Eine familienbasierte Infrastruktur zum dynamischen Weben von Aspekten in C++-Programmen

Diplomarbeit im Fach Informatik

vorgelegt von

Reinhard Tartler

geb. am 20.08.1980 in Nürnberg

Angefertigt am

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Prof. Dr. Wolfgang Schröder-Preikschat** **Dr.-Ing. Olaf Spinczyk**
Beginn der Arbeit:	23. Juni 2006
Abgabe der Arbeit:	17. Dezember 2006

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 17. Dezember 2006 _____

Kurzzusammenfassung

Aspektorientierte Programmierung (AOP) führt neue Techniken zur Modularisierung von quer schneidenden Belangen ein, was eine neue Art von Abstraktion bei der Entwicklung von Software bedeutet. Während das Prinzip der AOP keine Aussagen über den Zeitpunkt deren Anwendung macht, arbeiten die momentan verfügbaren Weber für die Programmiersprache C++ ausschließlich für den Zeitpunkt der Übersetzung der Anwendung. Diese Arbeit tritt den Versuch an, auf Basis der aspektorientierten Programmiersprache AspectC++ einen dynamischen Aspektweber zu entwickeln, bei dem derselbe Aspekt sowohl statisch als auch dynamisch gewoben werden kann. Dabei werden die technischen Schwierigkeiten der Umsetzung diskutiert und die Anwendbarkeit an bestehenden Anwendungen geprüft.

Abstract

Aspect-oriented programming (AOP) introduces new techniques for separating concerns into modules, which means a new kind of abstraction in the development of software. While the principle of AOP does not make statements about the time of its application, current weavers for the programming language C++ work exclusively at compilation time of the application. Using the aspect-oriented programming language AspectC++, this work attempts to develop a dynamic aspect weaver with which the same aspect can be woven both statically and dynamically. Thereby the technical difficulties are discussed, and the practicality is tested with existing applications.

Inhaltsverzeichnis

1 Einleitung und Motivation

1.1 Motivation

Ein Grundprinzip zur Bewältigung der Komplexität bei der Entwicklung von Software ist, nicht mehrere Probleme gleichzeitig anzugehen, sondern immer nur eines zu behandeln. Dafür sind die Belange der Software so aufzuteilen, dass jeder Belang in einem einzelnem Modul isoliert werden kann, ohne dass detaillierte Kenntnisse über andere Module bei dessen Entwicklung notwendig sind.

Dieses *Prinzip der Trennung der Belange* (engl. *separation of concerns*) funktioniert mit traditionellen Entwicklungstechniken wie der objektorientierten Programmierung für viele funktionale Anforderungen sehr gut. Es können konzeptionell zusammenhängende Datenstrukturen und dazugehörige Funktionen in *Klassen* gekapselt werden. Diese Isolation vereinfacht die Wartung, Erweiterung und Adaption an andere Module einer funktional zusammenhängenden Komponente.

Bei der Entwicklung von komplexen Softwaresystemen bietet *aspektorientierte Programmierung* (AOP) [KLM+97] neue Möglichkeiten zur Abstraktion, die die Modularisierung von quer schneidenden Belangen erlaubt. Diese Art von Belangen lässt sich nicht in einzelnen Klassen kapseln, sondern betrifft eine Vielzahl von Komponenten. Aspektorientierte Programmierung erweitert den Ansatz der objektorientierten Programmierung um Sprachkonzepte und Werkzeuge, die sich zur Kapselung quer schneidender Belange gut eignen. Quer schneidende Belange werden dabei in *Aspekte* gekapselt, die an festgelegten Stellen in den Programmfluss *eingewoben* werden.

Das Prinzip der aspektorientierte Programmierung macht keine Aussage darüber, ob Aspekte dynamisch zur Laufzeit oder statisch zur Übersetzungszeit gewoben werden sollen. Es ist jedoch gerade bei statisch getypten Sprachen wie `C` oder `C++` so, dass Implementierungen für AOP zur Laufzeit wesentlich weniger und andere Sprachmerkmale anbieten als Implementierungen, die ganz zur Übersetzungszeit arbeiten. Dies erschwert die Umsetzung von Szenarien, welche sowohl eine statische als auch dynamische Umsetzung der AOP benötigen.

Viele Softwareprogramme nutzen die Möglichkeit, Funktionalität zur Laufzeit nachzuladen. Meistens handelt es sich dabei um Funktionalität, die sich in einzelnen Objektdateien kapseln lässt. Quer schneidende Funktionalitäten wie Sicherheits-, Performance- oder Überwachungsbelange lassen sich mit

herkömmlichen Ansätzen jedoch nur sehr schwer (und in vielen Fällen gar nicht) umsetzen. Genau dafür führt die aspektorientierte Programmierung das Sprachkonzept des *Aspekts* ein, der einen *quer schneidenden Belang* in einzelne programmiersprachliche Entitäten kapselt. Diese Technik verspricht komplexe Systeme um solche Aspekte elegant, effizient und wartbar erweitern zu können.

1.2 C++ und AOP

Komplexe Softwaresysteme verwenden immer häufiger das objektorientierte Paradigma bei Planung, Entwurf, Implementierung. Dabei kommt immer wieder der Vorschlag auf, Konzepte der aspektorientierten Programmierung mit bereits bestehenden Techniken in objektorientierten Sprachen wie `C++` zu implementieren.

`C++` selbst bietet mit seiner Template-Sprache Möglichkeiten, Klassen weitgehend zu parametrisieren, wie es z.B. Alexandrescu in [Ale01] vorschlägt. Um quer schneidende Belange zu kapseln schlägt Alexandrescu dazu die Technik *Policy Based Design* vor. Diese Techniken können in Grenzen ebenfalls genutzt werden, um einzelne Konzepte der *aspektorientierten Programmierung* umzusetzen. Für eine vollständige Umsetzung der AOP in `C++` kommt man aber nicht um eine Spracherweiterung wie AspectC++ [SGSP02] umhin [SL06]. Alexandrescus Techniken beruhen auf Kooperation mit den Klassen, die zu parametrisieren sind. Eine Grundeigenschaft der AOP ist jedoch die so genannte *Obliviousness* [FF00], also die Eigenschaft, dass sich der zu parametrisierende Code nicht über die Modifikationen bewusst sein muss.

Die Programmiersprache AspectC++ schreibt keineswegs vor, dass Aspekte nur statisch in einem Programm wirken können. Vielmehr skizzieren Wasif Gilani und Olaf Spinczyk in [GS05], wie eine Infrastruktur zum dynamischem *Weben* von Aspekten aussehen könnte. Diese Arbeit realisiert diesen Gedanken in Form eines familienbasierten Ansatzes. Dabei wird aufgezeigt, welche AOP Merkmale sich besonders gut vom statischem Modell ins dynamische umsetzen lassen, und wie viel Aufwand dafür betrieben werden muss.

1.3 Anpassbare und erweiterbare Systeme

Komplexe Softwaresysteme wie Betriebs- oder Datenbanksysteme nutzen häufig die Möglichkeit, zur Laufzeit zusätzliche Programmmodule nachzuladen. Diese Programmmodule können dabei auch zu einem späterem Zeitpunkt als

das System an sich entwickelt worden sein. So unterstützt beispielsweise der Linux Kern Erweiterungen über *ladbare Kern Module* (LKM), über die zusätzliche Funktionalität wie Gerätetreiber nachgeladen werden können. Der Linux Kern stellt Infrastruktur zum laden, registrieren, entladen von LKMs bereit. Die Module nutzen dazu vordefinierte Schnittstellen im Kern; es entsteht dadurch eine Art Kooperationsvertrag zwischen den Modulen und dem Kern.

