Entwurf erfolgt eine Zerlegung der Merkmale entlang anderer Merkmale die sie beeinflussen.

In Abbildung 2.30 ist für einen String eine Trennung der Dimension *Betriebssystem* (*OS*) von der Dimension *Character* dargestellt. Entlang der Dimension *OS* sind die Betriebssysteme aufgetragen, für die eine Implementierung der Klasse vorliegt (Unix und Windows). Die Dimension *Character* enthält die Implementierungen für einfache 8 bit Zeichen (Char) und einen erweiterten 16 bit Zeichensatz (Wide Char – WChar). Die Implementierungen für die jeweiligen Betriebssysteme unterscheiden sich. So ist z. B. unter Windows eine andere Implementierung des einfachen Zeichensatzes notwendig als unter Unix. Innerhalb der Betriebssysteme ist außerdem eine unterschiedliche Implementierung für Char und WChar notwendig. Die Merkmale auf den beiden Dimensionen schließen sich gegenseitig aus, da entweder Char oder WChar, bzw. Windows oder Unix verwendet werden kann. Für jede Dimension kann nur eines der Merkmale ausgewählt werden. Jeder Punkt dieses 2-Dimensionalen Raums bildet in diesem Fall eine gültige Kombination der Merkmale ab (z. B. „Win / Char“ für die Windows-Implementierung mit dem einfachen Zeichensatz). Weitere Merkmale erweitern dieses Modell um zusätzliche Dimensionen.

Neben der Zerlegung von Merkmalen ist auch eine Zerlegung von Implementierungselementen (Klassen) oder anderen Softwareartefakten möglich. Ein Ausschnitt des Mehrdimensionalen *Hyperraums* bezüglich einer bestimmten Dimension wird als *Hyper Slice* bezeichnet. So stellt auch Abbildung 2.30 einen Hyperslice dar. In diesem Fall ist die betrachtete Dimension die der Implementierungselemente.

Der MDSOC Ansatz versucht die Abhängigkeiten der Merkmale untereinander zu

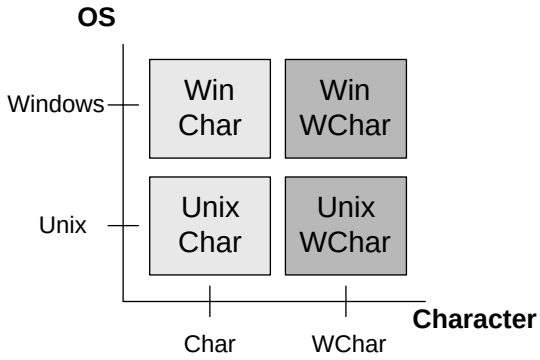

Abbildung 2.30: MDSOC – String mit 2 Merkmalsdimensionen

fassen. Er ist dadurch kompakter als der Kollaborationenentwurf und es lassen sich von vornherein ungültige Merkmalskombinationen ausschließen. Batory et al. zeigen, dass dieser Ansatz in ein Kollaborationenentwurf überführbar ist [BLS03].

2.6 Codetransformation

In den letzten Abschnitten wurden Spracherweiterungen vorgestellt, die bei der Entwicklung von Produktlinien hilfreich sind. Häufig ist für die Umsetzung solcher Spracherweiterungen die Entwicklung eines Compilers zu aufwendig. Aus diesem Grund wird der Quelltext häufig in eine andere Form umgewandelt, die von einem Compiler verarbeitet werden kann. Dies wird als *Codetransformationen* oder genauer als *Quellcode-Quellcode Transformation* bezeichnet.

Unter *Codetransformation* wird allgemein die Umwandlung von Programmcode in eine andere Form verstanden. Ein Beispiel hierfür ist die Transformation, die ein Compiler durchführt: Er wandelt Quelltext in Maschinencode um, der von einem Rechner ausgeführt werden kann. Diese Transformation wird auch als *Compiler Transformation* bezeichnet. Erfolgt die Umwandlung nicht in ausführbaren Code, sondern wiederum in Quelltext, wird der Begriff *Quellcode-Quellcode Transformation* verwendet. Häufig ist bei der Verwendung des Ausdrucks der Codetransformation eine solche Quellcode-Quellcode Transformation gemeint. Im Folgenden wird mit Codetransformation der allgemeine Vorgang der Umwandlung von Programmcode bezeichnet werden. Ist eine Unterscheidung zur Quellcode-Quellcode Transformation notwendig, so wird dies explizit angegeben.

Beispiele für die Verwendung einer Codetransformation sind die AHEAD Tool Suite und AspectC++, die beide Quelltext in objektorientierten Programmcode umwandeln. Auf diese Weise muss lediglich eine überschaubare Änderung am vorhandenen Code vorgenommen werden und die eigentliche Umwandlung in Maschinencode kann von Com-

pilern durchgeführt werden. Compiler sind für verschiedene Plattformen erhältlich, was die Verwendung über verschiedene Betriebssysteme hinweg erleichtert. Sie besitzen weiterhin eine hohe Leistungsfähigkeit in Bezug auf Optimierung während der Umsetzung. Ein weiterer Vorteil einer Quellcode-Quellcode Transformation besteht darin, dass der Quelltext in einer für den Entwickler lesbaren Form ausgegeben wird. Dies ermöglicht eine einfachere Fehlerbehebung bei der Implementierung einer Spracherweiterung.

Werden Quellcode-Quellcode-Transformationen verwendet, besteht das Problem, dass während des Debuggens einer Software eine Stelle des transformierten Programmcodes in eine Position des ursprünglichen Quelltext überführt werden muss, um an dieser einen Fehler zu beheben. Hier existieren für einige Programmiersprachen (z. B. C++) Möglichkeiten, im generierten Code dessen Ursprung durch Compilerdirektiven festzuhalten. Auf diese Weise kann die Fehlersuche und deren Behebung direkt am ursprünglichen Code vorgenommen werden. Der transformierte Programmcode hat damit für den Entwickler geringe Bedeutung. Bietet eine Programmiersprache diese Funktionalität nicht (z. B. Java), so bereitet die Entwicklung und Wartung von Software mit einem Codetransformationssystem entsprechend Probleme.

Die Aufgabe, die ein Codetransformator ausführt, lässt sich im Wesentlichen in drei Schritte unterteilen:

1. Quelltextanalyse,

2. Umwandlung des in einer Zwischenform vorliegenden Quelltextes,

3. Generierung von Code in der Ausgabesprache.

Nach der Analyse des Quelltextes liegt der Code in einer Zwischenform, dem *Abstrakten Syntaxbaum*, vor. Dieser enthält Informationen zur Syntax und zum Teil auch zur Semantik des zugrunde liegenden Quelltextes. Die Analyse des Quelltextes (*frontend*) und Synthese von Code in der Zielsprache (*backend*) sind dabei unabhängig voneinander. Bei der Quellcode-Quellcode Transformation erfolgt eine Umwandlung der Zwischenform, um die Syntax an die Ausgabesprache anzupassen. Bei einem Compiler ist eine solche Anpassung üblicherweise nicht notwendig, da während der Ausgabe direkt Maschinencode generiert werden kann. Hier erfolgt aber häufig eine Transformation, um Optimierungen des Quelltextes vorzunehmen.

Im Folgenden wird die Quellcode-Quellcode Transformation näher betrachtet, da sie für die Umsetzung der angestrebten Spracherweiterung dieser Arbeit verwendet wird.

2.6.1 Lexer

Die Analyse des Programmcodes unterteilt sich in zwei Schritte, der lexikalischen Analyse durch den *Lexer*[17] oder *Scanner*[18] und der darauf folgenden Analyse der Syntax durch den *Parser*.

Bei der lexikalischen Analyse wird der Quellcode, der als Folge von Zeichen vorliegt, in *Token* zerlegt. Dies sind Zeichenfolgen unterschiedlicher Länge, die eine bestimmte Bedeutung in der analysierten Sprache haben. Das können z. B. Zahlen, Operatoren oder Schlüsselwörter sein. In der Programmiersprache C++ ist z. B. das Schlüsselwort `class` ein solches Token, aber auch Zeichenfolgen wie `"Hallo Welt!"` sind lediglich einzelne Token. Die Länge ist hier nicht ausschlaggebend für die Entscheidung, ob eine Folge von Zeichen als zusammengehörig und damit als Token betrachtet wird, sondern ihre Verwendung in der untersuchten Sprache.

Die Ausgabe des Lexers ist eine Folge von Token (*Token Stream*). Die Token enthalten neben der Bedeutung oft auch Informationen zu Ihrem Ursprung. Der nächste Abschnitt beschreibt die darauf aufbauende Syntaxanalyse.

2.6.2 Parser

Die vom Lexer generierte Folge von Token wird im nächsten Schritt, der Quelltextanalyse, vom *Parser* weiterverarbeitet. Die Ausgabe des Parsers ist ein Abstrakter Syntaxbaum (*Abstract Syntax Tree – AST*), der syntaktische Informationen des Quelltextes enthält. Möglich ist diese Umwandlung mit Hilfe einer Grammatik, deren Regeln die Syntax des Quelltextes beschreiben.

Wie in Abbildung 2.31 dargestellt[19], repräsentiert ein AST die Token des analysierten Quelltextes, enthält aber zusätzlich Informationen über die Syntax. So ist ein binärer Operator ein Knoten, der an seinen Zweigen die zugehörigen Operanden hält, die wiederum zusammengesetzte Ausdrücke sein können. In Abbildung 2.31 ist dies der Knoten „BinaryExpr“ mit den beiden Knoten „SimpleName“. Wie in Abbildung für eine Quelltextzeile dargestellt, wird der gesamte Quelltext in einem solchen Baum abgebildet.

In einem weiteren Schritt wird der Kontext in den Ausdrücken überprüft. So wird z. B. sichergestellt, dass die Operanden einer Additionsoperation numerischen Typs sind. Wird eine Kontextüberprüfung vorgenommen, wird der zugehörige AST auch als *annotated AST* bezeichnet.

[17] aus dem Engl.: *lexical analyser*

[18] Die Bedeutung der Ausdrücke Lexer und Scanner ist in diesem Zusammenhang identisch.

[19] Die Bezeichnungen der Knoten des AST entsprechen der in PUMA (Abschnitt 2.6.4) verwendeten Notation.

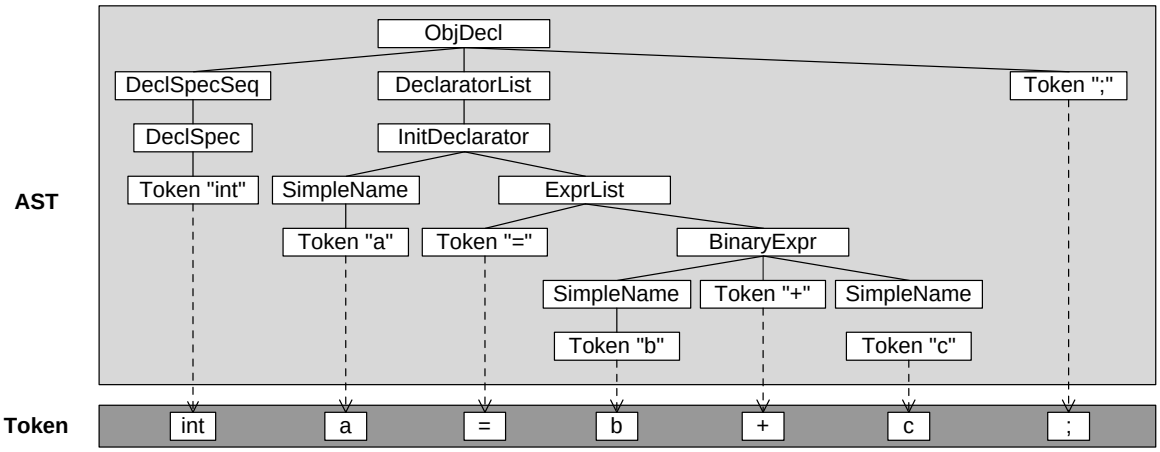

Abbildung 2.31: AST einer Addition

2.6.3 Transformationen des AST

Die interne Repräsentation des Quelltextes als AST kann verwendet werden, um im Falle eines Compilers Maschinencode oder im Falle einer Quellcode-Quellcode Transformation Quelltext zu erzeugen.

Vor der Ausgabe ist oft eine Umwandlung des AST notwendig. So auch für die Entwicklung von Spracherweiterungen, wenn Quelltext in einer anderen Programmiersprache ausgegeben wird. Für die ATS ist die Transformation einer Verfeinerung in eine Klasse (vgl. Abbildung 2.21) oder das Verschieben von Methoden zwischen Verfeinerungen und Klassen notwendig (vgl. Abbildung 2.22). AspectC++ verwendet Codetransformationen zum Einfügen des Advice-Code in den objektorientierten Code. Die Manipulation des AST findet vor der Übersetzung in die Maschinensprache statt und gehört damit zur statischen Metaprogrammierung.

Nach Abschluss der Transformation wird bei einer Quellcode-Quellcode Transformation der AST wieder als Quelltext ausgegeben. Dieser Schritt wird auch als Codegenerierung bezeichnet. Zum Teil ist es möglich während der Umwandlung des AST bereits eine dem entsprechende Modifikation des zugehörigen Token Streams vorzunehmen. In diesem Fall erfolgt die abschließende Ausgabe des Quelltextes lediglich durch Ausgabe des Token Streams.

2.6.4 PUMA

Für die im Rahmen dieser Arbeit entwickelte Spracherweiterung empfiehlt sich die Anwendung einer Quellcode-Quellcode-Transformation. Die Umsetzung der Transformation

erfolgt mittels der *PUMA* Bibliothek (*PUre MAnipulator*) [pur01, Spi03], auf die in diesem Abschnitt näher eingegangen wird.

PUMA ist eine Bibliothek zur C++ Quellcode Analyse, die zusätzlich Möglichkeiten zur Transformation des erstellten AST bietet. Im Rahmen der Entwicklung von AspectC++ wurde PUMA für das Parsen aspektorientierten Quelltextes erweitert[20]. Außerdem ist sowohl eine Verwendung unter Unix als auch unter Windows möglich. Auf Grund der Verwendung aspektorientierter Elemente in dieser Arbeit und der angestrebten Umsetzung sowohl für Unix als auch für Windows, sprechen diese Merkmale für die Verwendung von PUMA (siehe auch Abschnitt 4.2).

Wie auch in anderen Codetransformationssystemen, wird von PUMA aus den zu analysierenden Quellen ein AST erstellt. Grundlage bilden die beiden Softwaregeneratoren, *Orange* und *Lemon*[21]. Orange ist ein Generator für Lexer (eng. *lexical analyser generator*) und generiert den Teil der PUMA Bibliothek, der für die lexikalische Analyse zuständig ist. Der Parsergenerator Lemon erstellt die für die syntaktische Analyse wichtigen Teile der Bibliothek. Im Laufe der Entwicklung von PUMA wurden große Teile des Parsers durch handgeschriebenen Quelltext ersetzt [Spi03], so dass nur noch ein kleiner Teil von Lemon generiert wird. Eine Erweiterung der Syntax ist daher nur durch manuelle Erweiterung des Parsers möglich. Die Erweiterung des Scanners und Präprozessor Parsers ist aber durch Erweiterung der verwendeten Grammatik möglich.

Der vom Parser erzeugte Syntaxbaum kann mit Hilfe von PUMA manipuliert werden: Es können Teile kopiert, verschoben, gelöscht oder eingefügt werden. Auf diese Weise sind beliebige Transformationen möglich. Abschließend kann der modifizierten Syntaxbaums als Quelltext gespeichert werden.

[20] PUMA steht als Open Source zur Verfügung. Siehe http://aspectc.org
[21] http://www.hwaci.com/sw/lemon/lemon.html

Kapitel 3

Analyse existierender Ansätze

Im vorangegangenen Kapitel wurden Grundlagen zur Behandlung von Crosscuts in der Softwareentwicklung vorgestellt. Dabei wurde auf merkmalsorientierte und aspektorientierte Programmierung eingegangen. Auf diesen Ansätzen bauen weitere auf, die sowohl FOP als auch AOP Konzepte verwenden. Alle Ansätze versuchen durch die Modularisierung von Merkmalen, die in OOP über mehrere Klassen verteilt vorliegen, Verständlichkeit und Wiederverwendbarkeit von Programmcode zu erhöhen. Das dabei verwendete Prinzip *Separation of Concerns* stellt die Grundlage zur Entwicklung von Produktlinien dar. Weitere in diesem Zusammenhang wesentliche Eigenschaften sind Konfigurierbarkeit, Erweiterbarkeit und Wartbarkeit von Software. In diesem Kapitel werden die bereits vorgestellten und weitere Ansätze auf diese Eigenschaften hin untersucht.

Die detaillierte Betrachtung aller Ansätze ist nicht möglich, weshalb nur besondere Eigenschaften und wichtige Probleme der einzelnen Ansätze besprochen werden. Eine Zusammenfassung der Schwierigkeiten, die sich in den einzelnen Ansätzen darstellen, wird zum Abschluss des Kapitels gegeben. Für genauere Untersuchungen wird bei den jeweiligen Ansätzen auf entsprechende Literatur verwiesen.

Die OOP ist derzeit das erfolgreichste Programmierparadigma und wird in allen Bereichen der Wirtschaft zur Entwicklung von Software verwendet. Aus diesem Grund wird zu Beginn des Kapitels auch die OOP auf eine Eignung zur Produktlinienentwicklung untersucht.

3.1 Produktlinienentwicklung mit OOP

Mit Hilfe von Klassen bietet die OOP die Möglichkeit, große Softwarekomponenten zu entwickeln. Durch Vererbung kann eine Klasse erweitert werden, um ihr Funktionalität hinzuzufügen. Aggregation von Klassen wird verwendet komplexe Klassen oder Komponenten durch Zusammenfassen mehrerer Klassen zu bilden. Die Verwendung innerer

Klassen (siehe Abschnitt 2.1.3) bietet außerdem die Möglichkeit Klassen zu gruppieren und den Zugriff auf diese zu beschränken. Namensräume bieten eine ähnliche Möglichkeit, werden jedoch häufig verwendet, um ganze Bibliotheken zusammenzufassen. Eine Verwendung für Komponenten ist aber auch möglich.

Separation of Concerns

Die Trennung der Merkmale einer Software ist mit zunehmender Komplexität die Grundlage für die Skalierung bei der Entwicklung, Wartung und Evolution. Mit herkömmlicher OOP kann diese Trennung nicht immer vorgenommen werden. Eine Möglichkeit zur Umsetzung des Kollaborationenentwurfs besteht in der Zerlegung einer Klasse in eine Klassenhierarchie. Jede Klasse der Hierarchie implementiert dabei nur ein bestimmtes Merkmal (eine Rolle des Kollaborationenentwurfs). Die Umsetzung dieses Ansatzes ist aber praktisch nicht realisierbar:

- Für die Implementierung eines Merkmals muss die Klassenhierarchie erweitert werden, was umfangreiche Änderungen am Quelltext nach sich zieht.
- Die Konfiguration kann nur durch Metaprogrammierung mit Präprozessoranweisungen erfolgen[1].

Verdeutlicht wird dies in Abbildungen 3.1 - 3.3 am Beispiel eines Arrays mit den Merkmalen „Synchronisation“ und „Sortierung“[2].

```
1  //Include der Konfiguration
2  #include "ArrayCfg.h"
3
4  //Einfaches Array
5  class Array_Simple { .. };
6
7  //Sortierung eines Arrays
8  //Basisklasse durch Konfiguration festgelegt
9  class Array_Sort : public Array_Sort_Super { .. };
10
11 //Synchronisation eines Arrays
12 //Basisklasse durch Konfiguration festgelegt
13 class Array_Sync : public Array_Sync_Super { .. };
```

Abbildung 3.1: Alternative Merkmale in OOP (Array.h)

[1] Die Verwendung von Templates zur Konfiguration sei hier explizit ausgeschlossen, da Mixin Layers gesondert besprochen werden (Siehe 3.2.1).

[2] Das Beispiel soll einen Eindruck von Komplexität einer Konfiguration mit Präprozessoranweisung vermitteln. Der genaue Inhalt ist für das weitere Verständnis nicht von Belang.

```
1  //Für dieses Bsp. Basis von Array_Sort
2  //immer Array_Simple
3  #define Array_Sort_Super Array_Simple
4
5  // ** Array mit Synchr. und Sortierung **
6  #if defined(Sync) && defined(Sort)
7  //Festlegen der Basisklasse
8  #define Array_Sync_Super Array_Sort
9  //Definition der finalen Klasse
10 #define Array Array_Sync
11 #else //#if defined(Sync) && defined(Sort)
12
13 // ** Sync und Sort nicht kombiniert **
14 //Basisklasse für Synchronisation
15 #define Array_Sync_Super Array_Simple
16
17 //Array mit Synchronisation
18 #if defined(Sync)
19 #define Array Array_Sync
20 #else //#if defined(Sync)
21
22 //Array mit Sortierung
23 #if defined(Sort)
24 #define Array Array_Sort
25 #else //#if defined(Sort)
26
27 //Einfaches Array
28 #define Array Array_Simple
29 #endif  //#if defined(Sort)
30 #endif  //#if defined(Sync)
31 #endif  //#if defined(Sync) && ..(Sort)
```

Abbildung 3.2: Metaprogramm zur Konfiguration in OOP (ArrayCfg.h)

```
1 //Festlegen der Merkmale
2 #define Sort  //Sortierung
3 #define Sync  //Synchronisation
4
5 //include der Implementierung
6 #include "Array.h"
7
8 //Synchr. und sortiertes Array
9 Array a;
```

Abbildung 3.3: Konfiguration und Verwendung alternativer Merkmale in

Abbildung 3.1 zeigt die Implementierung der einzelnen Merkmale und muss für beliebige Klassen anwendbar sein. Somit müssen die Basisklassen (Zeilen 9 und 13) in einer Konfiguration festgelegt werden (Abbildung 3.2 Zeilen 3, 8 und 15). Die eigentliche Auswahl zu verwendender Merkmale gestaltet sich sehr einfach (siehe Abbildung 3.3, Zeilen 1-3). Bereits bei geringer Anzahl zu kombinierender Merkmale ist aber eine unübersichtliche Vorbereitung der Konfigurationen notwendig (siehe Abbildung 3.2). Werden weitere Merkmale verwendet, nimmt die Komplexität der Konfiguration exponentiell zu. Sind - wie bei der Produktlinienentwicklung üblich - mehrere Klassen von einem Merkmal betroffen, weitet sich die Konfiguration auf diese aus und führt zwangsläufig zu Code Tangling. Neben hohem Aufwand ergibt sich somit schlechte Wartbarkeit und ein geringes Maß an Wiederverwendbarkeit.

Eine weitere Möglichkeit zur Auswahl alternativer oder optionaler Komponenten besteht durch eine dynamische Konfiguration, also eine Kombination von Komponenten zur

Laufzeit. Hier besteht ein erhöhtes Maß an Flexibilität, was durchaus von Interesse ist. Die fehlende Möglichkeit zur statischen Konfiguration stellt aber Probleme in Bezug auf verwendete Ressourcen durch größeren Programmcode dar. Die Beeinflussung des Laufzeitverhaltens ist dadurch ebenfalls möglich. Eine Lösung des Skalierbarkeitsproblems wird nicht erreicht, da alle benötigten Komponenten implementiert werden müssen.

Wiederverwendbarkeit

Die Wiederverwendbarkeit von Softwarekomponenten ist allgemein mit OOP ohne große Probleme möglich. Auf Grund der besprochenen Probleme bei der Trennung von Merkmalen, ist dies nur für vollständige Komponenten in einer bestimmten Konfiguration möglich. Für Teile konfigurierbarer Komponenten (sofern dies überhaupt mit OOP möglich ist) oder bestimmte Merkmale, ist dies nur mit hohem Aufwand oder gar nicht erreichbar.

Schlussfolgerungen

Eine effektive Entwicklung von Produktlinien mit OOP ist nicht möglich. Probleme treten in allen Bereichen auf, die für die Produktlinienentwicklung von Interesse sind. Es sind daher andere Ansätze zu verwenden, die mit FOP (vgl. Abschnitt 2.3) bereits vorgestellt wurden. Zunächst wird untersucht, welche Möglichkeiten derzeit für die Programmiersprache C++ bestehen, Produktlinien zu entwickeln.

3.2 Erweiterungen von C++

Die Programmiersprache Java findet immer häufiger Anwendung, dennoch findet auch heute Softwareentwicklung zu großen Teilen mit der Programmiersprache C++ statt. Ursachen hierfür sind z. B. die Möglichkeiten der Hardware nahen Programmierung oder die weite Verwendung der Programmiersprache in bereits existierenden Projekten, bei denen eine Umstellung auf Java nicht möglich ist. Des Weiteren ist seit einiger Zeit mit ANSI C++ auch eine einheitliche Basis zur Programmierung in C++ vorhanden, die eine Entwicklung für verschiedene Betriebssysteme ermöglicht und keine virtuelle Laufzeitumgebung wie im Falle von Java benötigt.

Mit Mixin Layers und AOP existieren Ansätze die das Prinzip *Separation of Concerns* auch für die Programmiersprache C++ unterstützen. In diesem Abschnitt werden beide Ansätze auf ihre Anwendbarkeit bei der Produktlinienentwicklung untersucht.