AOP zur Laufzeit verspricht eine elegantere Lösung dieses Problems. Hier sind auf der Ebene der Basisanwendung keine Anpassungen für die Nutzung von Erweiterungsmodulen nötig. Der Aspektweber kümmert sich um die Aktivierung von Programmmodulen an theoretisch beliebigen Stellen im Programmfluss. Durch diese Flexibilität werden ganz neue Arten von Erweiterungen möglich. Während klassische Module sehr fokussiert Erweiterungen in ein System einbringen, wie besagte Gerätetreiber oder Unterstützung für neue Dateisysteme, können nachladbare Module mit AOP Merkmalen wesentlich weitreichendere Eingriffe im System vornehmen.

1.4 Struktur der Arbeit

Diese Arbeit gliedert sich wie folgt: Zunächst wird eine Einführung in das Vokabular der Sprache AspectC++ gegeben. Das darauf folgende Kapitel gibt einen Überblick über die aktuell verwendeten Techniken zur Implementierung von aspektorientierten Techniken im Umfeld der Sprache `C++`. Dabei wird schon angedeutet, dass *echte* AOP zur Laufzeit nur mit aufwändiger Infrastruktur zu bewerkstelligen ist.

Kapitel 4 stellt dann die im Rahmen dieser Arbeit entwickelte Infrastruktur vor. Es wird auf die einzelnen Komponenten ebenso wie deren Zusammenspiel eingegangen. Dabei ergibt sich auch ein möglicher Arbeitsablauf für die Entwicklung von Aspektmodulen, die zur Laufzeit gewoben werden können. Die folgenden Kapitel stellen dann die einzelnen Schwierigkeiten bei der Umsetzung der Infrastruktur vor, und erklären die verwendeten Konzepte. In Kapitel 8 wird aufgezeigt, warum sich ein familienbasierter Ansatz für dynamische Aspektweber anbietet und welche Vorteile er bringt. In Kapitel 9 werden die Infrastruktur sowie die Abläufe bei der Übersetzung eines Softwareprojekts mit dynamisch gewobenen Aspekten vorgestellt. Dabei wird die Umsetzung der einzelnen umgesetzten Sprachmerkmale demonstriert, analysiert und evaluiert.

In Kapitel 10 wird der entsehende Aufwand beim dynamischem Weben von Aspekten durch die entwickelte Infrastruktur in Form von Microbenchmarks

gemessen. Anschließend wird beispielhaft gezeigt, wie Anwendungen, die die Bibliothek ACE verwenden sowie der Proxyserver Squid, sinnvoll mit Aspekten zur Laufzeit erweitert werden können. Abschließend fasst ein Fazit die wichtigsten Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf möglicherweise weiter darauf aufbauende Arbeiten.

2 AOP mit AspectC++

AspectC++ [SLU05] ist eine vielseitig einsetzbare Spracherweiterung zu ISO C++. Als Sprache ist AspectC++ noch recht jung: Die ersten Ideen zur Sprache AspectC++ wurden im Jahre 2001 veröffentlicht. Sie stellt, gemessen an ihrem Sprachumfang und der Art und Weise ihrer Sprachmerkmale, eine ähnliche Erweiterung zu C++ wie AspectJ [KHH+01] zur Programmiersprache Java. Anstatt aber alle Eigenheiten von AspectJ zu übernehmen, wurde bei AspectC++ darauf geachtet, eine natürliche Erweiterung zu C++ zu sein, die sich gut in die Syntax von C++ eingliedert. Ziel ist es, dem Programmierer das Erlernen von AspectC++ zu erleichtern.

Als universelle Programmiersprache ist AspectC++ überall dort einsetzbar, wo auch C++ erfolgreich eingesetzt werden kann. Dies schließt sowohl sehr umfangreiche Softwareprojekte als auch Softwareprojekte für Systeme mit sehr begrenzten Betriebsmittelressourcen ein. Wichtiges Entwurfsziel von AspectC++ ist es, den kompletten Sprachumfang von C++ zu unterstützen. Dies bedeutet insbesondere, dass C++ *Templates* voll unterstützt werden, auch wenn dies den Parser deutlich schwieriger macht.

Dieses Kapitel stellt das von AspectC++ eingeführte Vokabular zur Umsetzung von aspektorientierter Programmierung vor und gibt einen Überblick über die angebotenen Sprachmerkmale. Es werden die wichtigsten Begriffe aus dem AspectC++-Jargon anhand eines Standardbeispiels für aspektorientierte Programmiersprachen präsentiert: Ein *tracing aspect*, dessen Programmausschnitt in Abbildung 1 aufgeführt ist.

```
1  #include <cstdio>
2  aspect Trace {
3    unsigned count__;
4
5    pointcut tracepoints() = execution("% ...::Buggy::%(...)");
6
7    advice tracepoints() : around() {
8      printf("[%d] Now executing: '%s'", count__,
9             %JoinPoint::signature());
10     tjp->proceed();
11     printf("[%d] method '%s' finished", count__,
12            %JoinPoint::signature());
13     ++count__;
14   }
15 };
```

Abbildung 1: Implementierung eines *tracing aspect* in der Sprache AspectC++

2.1 *pointcuts* und *match expressions*

Von allen Konzepten sind *pointcuts* wohl die wichtigsten in AspectC++. Sie legen fest, an welchen Stellen im Programm oder im Programmfluss Aspekte wirken sollen.

Prinzipiell gibt es in AspectC++ zwei Arten von Pointcuts: *name pointcuts* und *code pointcuts*. Erstere markieren Stellen in der statischen Programmstruktur wie Funktionen, Klassen und deren Methoden. Sie werden durch *match expressions* festgelegt, anhand derer Typen und Signaturen von Funktionen und Typbezeichnern gesucht werden können.

match expressions werden in AspectC++ als Zeichenketten dargestellt. So meint beispielsweise die Zeichenkette `"% ...::foo(int)"` jede Methode `foo()` unabhängig von Rückgabewert und in jedem Namensraum, die genau einen Funktionsparameter von Typ `int` erhält. `"% ...::operator %(...)"` trifft hingegen auf jede Typumwandlungsfunktion (oder -methode) sowie alle Operatoren in C++ zu. Die Zeichen `%` und `...` sind hierbei als Platzhalter für Bezeichnernamen bzw. für Platzhalter zu verstehen.

code pointcuts beschreiben das Auftreten eines Ereignisses im Programmfluss zur Laufzeit wie z.B. das Aufrufen einer Funktion oder die Instantiierung einer Klasse. Geeignete *pointcut*-Funktionen führen *name pointcuts* in *code pointcuts* über. Der *code pointcut* `call("void ...::measurePoint(...)")` beschreibt zum Beispiel den Aufruf jeder Funktion `measurePoint(...)`, die `void` als Rückgabewert hat, unabhängig ihrer Funktionsparameter. Im Beispielaspekt trifft daher die *match expression* in Zeile 5 auf jede Methode der Klasse `buggy` in jedem Namensraum zu.

In AspectC++ gibt es verschiedene Arten von *pointcut*-Funktionen: Die wichtigsten sind solche, die *name pointcuts* nach *name pointcuts* überführen und welche, die *name pointcuts* nach *code pointcuts* überführen.

2.2 *join points*

Ein *join point* ist eine Stelle im Kontrollflussfluss eines Programms, an denen Aspekte auf das Verhalten des Programms Einfluss nehmen können. Diese Stellen im Kontrollfluss haben Entsprechungen im Programmtext der Anwendung. Beim Aufruf einer Funktion wäre das der Funktionsaufruf oder bei der Ausführung einer Funktion die Zeilen ihrer Definition. Diese Stellen bzw. Abschnitte im Programmtext werden auch *join point shadows* genannt.

In AspectC++ unterscheidet man zwischen zwei verschiedenen Arten von

join points. Dabei versteht man unter einem *name join point* ein Element der Programmstruktur wie Namensraum, Typ, Funktion und Methode. Die andere Art wird *code join point* genannt. Unter diesem versteht man eine Stelle im Programm, an denen Ereignisse wie Instantiierung von Typen, Freigabe von Objekten, das Aufrufen von Funktionen oder Methoden sowie den Beginn der Ausführung einer Funktion oder Methode, auftreten können. *pointcuts* beschreiben immer eine Menge von *join points*.

Im Beispiel in Abbildung 1 kennzeichnet der *pointcut* aus Zeile 5, repräsentiert durch die Zeichenkette `"% ...::buggy::%(...)"`, diejenigen *name join points*, welche sich in jeder Methode der Klasse `buggy` befinden. Die *pointcut*-Funktion *execution* überführt diese *join points* in *execution join points* um, welche jede Ausführung dieser Methoden spezifizieren.

pointcut- Funktionen können durch logische Operatoren verknüpft werden. Dabei werden Mengenoperationen auf die von den *pointcuts* spezifizierten *join point* Mengen durchgeführt. Der Operator `&&` führt die Schnittmengenbildung und der Operator `||` die Vereinungsmenge von den durch die *pointcuts* spezifizierten *join point* Mengen aus.

2.3 Advice

In AspectC++ definiert *Advice* das Verhalten von Aspekten an einer Menge von *join points*. Die Deklaration von *Advice* erwartet eine *pointcut expression* zur Festlegung von denjenigen *join points*, an die *Advice* gebunden werden soll. Die Deklaration wird begleitet von einem Codefragment, welches angibt, ob der *Advice Code* bevor (`before()`, danach (`after()`) oder anstatt (`around()`) ausgeführt werden soll. Im Rumpf der Deklaration kann beliebiger Programmtext stehen. Dieser Programmtext kann zur Implementierung von den quer schneidenden Belangen verwendet werden. Daher nennt man *Advice* wie hier beschrieben auch *Code Advice*. Den in *Code Advice* enthaltenen Programmcode nennt man *Advice Code*. Er wird an allen *join points* ausgeführt, die bei der Advicedeklaration festgelegt wurden.

Advice ist das zentrale Sprachmittel zur Umsetzung von quer schneidenden Belangen. Die Anwendungen müssen dabei nicht auf den Advice vorbereitet werden. Vielmehr werden die *join points* eines Programms mit Hilfe von *pointcuts* an den Advicedeklarationen beschrieben. Die *join point shadows* sind also im Programmtext nicht sichtbar, sondern können nur mittels Werkzeugen *visualisiert werden*. Diese Eigenschaft nennt man im AOP Kontext auch *obliviousness*. [FF00]

Im Beispielaspekt in Abbildung 1 wird in Zeile 7 *around*-Advice definiert, der vor und nach jeder Ausführung (*pointcut*-Funktion `execution`) von Methoden der Klasse `buggy` eine Ausgabe mit der Signatur der ausgeführten Funktion und der bisherigen Anzahl der Ausführungen dieses *Advice Codes* ausgibt.

2.4 Aspekte

Aspekte modularisieren quer schneidende Belange in Form einer programmiersprachlicher Entität. In AspectC++ wird hierfür das Schlüsselwort `aspect` verwendet. Ein Aspekt ist dabei vom Aufbau und der Syntax her einer Klasse in C++ sehr ähnlich. In einem Aspekt können alle Sprachelemente vorkommen, die auch in Klassen möglich sind. Dazu gehören Attribute, Methoden, Typaliase, Unterklassen (*nested classes)*, Aufzählungen *enums* oder Typaliase (*typedefs*). Es ist möglich, in einem Aspekt von einem anderem Aspekt oder einer anderen Klasse zu *erben*. Die Syntax ist dabei genauso wie bei der Vererbung von Klassen in der Sprache `C++`.

Zusätzlich werden in Aspekten Advice Deklarationen eingeführt, die an durch *join points* spezifierte Stellen im Programmfluss das Verhalten des Programms zu beeinflussen. Die Eigenschaft, dass Advice an theoretisch beliebig vielen *join points* gebunden werden kann, wird auch *quantification* genannt. [FF00]

2.5 Benannte und virtuelle *pointcuts*

Das Schlüsselwort `pointcut` kann zur Einleitung einer *pointcut* Deklaration verwendet werden, um einen so genannten *benannten pointcut* zu definieren. Diese können in einer oder mehreren Deklarationen von Advice verwendet werden, um eine Menge von *join points* zu spezifizieren. Diese Art von Deklaration wird im Beispielaspekt in Zeile 5 demonstriert.

Die Deklaration eines *pointcuts* kann mit dem Schlüsselwort `virtual` versehen werden. In diesem Fall ist es möglich, in abgeleiteten Aspekten eine *pointcut* Deklaration ähnlich wie virtuelle Methoden in `C++` zu *überschreiben*.

Falls ein *pointcut* definiert werden soll um lediglich eine Schnittstelle vorzugeben, ist es möglich einen *pointcut* ähnlich wie bei Methoden aus `C++` als *pure virtual* zu definieren. In diesem Fall folgt nach den Schlüsselworten `virtual pointcut` keine *pointcut expression*, sondern wie bei *pure virtual methods* ein `= 0`. Im AspectC++ Programmtext kann dies dann so aussehen:

```
1    aspect A { virtual pointcut instrumented_with_context () = 0; };
```

Aspekte die mindestens so einen Aspekt besitzen heißen *abstrakt*. Sie werden nicht *instantiiert*, d.h. sie wirken nirgendwo im Programm. Wenn ein Aspekt von einem *abstraktem Aspekt* erbt, kann er den *pure virtual pointcut* implementieren, indem er eine neue *pointcut* Deklaration einführt. Diese Technik kann dazu verwendet werden, um Funktionalität generisch zu implementieren, etwa in einer Art Aspektbibliothek. Die konkrete Anwendung dieses Aspekts kann dann in einem wesentlich kürzerem Aspekt festgelegt werden.

2.6 Das Joinpoint-API

Das *Joinpoint-API* stellt dem Programmierer eine definierte Schnittstelle für den Zugriff auf Kontextinformation zur Laufzeit im *Advice Code* bereit. Kontextinformation wird in Advice gebraucht um Advice zu implementieren, der sich je nach Ausführungskontext anders verhält. Im Beispielaspekt in Abbildung 1 wird beispielsweilse in den Zeilen 9 und 12 auf Kontext zugegriffen, der die Signatur der Funktion beinhaltet, in dem sich der *aktuelle join point* befindet. Zur Veranschaulichung sei der Aspekt aus Abbildung 1 auf das folgende Programm angewendet:

```
1  struct Buggy {
2    void tester1() {};
3    void tester2() {};
4  };
5  int main() {
6    Buggy b;
7    b.tester1();
8    b.tester2();
9  }
```

In diesem Programm wirkt der Aspekt aus Abbildung 1 an zwei *join points*, nämlich an den Ausführungen der zwei Methoden `Buggy::tester1()` und `Buggy::tester2()`. Wird der Aspekt in das Programm gewoben, werden bei dessen Ausführung folgende Ausgaben gemacht:

```
[0] Now executing: 'void Buggy::tester1()'
[0] method 'void Buggy::tester1()' finished
[1] Now executing: 'void Buggy::tester2()'
[1] method 'void Buggy::tester2()' finished
```

Man unterscheidet zwischen *statischen* Kontextinformationen, die zur Übersetzungszeit feststehen, und *dynamischen* Kontextinformationen, welche erst

Typen und Konstanten (statischer Kontext)	
`That`	Typ des Aufrufers
`Target`	Typ des aufgerufenen Objektes
`Result`	Rückgabetyp
`JPID`	eindeutige Nummer für diesen *join point*
`JPTYPE`	*join point* Art (`AC::CALL` oder `AC::EXECUTION`)
`ARGS`	Anzahl der Funktionsparameter
`Arg<i>::Type`	Typ des n-ten Funktionsparameters
Laufzeitfunktionen und Zustand (dynamischer Kontext)	
`that()`	Zeiger auf das Aufruferobjekt
`target()`	Zeiger auf das aufgerufene Objekt
`result()`	Zeiger auf einen Puffer mit dem Rückgabewert
`static const char *signature()`	Signatur der Funktion
`void *arg (int n)`	n-ter Funktionsparameter
`Arg<i>::ReferredType *arg()`	getypter Zeiger auf den n-ten Funktionsparameter
`void proceed()`	Fortführung in `around()`-Advice

Tabelle 1: Übersicht über die Joinpoint-API

zur Laufzeit bekannt sind. Tabelle 1 gibt einen Überblick über die *Joinpoint-API*.