3.2.1 Mixin Layers

Die Verwendung von Mixin Layers wurde in Abschnitt 2.3.5 vorgestellt. Sie sind derzeit eine der wenigen Möglichkeiten zur Umsetzung des Kollaborationenentwurfs für C++[3]. Wie auch bei einfachen Mixins treten dabei eine Reihe von Problemen auf.

Praktische Gesichtspunkte

Die Implementierung von Mixin Layers in C++ erfolgt durch die Verwendung von Templates. Dies verursacht z. B. technische Probleme, da Funktionsdefinitionen meist in den Headerdateien platziert werden müssen. Es entstehen sehr große und unübersichtliche Dateien, da eine ganze Schicht in einer Datei implementiert werden muss. Bei einer Instanziierung aus mehreren Quellcodedateien ist außerdem nur die Verwendung von inline-Funktionen möglich, da andernfalls die Gefahr besteht, dass vom Compiler Funktionscode mehrfach erzeugt wird. Die Implementierung von Funktionen in gewöhnlichen C++ Dateien (`.cpp, .cc`, etc.) ist möglich, jedoch müssen dann Includes dieser Dateien bei der Instanziierung der Templates verwendet werden. Auch in diesem Fall besteht das Problem mehrfach definierter Funktionen. Die Lesbarkeit des Programmcodes wird durch Funktionsdefinitionen außerhalb der Klassen weiter verschlechtert: Die Angabe der Templateparameter und der zusammengesetzten Klassennamen (Abbildung 3.4, Zeilen 22-23), eine Verwendung von Funktionstemplates (Zeilen 29-31) oder gar Templatespezialisierungen ist nur noch schwer verständlich. Änderungen in Parametern führen außerdem zu umfangreichen Änderungen am Quelltext.

Konfigurierbarkeit

Die Konfigurierbarkeit ist eine grundlegende Eigenschaft, die bei der Entwicklung von Produktlinien erfüllt sein muss. Verglichen mit reiner OOP ist mit Mixin Layers zwar eine Konfiguration möglich, jedoch entsteht bei der Instanziierung der Templates durch die verschachtelte Angabe der Schichten kaum überschaubarer Quellcode. So werden z. B. in [Sic04] für einen konkreten Anwendungsfall mehrere Zeilen aufeinander folgenden Codes der Form `LayerX <LayerY <LayerZ < ... > > >` benötigt, um eine Software zu konfigurieren.

Konstruktorproblem

Das Konstruktorproblem, wie bereits für Mixins beschrieben (vgl. Abschnitt 2.3.5), führt auch für Mixin Layers dazu, dass entweder Standard-Konstruktoren verwendet werden

[3]Eine weitere Möglichkeit wird mit P++ in Abschnitt 3.2.2 vorgestellt.

```
1  //Mixin Layer
2  //Basislayer durch Parameter SUPER vorgegeben
3  template <class SUPER>
4  class Layer1 : public SUPER {
5  public:
6    //Verbindung als Innere Klasse
7    class Connection : public SUPER::Connection {
8    public:
9      template <class T>
10       void Send(const T& value){ .. };
11   };
12   //Weitere Innere Klasse
13   class Server : public SUPER::Server {
14   public:
15     //Erstellen einer neuen Verbindung
16     Connection* NewConn();
17   };
18 };
19
20 //Funktionsdefinition außerhalb einer Klasse
21 //zum Erstellen einer neuen Verbindung
22 template <class SUPER>
23 Layer1<SUPER>::Connection* Layer1<SUPER>::Server::NewConn() {
24   //erstellen einer neuen Instanz
25   return new Connection;
26 }
27
28 //Funktionstemplates in Mixin Layers
29 template <class SUPER>
30 template <class T>
31 void Layer1<SUPER>::Connection::Send(const T& value) { .. }
```

Abbildung 3.4: Implementierung von Mixin Layers

müssen oder eine Berücksichtigung aller Konstruktoren der Basisklassen notwendig ist. Eine Lösung wie bei den Mixins [EBC00] ist denkbar, wird aber durch die zusätzliche Verwendung innerer und äußerer Klassen erschwert.

Self Problem

Bei der Instanziierung einer Klasse innerhalb eines Mixin Layers wird eine Instanz der endgültigen, also vollständig verfeinerten Klasse benötigt. Da diese während der Implementierung nicht bekannt ist, ist ein Erzeugen eines solchen Objektes aus einem Mixin Layer heraus nicht möglich. Das Problem ist aus der OOP bekannt und wird als *Self Problem* [Lie86] bezeichnet.

Durch die Verwendung virtueller Funktionen wird dies umgangen. So wird in Abbildung 3.4, Zeile 25 eine Instanz der Klasse `Layer1::Connection` erstellt. In weiteren Schichten wird diese Funktion jeweils neu implementiert, um den jeweils korrekten Typ

der Schicht zu verwenden. Der Quelltext einer Schicht ist in anderen Schichten nicht verwendbar, obwohl er gleiche Funktionalität implementiert und sich nur bezüglich der erzeugten Klasse unterscheidet.

Für C++ Mixin Layers existiert eine weitere Lösung dieses Problems [BC90], bei der der Typ der finalen Klasse „nach oben“ an den Basislayer über einen Template Parameter weitergegeben wird (Zeile 3 der Abbildung 3.5). Auf diese Weise ist es möglich, Instanzen der finalen inneren Klassen zu erstellen (Zeile 15). Folge ist aber eine weitere Erhöhung der Komplexität der Implementierung und der Konfiguration (siehe Abbildung 3.6).

```
1  //Basislayer mit finalem Typen
2  //als Parameter
3  template <typename FINAL>
4  class Base {
5  public:
6    //Typdefinition zur Verwendung
7    //in erbenden Schichten
8    typedef FINAL Final;
9    ..
10   class Server {
11   public:
12     typename Final::Connection* NewConn() {
13       //Erstellen einer Instanz der
14       //finalen Klasse
15       return new Final::Connection;
16     };
17   };
18 };
```

Abbildung 3.5: Propagieren von Typen in Mixins – Definition

```
1  //Instanziierung mit Propagieren
2  //des finalen Typen
3  class Programm : public
4      Layer1 <Base <Programm> > {};
```

Abbildung 3.6: Propagieren von Typen in Mixins – Konfiguration

Werkzeugunterstützung

Die Unterstützung des Softwareentwicklers durch IDE, Debugger, etc. ist für Mixin Layers nicht vorhanden. So wird die Programmierung z. B. durch das Fehlen einer Übersicht der verwendeten Schichten oder einer automatischen Quellcodeergänzung erschwert. Beim Debuggen bereitet das Untersuchen von Variablen innerhalb der komplexen verschachtelten Schichtenstrukturen erhebliche Probleme. In Einzelfällen ist der Compiler außerdem nicht in der Lage, die recht komplexen Templates zu parsen[4]. Dies schränkt die Verwendung der Mixin Layers stark ein.

[4]Beispielsweise treten mit dem Microsoft Compiler des Visual Studio .NET 2003 Probleme auf, wenn Funktionsdefinitionen nicht innerhalb der Klasse vorliegen.

Entwurfsregeln

Entwurfsregeln, wie sie in AHEAD existieren, sind für Mixin Layers nicht vorhanden. Eine Schicht definiert zwar implizit ein Interface, welches die darüber liegende Schicht anbieten muss, bzw. die darunter liegende verwenden kann, weitere Beschränkungen der Konfiguration können aber nicht angegeben werden. Fehlkonfigurationen sind daher nur durch ausführliche Dokumentation zu verhindern oder im Nachhinein durch Tests festzustellen. Mit Hilfe der Template-Metaprogrammierung lassen sich dennoch Entwurfsregeln angeben. Eine solche Lösung ist aber sehr aufwendig zu entwickeln, schlecht zu warten und bedarf aufwendiger Anpassungen bei jeder Änderung.

Schlussfolgerungen

Neben den besprochenen Problemen existieren weitere, die die Verwendung von C++ Templates zur Implementierung von Mixin Layers erschweren. Für weitere Details sei auf [BC90] verwiesen. Zusammenfassend lässt sich feststellen, dass Mixin Layers für praktische Anwendungen nur bedingt geeignet sind.

3.2.2 P++

Singhal und Batory stellten mit *P++* [SB93] eine Umsetzung des GenVoca Konzeptes für C++ vor. Die Syntax der Spracherweiterung ähnelt der Syntax der Mixin Layers. Die Implementierung der Rollen wird ebenfalls durch innere Klassen vorgenommen. Diese werden in *Komponenten* zusammengefasst, die den äußeren Klassen der C++ Mixin Layers entsprechen. Die zu verwendende Basiskomponente wird auch hier durch Templateparameter vorgegeben. Eine zusätzliche Parametrisierung zur Erstellung generischer Komponenten ist ebenfalls möglich.

Ein Unterschied zu den C++ Mixin Layers ist die Verwendung von *Realms*. Sie stellen Schnittstellen für Komponenten dar und enthalten Klassen mit Methodendeklarationen. Die zugehörigen Implementierungen der Methoden werden von den Komponenten gestellt. Erweiterungen von Schnittstellen und deren Implementierung in den Komponenten erfolgt analog der Vererbung der OOP.

Die Konfiguration einer Software erfolgt in P++ wie von den C++ Mixin Layers bzw. der Instanziierung von Templates bekannt. Dies führt bei komplexen Komponenten bzw. Softwaresystemen auch hier zu langen Instanziierungen.

Mit Hilfe eines Codetransformationssystems wird P++ Code in herkömmlichen C++ Code transformiert, der abschließend von einem Compiler in Maschinencode übersetzt wird.

In dieser recht frühen Umsetzung des Kollaborationenentwurfs bestehen die gleichen

Probleme wie sie für C++ Mixin Layers besprochen wurden. Das Konstruktorproblem z. B. bleibt auch hier unberücksichtigt. Vorteile entstehen lediglich durch die indirekte Festlegung von Entwurfsregeln bei der Verwendung der Komponentenschnittstellen.

3.2.3 Aspektorientierte Programmierung

Wie in Abschnitt 2.4 gezeigt wurde, ist die AOP sehr gut für die Implementierung homogener Crosscuts geeignet. Dabei wird Redundanz in der Implementierung verhindert. Außerdem ist der eigentliche Quelltext frei von Codefragmenten, die Übersichtlichkeit und Verständlichkeit einschränken. Ihre Definition erfolgt in den Aspekten, getrennt vom übrigen Quelltext. Homogene Crosscuts sind allerdings nicht sehr häufig anzutreffen. Laufzeitanalysen (Logging, Profiling), Synchronisation oder Cachen von Berechnungswerten sind einige Beispiele für die sich AOP sehr gut eignet. Probleme entstehen im Hinblick auf die Entwicklung von Produktlinien. Im Folgenden sollen diese näher betrachtet werden.

Verständlichkeit des Quelltextes

Ein Vorteil der AOP ist die Modularisierung von Crosscutting Concerns. Die Umsetzung in Aspekten birgt aber Schwierigkeiten in Bezug auf die Verständlichkeit betroffener Quelltextabschnitte. Durch die Trennung von Aspekten und Klassen ist nicht ohne weiteres erkennbar, an welchen Stellen sich ein Joinpoint befindet. Im Quelltext kann so nicht erkannt werden, ob z. B. eine Methode durch einen Aspekt modifiziert wird oder nicht. Das Beachten aller Aspekte und damit globales Verständnis des Programmcodes ist daher notwendig, um einen bestimmten Programmabschnitt verstehen zu können. Unterstützung durch Werkzeuge könnte hier Abhilfe schaffen.

Neben dem erschwerten Verständnis des Quelltextes besteht ein Problem bei der Analyse existierender Aspekte. Da Pointcuts Suchausdrücke verwenden, um Codestellen für die Ausführung von Advice-Code zu definieren, ist nicht sichergestellt, dass alle notwendigen Quelltextstellen gefunden werden. Weitere Fehler entstehen, wenn keine Möglichkeit zur Überprüfung der Suchausdrücke besteht, wie es bei derzeitigen Implementierungen der Fall ist. So führen Schreibfehler in Suchausdrücken zum „Deaktivieren“ von Pointcuts oder zum Auslassen einzelner Joinpoints.

Merkmalszusammenhalt

Der Merkmalszusammenhalt *feature cohesion* [LHBC05] ist ein weiteres Problem der AOP. Häufig sind für die Implementierung eines Merkmals mehrere Aspekte und auch zusätzliche Klassen notwendig. Die AOP bietet keinerlei Möglichkeit alle Elemente, die

ein Merkmal ausmachen (Aspekte und Klassen) zu kennzeichnen und damit zusammenzufassen. Der Merkmalszusammenhalt ist wesentlich für das Verständnis des Quelltextes und die Konfiguration der Software (siehe Konfiguration).

Wartbarkeit

Die AOP beruht im Falle von AspectC++ auf der Verwendung von Codegeneratoren. Wie in Abschnitt 2.6.3 bereits beschrieben, bereitet dieses Vorgehen erhebliche Probleme in Bezug auf die Wartbarkeit und das Debuggen. Es besteht jedoch die Möglichkeit durch Compilerdirektiven Informationen im kompilierten Programmcode abzulegen, so dass während des Debuggens der ursprüngliche Quelltext bearbeitet werden kann. Dennoch besteht bei der AOP die Schwierigkeit, dass die im Debugger sichtbaren Objekte nicht mit den Klassen übereinstimmen, wie sie im Quelltext vorzufinden sind, wenn sie durch Aspekte modifiziert wurden.

Erweiterbarkeit

Die Erweiterbarkeit von AOP-Code ist mittels Vererbung und virtuellen Pointcuts gegeben. Probleme bestehen bei der Erweiterung vorhandenen OOP-Codes, auf den Aspekte einwirken. Hier besteht keine Möglichkeit der Steuerung bereits existierender Aspekte. Neu hinzugefügter Quelltext wird daher ebenfalls von den Aspekten modifiziert, auch wenn diese nicht für die Verwendung mit den neuen Quelltextelementen vorgesehen sind. Eine Modifikation existierender Pointcuts ist notwendig, um diese mit neuen Klassen und Funktionen verwenden zu können. Eine genaue und aufwendige Untersuchung vorhandener Aspekte sowie eine Erweiterung des Aspekt-Codes werden deshalb notwendig. Vorhandene Strukturen im Quelltext werden dabei möglicherweise zerstört.

Konfiguration

Die Konfigurierbarkeit in aspektorientierten Ansätzen ist üblicherweise auf das Aktivieren bzw. Deaktivieren von Aspekten und die Reihenfolge ihrer Anwendung beschränkt. Eine Gruppierung von Aspekten ist nicht möglich, so dass bei Merkmalen, die aus mehreren Aspekten bestehen, die einzelnen Aspekte aktiviert bzw. deaktiviert werden müssen. Dies könnte wiederum durch Unterstützung entsprechender Werkzeuge behoben werden. Da das Hinzufügen von Klassen unabhängig von den Aspekten erfolgt, besteht keine Möglichkeit Klassen zu konfigurieren, die für die Implementierung bestimmter Merkmale notwendig sind.

Häufig ist die Reihenfolge der Anwendung der Aspekte wichtig für deren Funktionalität. Eine solche Reihenfolge kann innerhalb von Aspekten festgelegt werden. Es besteht

jedoch keine Möglichkeit, dies von außen vorzunehmen, was die Wiederverwendbarkeit einschränkt [LHBC05].

Schlussfolgerungen

AOP bietet für einige Anwendungsfälle sehr elegante Lösungen. Dies betrifft zumeist homogene Crosscuts. Für die Verwendung zur Umsetzung des Kollaborationenentwurfs ist die AOP keine angemessene Lösung. Hier ist die merkmalsorientierte Programmierung überlegen. Die Unterstützung durch Werkzeuge kann einige der bestehenden Probleme beheben. Grundsätzliche Schwierigkeiten wie fehlender Merkmalszusammenhalt, schlechte Erweiterbarkeit und umständliche Konfiguration bleiben aber bestehen. Für weitere Diskussionen zu diesem Thema sei auf [LHBC05] verwiesen.

3.3 FOP und Java

Im letzten Abschnitt wurde auf Methoden eingegangen, die bei der Entwicklung von Produktlinien mit C++ hilfreich sind. Da sowohl C++ Mixin Layers als auch AOP keine ausreichende Unterstützung bieten, sollen nun Ansätze untersucht werden, deren Umsetzung für C++ diese Lücke schließen könnten.

Mit der AHEAD Tool Suite und Java Layers sind Java Erweiterungen vorhanden, die eine Nutzung von Java zur FOP ermöglichen. Während Java Layers eine Umsetzung der Mixin Layers ähnlich den C++ Mixin Layers sind, findet sich keine C++ Umsetzung, die den Erweiterungen der ATS ähnlich ist.

Im Folgenden werden beide Ansätze näher untersucht und Probleme betrachtet, die bei der Umsetzung des Kollaborationenentwurfs auftreten.

3.3.1 Die AHEAD Tool Suite

Die Jakarta Tool Suite (JTS) und ihre Weiterentwicklung, die AHEAD Tool Suite (ATS), führen Spracherweiterungen für Java ein, die speziell für die Implementierung von Kollaborationen vorgesehen sind. Dabei wird besonderer Wert auf die Lösung bereits besprochener Probleme gelegt (siehe Abschnitt 2.3.6). Das *Konstruktorproblem* wird durch das *Propagieren* von Konstruktoren gelöst und es wird die Verfeinerung vorhandener Konstruktoren ermöglicht. Das *Self Problem* wird gelöst, in dem bei der Instanziierung eines Objektes immer eine Instanz der vollständig verfeinerten Klasse erzeugt wird. Auch bei der Verwendung einer Klasse (z. B. bei der Vererbung) wird immer die endgültige Variante genutzt. Probleme bezüglich der Konfigurierbarkeit werden durch die Auslagerung in Konfigurationsdateien, den `.equation`-Dateien gelöst. Einige Probleme bleiben aber

auch hier offen und werden im Folgenden besprochen.

Homogene Crosscuts

Homogene Crosscuts können auch mit dieser Lösung nicht entsprechend umgesetzt werden. Hier ist Redundanz bei der Implementierung nicht zu umgehen. Folglich ist die Wartung des Quelltextes entsprechend aufwendig und fehleranfällig.

Crosscuts zwischen Merkmalen

Auftretende Abhängigkeiten zwischen einzelnen Merkmalen bereiten ebenfalls große Probleme. Sich überschneidende Merkmale, wie sie mit dem MDSOC Ansatz adressiert werden, müssen in der ATS auf eine Dimension reduziert werden. Ursache hierfür ist die eindimensionale Darstellung der Merkmale bei der Konfiguration. Überschneiden sich zwei oder mehr Merkmale, wie etwa im Beispiel des Strings der Abbildung 2.30, können diese nicht adäquat wiedergegeben werden[5].

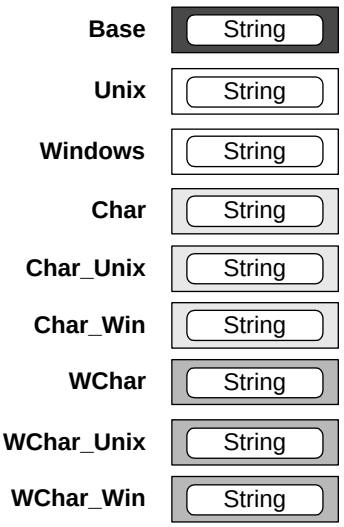

Abbildung 3.7: Crosscuts zwischen Merkmalen – Linearisierung

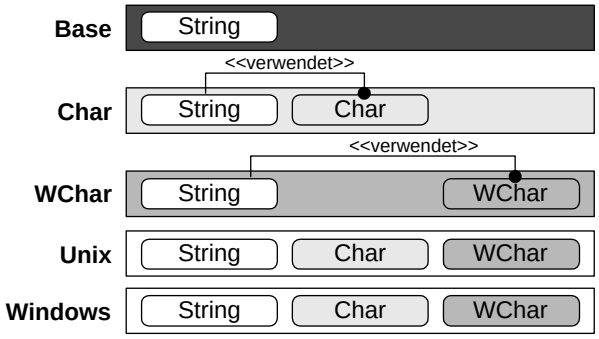

Abbildung 3.8: Crosscuts zwischen Merkmalen – Umwandlung in Klassen

Zur Lösung dieses Problems werden alle Kombinationen der Merkmale als eigenständige Merkmale definiert (siehe Abbildung 3.7). Die ursprünglich zweidimensionale Beziehung der Merkmale wird damit auf eine Dimension reduziert. Dies hat zur Folge,

[5]Unterschiedliche Implementierungen für verschiedene Betriebssysteme sind unter Java eher selten. An dieser Stelle ist lediglich das Überschneiden der Merkmale von Interesse und der technische Hintergrund belanglos.

dass die Konfiguration der eigentlichen Merkmale nicht mehr möglich ist, sondern die Teilmerkmale konfiguriert werden müssen (Layer `Char_*` und `WChar_*` der Abbildung 3.7). Die Merkmale selbst (`Unix`, `Windows`, `Char`, `WChar`) bleiben zusätzlich erhalten, da sie die von den anderen Merkmalen unabhängigen Implementierungen fassen. Des Weiteren ist mit steigender Anzahl abhängiger Merkmalsdimensionen eine exponentiell wachsende Anzahl von Merkmals-Kombinationen in der Konfiguration notwendig, die dem *Library Scalability Problem* entspricht (siehe Abschnitt 2.3.5). Die Konfiguration selbst kann in diesem einfachen Fall durch das Zusammenfassen von Merkmalen erleichtert werden, bereitet aber bei mehr als zwei Merkmalen Probleme.

Eine weitere Möglichkeit zur Lösung des Problems besteht in der Einführung einer neuen Klasse für eine der verwendeten Dimensionen (Abbildung 3.8). Im Kollaborationenentwurf werden Crosscuts beider Merkmalsarten in Crosscuts einer Merkmalsart mit einer Klasse je Merkmal der anderen Merkmalsart überführt. So sind in Abbildung 3.8 für die Dimension „Character Type“ die Klassen `Char` und `WChar` implementiert. Diese erhalten nun in den Merkmalen Unix bzw. Windows die jeweils systemspezifischen Implementierungen (Rollen der Klassen `Char` und `WChar` in den Schichten Unix und Windows der Abbildung 3.8). Auf diese Weise bleibt die getrennte Konfiguration der Merkmale Unix / Windows bzw. Char / WChar möglich. Diese „Transformation“ eines Merkmals in eine Klasse entspricht jedoch nicht dem Idealfall, da eigentlich die Erweiterung einer Klasse stattfinden sollte. Zum anderen stößt auch diese Lösung bei der Verwendung von mehr als zwei Merkmalsarten an seine Grenzen. Hier müssen wiederum Kombinationen mehrerer Merkmale entweder als eigene Merkmale oder als eigene Klassen (als Umsetzung der Merkmale) umgesetzt werden. Eine erneute Transformation der Merkmale in Klassen ist nicht möglich, da diese wiederum voneinander abhängig wären. Demnach ist eine Auflösung der mehrdimensionalen Zusammenhänge mit diesem Ansatz nicht ohne weiteres möglich.

Der von Batory et al. vorgestellte Ansatz [BLS03] führt eine Transformation des MDSOC Ansatzes auf eine Dimension vor. Dabei wird ein mathematisches Modell zur Beschreibung und Überführung in den GenVoca Ansatz verwendet. Angaben zur technischen Umsetzung werden allerdings nicht gemacht. Ein mathematisches Modell zur Beschreibung der Interaktionen zwischen Merkmalen wird außerdem in [LBN05] vorgestellt.

Feature Optionality Problem

Abhängigkeiten zwischen Merkmalen bedingen auch das Feature Optionality Problem: Werden Merkmale aus einer Konfiguration entfernt, so kommt es zu Problemen, wenn Klassen oder Methoden dieses Merkmals verfeinert werden. Die Konfiguration muss aber

auch in einem solchen Fall gültig bleiben[LBN05].

Ein Hinzufügen oder Entfernen anderer Merkmale muss also möglich sein, ohne ein verfeinerndes Merkmal modifizieren zu müssen. Mit der ATS ist dies nicht ohne weiteres möglich. Wird beispielsweise durch das Entfernen eines Merkmals eine Methode entfernt, die durch ein weiteres Merkmal modifiziert wird, so wird in der verfeinernden Klasse auf eine nicht existierende Methode verwiesen. Dies ist für viele Merkmale ein üblicher Sachverhalt und darf somit keine ungültige Konfiguration erzeugen.

Mit AOP wird dieses Problem gelöst, in dem die entsprechende Funktion von zugehörigen Pointcuts nicht mehr „ausfindig“ gemacht werden kann. Dies führt wie bereits besprochen zu Problemen, wenn Schreibfehler in Pointcuts oder Funktionsnamen vorhanden sind, da nicht zwischen fehlenden Merkmalen und Fehlern im Programmcode unterschieden werden kann.

Für FOP existiert eine Lösung [LBN05] analog der bereits vorgestellten Linearisierung eigentlich mehrdimensionaler Merkmale. Dabei wird für jede Kombination zweier sich beeinflussender Merkmale eine eigene Schicht erzeugt. Dementsprechend schlecht skaliert diese Lösung, was bei vielen voneinander abhängigen Merkmalen in einer nicht überschaubaren Anzahl von Teilmerkmalen resultiert.