Zu den statischen Kontextinformationen gehören beispielsweise die Anzahl und Typen der Funktionsparameter sowie der Rückgabetyp der Funktion, in der sich der *join point* befindet, in dessen Kontext der *Advice Code* ausgeführt wird. Alle diese Informationen haben die Eigenschaft, dass sie in Form von Typen und Konstanten vorliegen. AspectC++ stellt dazu einen impliziten Typ `JoinPoint` zur Verfügung, der diese Konstanten und Typen als Untertypen bereitstellt. Die dynamischen Kontextinformationen sind in Tabelle 1 aufgelistet. Dazu gehören z.B. die aktuell verwendeten Funktionsparameter oder der Rückgabewert. AspectC++ erlaubt den Zugriff auf den dynamischen Kontext über ein implizit bereitgestelltes Objekt mit dem Namen `tjp`.

```
1  #include <iostream>
2  using namespace std;
3
4  aspect retvaltracer {
5    advice tracepoints() : before() {
6      cout << *result () << endl;
7    }
8  };
```

Abbildung 2: Einfacher, generischer *tracing aspect*

2.7 Generic Advice

Generisches Programmieren bedeutet typunabhängiges Programmieren, also das Schreiben von Programmen, die unabhängig von den verwendeten Typen funktionieren. In C++ wird dazu Code aus Programmtext in Form einer Programmschablone (*template*) für jeden Typ erzeugt. Dies ermöglicht eine neue Form der Abstraktion: Eine Abstraktion von Typen.

Die Joinpoint-API stellt für jeden *join point* Typinformationen bereit. Damit ist es möglich, *generic advice* [LBS04] zu implementieren. Darunter versteht man *Advice*, der die statische Typinformation der Joinpoint-API benutzt.

generic advice ist dabei ein fundamentales Merkmal bei der Entwicklung von nicht-trivialem Advice. Der *tracing aspect* aus Abbildung 2 ist dabei insofern generisch, dass C++ anhand des zurückgegebenen Typs der Joinpoint-API Funktion `result()` zur Übersetzungszeit die passende Implementierung des Ausgabeoperators `operator<<` für die Ausgabe auf das Ausgabeobjekt `cout` wählt:

generic advice bietet durch seinen vielfältigen Vorrat aus Typinformationen umfassende Anwendungsmöglichkeiten für *template metaprogramming*, also das Schreiben vom C++ Teilprogrammen, die vollständig zur Übersetzungszeit vom C++ Übersetzer ausgeführt werden.

In [Tar05] wurde z.B. der in Abbildung 3 abgebildete generische Advice verwendet, um den Platzbedarf von Funktionsparametern zu ermitteln, auf den ein Aspekt wirkt. Das Template `ArgumentSizeCalculator` wird dabei aus dem *Advice Code* heraus mit dem von AspectC++ implizit bereit gestelltem Typen `JoinPoint` parametrisiert. Dabei kann das *template metaprogram* aus Abbildung 3 auf die Joinpoint-API aus Tabelle 1 zurückgreifen. In den Zeilen 4 und 7 werden Fallunterscheidungen für den Fall gemacht, dass der betrachtete Funktionsparameter eine Referenz ist. In diesem Fall wird nur der Platz für einen Zeiger, nicht der referenzierte Typ auf dem Stapelspeicher be-

```
1  #include <iostream>
2  aspect ArgumentSizePrinter {
3
4     template <typename T>
5     struct TypedSizeCalculator { enum { RESULT = sizeof(T) }; };
6
7     template <typename T>
8     struct TypedSizeCalculator<T&> { enum { RESULT = sizeof(T*) }; };
9
10    template <typename T> class ArgumentSizeCalculator {
11
12      template <typename TJP, int i> struct Sum_N {
13       enum { RESULT =
14         TypedSizeCalculator<typename TJP::template Arg<i-1>::Type>::RESULT
15           + Sum_N<TJP, i-1>::RESULT };
16        };
17
18      template <typename TJP> struct Sum_N <TJP, 0> {
19       enum { RESULT = 0};
20      };
21
22      public:
23      enum { RES = Sum_N<T, T::ARGS>::RESULT };
24    };
25
26    advice tracepoints() : before () {
27      cout << JoinPoint::signature() << " has " << JoinPoint::ARGS
28          << " parameters which need in sum "
29          << ArgumentSizeCalculator<JoinPoint>::RES
30          << " bytes on the stack."
31          << endl;
32    }
33 };
```

Abbildung 3: *generic advice* zur Berechnung des Platzbedarfs von Funktionsparametern

legt. Beide Funktionen benutzen den `sizeof()` Operator zur Ermittlung des Platzbedarfs. In Zeile 12 wird ein inneres *Hilfstemplate* verwendet, welches sowohl auf die Joinpointklasse als auch mit einer Ganzzahl parametrisiert wird. In Zeile 15 wird der Platzbedarf mit den *templates* aus den Zeilen 4 und 7 rekursiv berechnet. Das *template* aus Zeile 19 ist dabei der Rekursionsabbruch für den Fall, dass die Funktion keine Funktionsparameter besitzt. Das Ergebnis dieser Berechnung kann von Advice mittels dem Typ `RES` aus Zeile 23 abgefragt werden.

```
1  struct GeoBase { virtual ~GeoBase() {}; };
2
3  pointcut shapes() = "Circle" || "Polygon";
4
5  aspect Geometry{
6    advice shapes() : slice class : public GeoBase {
7      bool m_shaded;
8      void shaded(bool state) {
9        m_shaded = state;
10     }
11   };
12 };
```

Abbildung 4: Fortgeschrittenere Einfügungen

2.8 Einfügungen (*Introductions*)

Einfügungen können zur Erweiterung von Klassen um neue programmiersprachliche Elemente benutzt werden. In AspectC++ werden Einfügungen syntaktisch ebenfalls mit dem Schlüsselwort `advice` eingeleitet. Daher unterscheidet man in AspectC++ zwischen *Code Advice* und *Introduction Advice*. Neben Einfügungen von Attributen sind auch andere Arten von Einfügungen möglich. Dazu gehören Einfügungen von `friend`-Deklarationen, Methoden und Basisklassen. Abbildung 4 zeigt kompliziertere Einfügungen. In Zeile 1 wird eine Klasse `GeoBase` definiert, welche einen virtuellen Destruktor besitzt. Zeile 3 legt einen *pointcut* fest, der auf alle Klassen mit dem Namen `Circle` oder `Polygon` zutrifft. Der Aspekt `Geometry` leitet dann in Zeile 6 *Introduction Advice* ein, der insgesamt drei Einfügungen macht.

2.9 Weitere *pointcut*- Funktionen

Die Sprache AspectC++ stellt weitere *pointcut*- Funktionen zur Manipulation von *pointcuts* bereit. Man unterscheidet zwischen statisch und dynamisch evaluierbaren *pointcut*- Funktionen.

2.9.1 *pointcut*- Funktionen zur dynamischen Typprüfung

Die *pointcut*- Funktion `that(`*`type pattern`*`)` wird verwendet, um in *Advice Code* von *execution*-Advice zu prüfen, ob das Objekt der Methode zur *Ausführung* einem gegebenem Typmuster genügt. Der Benutzer kann damit *Advice Code* schreiben, der zum Beispiel nur dann zur Ausführung kommt, wenn die Methode aus einer abgeleiteten Klasse heraus aufgerufen wird. Die Pointcut-

```
1  class Bus {
2    void out (unsigned char);
3    unsigned char in ();
4  };
5
6  aspect BusIntSync {
7    pointcut critical() = execution("% Bus::%(...)");
8    advice critical() && !cflow(execution("% os::int_handler()")) : around()
         {
9      os::disable_ints();
10     tjp->proceed();
11     os::enable_ints();
12   }
13 };
```

Abbildung 5: Beispiel für cflow()

Funktion `target(type pattern)` arbeitet analog für *execution* Advice. Hier wird überprüft, ob eine aufgerufene Methode zu einem gegebenem Typmuster passt.

Wie ihr Name bereits andeutet, werden diese *pointcut*- Funktionen dynamisch zur Übersetzungszeit ausgewertet. Sie binden die Ausführung von *Advice* an zur Ausführungszeit relevante Typen. Dies ist dann durch syntaktische Analyse ermittelbar, wenn Zeigervariablen auf Instanzen abgeleiteter Klassen zeigen. Dies macht es möglich, die Ausführung von *Advice Code* auf konkrete Instanzen von bestimmten Klassen zuverlässig zu beschränken.

2.9.2 Kontrollfluss

Die *pointcut*- Funktion `cflow()` erlaubt es zur Laufzeit zu prüfen, ob ein bestimmter *join point* im Programmablauf durchschritten wurde. Damit wird der vergangene Verlauf des Kontrollflusses zu einer weiteren Form von abprüfbarem Kontext. Es sei das Beispiel in Abbildung 5 aus der AspectC++ Sprachreferenz betrachtet, welches die Aktivierung von *Advice Code* abhängig vom vergangenen Kontrollflussverlauf des Programms macht und die Verwendungsweise der Funktion `cflow()` demonstriert.

Es wird eine Klasse `Bus` deklariert, die als Teil eines Betriebssystemkerns den Zugriff auf ein Gerät mittels eines speziellen Busses implementiert. Die Ausführung der Methoden `out()` und `in()` soll nicht unterbrochen werden. Dies stellt einen klassischen quer scheidenden Belang dar, und wird mit dem Aspekt `BusIntSync` implementiert. Der Programmcode der Klasse `Bus` soll mittels der Sperrung von Unterbrechungen geschützt werden. Er kann dabei sowohl aus Unterbrechungsbehandlungen im Kern, der Unterbrechungen be-

```
1  aspect Logger {
2    pointcut calls() = call("% transmit(...)") && within("Transmitter");
3
4    advice calls() : around() {
5
6        cout << "transmitting ... " << flush;
7        tjp->proceed();
8        cout << "finished." << endl;
9    }
10 };
```

Abbildung 6: Die pointcut- Funktion within()

reits gesperrt hat, als auch von Programmcode aufgerufen werden, für die Unterbrechungen erlaubt sind heraus aufgerufen werden. Daher darf dieser Aspekt nicht wirken, wenn die Methoden `in()` und `out()` aus Unterbrechungsbehandlungen heraus aufgerufen werden. Abbildung 5 zeigt, wie die *pointcut*- Funktion `cflow()` zur Realsierung dieser Anforderung verwendet werden kann.

2.9.3 Statische evaluierbare *pointcut*- Funktionen

AspectC++ kennt drei statisch evaluierbare *pointcut*- Funktionen. Diese sind im Einzelnen:

`within()` liefert alle *code join points*, welche sich lexikalisch innerhalb des angegeben *name pointcuts* befinden.

`base()` liefert alle *name join points*, welche Basisklassen kennzeichnen, von denen Klassen im angegebenen *pointcut* erben.

`derived()` liefert alle *name join points*, welche Klassen kennzeichnen, die von den Klassen im angegebenen *pointcut* erben.

Sie werden meistens zusammen unter Verwendung von algebraischen Operationen mit anderen *pointcut*- Funktionen verwendet, um *join point* Mengen zu schneiden oder zu vereinigen. Das Codebeispiel in Abbildung 6 ist aus der AspectC++ Sprachreferenz entnommen und zeigt die Verwendung der Funktion `within()`. Der *pointcut* `calls()` besteht aus dem Schnitt zweier *pointcuts* und beschreibt alle Methoden `transmit()` aus der Klasse `Transmitter`.

Diese drei *pointcut*- Funktionen haben ihren Namen, weil das Ergebnis ihrer Evaluierung ausschließlich von der syntaktischen Struktur ihrer *pointcuts* abhängt. Es ist also insbesondere keine Evaluierung zur Laufzeit nötig.

2.10 Zusammenfassung

In diesem Abschnitt wurde ein Einblick in die aspektorientierte Programmierung mit AspectC++ gegeben. Es wurden dabei wichtige und in dieser Arbeit verwendete Sprachmerkmale vorgestellt. Dabei wurden auch wichtige Konzepte der AOP eingeführt. Hier eine Kurzübersicht über das in AspectC++ und dieser Arbeit verwendete Vokabular:

join points sind Stellen im Programmfluss, an denen Aspekte den Programmfluss eines Programms beeinflussen können (*code join points*, oder statische Strukturen im Programmaufbau (*name join points*).

pointcuts sind deklarative Beschreibungen von Verbindungspunktmengen in Form einer eigenen Sprache, der *pointcut*-Sprache. In AspectC++ sind *match expressions*, welche *pointcuts* festlegen, als Zeichenketten implementiert.

Advice Code ist die Modularisierung von quer schneidenden Belangen in Form von Programmtext, der auf *Kontext* zugreifen kann. *Advice*-Definitionen sind Teil der Implementierung von Aspekten.

Kontext: Zu jedem Verbindungspunkt kann der *Advice Code* Informationen zu seinem Kontext erfragen, in dem er eingesetzt wird. Zum Kontext zählt beispielsweise: Aktuelle Funktionsparameter und deren Typen, Rückgabewert (und -typ) (*result*), das Objekt der aufrufenden Funktion (*that*) oder das Objekt der aufrufenden Funktion (*target*).

JoinPoint-API Stellt dem *Advice Code* eine Programmierschnittstelle zur Verfügung, um Informationen über seinen Ausführungskontext abfragen zu können.

Einfügungen (*Introductions*) Hierunter versteht man Einfügungen von zusätzlichen Sprachelementen wie Deklarationen von Methoden, Variablen etc. in Klassen.

3 Statisches und dynamisches Weben

Ein Ziel der *aspektorientierten Programmierung* ist die Vereinfachung der Erweiterung von Softwareprojekten um quer schneidende Belange, die sich nicht mittels klassischer (objektorientierter) Programmierung modularisieren lassen. Zur Umsetzung dieses Ziels werden sowohl neue Sprachen, aber auch neue Werkzeuge zur Übersetzung, Erstellung und Fehlersuche eingeführt.

Das Prinzip der aspektorientierten Programmierung macht an sich noch keine Aussagen über den Zeitpunkt des *Webens* der Aspekte. Grundsätzlich unterscheidet man zwischen zwei Alternativen: Aspekte können entweder *statisch*, also zusammen mit der Anwendung, oder *dynamisch*, also nach dem Starten der übersetzten Anwendung in das System eingebracht werden.