Wartbarkeit

Die Wartbarkeit von Software profitiert von der Verwendung der FOP. Mit fehlender Unterstützung durch Werkzeuge, wie etwa für das Debuggen, entstehen neue Probleme, die bei anderen Ansätzen nicht existieren. Die Verwendung einer Quellcodetransformation führt zum Problem, dass der Debugger mit dem transformierten Quelltext arbeitet. Vom Entwickler werden jedoch Modifikationen am ursprünglichen Quelltext vorgenommen. Hier besteht demnach Notwendigkeit weiterer Unterstützung durch entsprechende Werkzeuge.

Schlussfolgerungen

Insgesamt ist die Umsetzung der FOP mit der ATS nahezu vollständig. Auftretende Probleme sind zum Teil dem Ansatz selbst (Feature Optionality Problem, homogene Crosscuts, Crosscuts zwischen Merkmalen), bzw. der zurzeit noch fehlenden Unterstützung durch entsprechende Werkzeuge zuzurechnen.

3.3.2 Java Layers

Neben der Möglichkeit Mixin Layers mit C++ Templates umzusetzen, ist mit den *Java Layers (JL)* eine ähnliche Implementierung für die Programmiersprache Java vorhan-

den [CL01]. Im Weiteren werden die Probleme besprochen, die sich wesentlich von der Implementierung der Mixin Layers unter C++ unterscheiden.

Erweiterungen

Außer den bereits erwähnten Eigenschaften von C++ Mixin Layers werden mit JL weitere, für die Implementierung von Kollaborationen wesentliche, Konstrukte eingeführt. Zu den wichtigsten gehören:

- Propagieren von Konstruktoren ähnlich der ATS
- Definieren von Einschränkungen für die parametrisierten Basisklassen
- Damit in Zusammenhang steht die Möglichkeit für die späteren Basisklassen Schnittstellen zu definieren, die diese implementieren müssen (*constrained parametric polymorphism*)[6]
- Verwendung virtueller Typen

Die Verwendung von Schnittstellendefinitionen ermöglicht es, Einschränkungen für zu verfeinernde Klassen und ganze Schichten zu definieren. Einschränkungen wie sie mit den Entwurfsregeln von GenVoca möglich sind, können allerdings nicht gemacht werden.

Self Problem

Virtual Types sind aus der Programmiersprache *Beta* bekannt. Für andere Programmiersprachen wurden sie in [BOW98, Tho97] untersucht. Dabei handelt es sich um Typen, deren endgültiger Typ erst zum Zeitpunkt der Instanziierung eines Objektes einer Klasse festgelegt wird. Diese erlauben es in JL Objekte der am weitesten verfeinerten Variante einer Klasse zu erstellen. Wie bereits für C++ Mixin Layers gesehen, ist das *Self Problem* eine grundsätzliche Schwierigkeit bei der Implementierung auf Schichten basierender Ansätze. Hier besteht generell die Notwendigkeit eine Instanz der finalen Klasse einer Reihe von Verfeinerungen zu erzeugen.

So erzeugt der Aufruf in Zeile 16 der Abbildung 3.4 kein Objekt der endgültigen Klasse, sondern ein Objekt der Klasse `Layer1::Node`. Dies wird bei den C++ Mixin Layers durch das Propagieren der finalen Klasse verhindert. In JL erfolgt dies durch die Verwendung virtueller Typen. Durch einen Aufruf der Form `new This()` kann eine Instanz der verfeinerten Klasse erstellt werden.

[6] Genauer müssen die Klassen *deep subtypes* dieser Schnittstellen sein, für eine genauere Beschreibung der verwendeten Schnittstellendefinitionen und weitere Details sei auf [Sma99] verwiesen.

Konstruktorproblem

Beim Propagieren von Konstruktoren, wie bereits für die ATS gezeigt, werden Konstruktoren in den Verfeinerungen erzeugt, so dass die Basisklassen initialisiert werden können. Ein zu propagierender Konstruktor wird bei JL mit dem Schlüsselwort `propagate` versehen. Sind bereits Konstruktoren in der Subklasse vorhanden, werden deren Signaturen angepasst. Die Initialisierung der Basisklasse erfolgt automatisch.

Schlussfolgerungen

JL bieten einige Erweiterungen zu den C++ Mixin Layers, die die Umsetzung des Kollaborationenentwurfs erleichtern. Dabei werden das Konstruktorproblem und das Self Problem gelöst. Außerdem können Entwurfsregeln durch das Definieren von Schnittstellen erstellt werden. Eine bessere praktische Anwendbarkeit im Vergleich zu C++ Mixin Layers ergibt sich durch die einfachere Syntax. Einige Probleme wie die unübersichtliche Konfiguration, fehlende Unterstützung durch Werkzeuge und unübersichtlicher Quelltext (durch die Definition einer ganzen Schicht in einer Datei) bleiben bestehen. Außerdem sind die für den Kollaborationenentwurf generell zutreffenden Probleme (Feature Optionality Problem, Behandlung homogener Crosscuts, etc.) auch hier vorhanden.

3.3.3 Delegation Layers

Delegation Layers [Ost01] sind eine weitere Implementierung der Mixin Layers für Java, die zwei Unterschiede zu den bereits vorgestellten Implementierungen mit sich bringen: Konfiguration zur Laufzeit (*runtime composition*) und eine erweiterte Wiederverwendung von Code in anderen Kollaborationen.

Runtime Composition

Im Gegensatz zu herkömmlichen Mixin Layers, die eine statische Konfiguration verwenden, kann die Konfiguration in Delegation Layers auch erst zur Laufzeit erfolgen. Diese dynamische Konfiguration hat den Vorteil, dass ein Anwender während der Laufzeit eines Programms entsprechende Merkmale auswählen kann und eröffnet somit eine wesentlich höhere Flexibilität. Die Konfiguration einer Komponente wird, wie für Mixin Layers üblich, im Quelltext vorgegeben. Als Basislayer werden dabei bereits instanziierte Komponenten verwendet (Zeile 14 in Abbildung 3.9). Instanzen von Komponenten lassen sich auf diese Weise zur Laufzeit mit neuen Merkmalen ausstatten.

```
1  class A {
2    void foo() { print("A"); }
3  }
4  class B extends A {
5    void foo() { print("B"); super.foo();}
6  }
7  class C extends A {
8    void foo() { print("C"); super.foo();}
9  }
10 ...
11 A a = new C();
12 a.foo(); // prints "CA"
13 A a = new C<new B()>(); //dynamische Konfiguration
14 a.foo(); // prints "CBA"
```

Abbildung 3.9: Dynamische Konfiguration in Delegation Layers

Weitere Eigenschaften

Wie auch in den Java Layers werden bei den Delegation Layers virtuelle Typen verwendet, um das *Self Problem* zu lösen. In Kombination mit der dynamischen Komposition der Delegation Layers wird die Instanziierung von Klassen ermöglicht, deren Typen der endgültigen Klasse entsprechen, auch wenn die Konfiguration der zugehörigen Komponente erst zur Laufzeit erfolgt.

3.4 Weitere Ansätze

Neben den merkmals- und aspektorientierten Ansätzen existieren weitere, die entweder eine etwas andere Herangehensweise verwenden (MDSOC), oder Prinzipien aus der FOP und der AOP vereinen (Jiazzi, Aspectual Collaborations, Caesar). Dabei wird versucht einige der bekannten Probleme der FOP bzw. AOP zu lösen.

3.4.1 Hyper/J

Die Trennung von Merkmalen in mehrere Dimensionen beim MDSOC Ansatz ist in einigen Punkten dem Kollaborationenentwurf überlegen. So werden voneinander abhängige Merkmalsdimensionen auf kompaktere Weise beschrieben. Redundanz bei der Beschreibung bleibt aus. Die Überführung dieses Ansatzes in eine Schichtenstruktur wie in AHEAD ist möglich [BLS03], führt aber zu Redundanz in der Konfiguration einer Komponente. Weiterhin entsteht die Notwendigkeit, bereits im multidimensionalen Entwurf enthaltene Informationen in Entwurfsregeln zu fassen. Der MDSOC Ansatz kann daher als allgemeinerer Ansatz aufgefasst werden.

Mit *Hyper/J*[7] existiert eine Implementierung dieses Ansatzes für Java. Im Gegensatz zur AHEAD Tool Suite wird hier auf die Verwendung gewöhnlichen Java-Codes gebaut. Auf diese Weise können in Projekten die als OO-Programmcode vorliegen, Klassen verfeinert werden, ohne eine vollständige Neustrukturierung des Quelltextes vornehmen zu müssen. Die Verfeinerungen selbst sind ebenfalls herkömmlicher Java-Code.

Konfiguration

Die in binärer Form vorliegenden Komponenten, werden statisch miteinander verknüpft. Die Konfiguration benötigt entsprechende Beschreibungsmöglichkeiten, wie Klassen und Methoden verschiedener Merkmale zu verbinden sind. Dies hat den Nachteil, dass die Konfiguration selbst umfangreich und komplex ausfällt. Erst in ihr werden die eigentlichen Merkmale definiert. Weiterhin sind für Methoden häufig zusätzliche *Summary Functions* notwendig, die festlegen, wie Rückgabewerte behandelt werden, wenn eine Methode einer Klasse verfeinert wird [TO00]. Ursache dessen ist, dass innerhalb der Verfeinerungen Rückgabewerte der verfeinerten Funktion nicht behandelt werden können. Hier wird so genannter *Glue Code* notwendig, der die einzelnen Merkmale verbindet.

Ein wesentlicher Unterschied zu herkömmlichen merkmalsorientierten Ansätzen ist die Verwendung von Suchausdrücken innerhalb der Konfiguration. Ähnlich der AOP werden gleichzeitig mehrere Klassen erweitert. Auf diese Weise ist auch für homogene Crosscuts eine bessere Wiederverwendbarkeit gegeben. In der Konfiguration kann für einzelne Methoden gesteuert werden, wie sie mit anderen Methoden zu kombinieren sind. Diese Flexibilität ist notwendig, um die Probleme, die sich aus überschneidenden Merkmalen ergeben, zu lösen.

Ein weiteres Problem besteht, da bei der Entwicklung einer Verfeinerung kein direkter Einfluss auf zu verfeinernde Methoden genommen werden kann. Mit Hilfe der Summary Functions kann nur der Rückgabewert einer Methode angepasst werden. Die Entscheidung, ob eine Methode durch eine Verfeinerung überschrieben oder modifiziert wird, muss während der Konfigurierung getroffen werden.

Feature Optionality Problem

Das Feature Optionality Problem tritt in Hyper/J nicht auf, da die Abhängigkeiten zwischen einzelnen Merkmalen während der Konfigurierung berücksichtigt werden können.

[7] http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm

Homogene Crosscuts

In Hyper/J werden homogene Crosscuts mit Hilfe AOP ähnlicher Suchausdrücke umgesetzt. So können sowohl heterogene als auch homogene Crosscuts entsprechend behandelt werden. Die Modifikation von Funktionsparametern wie sie bei der AOP verwendet wird, ist nicht möglich.

Merkmalszusammenhalt

Der Zusammenhalt der Merkmale wird bei Hyper/J erst durch die Konfiguration definiert. Der unabhängig davon vorliegende Quelltext bietet keine Möglichkeit zur Zusammenfassung von Merkmalen. Ein strukturiertes Speichern der Quelltextdateien ist möglich (z. B. ein Ordner je Merkmalsdimension), es werden aber auf Grund der Einschränkungen des Dateisystems keine Zusammenhänge zwischen den Merkmalsdimensionen festgehalten.

Schlussfolgerungen

Hyper/J kann sowohl homogene, als auch heterogene Crosscuts behandeln. ist dabei aber der AOP bzw. FOP unterlegen. Die Behandlung beliebiger mehrdimensionaler Zusammenhänge ist mit Hyper/J möglich, bedarf aber einer sehr komplexen Konfiguration. Das Feature Optionality Problem wird ebenfalls mit Hilfe der Konfiguration gelöst. Die praktische Anwendung ist derzeit auf Grund fehlender Unterstützung durch Werkzeuge nur schwer möglich.

3.4.2 Jiazzi

Jiazzi ist ein System zur Unterstützung komponentenbasierter Softwareentwicklung für Java. Vorgestellt in [MFH01] ermöglicht es in binärer Form (Java Bytecode) vorliegende Komponenten zu verknüpfen. Das Zusammenfügen der Komponenten wird vom Jiazzi Linker vorgenommen, der Jiazzi Komponenten mit herkömmlichen Javaklassen verbindet. So wird ein Ausliefern binärer Komponenten ohne den zugrunde liegenden Quellcode ermöglicht, was für den kommerziellen Einsatz oft grundlegende Voraussetzung ist.

Jiazzi unterstützt während der statischen Verknüpfung auch die Verwendung von Mixins. In den Signaturen der Komponenten, können zu diesem Zweck ähnlich den Mixin Layers die Basisklassen parametrisiert vorgegeben werden.

Neben der Verwendung von Jiazzi zur Implementierung des Kollaborationenentwurfs wurde in [MH03] eine Möglichkeit vorgestellt Jiazzi zur AOP zu verwenden. Damit besteht die Möglichkeit aspektorientierte Konzepte auf herkömmlichem Java-Code

anzuwenden. Die Möglichkeiten Pointcuts zu definieren, unterscheiden sich von denen herkömmlicher AOP Implementierungen. Pointcuts werden mit Hilfe der Verknüpfungssprache (*linking language*) von Jiazzi angegeben. Hier kann wie auch bei anderen Implementierungen auf die Methoden, Membervariablen oder Argumentlisten zugegriffen werden.

Ähnlich Hyper/J bietet damit Jiazzi die Möglichkeit homogene Crosscuts durch eine aufwendige Konfiguration (unter Verwendung der Verknüpfungssprache) zu behandeln. Heterogene Crosscuts werden dabei ähnlich den Java Layers durch die Verwendung parametrisierter Basisklassen implementiert.

Die Konfiguration mit der Verknüpfungssprache von Jiazzi ist im Vergleich zu anderen Ansätzen sehr kompliziert, da sie direkt auf der Ebene des Linkers arbeitet. Die Verwendung der Schnittstellen bereitet dabei besondere Probleme, wenn zwischen den Implementierungseinheiten (Klassen und Komponenten) komplexe Beziehungen bestehen [LHBC05].

3.4.3 Caesar

Aufbauend auf Java wurde mit *Caesar* [MO04] eine Programmiersprache vorgestellt, die Vorteile der merkmalsorientierten und aspektorientierten Programmierung vereinen soll. Die Umsetzung des Kollaborationenentwurfs erfolgt über Innere Klassen ähnlich den Mixin Layers. Aspekte werden in Caesar ähnlich wie die inneren Klassen der Schichten behandelt und liegen neben diesen innerhalb der äußeren, merkmalsbildenden Klasse.

Mit Hilfe *Bidirektionaler Schnittstellen* (*bidirectional interfaces – BI*) werden außerdem Regeln zum Entwurf im Quelltext definiert. Diese Schnittstellen bieten sowohl die Möglichkeit Methoden zu definieren, die von der implementierenden Klasse geliefert werden (`provided`), als auch Methoden die von dieser benötigt werden (`expected`).

Caesar konzentriert sich insbesondere auf Aspekte und deren dynamische Konfiguration. So kann zur Laufzeit festgelegt werden, ob ein Aspekt an ein Objekt gebunden wird (*deployment*). Außerdem ist eine statische Bindung möglich, die zur Konfigurationszeit durch Hinzufügen oder Entfernen von Aspekten gesteuert wird.

Sowohl die dynamische, als auch die statische Konfiguration selbst erfolgt bei Caesar durch Instanziierungen innerhalb des Quelltextes. Sie ist daher der Konfiguration wie sie aus Mixin Layers bekannt ist sehr ähnlich.

Caesar vereint somit Elemente der FOP und der AOP. Die Konfiguration erfolgt im Quellcode und bringt ähnliche Probleme mit sich, wie sie bereits bei Mixin Layer basierten Ansätzen besprochen wurden. Die Überprüfung einer Konfiguration kann zum Teil mit Hilfe von Schnittstellen erfolgen. Entwurfsregeln wie sie mit AHEAD möglich sind

können nicht definiert werden. Des Weiteren folgt die Umsetzung des Kollaborationenentwurfs nicht der GenVoca / AHEAD Theorie und erlaubt daher keine Optimierungen wie in [BO92] vorgeschlagen.

3.4.4 Aspectual Collaborations

Eine weitere Implementierung, die Konzepte der AOP und FOP für Java vereint sind die *Aspectual Collaborations (AC)* [LLO03].

Der Aufbau einer Kollaboration in AC ist der von Caesar ähnlich. Es werden jedoch keine äußeren Klassen verwendet, sondern alle Klassen, die sich in einer Datei befinden zu einer Kollaboration gezählt. Diese wird mit dem Schlüsselwort `collaboration` zu Beginn der Datei deklariert. Verfeinerungen werden nicht mit dem Schlüsselwort `class`, sondern mit `participant` definiert. In ihr können wie auch bei Caesar erwartete Methoden mit `expected` deklariert werden. Aspektorientierte Elemente befinden sich im Vergleich zu Caesar jedoch nicht innerhalb der Kollaboration, sondern innerhalb einer Verfeinerung als `aspectual methods`.

Ähnlichkeit zu Hyper/J findet sich bei der Verbindung von Kollaborationen die in AC aber im Quelltext stattfindet. Zu diesem Zweck nehmen *Attachments* die Zuordnungen der Verfeinerungen zu den verfeinerten Methoden vor. Hier ist außerdem wie auch in Hyper/J eine Verwendung von aspektorientierten Beschreibungen wie z. B. `around` möglich, um die Vorgehensweise bei der Verfeinerung zu steuern. Die Konfiguration ist daher der von Hyper/J ähnlich. Sie ist sehr flexibel, dafür muss jedoch eine hohe Komplexität in Kauf genommen werden.

Das Konstruktorproblem wird bei diesem Ansatz derzeit nicht berücksichtigt. Hier ist ein Verfeinern von Konstruktoren mit dem Erweitern der Signaturen geplant.

3.5 Zusammenfassung

In diesem Kapitel wurden Ansätze untersucht, mit denen eine Entwicklung von Produktlinien ermöglicht wird. Dabei wurde festgestellt, dass die OOP generell unbrauchbar für die Produktlinienentwicklung ist. Bei den auf Schichten basierenden Umsetzungen der FOP (*C++ Mixin Layers*, *P++*, *Java Layers*, *Delegation Layers*, *AHEAD*) wurde festgestellt, dass homogene Crosscuts nur unter Einführung von Redundanz implementiert werden können. Die Implementierungen der AOP (*AspectC++*, *AspectJ*[8]) sind sowohl geeignet homogene als auch heterogene Crosscuts zu implementieren. Es fehlen ihnen

[8]Hier wurde lediglich AspectC++ besprochen, die aufgeführten Probleme sind aber grundsätzlicher Art und betreffen den AOP Ansatz an sich.

aber wesentliche Eigenschaften wie z. B. Merkmalszusammenhalt, Erweiterbarkeit und Wartbarkeit, wodurch sie den FOP Implementierungen unterlegen sind.

Ansätze, die sich beide Paradigmen zu Nutze machen (*Jiazzi*, *Caesar*, *Aspectual Collaborations*, *Hyper/J*) versuchen durch AOP Konzepte Probleme des FOP Ansatzes zu lösen. Dies gelingt zur zum Teil, so dass Nachteile gegenüber den reinen FOP und AOP Implementierungen entstehen. Als Umsetzung des MDSOC Ansatzes für Java spielt *Hyper/J* eine gesonderte Rolle, da Abhängigkeiten zwischen Merkmalen besser dargestellt und implementiert werden können.

Es wurden einige grundlegende Probleme bei der Entwicklung von Produktlinien herausgestellt. Diese betreffen die zugrunde liegenden Ansätze oder speziell deren Umsetzungen. Im Folgenden werden diese Probleme noch einmal zusammengefasst. Tabelle 3.1 stellt die einzelnen Ansätze in Beziehung zu den aufgeführten Problemen. Dabei wird ausreichende Behandlung der dargestellten Probleme durch „+“ gekennzeichnet und unzureichende Behandlung durch „-“. Da zum Teil feine Abstufungen möglich sind, wird eine teilweise vorhandene aber nicht vollständige Behandlung eines Problems mit „+/-“ gekennzeichnet.

1. **Separation of Concerns** ist grundlegendes Ziel der Produktlinienentwicklung. Die Merkmale werden gekapselt und liegen getrennt von den Klassen vor. Die Erstellung von Software kann daher durch eine Beschreibung mit Hilfe der Merkmale erfolgen.

   (a) **Heterogene Crosscuts** werden insbesondere von den auf Kollaborationen basierenden Ansätzen ausreichend behandelt. Die Umsetzung mit AOP benötigt größeren Aufwand und führt dabei andere Probleme ein (z. B. Wartbarkeit, Erweiterbarkeit).

   (b) **Homogene Crosscuts** werden von den auf Schichten basierenden Ansätzen nur mit der Einführung von Redundanz behandelt. Dies bleibt in AOP und kombinierten Ansätzen aus.

   (c) **Crosscuts zwischen Merkmalen** werden in geringem Ausmaß von allen Ansätzen behandelt werden. Eine ausreichende Lösung bietet nur die Trennung solcher Merkmale wie in Hyper/J, was mit entsprechendem Konfigurationsaufwand verbunden ist.

2. Der **Merkmalszusammenhalt** wird mit Hilfe der Kollaborationen umgesetzt. AOP bietet dabei kaum, und kombinierte Ansätze im Vergleich zur AOP geringere Unterstützung.

3. Das **Konstruktorproblem** wird nur von Java Layers und AHEAD sowie der AOP ausreichend gelöst. Die Lösung der C++ Mixin Layers ist keineswegs praktikabel.

4. Das **Self Problem** wird durch *virtuelle Typen* (Delegation Layers, Java Layers), parametrisierte Typen (C++ Mixin Layers) oder die automatische Verwendung der finalen Typen (ATS) gelöst. Andere Ansätze bieten keine Lösung.

5. Die Lösung des **Feature Optionality Problems** erfolgt derzeit nur unter Verwendung von Suchausdrücken. Dies bringt genannte Probleme in Bezug auf Programmierfehler mit sich.

6. Die **Erweiterbarkeit** bereitet insbesondere bei aspektorientierten und zum Teil bei kombinierten Ansätze Probleme, da für Aspekte nur schwer kontrollierbar ist, ob sie auf neu entwickelte Teile der Software wirken.

7. Eine **Überprüfung des Entwurfs** erfolgt in einigen Ansätzen durch die Verwendung von Schnittstellen. Im Falle von AHEAD werden Entwurfsregeln verwendet, die auch semantische Informationen fassen können. Ähnliches ist für C++ Mixin Layers mit Hilfe der Template-Metaprogrammierung denkbar, unter praktischen Gesichtspunkten aber nicht umsetzbar.

8. **Praktische Anwendbarkeit**

   (a) Die **Programmierung** selbst wird in allen Ansätzen durch eine mangelnde Unterstützung von Werkzeugen erschwert. Bei C++ Mixin Layers und P++ kommen komplizierte Sprachkonstrukte hinzu. AOP erschwert die Programmierung, da keine direkte Verbindung der Aspekte mit den betroffenen Codestellen besteht.

   (b) Die **Konfiguration** fällt in einigen kombinierten Ansätzen (Hyper/J, Jiazzi, Aspectual Collaborations) sehr aufwendig aus. Die ATS bietet hier die einfachsten Möglichkeiten und bietet außerdem Möglichkeiten zur Optimierung auf Basis der algebraischen Beschreibung. Einige Ansätze (AHEAD, Hyper/J, Jiazzi, Aspectual Collaborations) ermöglichen außerdem die Konfiguration außerhalb des Quelltextes.

   (c) Das **Debuggen** von Software wird bisher von keinem der betrachteten Ansätze unterstützt. Die Verwendung von AOP Konzepten führt zu weiteren Problemen, da Joinpoints im Quelltext nicht sichtbar sind.

Es wird deutlich, dass herkömmliche Ansätze der FOP und AOP den kombinierten Ansätzen in einigen Bereichen nachstehen. Die kombinierten Ansätze bieten zum Teil nur unvollständige Umsetzungen des merkmalsorientierten und aspektorientierten Paradigmas.

Für C++ stehen bislang C++ Mixin Layers, P++ und AspectC++ zur Verfügung. Damit können im Hinblick auf Produktlinien weder heterogene noch homogene Crosscuts

	1a) Heterogene Crosscuts	1b) Homogene Crosscuts	1c) Crosscuts zw. Merkmalen	2. Merkmalszusammenhalt	3. Konstruktorproblem	4. Self Problem	5. Feature Optionality	6. Erweiterbarkeit	7. Entwurfsregeln	8a) Programmierung	8b) Konfiguration	8c) Debugging
C++ Mixin Layers	+	-	+/-	+	+/-	+/-	-	+	-	-	+/-	+/-
P++	+	-	+/-	+	-	+/-	-	+	+/-	-	+/-	-
Java Layers	+	-	+/-	+	+	+	-	+	+/-	+/-	+/-	-
Delegation Layers	+	-	+/-	+	-	+	-	+	-	+	+/-	-
AHEAD (ATS)	+	-	+/-	+	+	+	-	+	+	+	+	-
AspectC++	-	+	+/-	-	+	+/-	+/-	-	-	+/-	-	-
Hyper/J	+	+/-	+	+/-	-	-	+	+/-	-	+/-	+/-	-
Jiazzi	+/-	+/-	+/-	+	-	-	+/-	+/-	+/-	+/-	+/-	-
Caesar	+	+	+/-	+	-	-	+/-	+/-	+/-	+/-	-	-
Aspectual Collaborations	+	+	+/-	+	-	-	+	+/-	+/-	+/-	+/-	-

Tabelle 3.1: Vergleich verschiedener Ansätze zur Entwicklung von Produktlinien

hinreichend behandelt werden. Die Umsetzung des Kollaborationenentwurfs ist mit C++ Mixin Layers und P++ nur bedingt realisierbar und praktisch nicht anwendbar. Die Anwendung von AspectC++ auf C++ Mixin Layers und P++ ist aus technischen Gründen nicht möglich. Eine effektive Entwicklung von Produktlinien mit C++ ist daher derzeit nicht realisierbar.