Die heutigen verfügbaren Werkzeuge stützen sich meist aus pragmatischen Gründen auf das Weben von Aspekten zusammen mit der Anwendung. Diese Arbeit zeigt auf, welche weiteren Schritte für die Umsetzung zum *Weben zur Laufzeit*, also nachdem die Anwendung bereits gestartet wurde, notwendig sind.

3.1 Weberbindung

Für die Umsetzung statischen Webens kommen grundsätzlich zwei Herangehensweisen in Frage:

Bei Umsetzung auf *Binärebene* (z.B. [DFL+05]) wird die Anwendung mit herkömmlichen Werkzeugen entwickelt und übersetzt. Der Weber manipuliert dann anschließend den übersetzten Bytecode bzw. das erzeugte ausführbare Programm.

Bei Umsetzung auf *Quelltextebene* [SL06] manipuliert der Weber vor dem Übersetzer den Programmtext.

Diese Unterscheidung wird in dieser Arbeit unter dem Begriff *Bindung* zusammengefasst.

Die Umsetzung auf Binärebene geschieht durch Manipulation der erzeugten Programme. An den von *pointcuts* erfassten *join point* werden Funktionsaufrufe auf passende Hilfskonstrukte umgesetzt, die dann den *Advice* aktivieren. Dieser Ansatz kann mit vergleichsweise wenig Aufwand sehr effizient umgesetzt werden.

Für den speziellen Fall der Programmiersprache `C++` ist dieser Ansatz für die meisten C++ Anwendungen und Bibliotheken nicht geeignet. Die Manipulation von Funktionsaufrufen setzt voraus, dass der Funktionsaufruf tatsächlich existiert, und nicht durch *Inlining* in die aufrufende Funktion übersetzt wurde. Auf genau diese Optimierungstechnik setzen jedoch viele C++ Bibliotheken, darunter auch die C++ Standardbibliothek `STL`.

Die andere Möglichkeit zur Umsetzung von AOP ist die Bindung auf Quelltextebene. Hier geschieht die Manipulation des Programms noch vor der eigentlichen Übersetzung. Der Quelltext der Aspekte wird also direkt in die ursprünglichen Klassen *gewoben*. Dadurch können Optimierungen im Übersetzer wie *Inlining*, bei der (kurze) Funktionen an den Ort des Aufrufs eingebettet werden, greifen. Da sich viele C++ Anwendungen und Bibliotheken auf diese Optimierungstechnik verlassen, kommen für diese Domäne von Anwendungen nur Weber mit Bindung auf Sprachebene in Frage.

Weber, die auf Quelltextebene arbeiten, haben die Eigenschaft, dass die Aspekte bereits zur Übersetzungszeit bekannt sein müssen. Wird ein Aspekt überarbeitet, verbessert oder korrigiert muss die Anwendung neu gewoben, übersetzt und neu gestartet werden.

3.2 Der Übersetzer `ac++`

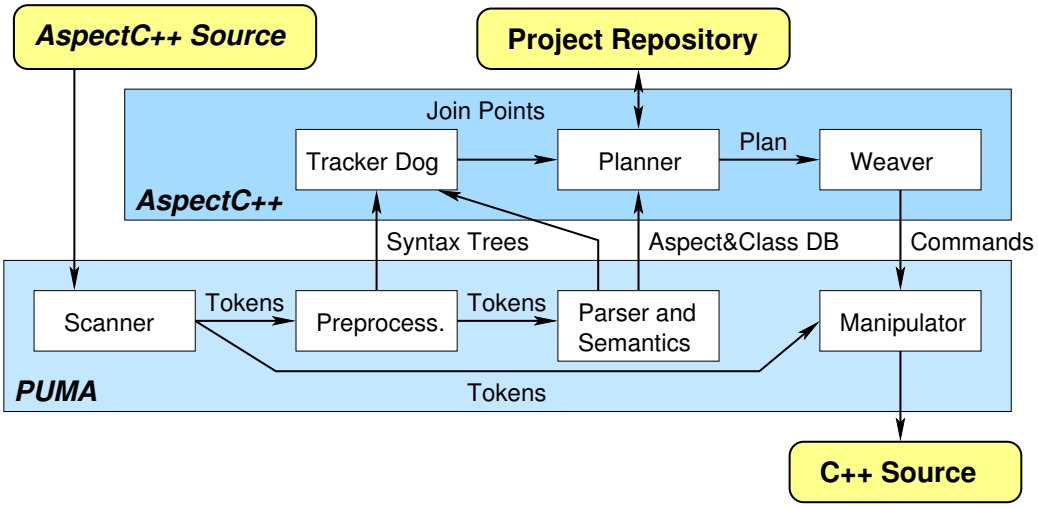


Abbildung 7: Die Architektur des Übersetzers `ac++`. [SL06]

Der Übersetzer `ac++` arbeitet als Präprozessor für einen nachgeschalteten

C++-Übersetzer. Zur syntaktischen und semantischen Analyse wird eine Bibliothek namens PUMA verwendet. PUMA (***Pu**re **Ma**nipulator*) ist dabei eine Bibliothek zur Analyse und Manipulation von C++ Programmtext. Abbildung 7 zeigt die Architektur von `ac++`. Zunächst wird das Projekt von PUMA Übersetzungseinheit um Übersetzungseinheit lexikalisch, syntaktisch und semantisch analysiert. AspectC++ baut dann anhand der gefundenen Aspekte im Projekt einen *Plan* auf, der bestimmt, an welchen Stellen diese *wirken* sollen. Anschließend wandelt der *Weber* diesen *Plan* in konkrete Manipulationsanweisungen um, die auf dem C++ Syntaxbaum dieser Übersetzungseinheit operieren. Das Ergebnis ist ein C++ Programm mit *eingewobenen* Aspekten, welches mit einem gewöhnlichem C++ Übersetzer wie dem GCC übersetzt werden kann.

`ac++` ist also ein Weber mit Bindung auf Quelltextebene. Alle Aspekte, *join points* und *pointcuts* (siehe Abschnitte 2.1 und 2.2) sind hierbei zur Übersetzungszeit bekannt und können zur Laufzeit des Programms nicht mehr beeinflusst werden. Der generierte C++ Code benutzt hierbei anspruchsvolle Hilfskonstrukte, die in [Tar05] vorgestellt und evaluiert wurden. Es konnte gezeigt werden, dass sich die generierten Codemuster von Übersetzern wie dem *GCC* gut optimieren lassen.

3.3 Weben zur Laufzeit

Aspektorientierte Techniken beieinflussen das Verhalten des Ablaufs und in Teilen auch die Struktur von Programmen. Diese Techniken können dazu verwendet werden, um quer schneidende Belange zu implementieren. Die bisher gezeigten Beispiele lassen sich zur Übersetzungszeit problemlos umsetzen. Es gibt jedoch auch Anwendungsfälle, in denen es Sinn macht, Aspekte erst zur Laufzeit in ein System einzubringen.

AOP zur Laufzeit kann für verschiedene Ziele eingesetzt werden. Ein Ziel kann die Anpassung und/oder Erweiterung von Software um quer schneidende Belange sein, also Belange, deren Modularisierung mit klassischen Werkzeugen nicht möglich ist. Statische Anpassungen zur Übersetzungszeit bedeuten dabei Änderungen an der Struktur des Programms. Sie implizieren damit die Notwendigkeit, das laufende Programm außer Betrieb zu nehmen, die betroffenen Strukturen zu überarbeiten und eine neue Version zu starten. Dynamische Anpassungen mittels AOP zur Laufzeit versprechen solche Wartungsarbeiten an Programmen durchführen zu können, bei denen die Unterbrechung ihres Dienstes unerwünscht ist. Dazu gehören Dienste, deren Unterbrechung Datenverlust, Einnahmeausfälle oder Sicherheitseinbußen be-

deuten können. Denkbar sind beispielsweise Fehlerkorrekturen im Sinne von *hot fixes*, welche in eine laufende Anwendung geladen werden können, ohne dass diese neu gestartet werden muss.