Kapitel 4

FeatureC++

In Abschnitt 3.2 wurde bereits die Notwendigkeit angesprochen, die Entwicklung von Produktlinien auch mit C++ zu ermöglichen. Im letzten Kapitel wurde gezeigt, dass derzeit keine Umsetzung des Kollaborationenentwurfs für C++ existiert, mit dem eine effektive Entwicklung von Produktlinien möglich ist. Mit AspectC++ ist eine AOP Implementierung vorhanden, deren Ziel hauptsächlich die Behandlung homogener Crosscuts ist. Die Realisierung heterogener Crosscuts fällt mit AOP aufwendiger und fehleranfälliger aus als mit FOP. Außerdem ist die AOP in Bezug auf Erweiterung und Wartbarkeit der FOP unterlegen.

Dieses Kapitel stellt FeatureC++, eine Erweiterung der Programmiersprache C++ vor. Mit ihr werden Ansätze der FOP und AOP vereint, um sowohl heterogene als auch homogene Crosscuts behandeln zu können. Dabei werden Vorteile beider Ansätze ausgenutzt und eine effektive Entwicklung von Produktlinien ermöglicht. Es wird außerdem gezeigt, dass FeatureC++ weitere der im Abschnitt 3.5 genannten Probleme löst.

4.1 Sprachentwurf

Unter den existierenden Implementierungen der FOP ist die Umsetzung von AHEAD in der ATS von besonderem Interesse. Sie ermöglicht eine einfache Programmierung sowie eine einfache externe Konfiguration von Software. Verfeinerungen liegen strukturiert in einzelnen Dateien vor, der Merkmalszusammenhalt bleibt aber dennoch erhalten. Mit der DRC-Anwendung der ATS ist außerdem die Überprüfung eines Entwurfs möglich.

Auf Grund dieser Vorteile gegenüber anderen FOP Implementierungen soll im Folgenden eine Umsetzung des AHEAD Ansatzes für C++ entwickelt werden. Eine ausreichende Behandlung homogener Crosscuts ist aber auch mit der ATS nicht möglich (siehe Abschnitt 3.3.1). Da sich aspektorientierte Ansätze besser eignen, wird eine Erweiterung des Ansatzes zur Verwendung aspektorientierter Konzepte untersucht.

4.1.1 FOP in C++

Bei der Umsetzung des Kollaborationenentwurfs muss die Dekomposition von Software (Separation of Concerns) und die spätere Komposition der einzelnen Elemente betrachtet werden. In AHEAD erfolgt die Dekomposition in einzelne Merkmale unter Verwendung von Schichten. Sie entsprechen den Kollaborationen und bilden die Grundlage für den Zusammenhalt der Merkmale.

Organisation der Implementierungselemente

Die ATS bildet die Schichten des Entwurfs auf Ordner des Dateisystems ab. Die Zerlegung der Klassen in Rollen wird durch Verfeinerungen wiedergegeben, die in Dateien im Ordner der jeweiligen Schicht abgelegt werden. Eine Zuordnung von Merkmalen als Teile anderer Merkmale, und damit eine Definition von Teilmerkmalen, ist über eine Verzeichnishierarchie ebenfalls möglich.

Bei der Komposition von Software ist die Reihenfolge der Anwendung der Verfeinerungen wichtig. Diese wird in der ATS durch die Angabe der verwendeten Schichten in den `.equation`-Dateien bestimmt. Zum Ausschluss ungültiger Kombinationen werden Entwurfsregeln definiert, die während der Zusammenstellung überprüft werden. Für die Erweiterung von C++ werden folgende Eigenschaften übernommen:

- Ablegen der Implementierungsdateien in Ordnerstrukturen entsprechend den Merkmalen bzw. Schichten,
- Konfiguration über `.equation`-Dateien und
- Entwurfsregeln zur automatisierten Überprüfung einer Konfiguration unter Verwendung des DRC-Moduls der AHEAD.

Syntax der Spracherweiterung

Neben diesen technischen Anforderungen müssen zur Umsetzung der Lösung Erweiterungen der Programmiersprache C++ vorgenommen werden. Die Nähe der Spracherweiterung zur AHEAD Umsetzung soll erhalten bleiben. Dementsprechend ergeben sich folgende Erweiterungen für C++:

- **Konstanten** werden wie in AHEAD als gewöhnliche Klassen definiert (siehe Abbildung 4.1, Zeilen 4 und 5).
- **Verfeinerungen** werden durch das Schlüsselwort `refines` gekennzeichnet (Zeile 15). Zur Bezeichnung wird der Name der Konstanten verwendet, im angegebenen

```
1  /** Layer Base **/
2
3  //Liste mit Knoten
4  class Node { .. };
5  class List {
6    void Add(Node* node) {
7      //Knote zur Liste hinzufügen
8      ...
9    }
10 };
11
12 /** Layer Sync **/
13
14 //Synchronisation der Liste
15 refines class List {
16   SyncObject sync;
17
18   //Synchronisiertes hinzufügen
19   //eines Knotens
20   void Add(Node* node) {
21     LockObject lock(sync);
22     super::Add(node);
23   }
24 };
```

Abbildung 4.1: Verfeinerungen in FeatureC++

```
1  /** Layer Sort **/
2
3  class SortedList : public List {
4    //Implementierung einer Sortierung
5    ...
6  }
7
8  /** Layer Serialize **/
9
10 //Interface zur Serialisierung
11 class Serializable {
12   virtual void Serialize(Stream& s) = 0;
13 }
14
15 //Erweiterung der Liste mit Serialisierung
16 refines class List : public Serializable {
17   virtual void Serialize(Stream& s) {
18     //Liste in Stream s schreiben
19     ...
20   }
21 };
```

Abbildung 4.2: Vererbung in FeatureC++

Beispiel also `List`. Verfeinerungen sind auch in Kombination mit dem Schlüsselwort `struct` möglich. Wie auch in C++ sind in diesem Fall alle Member per default öffentlich.

- Das **Verfeinern von Methoden** erfolgt durch das Überschreiben dieser. Der Zugriff auf Methoden verfeinerter Klassen erfolgt mit dem Schlüsselwort `super`. Der qualifizierte Name wird durch die Verwendung von `"::"` gebildet (Zeile 22).

- **Vererbung und Mehrfachvererbung** wird sowohl für Konstanten als auch für Verfeinerungen unterstützt (Zeilen 3 und 16 in Abbildung 4.2). Diese können verwendet werden, um in einer Verfeinerung z. B. Schnittstellen zu implementieren (Zeile 16). Die Syntax entspricht der in C++ üblichen. Wird bei der Vererbung eine Klasse der gleichen Bibliothek verwendet, so wird immer von der am weitesten verfeinerten Variante geerbt. In Zeile 3 der Abbildung 4.2 wird demnach von der bereits verfeinerten Liste geerbt, die z. B. bereits das Merkmal Synchronisation enthält.

- **Konstruktoren** müssen in Verfeinerungen nicht neu definiert werden. Die Initialisierung verfeinerter Klassen erfolgt mit dem Schlüsselwort `super`. Die Syntax

entspricht der Initialisierung von Basisklassen der OOP.

- **Statische Member von Klassen** können wie in der OOP definiert werden. Die Initialisierung erfolgt außerhalb der Definition der Verfeinerung, aber in derselben Datei.

Da sich C++ in einigen Bereichen von Java unterscheidet, sind C++ spezifische Erweiterungen hilfreich bzw. notwendig:

```
1  /** Layer Type **/
2
3  //Typisierter Knoten: Typ des
4  //Elements als Templateparameter
5  template <typename ELEMENT>
6  class TypedNode : public Node {
7    TypedNode(const ELEMENT& e)
8      :data(e) {}
9    ELEMET data;
10 };
11
12 //Typisierte Liste: Typ des
13 //Elements als Templateparameter
14 template <typename ELEMENT>
15 refines class List {
16   //Element hinzufügen
17   void Add(const ELEMENT& e) {
18     //neuen Knoten erzeugen und
19     //der Liste hinzufügen
20     super::Add(new TypedNode<ELEMENT>(e));
21   }
22 };
```

Abbildung 4.3: Templates in FeatureC++

```
1  /** Layer Sync **/
2
3  //Definition wie bisher OHNE
4  //Template Parameter
5  refines class List {
6    ...
7    //Verwendung des Templates
8    void Add(Node* node) {
9      LockObject lock(sync);
10     super::Add(node);
11   }
12 };
```

Abbildung 4.4: Verfeinerung von Templates in FeatureC++

- **C++ Templates** bieten die Möglichkeit, auf einfache Weise von bestimmten Datentypen zu abstrahieren. Um diese Funktionalität nicht zu verlieren, ist die Definition von Templates möglich. Werden in einer Verfeinerung Templateparameter eingeführt, so werden folgende Verfeinerungen ebenfalls zu Templates. Die Definition einer solchen Verfeinerung erfolgt wie in C++ üblich (siehe Abbildung 4.3, Zeilen 5 und 14). Die Templateparameter können in folgenden Verfeinerungen verwendet werden. Ihre Neudefinition ist nicht erlaubt (siehe Abbildung 4.4). Die Einführung weiterer Templateparameter in den folgenden Verfeinerungen ist möglich, wird aber bei der Implementierung des Prototyps nicht unterstützt.

- **Globale Methoden** werden weiterhin unterstützt, da sie häufig bei der Implementierung binärer Operatoren Verwendung finden. Die Integration dieser entspricht

der Erweiterung der FOP auf prozedurale Quelltextelemente. Verfeinerungen globaler Methoden werden vorerst nicht unterstützt.

- Die **Verwendung von Includes** ist nicht ohne weiteres möglich bzw. sinnvoll: Zu verwendende Dateien, in denen sich Definitionen anderer Klassen der gleichen Bibliothek befinden, sind über einen zum Zeitpunkt der Implementierung nicht bekannten Pfad erreichbar. Alle Klassen der eigenen Bibliothek können daher ohne die Angabe von Includes verwendet werden. Dateien aus anderen Bibliotheken können nach wie vor über die Präprozessordirektive `#include` eingebunden werden.

Neben diesen allgemeinen Erweiterungen der OOP soll die Implementierung strengeren Regeln folgen. Dies trägt zur Übersichtlichkeit der Implementierung und Ordnerstruktur bei und ermöglicht eine Zuordnung von Verfeinerungen bereits auf Dateiebene. Daher müssen:

- Dateien zur Definition genau einer Klasse bzw. Verfeinerung verwendet werden,
- der Name einer Datei dem Namen der enthaltenen Klasse entsprechen,
- Deklaration und Definition von Methoden in einer Datei erfolgen,
- Templateparameter bei Methodendefinitionen weggelassen werden.

4.1.2 Behandlung Homogener Crosscuts

Die vorgestellten Erweiterungen von C++ erlauben es, heterogene Crosscuts zu implementieren. Häufig sind homogene Crosscuts vorhanden, deren Implementierung mit herkömmlicher FOP ein großes Maß an Redundanz zur Folge hat. Dadurch erschwert sich die Wartung, da der über viele Klassen und Methoden verteilte Programmcode bei einem Fehler korrigiert werden muss. Aus diesem Grund wird eine Erweiterung des Konzeptes untersucht.

Ein wichtiger Punkt bei der Implementierung von crosscutting Concerns ist der Merkmalszusammenhalt (Feature Cohesion), also die Zusammenfassung von Implementierungselementen, die ein Merkmal ausmachen. In Verbindung mit einem entsprechenden Konfigurationsprozess lässt sich dadurch ein Merkmal leicht zu einer Software hinzufügen oder von ihr entfernen. Außerdem wird das Verständnis des Programmcodes erleichtert. Aspektorientierte Elemente sollten daher ebenfalls in die Schichten des FOP Ansatzes integriert werden.

Multi Mixins

Zur Verhinderung von Redundanz bei der Implementierung von Verfeinerungen, genügt bereits eine Zusammenfassung gleichen Quellcodes. Wie auch bei den Pointcuts der AOP können daher Suchausdrücke verwendet werden, um mehrere Klassen oder Methoden zu definieren, auf die eine Verfeinerung angewandt werden soll. Diese auf Suchausdrücken basierenden *Multi Mixins* [ALRS05b] bilden eine einfache Möglichkeit zur besseren Behandlung homogener Crosscuts. Die Umsetzung erfolgt analog den Suchausdrücken der AOP:

- Es können Suchausdrücke für Namen zu verfeinernder Klassen verwendet werden. In Abbildung 4.5 wird mit dem Ausdruck `%List` eine Verfeinerung aller Klassen ermöglicht, deren Namen mit „`List`“ enden. Im Beispiel aus Abbildung 4.2 sind demnach die Klassen `List` und `SortedList` von der Verfeinerung betroffen.
- Eine weitere Möglichkeit zur Verwendung solcher Suchausdrücke besteht bei der Verfeinerung von Methoden: Es können mehrere Methoden einer Klasse verfeinert werden. Der Suchausdruck in Abbildung 4.6 verfeinert alle Methoden der Klasse `List`. Eine einfache Verwendung von Wildcards wie bei Klassennamen ist nicht ohne weiteres möglich, da auf die verfeinerten Methoden zugegriffen werden muss. Wie im Beispiel angedeutet, besteht die Möglichkeit der Verwendung weiterer AOP spezifischer Schlüsselwörter wie `before`, `after`, `around`, etc., oder der Verwendung AOP ähnlicher Konstrukte zum Zugriff auf Parameter eines Funktionsaufrufs über Schlüsselwörter der AOP wie `args`.

```
1  /** Layer Logging **/
2
3  //Verfeinerung aller Klassen,
4  //die auf "List" enden
5  refines class %List {
6
7    //Verfeinerung der Methode "Add"
8    void Add(Node* node){
9      super::Add(node);
10
11     //Logging
12     cout << "Added node: "
13          << node->ToString();
14   }
15 };
```

Abbildung 4.5: Wildcards für Klassen in Multi Mixins

```
1  /** Layer Sync **/
2
3  //Synchronisation der Klasse List
4  refines class List {
5    SyncObject sync;
6
7    //Verfeinerung aller Methoden
8    % %(...) : before() {
9      //Synchronisation
10     LockObject lock(sync);
11   }
12 };
```

Abbildung 4.6: Wildcards für Methoden in Multi Mixins

Eine Kombination der Suchausdrücke für Methoden- und Klassennamen ermöglicht es, Suchausdrücke für beliebige Klassen / Methoden Kombinationen zu erstellen. Des Weiteren besteht die Möglichkeit, durch die Verwendung boolescher Operatoren wie auch in der AOP Suchausdrücke zu kombinieren. Diese Umsetzung bietet klare Vorteile gegenüber reiner AOP:

1. Es werden lediglich Klassen verfeinert, die sich in der Verfeinerungshierarchie weiter oben befinden, es kann also eine Eingrenzung der modifizierten Klassen anhand verwendeter Merkmale getroffen werden. Die Verfeinerungen fügen sich daher sehr elegant in den Kollaborationenentwurf ein.

2. Im Falle von Verfeinerungen einer oder weniger Klassen (durch Auflistung dieser im Suchausdruck) ist eine einfachere Zuordnung der Verfeinerung zur verfeinerten Klasse möglich. Dennoch können auch hier Schreibfehler bei der Definition der Suchausdrücke zu Problemen führen.

Ein Problem entsteht durch die Verwendung von Suchausdrücken mit variablen Argumentlisten. Lohmann et al. zeigen, dass aber auch hier ein typsicherer Zugriff auf die verwendeten Argumente möglich ist [LBS04].

Aspectual Mixin Layers

Mit AOP lassen sich homogene Crosscuts sehr gut behandeln. Ihr größter Nachteil ist der fehlende Merkmalszusammenhalt. In Abschnitt 3.4 wurden *Caesar* und *Aspectual Collaborations* vorgestellt, die aspektorientierte Elemente mit Merkmalsorientiertem Programmcode verbinden. Dabei werden alle zu einem Merkmal gehörenden Implementierungselemente (Klassen und Aspekte) in der Kollaboration zusammengefasst. Auf diese Weise werden sowohl heterogene als auch homogene crosscuts angemessen implementiert. Dieser Ansatz lässt sich auf FeatureC++ übertragen: Durch die Positionierung von Aspekten innerhalb der Schichten werden diese mit Klassen gruppiert und auf einfache Weise konfiguriert.

Ein Kollaborationendiagramm dieser *Aspectual Mixin Layers (AML)*[ALRS05b] ist in Abbildung 4.7 dargestellt. Der Layer `Sync` implementiert Synchronisation unter Verwendung der Klassen `SyncObject` und `LockObject` und des Aspektes `SyncAspect`. In den Abbildungen 4.8 und 4.9 ist die zugehörige Implementierung des Layers `Sync` dargestellt. Die Klasse `SyncObject` (siehe Abbildung 4.8) implementiert ein Objekt zur Synchronisation, dass in Kombination mit der Klasse `LockObject` verwendet wird (der Konstruktor sperrt das Synchronisationsobjekt – Zeile 14 der Abbildung). Der Aspekt führt das Synchronisationsobjekt `_sync` in die Klasse `List` ein (Zeile 7). Der Pointcut `lock` sperrt in

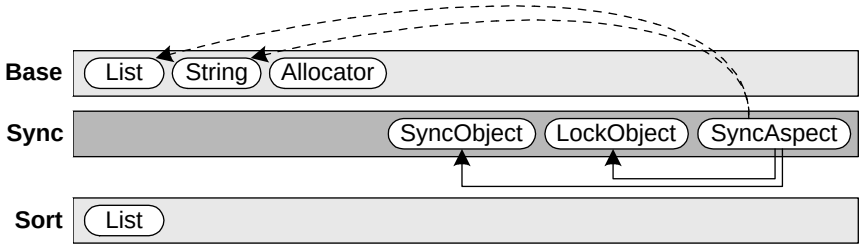

Abbildung 4.7: Kollaborationendiagramm von AML

```
1  /** Layer Sync **/
2
3  //Klasse zur Synchronisation
4  class SyncObject {
5    void lock() { .. }
6  };
7
8  //Klasse zum Sperren
9  //mit Synchronisationsobjekten
10 class LockObject {
11   SyncObject& _sync;
12   LockObject(SyncObject& sync)
13   :_sync(sync) {
14     _sync.lock();
15   }
16 };
```

Abbildung 4.8: AML – Synchronisationsklassen

```
1  /** Layer Sync **/
2
3  //Aspekt zur Synchronisation
4  aspect Synchronize {
5    //Einführen eines Sync. Objektes
6    pointcut list() = classes("List" || "String");
7    advice list() : SyncObject _sync;
8
9    //Pointcut Definition
10   pointcut lock() = execution("% List::%(...)"
11                               || "% List::%(...)");
12
13   //Advice zum Sperren
14   advice lock() : before() {
15     //Synchronisation
16     LockObject lock(tjp->that()->_sync);
17   }
18 };
```

Abbildung 4.9: AML – Synchronisationsaspekt

Verbindung mit dem zugehörigen Advice (Zeilen 14-17) das Synchronisationsobjekt der Liste bei jedem Aufruf einer Funktion der Klasse.

Bei dieser Vorgehensweise nehmen die Aspekte lediglich auf darüber liegende Schichten, bzw. die Schicht ihrer Implementierung Einfluss. Methoden der Klasse `List`, die im Layer `Sort` implementiert werden, sind daher von den Aspekten nicht betroffen.

Da Aspekte lediglich auf die oberen Schichten Einfluss nehmen, wird das Problem der Erweiterbarkeit aspektorientierter Ansätze gelöst: Die Wirkung der Aspekte auf neu implementierte Merkmale wird gezielt durch deren Platzierung innerhalb der Konfiguration gesteuert, ohne Änderungen an Pointcuts vorzunehmen.

Aspectual Mixins

Mit *Aspectual Mixins* werden AOP Konzepte direkt auf Verfeinerungen angewandt [ALRS05b]. So werden innerhalb der Verfeinerungen Pointcuts und Advice-Code definiert. Im Vergleich zu Multi Mixins ist dabei der Zugriff auf andere Klassen möglich. Eine Auswahl der verfeinerten Klassen erfolgt erst durch Definition der Pointcuts. Dies ist hilfreich, wenn Klassen einheitlich verfeinert werden und an anderen Klassen geringe Änderungen vorgenommen werden.

Existierende Klassen sind mit Aspectual Mixins auf Grund der Syntax der FOP Elemente einfach erweiterbar. Bei der Verfeinerung einer Klasse kann so auf die aufwendige Definition von Pointcuts und Advice-Code verzichtet werden. Eine Kombination von Aspectual Mixins und Multi Mixins scheint außerdem möglich. Hierbei könnten mehrere Klassen auf gleiche Weise verfeinert werden und zusätzlich Elemente der AOP Verwendung finden. Die ausführliche Analyse solcher Kombinationen ist jedoch nicht Gegenstand dieser Arbeit.

Auswahl einer Methode

Die drei vorgestellten Varianten zur Integration von AOP Konzepten in die FOP sind bei der Lösung unterschiedlicher Probleme der FOP hilfreich. Für die Verwendung in der prototypischen Umsetzung wird zunächst nur eine der dargestellten Methoden implementiert.

Die umfangreichsten Möglichkeiten bieten Aspectual Mixins und AML. Mit ihrer Hilfe werden alle mit der AOP erreichbaren Effekte erzielt. Während Aspectual Mixins zum Teil eine einfachere Syntax bei der Anwendung bieten, bestehen noch Unklarheiten in Bezug auf Instanziierung der Aspektelemente. Die Implementierung der AML profitiert außerdem von der Verwendung von AspectC++, da sich die Aspekte nur geringfügig von herkömmlichen Aspekten unterscheiden. Für die Entwicklung des Codetransformationssystems werden daher zunächst AML verwendet.

Aspect Refinements

Wie auch Klassen sind Aspekte häufig von mehreren Merkmalen betroffen. Die Anwendung des FOP-Ansatzes (Zerlegung der Aspekte entsprechend der sie betreffenden Merkmale) auf Aspekte ist daher der nächste logische Schritt. Auf diese Weise können Merkmale, die Anlass zur Implementierung mit Aspekten geben, in Teilmerkmale zerlegt werden. Weiterhin ist die Zerlegung der Aspekte bzgl. anderer Merkmale möglich.

Ein Beispiel für die Zerlegung eines Merkmals in Teilmerkmale ist in Abbildung 4.10 dargestellt. Das Merkmal Logging wird in die Teilmerkmale Log und LogDebug

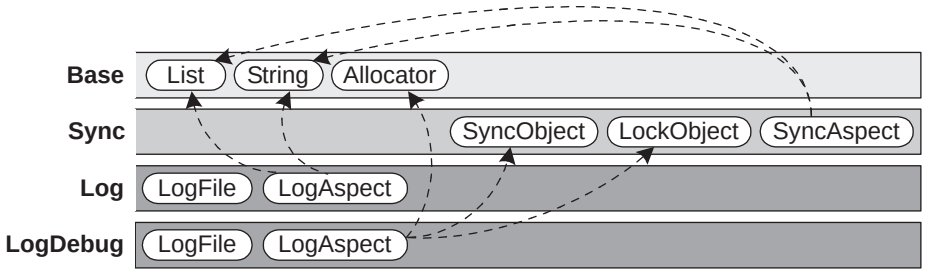

Abbildung 4.10: Kollaborationendiagramm mit Aspektverfeinerung in AML

zerlegt. Die gestrichelten Linien verweisen auf die von Pointcuts betroffenen Klassen. In der Kollaboration LogDebug wird das Logging auf Methoden erweitert, die lediglich für das Debuggen notwendig sind und beispielsweise zu Einschränkungen bezüglich der Leistungsfähigkeit führen.

Für die Umsetzung dieser Lösung ist eine Verfeinerung von Aspekten notwendig, wie sie auch für Klassen zu implementieren ist. Daher muss eine Verwendung der Schlüsselwörter `refines` und `super` analog ihrer Verwendung in Klassenverfeinerungen ermöglicht werden. Mit `super` soll so auch der Zugriff auf Pointcuts des verfeinerten Aspektes möglich sein.

In den Abbildungen 4.11 und 4.12 ist die zugehörige Implementierung der Klasse `LogFile` und des Aspektes `LogAspect` dargestellt. Die Verfeinerung der Klasse `LogFile` implementiert ein erweitertes Format für das Logging. Die Verfeinerung des Aspektes erweitert den Pointcut, so dass dieser weitere Klassen einbezieht. Das Objekt `_logFile` innerhalb des Aspektes (siehe Abbildung 4.12, Zeile 5) ist dabei immer eine Instanz der am weitesten verfeinerten Variante der Klasse `LogFile`. Bei der Verwendung des Layers `DebugLog` wird demnach auch ein Objekt der Klasse `LogFile` dieses Layers erzeugt.