Ein anderes Ziel kann die Erweiterung von Software um zusätzliche, möglicherweise erst nach Fertigstellung der Basisanwendung entwickelte Komponenten, welche mittels Advice an (theoretisch) beliebigen Stellen der Anwendung deren Ablauf beeinflussen können. Dies kann mit dem Sprachmerkmal *Introductions* realisiert werden, welches für die Sprache AspectC++ in Unterabschnitt 2.8 vorgestellt wurde.

Erweiterungen, die C/C++ Anwendungen um quer schneidende Belange erweitern sind mit klassischen Werkzeugen ungewöhnlich, hauptsächlich weil sie technisch sehr anspruchsvoll umzusetzen sind. Dennoch tritt diese Arbeit den Versuch an, Systeme mittels aspektorientierten Konzepten zur Laufzeit zu erweitern.

3.4 Aspektmodule als Softwareerweiterungen

Anpassungen und Erweiterungen von bestehender Software sind gängige Techniken in vielen Softwareprojekten. Diese Prozesse werden sowohl bei der Entwicklung aber auch bei der Wartung eingesetzt, um Programme zu analysieren, erweitern und zu verbessern. Werden Aspekte zur Laufzeit gewoben, so stellt dies in gewisser Form immer eine Art Erweiterung des Systems dar.

Erweiterbare Systeme bestehen aus einem *Basisprogramm*, welches zur Laufzeit (und damit nach seiner Auslieferung und Inbetriebnahme) durch *Erweiterungsmodule* um zusätzliche Funktionalität erweitert wird. Dabei *kennt* zwar das Erweiterungsmodul das Basisprogramm, aber nicht zwangsläufig umgekehrt. Unter *Kennen* versteht man in diesem Zusammenhang die Information über interne Strukturen wie Datentypen oder Funktionen, die aufgerufen und benutzt werden können.

Eine weit verbreitete Eigenschaft von Erweiterungsmodulen ist es, dass Module normalerweise in beliebiger Reihenfolge geladen und entladen werden können. Dies ist für nachladbare Aspektemodule nicht unbedingt der Fall. Dieser Unterabschnitt erklärt diese Erkentnis im Detail.

3.4.1 Erweiterungen ohne AOP

Ohne aspektorientierte Programmierung ist die Kenntnisbeziehung in Systemerweiterungen in gewisser Weise bidirektional. Die Basisanwendung gibt

eine Schnittstelle für (noch unbekannte) Module vor, mit der diese sich registrieren können. Das Basisprogramm nutzt ebenfalls diese Schnittstelle, um Module aktivieren zu können. Diese Adaptionsschnittstelle stellt in gewisser Weise eine *Adaptionsvereinbarung* dar, die festlegt, in welcher Weise Erweiterungen am System überhaupt möglich sind. Insofern besteht eine gegenseitige *Kenntnisbeziehung* zwischen Basisprogramm und möglichen Erweiterungen. Erweiterungsmodule können dabei nur in sehr engen Grenzen Anpassungen an einer bestehenden Anwendung vornehmen. Die *Adaptionsvereinbarung* legt dabei fest, an welchen Stellen der Programmabfluss die Erweiterung überhaupt beachtet.

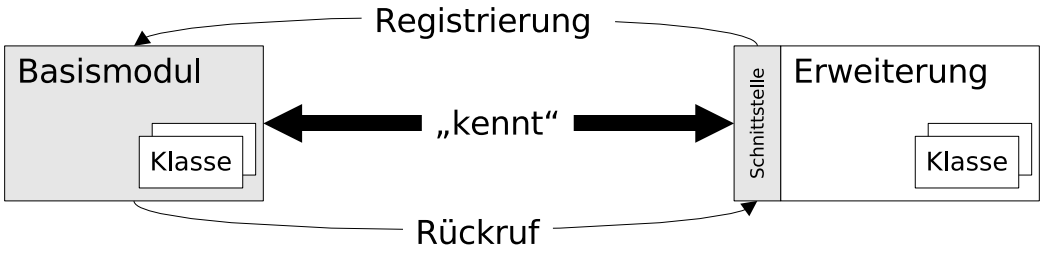


Abbildung 8: Bei Erweiterungen ohne AOP gibt die Anwendung die von den Modulen implementierte Schnittstelle vor.

In Abbildung 8 wird dieser Umstand veranschaulicht. Die grau unterlegten Kästen in der Abbildung sind Entscheidungen und Vorgaben, die beim Entwurf der Anwendung festgelegt werden. Teile dieser Vorgaben werden von einem (möglicherweise später) entworfenem Erweiterungsmodul implementiert.

3.4.2 Erweiterungen mit AOP

Mit Hilfe von aspektorientierter Programmierung wird die Kenntnisbeziehung unidirektional, das heisst, dass die Basisanwendung muss nicht mehr auf Erweiterungen *vorbereitet* werden muss. Es ist also nicht mehr nötig, eine Schnittstelle vorzugeben, denn Aspekte sind in der Lage an (theoretisch) beliebigen Stellen im Programmfluss Veränderungen einzuführen. Es existiert daher keine *Adaptionsvereinbarung*, wie im vorigem Abschnitt. Aspekte wirken global, das heisst sie können das Verhalten von bestehenden Programmen beeinflussen, ohne dass diese darauf vorbereitet werden müssen. Die *Kenntnisbeziehung* verläuft daher nur in Richtung des Basisprogramms.

Sollen Erweiterungen auch nach Inbetriebnahme an laufenden Programmen

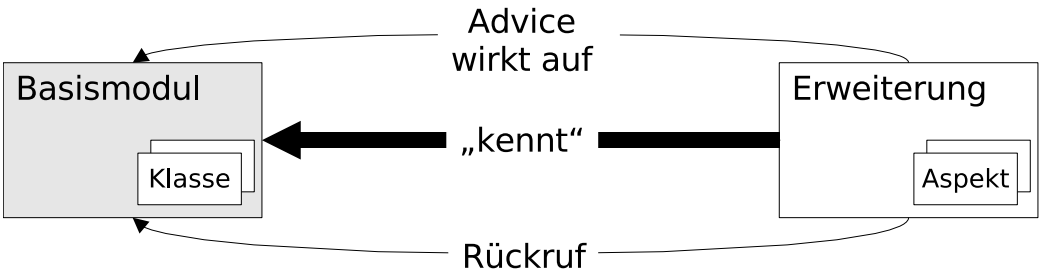


Abbildung 9: Erweiterungen mit AOP benötigen keine Schnittstelle in den Erweiterungsmodulen.

geschehen, so ist eine Infrastruktur nötig, welche es erlaubt, Aspekte zur Laufzeit zu weben. Statische Weber arbeiten zur Übersetzungszeit. Sie können die nötigen Bindungen selber herstellen, weil sie die vollständigen Quellen vorliegen haben und diese auch beliebig transformieren können. Um Aspekte zur Laufzeit weben zu können ist zusätzliche Infrastruktur nötig, diediese Bindungen bereit stellt.

Das Weben von Aspekten zur Laufzeit setzt immer eine Unterstützung zur Laufzeit voraus. Diese Unterstützung kann bei interpretierten Sprachen im Interpreter und bei Programmen, die in Sprachen wie Java geschrieben wurden, in einer virtuellen Maschine implementiert sein. Anwendungen in `C/C++` werden üblicherweise weder interpretiert noch laufen sie in einer virtuellen Maschine ab. Daher ist es notwendig, die Anwendung zu manipulieren und die Laufzeitunterstützung in die Anwendung zu binden. Diese Manipulation kann entweder auf Quelltextebene auf Basis von Quelltexttransformation oder auf der Ebene des ausführbaren Programms geschehen.