4.2 Codetransformation

Wie im Abschnitt 2.6 erläutert, ist eine Quelltext-Quelltext-Transformation eine einfache Möglichkeit zur Implementierung einer Spracherweiterung. Eine direkte Umsetzung in Maschinencode bietet einige Vorteile, ist im Hinblick auf den Aufwand der Implementierung eines Prototyps nicht geeignet. In diesem Abschnitt wird deshalb die Umsetzung der Spracherweiterung mit Hilfe einer Quelltext-Quelltext-Transformation beschrieben.

Grundlage für die Codetransformation bildet die PUMA Bibliothek (siehe Abschnitt 2.6.4). Da PUMA bereits das Parsen aspektorientierten Quelltextes unterstützt, wird die

```
1  /** Layer Log **/
2
3  //Klasse zum Schreiben von Log-
4  //informationen in eine Datei
5  class LogFile {
6    void log(const char*) { .. }
7  };
8
9  /** Layer LogDebug **/
10
11 //Verfeinerung mit geändertem
12 //Format der Ausgaben
13 refines class LogFile {
14   void setLogFormat() { .. }
15   LogFile() {
16     setLogFormat();
17   }
18 };
```

Abbildung 4.11: Verfeinerung einer Klasse in FeatureC++

```
1  /** Layer Log **/
2
3  //Aspekt zum Logging
4  aspect LoggingAspect {
5    LogFile _logFile;
6
7    //Pointcut Definition
8    pointcut log() = call("% List::%(...)"
9                       || "% String::%(...)");
10
11   //Advice zur Logausgabe
12   advice log() : before() {
13     //Logging der Signatur
14     _logFile.log(tjp->signature());
15   }
16 };
17
18 /** Layer LogDebug **/
19
20 //Verfeinerung des Aspekts
21 refines aspect LoggingAspect {
22   //Pointcut erweitern
23   pointcut log() = call("% Allocator::%(...)"
24                       || "% SyncObject::%(...)")
25                       || "% LockObject::%(...)")
26                       || super::log());
27 };
```

Abbildung 4.12: Verfeinerung von Aspekten in FeatureC++

Bibliothek auch für die aspektorientierten Elemente der Spracherweiterung verwendet.

4.2.1 Auswahl einer geeigneten Transformation

Mit der ATS besteht die Möglichkeit merkmalsorientierten Quelltext einer Klasse in eine Klassenhierarchie oder eine einzelne Klasse (*Jampack*) umzuwandeln. Für die Umwandlung in eine Klassenhierarchie wird in der ATS der Name einer erbenden Klasse aus dem eigentlichen Namen der Klasse und dem Namen der Schicht gebildet (vgl. Abbildung 2.25).

Beide Möglichkeiten der Transformation (Jampack und Klassenhierarchie) haben Vor- und Nachteile. So sind mit einer Klassenhierarchie auch im transformierten Code die ursprünglichen Strukturen leicht auffindbar, was beim Debuggen hilfreich ist. Probleme entstehen, wenn in Funktionen der `this`-Zeiger verwendet wird. In diesem Fall ist eine Typumwandlung in den finalen Typen notwendig. Abbildung 4.13 stellt dies anhand der Klasse `String` dar. Die Verfeinerung (Zeile 15) implementiert eine Methode zum Anfügen von Zahlen an einen String. Abbildung 4.14 zeigt die Transformation der Verfeinerungen

```
1  /* Layer Base */
2  class String {
3    //Anfügen eines anderen Strings
4    String& operator <<
5              (const char* other) {
6
7      ... //other hinzufügen
8
9      //Zurückgeben des Objektes selbst
10     return *this;
11   }
12 };
13
14 /* Layer Numbers */
15 refines class String {
16   //Anfügen eines Integers zum String
17   String& operator << (int i) {
18
19     ... //int i hinzufügen
20
21     //Zurückgeben des Objektes selbst
22     return *this;
23   }
24 };
```

Abbildung 4.13: Konstante mit Verfeinerung in FeatureC++

```
1  /* Transformiert aus Layer Base */
2  class String_Base {
3    //Anfügen eines anderen Strings
4    String& operator <<
5              (const char* other) {
6
7      ... //other hinzufügen
8
9      //Zurückgeben des Objektes selbst
10     return *((String*)this);
11   }
12 };
13
14 /* Transformiert aus Layer Numbers */
15 class String : public String_Base {
16   //Anfügen eines Integers zum String
17   String& operator << (int i) {
18
19     ... //int i hinzufügen
20
21     //Zurückgeben des Objektes selbst
22     return *this;
23   }
24 };
25
26 //Verwendung der Klasse "String"
27 String s;
28 s << "Hallo␣" << 47;
```

Abbildung 4.14: Transformation in eine Klassenhierarchie

in eine Klassenhierarchie. In Zeile 10 ist eine Typumwandlung der Klasse `String_Base` notwendig, um die Klasse `String` zu erhalten. Nur so ist eine Hintereinanderausführung des `<<`-Operators möglich (siehe Abbildung 4.14, Zeile 28).

Die Verwendung einer solchen Typumwandlung ist grundsätzlich kein Problem, da lediglich Instanzen der finalen Klasse erstellt werden. Die Typumwandlung ist daher semantisch immer korrekt. In der ATS wird dies durch die Verwendung des Schlüsselwortes `abstract` bei der Definition der Klasse sichergestellt. Dies ist unter C++ nicht möglich. Aushilfsweise kann eine rein virtuelle Methode verwendet werden. Diese wird erst in der finalen Klasse implementiert, so dass lediglich Instanzen dieser Klassen erstellt werden können. Da dies kein grundsätzliches Problem darstellt, wird zunächst darauf verzichtet.

Die Typumwandlung des `this`-Zeigers bereitet keinerlei Probleme, wenn dieser in einer Methode als Rückgabewert verwendet wird. Wird er hingegen für den Zugriff auf das Objekt verwendet, ist dies nur möglich, wenn die Definition der Methode nach der Definition der finalen Klasse der Hierarchie erfolgt. Andernfalls ist kein Zugriff möglich. Weiterhin ist durch diese Vorgehensweise der Zugriff auf private Member bei der Ver-

wendung des `this`-Zeigers nicht mehr möglich.

Für die Codeumwandlung unter C++ bietet sich außer den zwei genannten Transformationen eine weitere: Die Umwandlung in C++ Mixin Layers. Auf diese Weise ist auch ohne die Verwendung der Spracherweiterung eine Konfiguration mit dem transformierten Quelltext möglich. Es ergeben sich aber Probleme, wie sie bereits in 3.2.1 beschrieben wurden. Sie führen zur Einschränkung der Möglichkeiten der Spracherweiterung.

Da die Spracherweiterung auch aspektorientierte Ansätze enthält, ist eine einfache Implementierung dieser notwendig. Das Parsen aspektorientierter Quelltextelemente ist durch die Verwendung von PUMA möglich. Hier erfolgt eine Erweiterung im Hinblick auf Aspektverfeinerungen. Rein merkmalsorientierte Quelltextteile werden in objektorientierten Quelltext umgewandelt. Daher liegt es nahe, aspektorientierte Elemente in AOP Code umzuwandeln. Dieser kann durch Verwendung von AspectC++ auf den transformierten objektorientierten Code angewandt werden.

Für die Verwendung von Aspekten innerhalb der Schichten ergibt sich ein Vorteil bei der Verwendung einer Klassenhierarchie. Eine Steuerung der Wirkung der Aspekte auf einzelne Schichten erfolgt durch Verwendung der transformierten Klassennamen. Hierzu ist eine Umwandlung der Pointcuts in den Aspekten notwendig (vgl. Abschnitte 4.1.2 und 4.2.2). Die Verwendung von Jampack lässt dies nicht auf so einfache Weise zu. Mixin Layers sind an dieser Stelle ebenfalls auszuschließen, da AspectC++ derzeit keine Pointcuts in Template-Code unterstützt.

Die prototypische Umsetzung der Spracherweiterung erfolgt daher durch die Transformation in eine Klassenhierarchie. Aspektverfeinerungen werden in eine Vererbungshierarchie von Aspekten transformiert (vgl. Abschnitt 2.4).

4.2.2 Transformation in eine Klassenhierarchie

Alle für die Codetransformation notwendigen Schritte werden in diesem Abschnitt dargestellt. Dabei wird zunächst die Transformation merkmalsorientierten Quelltextes und anschließend die Umwandlung aspektorientierter Elemente besprochen.

Merkmalsorientierter Quelltext

Die Transformation des FOP Codes soll analog der Codetransformation der ATS erfolgen. Einige Besonderheiten müssen dabei auf Grund der Unterschiede der Spracherweiterungen, sowie der Eigenschaften von C++ berücksichtigt werden. Die folgende Übersicht zeigt die notwendigen Transformationen.

Klassen und Verfeinerungen Klassen und deren Verfeinerungen werden wie auch in der ATS in eine Klassenhierarchie transformiert. Structs und deren Verfeinerungen werden analog behandelt. Die Bezeichnung der erzeugten Klassen erfolgt mit dem Namen der jeweiligen Klasse und dem Namen der Schicht aus dem die Klasse oder Verfeinerung stammt. Als Trennung zwischen beiden Elementen wird ein Unterstrich (`"_"`) verwendet. Die Auswahl des Zeichens ist willkürlich, jedoch wird der Unterstrich häufig zur Trennung zweier Wörter in einem Namen verwendet und bereitet keinerlei Probleme in Verbindung mit AspectC++. Bei der am weitesten verfeinerten Klasse (der finalen Klasse) entfällt der Name der Schicht. Es sind C++ typische Vererbungen zu erzeugen. Hierbei wird öffentliche Vererbung verwendet[1]. Sind Verfeinerungen vorhanden, ohne dass eine zu verfeinernde Basisklasse existiert, werden die Verfeinerungen nicht transformiert und eine Warnung ausgegeben. Dies ist notwendig, um dem Feature Optionality Problem zu begegnen (siehe Abschnitte 3.3.1 und 4.5.2).

This-Zeiger Die Verwendung des `this`-Zeigers bezieht sich immer auf den Typen der finalen Klasse. Hierzu ist es notwendig, für sämtliche Verwendungen des `this`-Zeigers eine Typumwandlung in einen Zeiger auf den Typen der Klasse vorzunehmen (vgl. Abschnitt 4.2.1 und Abbildung 4.13).

Mehrfachvererbung Zur Unterstützung der Mehrfachvererbung sind an Verfeinerungen angegebene Basisklassen in die Vererbung einzubeziehen. Da immer von der am weitesten verfeinerten Variante einer Klasse geerbt wird, ist der Namen der finalen Klasse zu verwenden.

Zugriffssteuerung In verfeinerten Methoden muss der Zugriff auf die Member der Basisklasse ermöglicht werden. Hierzu ist eine Transformation des Schlüsselwortes `super` in den Namen der Basisklasse notwendig.

Überladene Methoden Methoden, die durch das Überladen in Verfeinerungen nicht mehr sichtbar sind (*Method Hiding*, vgl. [Str86]) werden durch Propagieren[2] der Methoden sichtbar gemacht. Der für OOP zum Teil hilfreiche Mechanismus des Method Hiding findet hier keine Anwendung, da es sich nicht um eine Vererbungshierarchie im eigentlichen Sinne handelt. Vielmehr liegt eine Zerlegung einer Klasse in eine Vererbungshierarchie vor. Aus diesem Grund ist es in den seltensten Fällen sinnvoll, Methoden durch

[1] Nur durch öffentlichen Zugriff ist die Verwendung öffentlicher Methoden aller Verfeinerungen möglich.

[2] Unter *Propagieren von Methoden* wird das Erzeugen einer Methode in einer Subklasse verstanden. In dieser erfolgt dann lediglich der Aufruf der Methode der Basisklasse.

Methoden gleichen Namens nicht zu propagieren. Sollte das „Verstecken“ solcher Methoden dennoch notwendig sein, ist dies dem Anwender möglich, in dem er die Methode in der Verfeinerung mit dem Zugriffsrecht `private` überschreibt.

Konstruktoren Konstruktoren in Verfeinerungen müssen nicht neu definiert werden. Daher werden auch die Konstruktoren der Basisklasse propagiert. Hierzu muss die Initialisierung der Basisklasse mit dem propagierten Konstruktor erfolgen. Werden Konstruktoren verfeinert oder hinzugefügt, wird weiterhin die Initialisierung der Basisklasse mit dem Schlüsselwort `super` berücksichtigt. Es erfolgt die Initialisierung wie in C++ üblich mit dem Namen der Basisklasse.

Propagieren von Methoden Beim Propagieren von Methoden (sowohl Konstruktoren, als auch einfache Methoden) müssen die Zugriffsrechte beachtet werden. Diese werden entsprechend der Zugriffsrechte der Basisklasse propagiert. Dabei ist auf die unterschiedlichen Zugriffsrechte von Klassen und Structs zu achten.

Templates Für die Verwendung von Templates ist es notwendig, die Definition der Template-Parameter für Template-Verfeinerungen zu generieren. Dies betrifft sowohl die Definition der Parameter vor den erzeugten Klassen (inklusive Standardparameter), als auch vor den Methodendefinitionen der Klassen. Im letzteren Fall ist darauf zu achten, dass Standardparameter nicht verwendet werden dürfen.

Statische Member Initialisierungen statischer Member müssen in den C++ Implementierungsdateien erfolgen. Gleiches gilt für die Definitionen von Funktionen und Variablen im globalen Namensraum. Hier ist zusätzlich eine Deklaration in der Headerdatei zu generieren.

Ausgabe Die erzeugten Klassen werden wie für C++ üblich in Headerdateien und Implementierungsdateien getrennt. Methoden außerhalb von Klassen oder Verfeinerungen werden dabei in die C++ Implementierungsdateien ausgegeben. Ausgenommen hiervon sind Methoden, die als `inline` deklariert sind. In jeder erzeugten Headerdatei wird eine Forwarddeklaration der finalen Klasse generiert. Auf diese Weise ist es möglich den Typen der Finalen Klasse bereits vor der eigentlichen Deklaration zu verwenden. Um wiederholtes Parsen der Header durch den Compiler zu verhindern, werden die für C++ üblichen Präprozessordirektiven `"#ifndef .. #define .. #endif"` generiert. Diese umschließen den gesamten Quelltext der Headerdatei.

Die Deklarationen der Klassen benötigen zum Teil Typen anderer Klassen des Projektes. In einigen Fällen ist hier eine Forwarddeklaration des benötigten Typen ausreichend,

in anderen Fällen ist ein Include der Headerdatei in der die andere Klasse definiert ist notwendig. Hierzu wird für jede Headerdatei festgestellt, ob eine Forwarddeklaration verwendeter Typen ausreicht. In diesem Fall werden entsprechende Forwarddeklarationen erzeugt. Ist dies nicht der Fall, müssen Includes der entsprechenden Headerdateien generiert werden. Hierbei ist auf zyklische Includes zu achten, da ein herkömmlicher C++ Compiler diese nicht verarbeiten kann[3]. In einem solchen Fall ist eine entsprechende Fehlermeldung mit dem Zyklus auszugeben.

Die gleichzeitige Verwendung unterschiedlich konfigurierter Klassen und Komponenten in einer Anwendung wird ermöglicht. Hierzu werden sämtliche Klassen und Methoden in einem Namensraum gehalten. Der Name ergibt sich aus dem Namen der Konfiguration. Zur Vereinfachung der späteren Verwendung einer erzeugten Komponente oder Bibliothek wird eine weitere Headerdatei erstellt, die alle notwendigen Includes enthält und die Verwendung des Namensraums mittels `using`-Direktive vorsieht.

Aspektorientierte Quelltextelemente

Wie in Abschnitt 4.1.2 besprochen, werden auch Aspekte und deren Verfeinerungen in eine Vererbungshierarchie transformiert. Die Namensgebung der Aspekte erfolgt dabei wie bereits für Klassenhierarchien beschrieben. Gleiches gilt für die Transformation von Methoden. Die Transformation der Pointcuts bedarf genauerer Analyse. Dabei ist die Wirkung der Pointcuts auf einzelne Schichten und die Verwendung des Schlüsselwortes `super` innerhalb der Pointcuts zu beachten.

Die Umwandlung der Pointcuts muss so erfolgen, dass ein transformierter Pointcut auf alle Rollen einer Klasse wirkt, die sich in derselben Schicht wie der Aspekt oder in darüber liegenden Schichten befinden. Für den `call` Pointcut oder den `classes` Pointcut (weitere siehe Tabelle 4.1) ist eine solche Transformation notwendig. Für andere vordefinierte Pointcuts wird dies gesondert betrachtet.

In den Abbildungen 4.15 und 4.16 ist die Transformation für einen `call`-Pointcut dargestellt, wie sie für einen Entwurf analog zu Abbildung 4.10 erfolgt[4]. Bei Verwendung der Layer `Base` und `Log` wirkt daher der Pointcut auf beide Layer. Ein darunter definierter Layer (z. B. `Sync` in Abbildung 4.10) ist von diesem Pointcut nicht betroffen.

Der Pointcut `derived` kann nicht auf diese Weise transformiert werden, da hier al-

[3]Das wiederholte Einbinden der Header wird durch die bereits erwähnten Präprozessordirektiven verhindert. Dennoch kann ein Zyklus nicht verarbeitet werden, da so nicht alle Klassendefinitionen bekannt sind.

[4]Die dargestellte Transformation entspricht nur der theoretisch notwendigen Transformation und wird hier zur Vereinfachung angegeben. Auf Grund technischer Umstände entspricht diese nicht der tatsächlich verwendeten Transformation, die in Abschnitt 4.3.3 genauer beschrieben wird. Für das Verständnis ist aber die dargestellte Transformation ausreichend.

```
1 /* Layer Log */
2 aspect LoggingAspect {
3   //Pointcut Definition
4   pointcut log() = call(
5               "% String::%(...)");
6
7   ..
8 };
```

Abbildung 4.15: Aspekt in FeatureC++

```
1 /* Aus Layer Log generierter Code */
2 aspect LoggingAspect {
3   //Pointcut Definition
4   pointcut log() = call(
5                "% String_Base::%(...)"
6             || "% String_Log::%(...)");
7   ..
8 };
```

Abbildung 4.16: Transformation von Aspekten

Pointcut	Transformation
call construction destruction execution within classes target that	Klassen der Schicht des Aspektes und aller darüber liegende Schichten
base derived	Klasse des Layers
result args	finale Klasse
cflow	keine Transformation notwendig

Tabelle 4.1: Transformation von Pointcuts in FeatureC++

le erbenden Klassen betroffen sind. Es ist daher eine Transformation des Suchausdrucks sinnvoll, der nur auf Klassen der Schicht zutrifft, aus dem der Aspekt stammt. In Verwendung mit AspectC++ würde dieser auf alle erbenden Klassen wirken, was den weiteren Verfeinerungen entspricht. Hier ist aber auch eine Anwendung des Pointcuts auf die finale Klasse möglich, so dass alle Erbenden der finalen Klasse betroffen sind. Dies entspricht der Verwendung im Sinne der OOP. Ähnliches gilt für den Pointcut `base`. Hier ist eine Transformation sowohl in den Namen der am wenigsten verfeinerten Klasse, als auch in den Namen der Klasse der Schicht des Aspektes möglich. Ersteres entspricht der Verwendung wie sie von AOP in Kombination mit objektorientiertem Code bekannt ist. Letzteres hingegen stellt eine gezielte Steuerung der Wirkung innerhalb der Schichten dar.

In Kombination (z. B. `"!base(..) && !derived(..)"`) ergeben sich weitere Möglichkeiten, die je nach Implementierung z. B. nur die Schicht des Aspektes betreffen. Mit der Einführung neuer Pointcuts lassen sich alle genannten Effekte erzielen. Eine genaue Untersuchung der einzelnen Kombinationsmöglichkeiten im Hinblick auf Anwendbarkeit ist daher notwendig, um sinnvolle Transformationen zu ermitteln. Dies kann im Rahmen dieser Arbeit nicht erfolgen und Bedarf intensiver Analyse unter Zuhilfenahme der prototypischen Umsetzung.

Der Pointcut `cflow`, der für eine Analyse des Aufrufkontextes verwendet wird, bedarf keiner Transformation. Er wird lediglich in Zusammenhang mit anderen Pointcuts wie etwa `execution` verwendet.

Ein weiterer Unterschied muss für die Pointcuts `result` und `args` gemacht werden. Diese betreffen Parameter von Funktionsaufrufen, bzw. den Rückgabewert. Werden Klassen der eigenen Bibliothek referenziert, sind die finalen Klassen zu verwenden, da auch im FOP-Code in der Signatur von Funktionen immer die finalen Klassen verwendet werden. Signaturen mit anderen Klassen einer Vererbungshierarchie als Parameter treten grundsätzlich nicht auf.

Tabelle 4.1 fasst alle Pointcuts und ihre Transformationen zusammen. Wie bereits erwähnt, ist für die Pointcuts `derived` und `base` eine genauere Untersuchung notwendig und hier nur die für die prototypische Umsetzung verwendete Transformation angegeben. Für die anderen Pointcuts sind die angegebenen Transformationen plausibel, bedürfen aber ebenfalls genauerer Untersuchungen. Dies erfolgt zum Teil in Abschnitt 4.4.

```
1 //Verfeinerung des Aspekts
2 aspect LoggingAspect : public LoggingAspect_Log {
3   //Pointcut erweitern
4   pointcut log() = call("% Allocator::%(...)"
5                      || "% SyncObject::%(...)"
6                      || "% LockObject::%(...)"
7                      || LoggingAspect_Log::log() );
8 };
```

Abbildung 4.17: Transformation von Aspektverfeinerungen

Die Verwendung von `super` in Pointcuts erfolgt ähnlich der Verwendung in normalen Verfeinerungen (vgl. 4.1.2). Bei der Transformation muss daher das Schlüsselwort `super` ebenfalls in den Namen des Basisaspektes umgewandelt werden. Dementsprechend ergibt sich für eine Verfeinerung des Aspektes aus Abbildung 4.12 eine Transformation wie in Abbildung 4.17 dargestellt[5]. Die Wirkung des überschriebenen Aspektes wird dadurch

[5]Die Transformation entspricht wiederum nicht der tatsächlich verwendeten (siehe Abschnitt 4.3.3), ist aber für das Verständnis ausreichend.

um die neu definierten Punkte im Programmcode erweitert.

Die Definition von Aspekten unter AspectC++ ist auf den globalen Namensraum beschränkt. Es erfolgt demnach keine Integration der Aspekte in den unter 4.2.2 beschriebenen Namensraum. Eine Verwendung des Namensraums mit einer `using`-Direktive ist jedoch notwendig, um die darin enthaltenen Klassen aus dem Advice-Code ansprechen zu können.

4.3 Implementierung

Nachdem die Syntax von FeatureC++ und die benötigte Transformation zur Überführung in objektorientierten Quelltext dargestellt wurde, wird im folgenden Kapitel auf einige Details der Implementierung des Prototypen eingegangen, dessen Grundlage die PUMA Bibliothek (siehe Abschnitt 2.6.4) bildet.

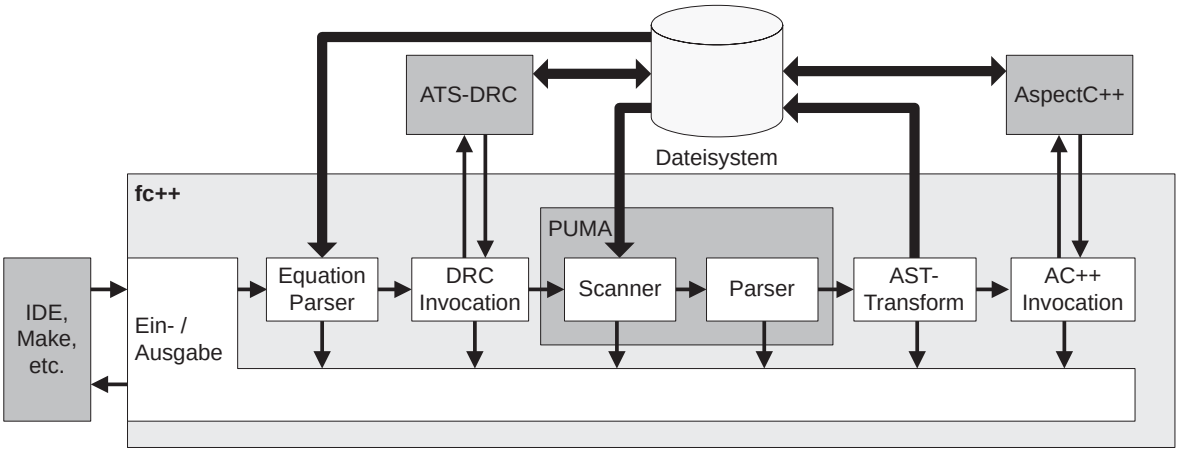

Abbildung 4.18: FeatureC++ – Systemübersicht

Ein Überblick über das Codetransformationssystem und den Transformationsvorgang gibt Abbildung 4.18. Nach dem Auswerten der Startparameter liest der Equation-Parser die `.equation`-Datei ein und ermittelt die für die Konfiguration notwendigen Quelltextdateien. Existierende Entwurfsregeln werden vom Design Rule Check der ATS (ATS-DRC) verwendet, um die Korrektheit der Konfiguration zu überprüfen. Die PUMA-Bibliothek führt daraufhin die lexikalische Analyse und das Parsen des Quelltextes durch. Die erzeugten Syntaxbäume werden in Syntaxbäume transformiert, die objektorientiertem bzw. aspektorientiertem Quelltext entsprechen (AST-Transform). Nach Abschluss der Transformation werden C++ und AspectC++ Ausgaben erzeugt. Eventuell generierter aspektorientierter Programmcode wird mit AspectC++ in den objektorientierten

Code gewebt. Fehlermeldungen der aufgerufenen Prozesse, sowie Ausgaben des Parse- und Transformationsvorgangs werden an den aufrufenden Prozess ausgegeben. Dies erfolgt in der für das Frontend üblichen Schreibweise[6].

Im Folgenden wird zunächst das Einlesen der Eingabe, das Auswerten der Konfiguration und das Ausführen der Design Rule Checks dargestellt. Daraufhin wird die notwendige Anpassung von PUMA zum Parsen von FeatureC++ Quelltext erläutert. Die Transformation der resultierenden Syntaxbäume ist Hauptaufgabe des Codetransformationssystems und wird daher etwas umfangreicher betrachtet. Abschließend wird das Weben des erzeugten AOP-Quelltextes mit Hilfe von AspectC++ dargestellt.

4.3.1 Vorbereiten der Codetransformation

Ausgangspunkt für die Transformation des merkmalsorientierten in objektorientierten Quelltext ist die Konfiguration der zu erstellenden Komponente oder Bibliothek. Diese liegt bei FeatureC++, wie auch in der ATS, in einer `.equation`-Datei vor, die eine Auflistung der zu verwendenden Schichten enthält. Vor der Codetransformation steht daher auch das Einlesen der `.equation`-Datei, die über die Kommandozeile übergeben wird. Treten dabei Fehler auf, werden diese ausgegeben und der Prozess abgebrochen. Die Ordner des Dateisystems, welche die zu verwendenden Quelltextdateien enthalten ergeben sich aus dem Projektverzeichnis und dem Namen der Schicht.

Im nächsten Schritt werden existierende Entwurfsregeln überprüft. Diese finden sich in den Verzeichnissen der jeweiligen Schichten in `.drc`-Dateien. Der Aufbau entspricht den in der ATS verwendeten Dateien und die Überprüfung erfolgt mit der DRC-Anwendung der ATS. Hierzu ist die ATS-Bibliothek, sowie Java-Runtime ab v1.4 notwendig. Der Pfad zur DRC-Anwendung ist in einer Konfigurationsdatei bereitzustellen. Die Überprüfung erfolgt durch Erzeugen eines Java Prozesses, dem die DRC-Anwendung und die zu überprüfenden Dateien als Parameter übergeben werden. Ausgaben des Java-Prozesses werden an die Standardausgabe weitergeleitet, so dass diese für den aufrufenden Prozess sichtbar sind. Tritt hierbei ein Fehler auf, wird ebenfalls die weitere Ausführung abgebrochen.

Ist die Überprüfung der Entwurfsregeln abgeschlossen, kann mit dem Parsen des Programmcodes fortgefahren werden.

[6] Hier wird zwischen GCC und Microsoft Visual Studio unterschieden, um eine Auswertung ausgegebener Quelltextstellen zu ermöglichen.

4.3.2 Parsen der Quellen

Das Parsen des Quelltextes findet schichtweise statt. Beginnend mit der obersten Schicht, dem Basislayer, hin zur am weitesten verfeinerten Schicht werden alle enthaltenen Dateien geparst. Diese Reihenfolge ist notwendig, um bereits während des Parsens die Überprüfung von verwendeten Klassen- und Methodennamen vorzunehmen. In den Vorgang des Parsens einer Quelltextdatei gehen alle aus bereits geparsten Dateien ermittelten Quelltextelemente wie Klassen und Methoden ein. Auf diese Weise kann vorausgesetzt werden, dass alle benötigten Elemente auch ohne die Verwendung von Includes bekannt sind. Weiterhin müssen somit alle Dateien nur einmal geparst werden.

Diese Vorgehensweise bewirkt weiterhin, dass bei einer Verwendung von Klassen derselben Schicht die Dateien, die diese Klassen enthalten, vorher geparst werden. Wurde eine entsprechende Datei noch nicht analysiert, wird das Parsen der aktuellen Quelltextdatei abgebrochen und mit der nächsten Datei fortgefahren. Nach der Analyse aller Dateien einer Schicht wird mit den zuvor ausgelassenen Dateien fortgefahren. Dieser Vorgang wird wiederholt, bis sämtliche Dateien geparst wurden. Kommt es hierbei zu einer Kreuzreferenz von Dateien, wird der Vorgang abgebrochen und die fehlerhaften Dateien ausgegeben. Dies ist z. B. der Fall, wenn zwei Klassen gegenseitig voneinander erben.

Erweiterung von PUMA

Nachdem die generelle Vorgehensweise beim Parsen des Quellcodes vorgestellt wurde, soll nun die notwendige Erweiterung der PUMA Bibliothek zum Parsen des FeatureC++ Quelltext dargestellt werden. Als Basis dient die PUMA Bibliothek v1.0.

Grundlage der Analyse des Quelltextes bilden der Lexer-Generator *Orange* und der Parsergenerator *Lemon*, deren Regeln angepasst werden müssen, um die modifizierte Syntax von FeatureC++ zu verarbeiten. Dies betrifft die Analyse der Schlüsselwörter `refines` und `super`. Hingegen werden AspectC++ spezifische Schlüsselwörter (z.B. `aspect`) bereits von PUMA erkannt. Lemon und Orange erzeugen Quelltext, der Grundlage für die PUMA Bibliothek darstellt. Wie bereits in Abschnitt 2.6.4 erläutert, wird aber nur ein kleiner Teil des PUMA Quelltextes automatisch generiert. Der übrige Quelltext wird auf herkömmliche Weise implementiert, so dass auch dessen Erweiterung per Hand erfolgen muss. Abbildung 4.19 gibt einen Überblick über die notwendigen Erweiterungen.

Ein Präprozessorparser verarbeitet die vom Scanner erzeugte Folge von Zeichen (engl. *token stream*) und expandiert enthaltene Makros. Die Makro-bereinigte Token Folge wird vom C++ bzw. AspectC++ Parser verarbeitet, der einen C++ bzw. AspectC++ Syntaxbaum erzeugt.

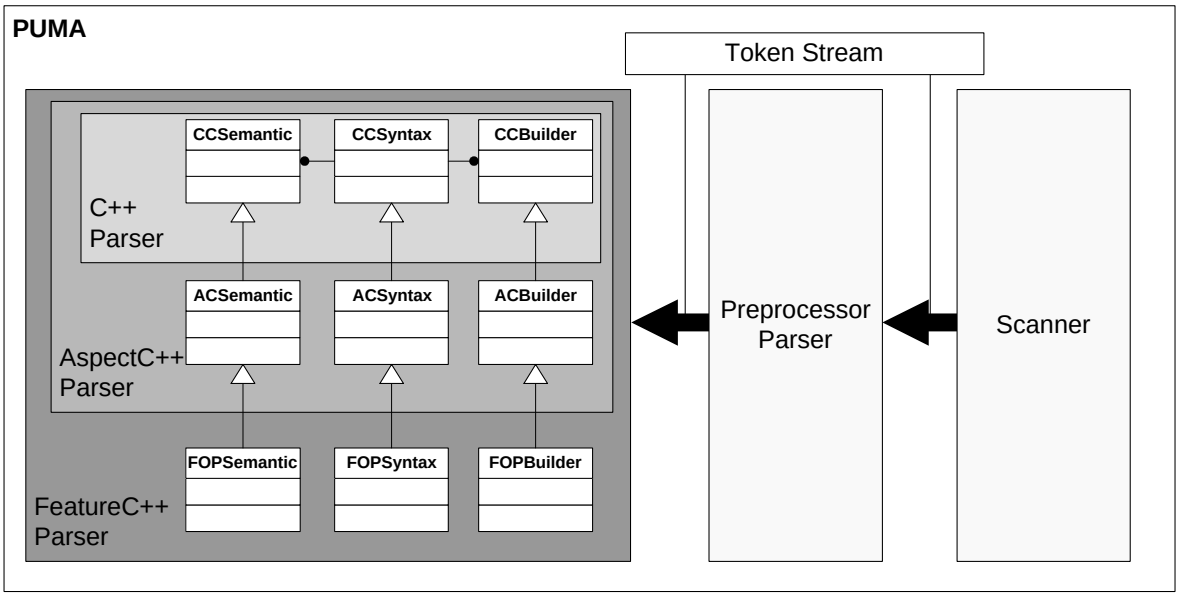

Abbildung 4.19: Erweiterung der PUMA-Bibliothek

Für die Erweiterung der Syntax auf merkmalsorientierten Quelltext ist eine Anpassung des Parsers notwendig. Als Basis wird hierzu der AspectC++ Parser verwendet. Dieser erweitert bereits den C++ Parser und ermöglicht es sowohl AspectC++ als auch C++ Quelltext zu verarbeiten. Im Wesentlichen ist eine Erweiterung der drei Kernkomponenten `ACSyntax`, `ACSemantic` und `ACBuilder` notwendig (vgl. Abbildung 4.19). Die neuen Klassen werden analog bezeichnet und beginnen mit der Zeichenfolge `"FOP"`. Die auf `"Syntax"` endenden Klassen sind für die syntaktische Analyse des Quelltextes zuständig. Die Semantik des analysierten Quelltextes wird von den Klassen `CCSemantic`, `ACSemantic` und `FOPSemantic` überprüft[7]. Bei syntaktischer und semantischer Korrektheit werden von den `"Builder"`-Klassen entsprechende Knoten des Syntaxbaums erzeugt.

Nachdem alle Quellcodedateien verarbeitet wurden, liegt je Datei ein Syntaxbaum vor. Dieser ist seiner Schicht zugeordnet, und kann im Folgenden transformiert werden.

4.3.3 Codetransformation

Bei der Transformation der generierten Syntaxbäume sind folgende Vorgänge zu unterscheiden:

[7] An dieser Stelle findet keine vollständige Überprüfung der Semantik statt. Es wird lediglich eine zur Erstellung des Syntaxbaums notwendige Überprüfung vorgenommen. Eine erweiterte Semantikanalyse des erstellten AST ist nach dem Parsen möglich.

- Die Transformation von Klassen, Aspekten und Verfeinerungen,
- die Umwandlung von Methoden innerhalb und außerhalb von Klassen,
- das Propagieren von Methoden der Basisklasse,
- und das Transformieren von Pointcuts.

Im Folgenden sollen diese Teile gesondert betrachtet werden, da sich die notwendigen Transformationen unterscheiden. Zuvor werden die benötigten Operationen am AST näher betrachtet.

Modifikationen des AST

Grundlegende Operationen an einem Baum - wie es auch der AST ist - sind das Einfügen und Löschen von Knoten. Aus diesen lassen sich zusammengesetzte Operationen wie das Kopieren und Verschieben von Knoten erstellen. Anwendung findet z. B. das Kopieren von Knoten beim Propagieren von Methoden, in dem der Knoten der ursprünglichen Methode dupliziert wird.

Die PUMA-Bibliothek besitzt bereits Möglichkeiten solche Transformationen durchzuführen. Diese sind jedoch nicht ausreichend, da hierbei kein neuer Syntaxbaum erzeugt wird, sondern lediglich die zugrunde liegende Folge von Token bearbeitet wird. Eine weitere Modifikation eines kopierten Teils auf der Ebene des AST ist damit erst nach erneutem Parsen der kopierten Zeichenfolge möglich. Dies ist nur bei einer syntaktisch und zumindest zum Teil semantisch korrekten Operation möglich. Da sich z. B. ein kopierter Bereich in einer neuen Umgebung befindet, ist unter Umständen weder semantische noch syntaktische Korrektheit gegeben. Um einen korrekten Syntaxbaum zu erhalten, wäre demnach eine weitere Modifikation auf Ebene der Token oder eine Anpassung des Parse-Vorgangs notwendig.

Neben den Knoten des AST finden sich in PUMA weitere Klassen, die einen Zugriff auf Quelltextelemente auf einer höheren Abstraktionsebene ermöglichen. So ist z. B. der Zugriff auf Funktionen und deren Parameter möglich, ohne durch den Syntaxbaum traversieren zu müssen. Diese Objekte sind nach Operationen auf dem AST durch PUMA ebenfalls nicht vorhanden, aber wesentlich für Transformationen folgender Schichten.

Aus diesen Gründen wurde der Syntaxbaum so erweitert, dass echte Kopieroperationen des AST möglich sind. Hierbei werden Knoten des Baums rekursiv kopiert und neue Knoten an anderer Stelle erzeugt. Außerdem wurde das Kopieren einiger ausgewählter Objekte zum abstrakteren Zugriff auf den AST ermöglicht. Dabei werden ebenfalls zugrunde liegende Knoten des AST kopiert.

Transformation von Klassen, Aspekten und Verfeinerungen

Während der Codetransformationen müssen Verfeinerungen von Klassen und Aspekten in Klassen bzw. Aspekte umgewandelt werden. Außerdem muss eine Vererbung zum Element der darüber liegenden Schicht generiert werden. Bei Klassen und Aspekten selbst, also den Konstanten in AHEAD, ist eine solche Transformation nicht notwendig. Hier erfolgt lediglich eine Transformation des Klassennamens.

Für Aspekte und deren Verfeinerungen sind identische Transformationen durchzuführen. Da der Syntaxbaum einer Verfeinerung der zu generierenden Klassen- bzw. Aspektdefinition sehr ähnlich ist, können diese durch wenige Veränderungen transformiert werden. Hierzu muss der Knoten entfernt werden, der das Schlüsselwort `refines` enthält und der Typ der Definition von einer Verfeinerung in eine Klasse bzw. einen Aspekt umgewandelt werden. In AspectC++ kann nur von abstrakten Aspekten geerbt werden. Daher wird in allen verfeinerten Aspekten ein abstrakter Dummy-Pointcut erzeugt.

Eine wesentlich umfangreichere Transformation ist für Verfeinerungen von Templates notwendig. Neben den bereits besprochenen Transformationen ist das Erzeugen der Templatedefinition notwendig. Zur Vereinfachung kann die Definitionen des Basislayers kopiert werden. Die generierte Vererbung unterscheidet sich ebenfalls, da Templateparameter an die Basisklasse weitergegeben werden müssen. In Abbildung 4.22 sind die notwendigen Transformationen des AST der Klasse `Array` aus Abbildung 4.20 dargestellt. Abbildung 4.21 zeigt den zugehörigen transformierten Quelltext.

```
1 /** Layer Base **/
2 template <class ELEMENT>
3 class Array {..};
4
5
6 /** Layer Sync **/
7 refines class Array : public SyncObject {..};
```

Abbildung 4.20: Templateverfeinerungen in FeatureC++

```
1 /** Generiert aus Layer Base **/
2 template <class ELEMENT>
3 class Array_Base {..};
4
5 /** Generiert aus Layer Sync **/
6 template <class ELEMENT>
7 class Array_Sync :
8     public Array_Base<ELEMENT>,
9     public SyncObject {..};
```

Abbildung 4.21: Transformation von Templateverfeinerungen

Alle neuen Knoten des Syntaxbaums in Abbildung 4.22 sind dunkelgrau hinterlegt. Modifizierte Elemente, wie Klassennamen, sind fett umrandet. Die Templatedefinition wurde aus Gründen der Übersichtlichkeit zum Teil ausgelassen. Sie ist mit den Knoten „TemplateParamList“ und „TemplateParamDecl“ angedeutet. Die Definition der Verfeinerung mit dem Schlüsselwort `refines` wurde durch die Klassendefinition ersetzt (Kno-

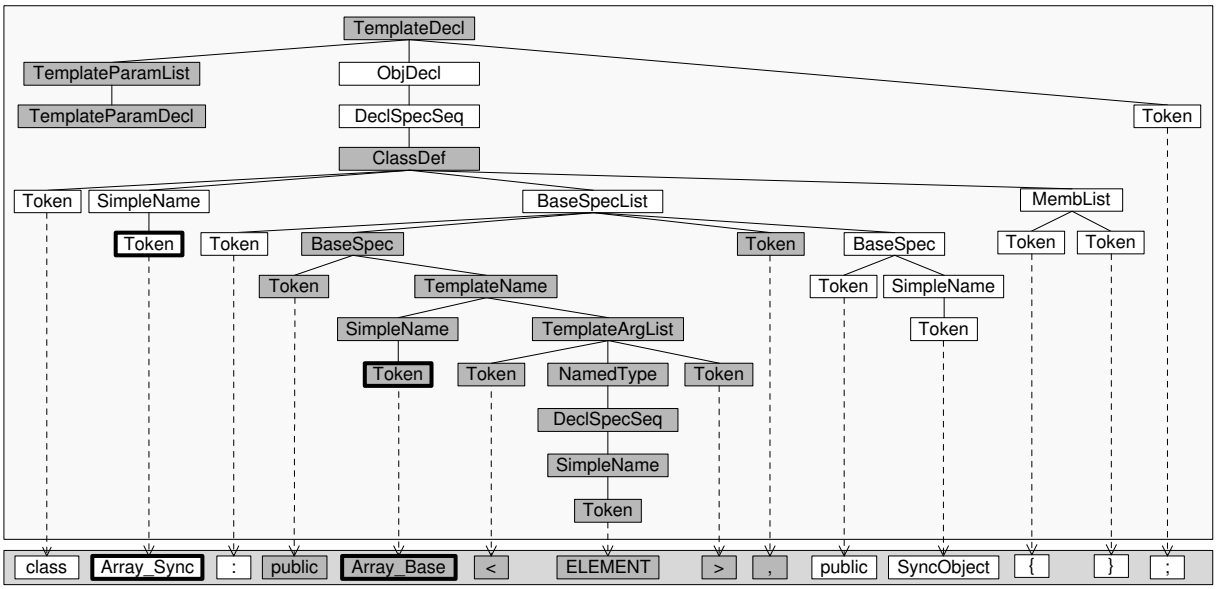

Abbildung 4.22: Codetransformation mit PUMA – AST und Token

ten ClassDef).

Das Erzeugen der Vererbung erfolgt durch das Generieren einer Basisklassendefinition (Knoten BaseSpec). Da im angegebenen Beispiel bereits eine Vererbung existiert, muss diese zur existierenden Liste hinzugefügt werden (Knoten BaseSpecList).

Transformation von Methoden

Nach dem Transformieren einer Verfeinerung oder Klasse müssen die enthaltenen Methoden angepasst werden. Diese sind innerhalb oder außerhalb der Klassen definiert. In beiden Fällen ist das Ersetzten von Aufrufen der Basisklasse notwendig. Sie werden durch die Verwendung von `super` ermöglicht und müssen durch den transformierten Namen der Basisklasse ersetzt werden. Bei der Transformation von Konstruktoren ist auch `super` bei der Initialisierung der Basisklasse zu ersetzen. Im Falle von Templates sind außerdem die Templateparameter beim Aufruf von Methoden der Basisklasse zu erzeugen. Hierzu werden entsprechende Informationen aus der Basisklasse ermittelt und die Syntaxbäume der Methoden mit Knoten für Templateparameter erweitert.

Die Quelltextelemente, die sich außerhalb der Klasse befinden (Methodendefinitionen der Klasse, Initialisierungen statischer Member und globale Methoden) müssen entsprechend ihrer Zugehörigkeit zur Klasse transformiert werden. Dabei sind die Namen der Klassen anzupassen. Bei Templateklassen sind außerdem Templatedefinitionen zu erzeugen, wobei an dieser Stelle keine Template-Standard-Parameter verwendet werden

dürfen.

Wird in den Methoden einer Klasse der `this`-Zeiger verwendet, muss dieser in den Typen der finalen Klasse umgewandelt werden (siehe Abschnitt 4.2.2). Hierzu wird jedes Vorkommen mit einer statischen Typumwandlung versehen.

Propagieren von Methoden

Neben dem Anpassen der bereits existierenden Methoden werden bei der Transformation Konstruktoren und überladene Methoden propagiert. Grundlage bildet die Definition der jeweiligen Methode der Basisklasse. Der entsprechende Knoten des AST wird in den Syntaxbaum der Verfeinerung kopiert und die Namen der Klasse angepasst. Dabei wird der Inhalt des Funktionsrumpfes ausgelassen. An dessen Stelle wird der Aufruf des Konstruktors der Basisklasse erzeugt. Beim Propagieren von Konstruktoren wird an Stelle des Funktionsaufrufs die Initialisierung der Basisklasse generiert.

Transformation von Pointcuts

Die Transformation des Advice-Codes in Aspekten entspricht der bereits besprochenen Transformation von Methoden. Pointcuts hingegen erfordern eine andere Behandlung. Da PUMA während des Parsens für Suchausdrücke keinen Syntaxbaum erstellt[8], ist vor der Transformation die Analyse der Suchausdrücke notwendig.

Alle Suchausdrücke, die auf Klassen oder Klassenattribute zutreffen, werden transformiert. Dabei werden alle auf einen Suchausdruck zutreffenden Objekte ermittelt und jeweils ein neuer Suchausdruck erstellt. Diese werden mit logischem Oder verknüpft und ersetzen so den ursprünglichen Suchausdruck.

Ein weiterer wesentlicher Punkt ist das Verfeinern von Pointcuts (vgl. Abbildung 4.17). Generell können in AspectC++ mit dem Schlüsselwort `virtual` definierte Pointcuts überschrieben werden. Pointcuts werden vor dem Übersetzen ausgewertet, weshalb das Schlüsselwort `virtual` in diesem Fall eine andere Bedeutung hat als es von der OOP her bekannt ist. Das statische Auswerten der Pointcuts führt in Kombination mit `virtual` zu den folgenden Ergebnissen beim Weben:

- Das Überschreiben eines **nicht-virtuellen Pointcuts** ist in AspectC++ nicht möglich.
- Ein **virtueller Pointcut** kann überschrieben werden. Es wird Programmcode entsprechend des neuen Pointcuts gewebt.

[8] die Elemente in den Suchausdrücken werden als Strings interpretiert

- Das Erweitern eines überschriebenen Pointcuts durch dessen „Aufruf“ im überschreibenden Pointcut ist nicht möglich. Ein „Aufruf“ eines anderen nicht-virtuellen Pointcuts des Basisaspekts ist hingegen möglich.

Um dennoch eine Transformation zu finden, die das Erweitern von Pointcuts ermöglicht, wird ein Hilfspointcut erstellt. Dieser enthält den eigentlichen Suchausdruck des zu überschreibenden Pointcuts (siehe Abbildung 4.24, Zeilen 3 und 4; Abbildung 4.23 enthält den zugehörigen Ausgangsquelltext).

```
1  //Layer Base
2  aspect logging {
3    pointcut virtual log()
4        = "% String::%(...)";
5
6
7    ..
8  };
9
10 //Layer ExtLog
11 refines aspect logging {
12   pointcut virtual log()
13       = "% Alloc::%(...)"
14         || super::log();
15
16
17 }
```

Abbildung 4.23: Verfeinerung von Pointcuts in FeatureC++

```
1  //Generiert aus Layer Base
2  aspect logging_base {
3    pointcut log_helper()
4        = "% String_base::%(...)";
5    pointcut virtual log()
6        = log_helper();
7    ..
8  };
9
10 //Generiert aus Layer ExtLog
11 aspect logging : public logging_base {
12   pointcut log_helper()
13       = "% Alloc_Base::%(...)"
14         || logging_base::log_helper();
15   pointcut virtual log()
16       = log_helper();
17 }
```

Abbildung 4.24: Transformation von Pointcuts

Der Pointcut wird demnach in zwei Pointcuts transformiert (Zeilen 3-6), wobei einer die Definition des Suchausdrucks enthält, und der andere lediglich das Überschreiben mit dem Schlüsselwort `virtual` ermöglicht. Wird der Pointcut nicht durch eine Verfeinerung überschrieben, so verweist er auf den ursprünglichen Suchausdruck (Zeile 6). Erfolgt in einem verfeinernden Aspekt ein Überschreiben (Abbildung 4.23, Zeile 14), so wird in der Transformation der neue Suchausdruck verwendet (Zeile 16). In diesem Fall wird außerdem das Schlüsselwort `super` durch den nicht virtuellen Pointcut des Basisaspektes ersetzt (Zeile 14). Durch die Verwendung des Hilfspointcuts im erbenden Aspekt (Zeilen 12-14 der Abbildung 4.24), ist die Skalierung dieser Umsetzung auf lange Verfeinerungshierarchien sichergestellt.

Weiterer generierter Quelltext

Nach der Transformation aller enthaltenen Elemente erfolgt die Ausgabe des Quelltextes. Dabei werden Deklarationen und Implementierungen durch Ausgabe in die entspre-

chenden Dateien getrennt. In Abschnitt 4.2.2 wurden bereits weitere zu generierende Quelltextelemente besprochen:

- Includes bzw. Forwarddeklarationen für referenzierte Klassen,
- Präprozessordefinitionen, um wiederholtes Parsen der Header zu verhindern,
- Namespace entsprechend des Namens der Konfiguration,
- Forwarddeklaration der finalen Klasse zu Beginn der Headerdatei,
- Include für die zu einer Implementierungsdatei gehörenden Headerdatei,
- Using-Deklaration für den Namespace in C++ Implementierungs- und Aspektdateien,
- Präprozessor Line-Direktiven.

Der Quelltext dieser Elemente wird während der Ausgabe generiert. Die Erzeugung der Includes und Forwarddeklarationen beruht dabei auf den während der Transformation gesammelten Informationen. Dazu wird jeder Zugriff auf eine andere Klasse registriert und ein Include für jede benötigte Datei, oder eine Forwarddeklaration der verwendeten Klasse erzeugt.

Das Generieren der Line-Direktiven erfolgt bei der Ausgabe der Token in eine Datei. Da jedes Token eine Kennzeichnung seines Ursprungs besitzt (Datei und Zeile) werden hierzu die Änderungen dieser Information betrachtet. Ändert sich die Datei, oder ändert sich die Zeilennummer im Widerspruch zum Zeilenumbruch, so wird eine Line-Direktive erzeugt:

```
#line <line_no> "<file_name>"
```

Hierbei ist `<line_no>` die Zeilennummer aus der die nächste Quellcodezeile stammt und `<file_name>` die zugehörige Datei.

Die zum Ausschluss des wiederholten Parsens durch den Compiler erzeugten Präprozessordirektiven haben folgende Form:

```
#ifndef FILE_NAME_H__INCLUDED
#define FILE_NAME_H__INCLUDED
..
#endif //FILE_NAME_H__INCLUDED
```

Die definierte Variable (im Beispiel `FILE_NAME_H__INCLUDED`) wird aus dem Dateinamen und dem Suffix `"__INCLUDED"` gebildet. Der Punkt des Dateinamens sowie Wechsel zwischen Groß- und Kleinbuchstaben werden durch Unterstriche (_) ersetzt. Innerhalb des durch `#ifndef..` und `#endif` eingeschlossenen Bereichs befindet sich der eigentliche Quelltext der Headerdatei.

4.3.4 Weben mit AspectC++

Nach dem Ausgeben des transformierten Quelltextes erfolgt die Anwendung der Aspekte auf den gesamten objektorientierten Programmcode. Hierzu wird AspectC++ verwendet, welches als Ausgabe rein objektorientierten Quelltext erzeugt. Die Ausführung von AspectC++ erfolgt analog der Ausführung des DRC-Werkzeugs der ATS. Treten während des Webens Fehler auf, werden diese an den aufrufenden Prozess (IDE, Kommandozeile, *Make*, etc.) weitergeleitet. Nach Abschluss dieses Teils wird der objektorientierte Programmcode von einem herkömmlichen Compiler in Maschinencode umgesetzt.

4.4 Untersuchung der Ergebnisse

Im letzten Kapitel wurde FeatureC++, eine Spracherweiterung für C++ und deren prototypische Umsetzung vorgestellt. Die Eigenschaften der entwickelten Erweiterung werden im Folgenden anhand eines konkreten Beispiels betrachtet, sowie Probleme und deren Lösungen untersucht. Außerdem soll der Ansatz mit anderen Umsetzungen der merkmals- und aspektorientierten Programmierung verglichen werden.

4.4.1 Fallstudie

Häufig ist in der Softwareentwicklung die Berücksichtigung unterschiedlicher Betriebssysteme notwendig. In Abschnitt 2.5 wurde die Entwicklung sich überschneidender Merkmale anhand der Klasse String dargestellt (vgl. Abbildung 2.30). Dabei wurde das Betriebssystem als ein Merkmal betrachtet, welches sich mit dem Merkmal „Character Type“ überschneidet. An diesem Beispiel wird gezeigt, wie auf einfache Weise solche Merkmale mit FeatureC++ implementiert werden können. Der Entwurf orientiert sich dabei am Kollaborationendiagramm der Abbildung 3.8. Im Folgenden werden nur Quelltextausschnitte aufgeführt. Eine Darstellung des vollständigen Quelltextes findet sich in Anhang A[9].

[9]Der entwickelte Prototyp sowie weitere Beispiele sind unter http://wwwiti.cs.uni-magdeburg.de/iti_db/fcc/ verfügbar.

Die verwendeten Merkmale sind in Abbildung 4.25 dargestellt. Die Merkmale „Char“, „WChar“, „Win“ und „Unix“ implementieren alternative Merkmale zur Verwendung des Zeichensatzes (C-Type) bzw. des Betriebssystems (OS) wie bereits in Abschnitt 2.5 vorgestellt. Die Merkmale „Data“ und „Public“ sind obligatorische Merkmale, die die Manipulation interner Daten (Data) bzw. öffentliche Operationen (Public) implementieren. Die optionalen Merkmale „Profile“ und „Sync“ implementieren Profiling[10] bzw. Synchronisation des Strings.

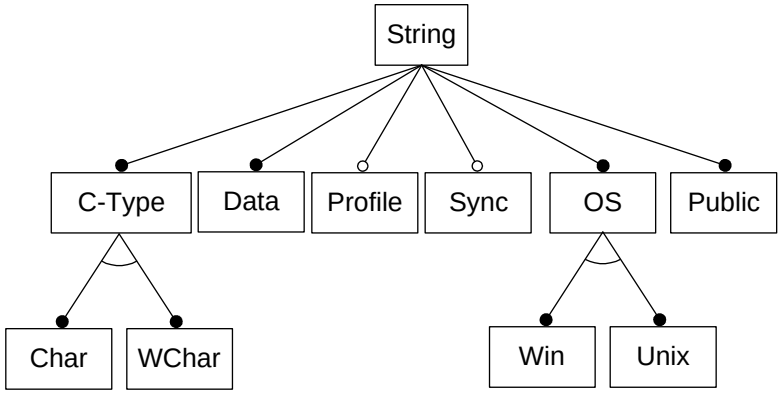

Abbildung 4.25: Merkmalsdiagramm eines konfigurierbaren Strings

Entwurf

Im Kollaborationenentwurf der Abbildung 4.26 sind die verwendeten Klassen, Aspekte (eckig umrandet) und Merkmale dargestellt. Zur Verdeutlichung der Skalierbarkeit des Ansatzes ist die Klasse `Mutex` enthalten. Diese implementiert ein systemabhängiges Synchronisationsobjekt. Sie findet im Merkmal „Sync“ in Verbindung mit dem Aspekt `Sync` zur Synchronisation Verwendung. Die Erweiterbarkeit der Aspekte wird mit Hilfe der Merkmale „Profile“ und „Sync“ gezeigt. Das Merkmal „Profile“ implementiert mit dem Aspekt `Profiler` die Analyse des Laufzeitverhaltens der Klasse `String`. Die Wirkung des Aspektes wird im Merkmal `Sync` auf die Klasse `Mutex` ausgeweitet.

Wie bereits in Abschnitt 3.3.1 gezeigt, erfolgt im Layer „Char“ und „WChar“ die Auswahl der zu verwendenden Klassen zur Manipulation von Zeichenfolgen (Klassen `Char` und `WChar`). Die Verfeinerungen der beiden Klassen in den Layern „Unix“ und „Windows“ implementieren die systemspezifischen Teile. Einen Ausschnitt der Verfeinerung der Klasse `Char` im Layer Windows zeigt Abbildung 4.27. In Abbildung 4.28

[10] Unter Profiling wird die Analyse von Programmen unter anderem zur Gewinnung von Informationen über die Laufzeit und Verwendung von Funktionen verstanden.

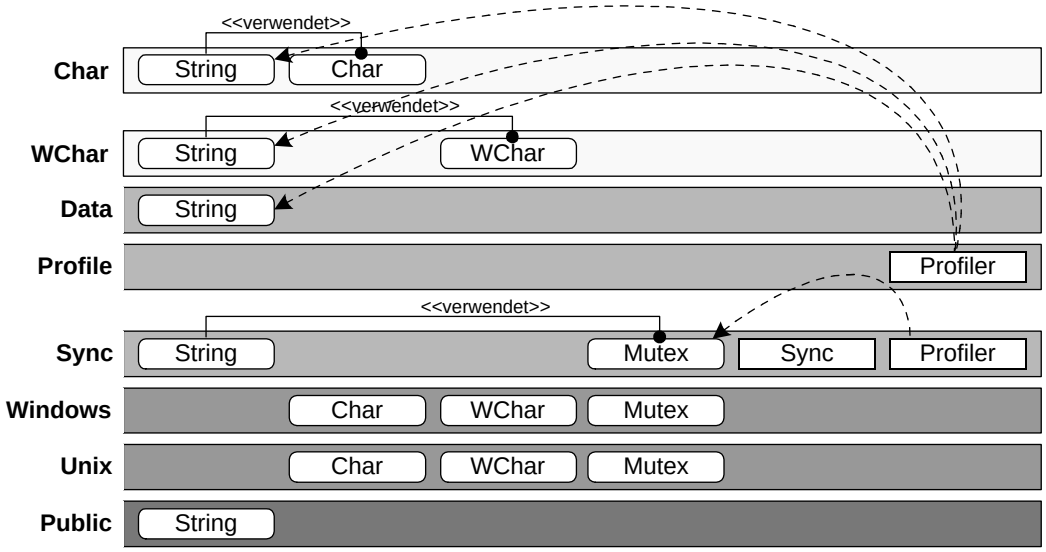

Abbildung 4.26: Kollaborationendiagramm eines konfigurierbaren Strings

ist die Verwendung der Klasse dargestellt. Es ist zu sehen, dass keine Verwendung von Präprozessordirektiven der Form `#ifdef WIN32` notwendig ist, um die Verwendung für ein Betriebssystem im Quellcode zu konfigurieren. Die Konfiguration des verwendeten Zeichentyps erfolgt ebenfalls vollständig außerhalb des Quellcodes.

```
1  #include <string.h>
2  #include <stdlib.h>
3
4  refines class Char {
5  public:
6    //Vergleichen zweier Strings
7    static int stricmp(const char* s1, const char* s2) {
8      return ::_stricmp(s1,s2);
9    }
10   ..
11 };
```

Abbildung 4.27: Verfeinerung von Klassen in FeatureC++

Gleiches gilt für die Implementierung der Synchronisation mit der Klasse `Mutex`. Diese wird in Zusammenhang mit dem Aspekt `Sync` verwendet. Dieser implementiert die Synchronisation der Klasse `String`. Durch die Positionierung des Aspektes oberhalb des Layers „Public“, der öffentliche Methode der Klasse `String` enthält, kann die Synchronisation auf die Datenmanipulation im Layer „Data“ beschränkt werden[11]. Dabei wird

[11]Eine zusätzliche Synchronisation weiterer Methoden würde das Laufzeitverhalten beeinflussen.

```
1  //Layer Char
2  refines class String {
3  public:
4    typedef char Elem;
5    typedef Char StringFunctions;
6  };
7
8  //Layer Data
9  refines class String {
10 public:
11   int CompareNoCase(const Elem* other) const {
12     return StringFunctions::stricmp(m_Buf,other);
13   }
14 };
```

Abbildung 4.28: FeatureC++ – Verwendung verfeinerter Klassen

ein klarer Vorteil gegenüber herkömmlicher AOP deutlich: Neben der Steuerung von Pointcuts über Klassennamen und Signaturen von Methoden ist eine Steuerung über Merkmale möglich.

Die Erweiterung der AOP auf FOP Elemente wird mit dem Aspekt `Profiler` deutlich. Durch seine Erweiterung im Layer „Sync“ wird das Profiling auf die Klasse `Mutex` ausgeweitet. So ist bei Verwendung des Merkmals Synchronisation ein getrenntes Profiling beider Klassen möglich. Dies erlaubt z. B. Aussagen zur Auswirkung der Synchronisation auf die Leistungsfähigkeit der Klasse String, indem die von der Aspektverfeinerung ermittelten Daten mit den vom Basisaspekt ermittelten Daten verglichen werden.

Auswertung

Bei der Implementierung des vorgestellten Beispiels wurde festgestellt, dass die entwickelte Spracherweiterung auf einfache Weise die Implementierung eines merkmalsorientierten Entwurfs erlaubt. Die Kombination mit aspektorientierten Elementen erlaubt die Behandlung homogener Crosscuts, ohne dabei den Merkmalszusammenhalt zu verlieren. Durch Verwendung der `#line`-Direktiven kann bei Fehlermeldungen während der Entwicklung direkt merkmalsorientierter Quelltext bearbeitet werden.

Es traten dennoch einige Probleme auf, die zum Teil bei der Entwicklung der Spracherweiterung nicht betrachtet wurden:

- Die Verwendung virtueller Methoden in den Verfeinerungen ist notwendig, da sonst die Möglichkeit besteht, dass nicht verfeinerte Varianten von Methoden an Stelle der verfeinerten aufgerufen werden. Dies stellt insbesondere ein Problem bei statischen Methoden dar, da diese nicht virtuell sein können.

- Fängt ein Aspekt Methodenaufrufe in Schichten ab, die sich oberhalb der Implementierung des Aspektes befinden, so ist aus dem Advice-Code kein Zugriff auf Member der betroffenen Klasse möglich, die erst in der Schicht des Aspektes definiert wurden. Die Kombination von Aspekten mit herkömmlichen Verfeinerungen von Klassen wird dadurch beeinträchtigt.

In Abschnitt 4.4.3 folgt ein Vergleich der Spracherweiterung zu weiteren Umsetzungen der FOP, AOP und Ansätzen, die Elemente beider Paradigmen besitzen. Zuvor wird die Anwendbarkeit des GenVoca-Ansatzes auf FeatureC++ untersucht.

4.4.2 Anwendung von GenVoca und AHEAD

GenVoca wurde als eine Theorie für die algebraische Beschreibung eines Programms bestehend aus Klassen und Verfeinerungen vorgestellt (siehe Abschnitt 2.3.4). Diese wurde mit AHEAD auf beliebige Elemente der Softwareentwicklung erweitert. Mit der ATS sind Werkzeuge vorhanden, die diese Theorie zum Teil umsetzen. Insbesondere besteht Unterstützung für die merkmalsorientierte Programmierung mit Java. Für die Umsetzung von FeatureC++ bildet die ATS die Grundlage. Darauf basierend erfolgte die Integration aspektorientierter Konzepte. Bisher wurde aber die Integration von FeatureC++ in das Konzept von GenVoca bzw. AHEAD nicht betrachtet. Eine vollständig integrierte Umsetzung von FeatureC++ ermöglicht die Konfiguration und auch automatisiertes Generieren einer Software, die Komponenten beider Programmiersprachen enthält.

Merkmalsorientierter Quelltext

Betrachtet man die Umsetzung von FeatureC++ im Hinblick auf merkmalsorientierten Quelltext, lässt sich feststellen, dass dieser nur in Bezug auf technische und C++ spezifische Aspekte von der Implementierung für Java in der ATS abweicht. Dennoch wurden die in Abschnitt 2.3.4 dargestellten algebraischen Hintergründe nicht betrachtet. So ist die Assoziativität des Verfeinerungsoperators ein wesentlicher Punkt:

$$C3 \bullet (C2 \bullet C1) = (C3 \bullet C2) \bullet C1.$$

Dadurch wird es möglich, Verfeinerungen auf andere Verfeinerungen anzuwenden, und erst später die Anwendung auf eine Konstante vorzunehmen. Dies ist z. B. notwendig, um Merkmale einer Software hierarchisch zu gestalten, und deren Transformation unabhängig von anderen Merkmalen vornehmen zu können. Auf diese Weise wird eine Grundlage für die Kombination großer Softwarekomponenten mit merkmalsorientierten Methoden geschaffen.

Die Code-Transformation in FeatureC++ erfolgt ausgehend vom Basislayer, also der Schicht, die in einer GenVoca Gleichung am weitesten rechts steht. Eine Verfeinerung von $C2$ durch $C3$, wie auf der rechten Seite der Gleichung, ist derzeit nicht möglich. Dies lässt sich durch Generieren einer neuen Verfeinerung umsetzen. Diese Verfeinerung enthält alle Elemente aus $C2$ und $C3$. Sind in beiden Ausgangsverfeinerungen Verfeinerungen von Methoden enthalten, müssen zur Unterscheidung der Methoden andere Namen verwendet werden. Hier bietet sich die Verwendung des Namens der Schicht wie auch bei der Generierung der Klassenhierarchien an. Die Transformation fällt damit ähnlich der *Jampack*-Transformation[12] der ATS aus.

Aspektorientierte Elemente

Die Transformation von Aspekten erfolgt wie auch die Transformation merkmalsorientierter Elemente beginnend mit dem Basislayer. Die Anwendung einer Aspektverfeinerung auf eine andere Aspektverfeinerung ist auch hier nicht vorgesehen. Eine nahe liegende Implementierung für diesen Fall ist ebenfalls das Erstellen neuer Aspektverfeinerungen. Ein Problem stellt dabei die Transformation der Pointcuts dar. Diese werden so transformiert, dass sie auf Klassenverfeinerungen derselben Schicht und der darüber liegenden Schichten wirken. Da auch eine Transformation der Klassenverfeinerungen erfolgt, muss die Transformation der Aspektverfeinerungen daran angepasst werden. Die Umsetzung dieser Lösung ist demzufolge sehr aufwendig, und würde die Komplexität der AspectC++ Suchausdrücke wesentlich erhöhen. Daher sollte versucht werden andere Lösungsmöglichkeiten zu finden.

4.4.3 Vergleich mit anderen Ansätzen

Die vorangegangenen zwei Abschnitte haben bereits teilweise eine Bewertung von FeatureC++ vorgenommen. Der folgende Abschnitt stellt einen Vergleich mit den in Abschnitt 3 besprochenen Ansätzen her.

Auf Grund der Ähnlichkeit zu AHEAD bzw. der AHEAD Tool Suite sind auch einige der Probleme im merkmalsorientierten Teil von FeatureC++ wieder zu finden. Durch Zusammenarbeit mit der aspektorientierten Erweiterung werden diese zum Teil gelöst und es zeigen sich grundsätzliche Vorteile. In Tabelle 3.1 sind wesentliche Probleme der betrachteten Ansätze aufgeführt. Bei einer Einordnung von FeatureC++ lassen sich die Vorteile der ATS und von AspectC++ wieder finden. So können sowohl homogene als auch heterogene Crosscuts adäquat behandelt werden. Durch die Zuordnung der Aspekte zu den Schichten bleibt der Merkmalszusammenhalt auch bei homogenen Crosscuts

[12] Vgl. Abbildung 2.22.

erhalten. Darüber hinaus ist die Erweiterbarkeit der verwendeten Aspekte gegeben. Diese wirken nur auf bereits vorhandene Schichten, können aber leicht auf neue Schichten ausgeweitet werden.

Die Anwendung der AOP auf merkmalsorientierten Quelltext erlaubt es optionale Merkmale besser zu berücksichtigen. Dies trifft insbesondere auf Crosscuts zu, die sehr viele Klassen und damit viele andere Merkmale betreffen. Das Feature Optionality Problem trifft aber weiterhin bei Merkmalen zu, die rein merkmalsorientiert implementiert werden.

Ursache des Feature Optionality Problems sind Crosscuts zwischen Merkmalen. Ein ähnliches Problem entsteht, wenn orthogonale Merkmale im Kollaborationenentwurf wiedergegeben werden (vgl. Abschnitt 2.5 und 4.4.1). Ursache ist die notwendige Transformation der eigentlich mehrdimensionalen Merkmalsbeziehungen in eine Dimension oder zusätzliche Klassen. Hier bietet FeatureC++ eine weitere Möglichkeit mit der Implementierung einer Merkmalsdimension in Aspekten (z. B. die Dimension „Character Type“ des Beispiels aus Abbildung 4.26). Probleme bestehen weiterhin bei vielen sich schneidenden Merkmalen. Dabei wird durch Linearisierung der Dimensionen die Konfiguration entsprechend komplexer und es entstehen Abhängigkeiten innerhalb der Konfiguration.

Die Konfiguration mit FeatureC++ gestaltet sich, wie auch bei der ATS, durch die Verwendung der `.equation`-Dateien flexibel und sehr einfach. Klare Vorteile gegenüber anderen Ansätzen ergeben sich durch Einbeziehung der Aspekte in die Konfiguration.

Weitere Vorteile bestehen bezüglich praktischer Gesichtspunkte. Im Gegensatz zur ATS ist durch die Verwendung von Präprozessordirektiven direktes Debuggen des FOP-Codes möglich. Außerdem ist der Gesamtüberblick über eine entwickelte Software besser als bei reiner AOP oder reiner FOP. Hier kann sowohl vom Gewinn an Übersichtlichkeit durch aspektorientierte Elemente als auch von der Übersichtlichkeit des FOP-Codes profitiert werden.

4.5 Ausblick und Erweiterungen

Es wurde gezeigt, dass weiterhin sowohl technische, als auch grundsätzliche Probleme bestehen. Diese geben Anlass zur Erweiterung von FeatureC++. Einige der Schwierigkeiten könnten bereits durch eine ausreichende Unterstützung von Entwicklungsumgebung oder Debugger gelöst werden, andere benötigen eine direkte Erweiterung des Ansatzes. Hierzu gehören das Feature Optionality Problem und das Konstruktorproblem. Mögliche Lösungen für einige Probleme werden daher in diesem Abschnitt diskutiert. Für weitere Diskussion sei ebenfalls auf [ALRS05a] verwiesen.

4.5.1 Jampack

Wie in den Abschnitten 4.4.1 und 2.3.4 beschrieben, bereitet die Verwendung einer Klassenhierarchie in einigen Fällen Schwierigkeiten. Andererseits ist hierdurch eine leichte Umsetzung der aspektorientierten Erweiterung möglich. Aus diesem Grund muss näher untersucht werden, in wie weit eine Umsetzung des *Jampack*-Ansatzes[13] notwendig und sinnvoll ist, bzw. ob eine Lösung der angesprochenen Probleme mit der derzeitigen Umsetzung möglich ist. Es sprechen folgende Punkte gegen die Verwendung einer Klassenhierarchie:

1. die Notwendigkeit virtueller Methoden,
2. die fehlende Zugriffsmöglichkeit auf Member von Verfeinerungen aus der Schicht des Aspektes und
3. der Verlust der Assoziativität von Verfeinerungen.

Eine mögliche Lösung zu Punkt 1 ist das Transformieren verfeinerter Methoden in virtuelle Methoden. Damit sind jedoch verfeinerte Methoden generell virtuell, was die Möglichkeiten der Verwendung virtueller Methoden in Bezug auf herkömmliche Vererbung einschränkt. Des Weiteren ist diese Lösung nicht auf statische Methoden anwendbar.

Alle drei genannten Probleme werden durch die Verwendung einer Jampack-Transformation gelöst. Neben diesen muss untersucht werden, ob weitere Vor- oder Nachteile des Jampack-Ansatzes gegenüber der Umsetzung in eine Klassenhierarchie bestehen.

4.5.2 Feature Optionality Problem

Ist ein Merkmal in einer Konfiguration nicht vorhanden, besteht ein Problem bei der Anwendung von Verfeinerungen der Klassen, die für dieses Merkmal notwendig sind. Um dies zu umgehen, dürfen solche Verfeinerungen keine Fehlermeldungen bei der Transformation auslösen. Gleiches trifft für die Verfeinerung von Methoden zu.

Werden solche Verfeinerungen lediglich ignoriert, können auch „echte“ Fehler nicht mehr vom Compiler beanstandet werden. Hier ist zwar eine Unterstützung durch die Entwicklungsumgebung hilfreich aber nicht ausreichend. Die derzeitige Umsetzung von FeatureC++ gibt bei Verfeinerungen nicht vorhandener Klassen eine Warnung aus. Wird

[13] *Jampack*-Transformation: Transformation in eine Klasse an Stelle einer Klassenhierarchie. Siehe Abschnitt 2.3.6.

eine nicht vorhandene Methode einer Basisklasse durch die Verwendung von `super` aufgerufen, bricht die Transformation mit einer Fehlermeldung ab.

Das Erkennen von Verfeinerungen von Methoden wäre durch eine Anpassung der Syntax ähnlich der AOP Syntax möglich. Durch die Verwendung der Schlüsselwörter `before` oder `after`, könnte ein expliziter Aufruf der verfeinerten Methode ausbleiben. Ein Aufruf einer nicht existierenden Methode mit `super` erzeugt nach wie vor eine Fehlermeldung. Das Verfeinern ist hingegen optional und findet nur Anwendung, wenn die zu verfeinernde Methode auch existiert. Auch die Verwendung von `around`, wie in der AOP ist möglich, so dass z. B. mit einem Aufruf von `proceed(...)` die Ausführung der verfeinerten Methode mit modifizierten Argumenten fortgesetzt werden kann. Eine Unterscheidung zu einem Methodenaufruf mit `super` ist dennoch möglich. Offen bliebe aber auch hier die Verfahrensweise bei einem Methodenaufruf einer nicht vorhandenen Methode mit `super`.

4.5.3 Crosscuts zwischen Merkmalen

Eine umfassende Klärung des Feature Optionality Problems Bedarf einer allgemeineren Betrachtung der Abhängigkeiten zwischen Merkmalen. Crosscuts zwischen Merkmalen sind Ursache des Feature Optionality Problems. Für vollständig unabhängige Merkmale tritt es demnach nicht auf. Es ist folglich nicht nur die Beziehung zwischen zwei Merkmalen zu betrachten, sondern die Interaktionen zwischen beliebigen Merkmalen.

Der MDSOC Ansatz ermöglicht dies durch die Trennung der Merkmale in Merkmalsdimensionen. In [LBN05] wird versucht alle Interaktionen zwischen verschiedenen Merkmalen zu extrahieren, um so bei einer Konfiguration nur notwendige Verfeinerungen anzuwenden. Dabei werden ähnlich dem MDSOC Ansatz verschiedene Merkmalsdimensionen behandelt. Es wird weiterhin versucht, eine algebraische Beschreibung der Konfiguration zu ermöglichen. Die intuitive mehrdimensionale Beschreibung wie im MDSOC Ansatz bleibt aber aus.

Zusammenfassend ist festzuhalten, dass zum Feature Optionality Problem und allgemein der Interaktionen zwischen Merkmalen derzeit keine umfassende Lösung existiert und dies Gegenstand aktueller Forschung ist.

4.5.4 Konstruktorproblem

Konstruktoren sind in der OOP von besonderem Interesse, da sie Möglichkeiten bieten, Objekte auf verschiedene Weise zu initialisieren. Da bei jeder Vererbung die Konstruktoren neu definiert werden müssen, ist eine Vielzahl von Konstruktoren oft hinderlich. In FeatureC++ wird dieses Problem innerhalb einer Verfeinerungshierarchie durch das

Propagieren der Konstruktoren verhindert[14]. In der ATS wird außerdem die Verfeinerung von Konstruktoren angeboten. Dies erfolgt in FeatureC++ durch das Neudefinieren eines Konstruktors, wobei die Initialisierung der verfeinerten Klasse mit `super` erfolgt.

Die explizite Verfeinerung von Konstruktoren wie in der ATS ist daher eine sinnvolle Erweiterung. Aber auch hier ist die Verfeinerung aller Konstruktoren notwendig, wenn auch nur wenige Member einer Klasse initialisiert werden. Aus diesem Grund wird eine Erweiterung der Syntax in Betracht gezogen. Der einfachste Fall ist die Initialisierung von Membervariablen, für die keinerlei Parameter notwendig sind. Hier werden alle vorhandenen Konstruktoren um den notwendigen Code zur Initialisierung verfeinert.

Neben diesem einfachen Fall sind weitere Varianten möglich, bei der Member mit Parametern des Konstruktors initialisiert werden. Hier besteht die Möglichkeit, Konstruktoren zu mischen, indem die Argumentlisten neuer Konstruktoren mit denen existierender Konstruktoren zu neuen Argumentlisten kombiniert werden. Die Initialisierung der Basisklasse erfolgt dabei automatisch. Eine solche Vorgehensweise wurde bereits in [CL01] untersucht. Das Feature Optionality Problem oder allgemeiner Crosscuts zwischen Merkmalen treffen hier ebenfalls zu, so dass dies in Zusammenhang mit den bereits erwähnten Problemen zu diskutieren ist.

4.5.5 Self Problem

Wie auch das Konstruktorproblem, besteht das Self Problem innerhalb einer Verfeinerungshierarchie nicht. Instanzen werden immer nur von der finalen Klasse erzeugt. Vererbung innerhalb der FOP bedingt wiederum dieses Problem. Auf Grund der nichtlinearen Struktur von Vererbungen lässt sich der Ansatz der FOP nicht auf die OOP übertragen. Die Verwendung virtueller Typen, die auch für typsichere Sprachen möglich sind [BOW98], ist an dieser Stelle hilfreich.

4.5.6 Library Scalability Problem

Die einfache Konfiguration alternativer Merkmalen einer Software ist ein wesentlicher Vorteil der FOP gegenüber der OOP. Werden innerhalb einer Software mehrere alternative Konfigurationen kleiner Komponenten oder Klassen benötigt, kann dies durch die Auslagerung dieser Klassen in eine Bibliothek ermöglicht werden. Aus dieser können die benötigten Konfigurationen erzeugt werden. Ist dies nur für wenige Alternativen der Fall (z. B. lediglich zwei Varianten einer Klasse), ist der Aufwand für die Auslagerung in eine Bibliothek zu groß. Eine Implementierung beider Varianten innerhalb des FOP-Entwurfs

[14] Bei der üblichen Verwendung der Vererbung innerhalb des merkmalsorientierten Entwurfs bleibt das Problem bestehen. Daher muss auch dieser Bereich weiter untersucht werden.

ist aber auch nicht optimal.

Der Erfolg der Mixins beruht auf der Anwendung einer Verfeinerung auf beliebige Klassen. Die Anwendung dieses Konzeptes auf den Kollaborationenentwurf, entspricht einer Mehrfachverfeinerung, wie sie auch mit den Multi Mixins besteht (vgl. Abschnitt 4.1.2). Im Gegensatz zum aspektorientierten Ansatz, mit dem ebenfalls mehrere Klassen verfeinert werden können, ist eine bessere Strukturierung der Verfeinerung gegeben. In diesem Fall müssen nicht für jede zu verfeinernde Methode Suchausdrücke definiert werden. Außerdem ist eine solche Mehrfachverfeinerung leichter verständlich, da sie ähnlich einer herkömmlichen Verfeinerung aufgebaut ist.

Kapitel 5

Zusammenfassung

In dieser Arbeit wurde das merkmalsorientierte Programmieren mit C++ untersucht. Nach einer Einleitung, welche die Notwendigkeit zur Entwicklung von Produktlinien mit C++ näher gebracht hat, wurden in Kapitel 2 für die weitere Arbeit relevante Grundlagen vermittelt. Dabei wurden nach Konzepten der OOP und des generischen Programmierens die aspektorientierte und merkmalsorientierte Programmierung sowie der MDSOC-Ansatz vorgestellt. Außerdem wurde eine Einführung in Codetransformationen gegeben, die für diese Arbeit von besonderem Interesse sind.

Eine Analyse bestehender Ansätze, die die Wiederverwendbarkeit und Erweiterbarkeit von Software zu verbessern versuchen, folgte in Kapitel 3. Dabei wurden nach Betrachtung der OOP reine FOP Ansätze und die AOP untersucht. Nach der Analyse kombinierte Ansätze, die sowohl Elemente der FOP als auch der AOP verwenden, wurden die bei der Produktlinienentwicklung auftretenden Probleme zusammengefasst.

In Kapitel 4 wurde FeatureC++ vorgestellt, eine Erweiterung zum merkmalsorientierten Programmieren unter C++. Die Umsetzung orientiert sich an der AHEAD Tool Suite, berücksichtigt aber C++ spezifische Eigenschaften wie z. B. Templates. Neben der reinen Umsetzung der FOP wurde die Integration der AOP untersucht. Dabei wurden drei Ansätze zur Integration von AOP und FOP entwickelt: *Multi Mixins*, *Aspectual Mixin Layers* und *Aspectual Mixins*. An ihnen wurde gezeigt, dass eine Anwendung der AOP auf FOP Quelltext möglich ist und sich das FOP Konzept auf Aspekte erweitern lässt. Weiterhin wurde in diesem Zusammenhang herausgestellt, dass sowohl FOP als auch AOP von dieser Kombination profitieren und Probleme beider Ansätze gelöst werden. Die Implementierung eines Prototyps zur Transformation von FeatureC++-Code in AOP- bzw. OOP-Code wurde daraufhin vorgestellt. Abgeschlossen wurde das Kapitel mit der Analyse des vorgestellten Ansatzes und seiner Implementierung sowie der Untersuchung bestehender Probleme und Erweiterungsmöglichkeiten.

Bei der Entwicklung und Analyse von FeatureC++ wurde folgen Eigenschaften

der Spracherweiterung herausgestellt:

- Wie auch in der ATS ist einfache Konfigurierbarkeit, die Behandlung heterogener Crosscuts und gute Erweiterbarkeit gegeben. Das Konstruktorproblem und das Self Problem werden ebenfalls gelöst.
- Durch Verwendung von Elementen der AOP ist die adäquate Behandlung homogener Crosscuts möglich. Das Feature Optionality Problem kann ebenfalls zum Teil gelöst werden.
- Aus der Verbindung der AOP und FOP ergeben sich weitere Vorteile:
  - Gute Erweiterbarkeit trotz der Verwendung von AOP,
  - Anwendung von FOP auf Aspekte und dadurch Vorteile z. B. in Bezug auf Crosscuts zwischen Merkmalen und
  - bessere Steuerung der Wirkung von Aspekten.
- Des Weiteren ergeben sich Vorteile aus der Berücksichtigung C++ spezifischer Eigenschaften:
  - Verwendung von Templates und damit die Möglichkeit bestimmte Merkmalsdimensionen nach der Konfiguration festlegen zu können,
  - Unterstützung von Mehrfachvererbung,
  - Vereinfachung der Programmierung durch automatisches Generieren von C++ Code für technische Zwecke (z. B. Includes).

Bei der Entwicklung des Prototyps wurden Aspectual Mixin Layers zur Umsetzung der Integration der AOP verwendet. Es wurde festgestellt, dass die Umsetzung in eine Klassenhierarchie, wie es mit dem derzeitigen Prototyp der Fall ist, einige Probleme aufwirft. Dazu gehören rein praktische Probleme, wie die Notwendigkeit der Verwendung virtueller Funktionen aber auch grundsätzliche Probleme, wie der Verlust der Assoziativität des Verfeinerungsoperators.

In der Analyse der Umsetzung wurden Probleme untersucht, die Gegenstand aktueller Forschung sind. An einem Beispiel wurde gezeigt, dass FeatureC++ auch unter praktischen Gesichtspunkten wesentliche Vorteile gegenüber den derzeit existierenden Möglichkeiten zur Produktlinienentwicklung besitzt. Dies trifft insbesondere auf Ansätze zu, die C++ verwenden. Einige vorgeschlagene Erweiterungen, wie die Verwendung von AOP Konzepten in Verfeinerungen von Klassen, vermögen diese Probleme zum Teil zu

lösen. Es wurde aber auch gezeigt, dass schon jetzt Vorteile gegenüber reinen Ansätzen der FOP und AOP, sowie kombinierten Ansätzen bestehen.

Die vorgestellte Erweiterung der FOP ist keineswegs an C++ gebunden. Die Anwendung auf andere Programmiersprachen ist ebenfalls möglich. Dadurch lässt sich die Konfiguration auf Software anwenden, deren Komponenten mit unterschiedlichen Programmiersprachen entwickelt werden. In Zusammenhang mit der ATS ist dies bereits jetzt möglich.

Mit der in dieser Arbeit entwickelten Lösung kann erstmals effektive merkmalsorientierte Programmierung in C++ erfolgen. Damit ist die Voraussetzung für die Entwicklung von Produktlinien gegeben. Durch die vorgeschlagene Erweiterung der FOP auf AOP Konzepte ergeben sich Vorteile gegenüber anderen Lösungen. Weiterhin kann der entwickelte Prototyp als Grundlage praktischer Untersuchungen aktueller Probleme verwendet werden.

Anhang A

Fallstudie

Im Folgenden wird der vollständige Programmcode des Beispiels aus Abschnitt 4.4.1 aufgeführt. Die systemabhängigen Implementierungen der Klasse `Mutex` sowie die Zeitmessung im Aspect `Profiler` sind nur angedeutet. Der Quelltext wurde mit FeatureC++ v0.3 getestet.

A.1 Klasse String

Layer Char

```
1 class String {
2 public:
3   //Typdefinition des verwendeten Characters
4   typedef char Elem;
5   typedef Char StringFunctions;
6 };
```

Layer WChar

```
1 class String {
2 public:
3   //Typdefinition des verwendeten Characters
4   typedef wchar_t Elem;
5   typedef WChar StringFunctions;
6 };
```

Layer Data

```
1 refines class String {
2   Elem* m_Buf;
3   int m_Length;
4   int m_BufSize;
```

```
5  public:
6    String()
7      :m_Buf(0),
8       m_Length(0),
9       m_BufSize(0) {}
10   virtual ~String() {
11     if (m_Buf)
12       free(m_Buf);
13   }
14
15   int length() {
16     return m_Length;
17   }
18   const Elem* c_str() {
19     return m_Buf;
20   }
21
22   int CompareNoCase(const Elem* other) {
23     return StringFunctions::stricmp(m_Buf,other);
24   }
25
26 protected:
27   void AddData(const Elem* other, int l) {
28     AllocBuf(m_Length+l+1);
29     memcpy(m_Buf+m_Length, other, l*sizeof(Elem));
30     m_Length+=l;
31     m_Buf[m_Length]=0;
32   }
33   void SetData(const Elem* other, int l) {
34     m_Length=0;
35     AddData(other, l);
36   }
37   void AllocBuf(int newSize) {
38     if (newSize <= m_BufSize)
39       return;
40
41     int addSize = newSize - m_BufSize;
42
43     int oldBufSize = m_BufSize;
44     m_BufSize+=addSize;
45     Elem *newBuf = (Elem*)malloc(m_BufSize * sizeof(Elem));
46
47     //nur altes kopieren wenn laenge > 0
48     if (m_Length)
49       memcpy(newBuf,m_Buf,(m_Length+1) * sizeof(Elem));
50     else
51       newBuf[0]=0;  //sonst letztes zeichen auf \0
52
53     //alten loeschen
54     if (oldBufSize)
55       free(m_Buf);
56
57     //zeiger umsetzen
58     m_Buf=newBuf;
59   }
60 };
```

Layer Sync

```
refines class String {
  Mutex m_Sync;
};
```

Layer Public

```
#define MAX_INT_STR_LEN (64 * 31 / 100) + 1 + 1

refines class String {
public:
  String() {  }
  String(const Elem* val) {
    operator << (val);
  }

  operator const Elem* () {
    return c_str();
  }

  String& operator = (const Elem* src) {
    SetData(src,StringFunctions::len(src));
    return *this;
  }

  String& operator << (const Elem* other) {
    AddData(other,StringFunctions::len(other));
    return *this;
  }

  String& operator << (int val)
  {
    int sz = MAX_INT_STR_LEN;
    Elem buf[MAX_INT_STR_LEN+1];
    return *this << StringFunctions::itos(val,buf,10);
  }
};
```

A.2 Klasse Char

Layer Char

```
class Char {
public:
  static int len(const char* s) {
    return ::strlen(s);
  }
};
```

Layer Windows

```
1  #include <string.h>
2  #include <stdlib.h>
3
4  refines class Char {
5  public:
6    static int stricmp(const char* s1, const char* s2) {
7      return ::_stricmp(s1,s2);
8    }
9    static char * itos(int val, char *buf, int radix) {
10     return ::_itoa(val,buf,radix);
11   }
12 };
```

Layer Unix

```
1  #include <string.h>
2  #include <ctype.h>
3
4  refines class Char {
5  public:
6    static int stricmp(const char* s1, const char* s2) {
7      return ::strcasecmp(s1,s2);
8    }
9    static char * itos(int val, char *buf, int radix) {
10     return ::_itoa(val,buf,radix);
11   }
12 };
```

A.3 Klasse WChar

Layer WChar

```
1  class WChar {
2  public:
3    static int len(const wchar_t* s) {
4      return ::wcslen(s);
5    }
6  };
```

Layer Windows

```
1  #include <string.h>
2  #include <stdlib.h>
3  #include <wchar.h>
4
5  refines class WChar {
6  public:
```

```
7   static int stricmp(const wchar_t* s1, const wchar_t* s2) {
8     return ::_wcsicmp(s1,s2);
9   }
10
11  static wchar_t * itos(int val, wchar_t *buf, int radix) {
12    return ::_itow(val,buf,radix);
13  }
14 };
```

Layer Unix

```
1  #include <string.h>
2  #include <ctype.h>
3
4  refines class WChar {
5  public:
6
7    static int stricmp(const wchar_t* /*s1*/, const wchar_t* /*s2*/) {
8      return ::_wcsicmp(s1,s2);
9    }
10
11   static wchar_t * itos(int val, wchar_t *buf, int radix) {
12     return ::_itow(val,buf,radix);
13   }
14 };
```

A.4 Klasse Mutex

Layer Sync

```
1  class Mutex {
2  public:
3    int m_nLockCount;
4    Mutex()
5      :m_nLockCount(0) {}
6
7    void lock() {
8      printf("lock\n");
9      execute_lock();
10     m_nLockCount++;
11   }
12
13   void unlock() {
14     printf("unlock\n");
15     m_nLockCount--;
16     execute_unlock();
17   }
18
19 protected:
20   virtual void execute_lock()=0;
21   virtual void execute_unlock()=0;
22 };
```

Layer Windows

```
1  refines class Mutex {
2  public:
3    virtual void execute_lock() {
4      //execute system lock function
5    }
6
7    virtual void execute_unlock() {
8      //execute system unlock function
9    }
10 };
```

Layer Unix

```
1  refines class Mutex {
2  public:
3    virtual void execute_lock() {
4      //execute system lock function
5    }
6
7    virtual void execute_unlock() {
8      //execute system unlock function
9    }
10 };
```

A.5 Aspekt Profiler

Layer Profile

```
1  #include <stdio.h>
2  aspect profiler {
3    pointcut virtual profile() = execution("% String::%(...)");
4    advice profile() : around() {
5      double t = 0;   //get actual time - dummy impl.
6      tjp->proceed();
7      double dt = 0 - t;  //calculate execution time - dummy impl.
8      printf("Execution Time (%s): %.3f\n",JoinPoint::signature(), dt / 1000.0);
9    }
10 };
```

Layer Sync

```
1  refines aspect profiler {
2    pointcut virtual profile() = execution("void Mutex::lock(...)") || super::profile();
3  };
```

A.6 Aspekt Sync

Layer Sync

```
1 aspect Sync {
2   pointcut str_sync() = execution("% String::%(...)");
3   advice str_sync() : around() {
4     ((String*)tjp->that())->m_Sync.lock();
5     tjp->proceed();
6   }
7 };
```

Literaturverzeichnis

[AB05] • Apel, S.; Böhm, K.: Towards the Development of Ubiquitous Middleware Product Lines. In *ASE'04 SEM Workshop*, LNCS, Band 3437. Springer, 2005.

[ALRS05a] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution. In *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05), in 19th European Conference on Object-Oriented Programming*, 2005.

[ALRS05b] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of Fourth International Conference on Generative Programming and Component Engineering*, 2005.

[Bat88] Batory, D.: Concepts for a DBMS synthesizer. In *ACM Principles of Database Systems Conference*. IEEE Computer Society Press, 1988.

[Bat98] Batory, D.: Product-Line Architectures, 1998.

[BC90] Bracha, G.; Cook, W.: Mixin-based inheritance. In *Proceedings of the 8th Conference on Object-Oriented Programming, Systems, Languages and Applications / European Conference on Object-Oriented Programming*, 1990.

[BCGS95] Batory, D.; Coglianese, L.; Goodwin, M.; Shafer, S.: Creating Reference Architectures: An Example from Avionics. In *Symposium on Software Reusability*, 1995.

[BCS00] Batory, D.; Cardone, R.; Smaragdakis, Y.: Object-Oriented Frameworks and Product-Lines, 2000.

[Big94] Biggerstaff, T.: The Library Scaling Problem and the Limits of Concrete Component Reuse. In *Proceedings of the 3rd International Conference on Software Reuse*. IEEE Computer Society Press, 1994.

[BLS03] Batory, D.; Liu, J.; Sarvela, J. N.: Refinements and Multi-Dimensional Separation of Concerns, 2003.

[BO92] Batory, D.; O'Malley, S.: The Design and Implementation of Hierarchical Software Systems with Reusable Components. In *ACM Transactions on Software Engr. and Methodology,*, 1992.

[BOW98] Bruce, K.; Odersky, M.; Wadler, P.: A statically Safe Alternative to Virtual Types. In *Proceedings European Conference on Object-Oriented Programming*, 1998.

[Bru02] Bruce, K. B.: *Foundations of Object-Oriented Languages*. MIT Press, 2002.

[BT97] Batory, D.; Thomas, J.: P2: A Lightweight DBMS Generator. *Journal of Intelligent Information Systems*, Band 9, Nr. 2, 1997.

[CBML02] Cardone, R.; Brown, A.; McDirmid, S.; Lin, C.: Using Mixins to Build Flexible Widgets. In *AOSD*, 2002.

[CC04] Colyer, A.; Clement, A.: Large-Scale AOSD for Middleware. AOSD, 2004.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming: Methods, Tools, and Applications*. Addison Wesley, 2000.

[CL01] Cardone, R.; Lin, C.: Comparing Frameworks and Layered Refinement. In *Proceedings of ICSE*, 2001.

[Dij72] Dijkstra, E. W.: The Humble Programmer. *CACM*, Band 15, Nr. 10, 1972.

[EBC00] Eisenecker, U. W.; Blinn, F.; Czarnecki, K.: A Solution to the Constructor-Problem of Mixin-Based Programming in C++. GCSE 2000 Workshop on C++ Template Programming, 2000.

[ES90] Ellis, M. A.; Stroustrup, B.: *The annotated C++ reference manual*. Adison-Wesley, 1990.

[GHJV95] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gri98] Griffel, F.: *Componentware*. dpunkt.verlag, 1998.

[HC91] Hooper, J.; Chester, R.: *Software Reuse*. Plenum Press, 1991.

[KCH+90] Kang, K.; Cohen, S.; Hess, J.; Nowak, W.; Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technischer Bericht, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[KHH+01] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G.: An Overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer Verlag, 2001.

[KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J.: Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, S. 220–242. Springer, 1997.

[LAS01] Leich, T.; Apel, S.; Saake, G.: Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems*. Springer, 2001.

[LBN05] Liu, J.; Batory, D.; Nedunuri, S.: Modeling Interactions in Feature Oriented Software Designs. ICFI, 2005.

[LBS04] Lohmann, D.; Blaschke, G.; Spinczyk, O.: Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In *GPCE*, 2004.

[LHBC05] Lopez-Herrejon, R. E.; Batory, D.; Cook, W.: Evaluating Support for Features in Advanced Modularization Technologies. In *European Conference on Object-oriented Programming*, 2005.

[Lie86] Liebermann, H.: Using prototypical objects to implement shared behavior in object-oriented systems. Technischer Bericht, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Mass. 02139, USA, 1986.

[LLO03] Lieberherr, K.; Lorenz, D. H.; Ovlinger, J.: Aspectual Collaborations: Combining Modules and Aspects. Technischer Bericht, College of Computer and Information Science, Northeastern University, 360 Huntington Avenue, Boston, USA, 2003.

[MFH01] McDirmid, S.; Flatt, M.; Hsieh, W. C.: Jiazzi: NewAge Components for OldFashioned Java, 2001.

[MH03] McDirmid, S.; Hsieh, W. C.: AspectOriented Programming with Jiazzi, 2003.

[MO04] Mezini, M.; Ostermann, K.: Variability Management with FeatureOriented Programming and Aspects. *ACM SIGSOFT*, 2004.

[NAT94] NATO: Standard for the Development of Reusable Software Components. Technischer Bericht, GTE Government Systems, 1994.

[Ost01] Ostermann, K.: Implementing Reusable Collaborations with Delegation Layers. In *Proceedings OOPSLA'01*, 2001.

[OT00] Ossher, H.; Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach, 2000.

[Par72] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. In *Communications of the ACM*, 1972.

[Par76] Parnas, D.: On the design and development of program families. In *IEEE Transactions on Software Engineering*, 1976.

[Par79] Parnas, D. L.: Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, Band SE-5, Nr. 2, 1979.

[pur01] pure-systems GmbH: *PUMAUsers Manual*, 2001.

[pur04] pure-systems GmbH: *AC++ Compiler Manual*, 2004.

[SB00] Smaragdakis, Y.; Batory, D.: Mixin-Based Programming in C++. In *GCSE*, 2000.

[SB93] Singhal, V.; Batory, D.: P++: A Language for Software System Generators. Technischer Bericht, Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, 1993.

[SGSP02] Spinczyk, O.; Gal, A.; Schröder-Preikschat, W.: AspectC++: An Aspect-Oriented Extension to C++. In *TOOLS Pacific*, 2002.

[Sic04] Sichting, H.: Middleware-architektur für mobile informationssysteme. Master's thesis, Otto-von-Guericke-Universität Magdeburg, 2004.

[Sma99] Smaragdakis, Y.: Implementing Large-Scale Object-Oriented Components, 1999.

[Spi03] Spinczyk, O.: Aspektorientierung und Programmfamilien im Betriebssystembau. *Lecture Notes in Informatics (LNI) – Dissertations*, 2003.

[SR03] Sutton, S. M.; Rouvellou, I.: Applicability of Categorization Theory to Multidimensional Separation of Concerns. Technischer Bericht, College of Computer and Information Science, Northeastern University, 360 Huntington Avenue, Boston, USA, 2003.

[Str86] Stroustrup, B.: *The C++ Programming Language.* Adison-Wesley, 1986.

[Szy97] Szyperski, C.: *Component Software - Beyond Object Oriented Programming.* Addison Wesley, 1997.

[Tho97] Thorup, K.: Genericity in Java with Virtual Types. In *Proceedings European Conference on Object-Oriented Programming*, 1997.

[TO00] Tarr, P.; Ossher, H.: Hyper/J User and Installation Manual, 2000.

[TOHS99] Tarr, P.; Ossher, H.; Harrison, W.; Sutton, S. M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, 1999.

[Wit96] Withey, J.: Investment Analysis of Software Assets for Product lines. Technischer Bericht, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996.

[Zam99] Zamir, S.: *Handbook of Object Technology.* CRC Press, 1999.

Selbständigkeitserklärung

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

Magdeburg, den 14. August 2005

Marko Rosenmüller