Manipulation bedeutet in diesem Zusammenhang, dass Methodenaufrufe im Programm durch Aufrufe auf Hilfskonstrukte wie zum Beispiel dem *Aspect Moderator* [CBE+00] ersetzt werden. Bei diesem Ansatz wird eine Schnittstelle eingeführt, an denen sich Aspekte registrieren, um von diesem Moderator zu Ereignissen wie Start der Ausführung oder Beendigung der Ausführung einer Funktion aufgerufen zu werden. Die Instrumentierung kann dabei entweder manuell durch den Programmierer, oder automatisch durch ein Werkzeug wie `DAO C++` geschehen, wie es in [AE04] vorgeschlagen wird.

3.4.3 Erweiterungsketten

Das Prinzip der aspektorientierten Softwareentwicklung beschreibt Aspekte als global wirkende Konstrukte. Im Fall dynamischen Webens impliziert dies die Vorstellung, dass Aspekte automatisch auch auf später nachgeladene oder bereits vorher geladene Softwaremodule wirken und beliebig ladbar und entladbar seien. Dies ist im speziellen Fall von C++ jedoch schwierig umzusetzen, weil übersetzte Module in C++ keine Typinformationen mehr besitzen. Das Vorhandensein von Typinformationen ist jedoch essentiell für die Unterstützung von *generic advice* (siehe auch Unterabschnitt 2.7), ohne den kaum sinnvolle Aspektimplementierungen in AspectC++ möglich sind.

In `C/C++` Programmen werden statische Typinformationen meist in Form von Headerdateien bereitgestellt, die für einzelne Übersetzungseinheiten eingebunden werden. Ein `C/C++` Modul stellt also immer eine Schnittstelle in Form von Headerdateien zur Verfügung, die von Erweiterungen eingebunden wird. In diesem Zusammenhang kann man diesen Sachverhalt auch als *Kenntnisbeziehung* betrachten: Erweiterungsmodule *kennen* ihre Basismodule.

In AspectC++ sind für die Übersetzung von Aspekten zweierlei Arten von statischen Typinformationen nötig. Einerseits müssen die Typen, auf denen die Aspekte wirken, *bekannt* sein. Andererseits entstehen bei der Evaluierung der *pointcuts* für jeden ermittelten *join point* weitere *join point* spezifische Typinformationen, die von Advice im Aspekt mittels der Joinpoint-API genutzt werden können. Im Falle von statischem Weben verursacht dies keine weitere Schwierigkeiten, da beide Arten von Typinformationen entweder bereits vorhanden sind, oder vom Aspektweber selbst ermittelt werden. Im Falle von dynamischem Weben sind die *join point* spezifischen Typinformationen nur für bereits übersetzte Module vorhanden.

Es entsteht also eine *Kette* von Erweiterungen. Jede Erweiterung kann dabei auf statische Typinformationen von allen *früheren* Gliedern dieser Kette zurückgreifen. Man könnte auch sagen, jede Erweiterung *kennt* nur ihre früheren Glieder. Es entsteht dadurch implizit eine *Kenntniskette* (Abbildung 10). Jedes Modul hat dabei Kenntnis über alle seine Vorgängermodule. Innerhalb jeder Erweiterung wirken Aspekte *statisch*. Damit ein Aspekt in der Kenntnishierarchie nach nach links in Bild 10 wirken kann, muss er *dynamisch* gewoben werden. Um in der Kenntnishierarchie nach rechts im Bild zu wirken, reicht es den selben Aspekt *statisch* zu übersetzen.

Damit ist es prinzipiell nicht ohne weiteres möglich Aspekte zu programmieren, die auf später nachgeladene Module *wirken*. Um dennoch Erweite-

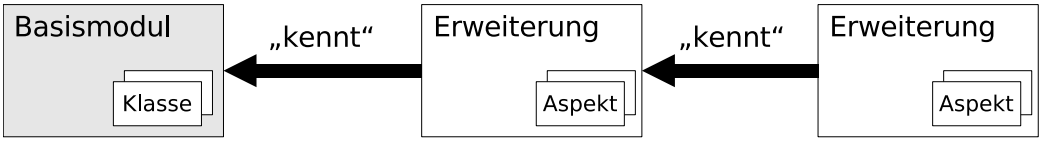


Abbildung 10: Schrittweise Erweiterungen mit Modulen, welche AOP Techniken auf zuvor geladene Module anwenden, implizieren eine Kette von Wissensbeziehungen.

rungsmodule mit potentiell quer schneidendem Charakter zur Laufzeit laden zu können, ist es notwendig, Einschränkungen in Kauf zu nehmen. Die Einschränkung dabei ist, dass weitere Module ebenfalls mit diesem Aspekt übersetzt werden müssen, oder zu verbieten, unabhängig voneinander übersetzte (dynamische) Aspektmodule *gleichzeitig* zu laden. In AspectC++ wird *Advice* pro *join point* instantiiert, also Code erzeugt. Die Lösung für *dynamisches Weben* besteht nun darin, die statische Typinformation aus den Vorgängermodulen zu extrahieren und die Instantiierung von *Advice* mittels der extrahierten Typinformation in den Erweiterungsmodulen vorzunehmen.

Diese Einschränkung bedeutet, dass Aspekte zur Laufzeit zwar möglich sind, impliziert aber eine künstlich aufgezwungene *Ladereihenfolge*. Diese Notwendigkeit entsteht aus der Tatsache, dass statische Typinformation nur durch Quelltextanalyse, meist ein Zwischen- oder Nebenprodukt der Übersetzung, entstehen kann.

Eine Konsequenz aus diesem Ansatz ist, dass ein dynamisches Erweiterungsmodul genau dann aktiv ist, wenn es geladen ist. Weiterhin folgt, dass ein Erweiterungsmodul, das zusätzlich zu einem bereits bestehendem Erweiterungsmodul geladen werden soll, voraussetzt, dass das frühere Erweiterungsmodul bereits geladen ist. Eine weitere Konsequenz ist, dass eine *Entladereihenfolge* erzwungen wird, denn es kann sein, dass ein früher geladenes Modul Einfügungen macht, die von einem späterem Modul verwendet werden. Damit dürfen früher geladene Erweiterungsmodule nicht entladen werden, solange sie von später geladenen noch verwendet werden.

Im manchen Situationen ist es jedoch wünschenswert, dass gewobene Aspekte *deaktiviert* werden können. Dadurch ist es möglich, Aspekte aus *jüngeren* Erweiterungsmodule zu deaktivieren, ohne dass das Modul zu entladen. Dies kann die Einschränkung der vorgegebenen Entladereihenfolge etwas mildern. Die Umsetzung der Deaktivierung kann entweder dabei explizit im Aspekt, also manuell vom Programierer, oder implizit durch den Weber implementiert werden.

3.5 Der *Single Language* Ansatz

Der vorige Abschnitt macht klar, dass in in einem erweiterbarem System abhängig davon, wo Aspekte in der Erweiterungskette auftreten, der Aspekt entweder *dynamisch* oder *statisch* gewoben werden muss. Es bietet sich daher an, für sowohl *statisches* als auch *dynamisches* Weben *dieselbe* Aspektsprache zu verwenden.

Dieser Ansatz bringt eine Reihe von Vorteilen. Zum einen muss der Programmierer keine neue Programmiersprache erlernen. Dies erhöht die Akzeptanz und erlaubt die Wiederverwendung von Programmstücken, die bereits in dieser Sprache vorliegen. Es ist außerdem möglich, bestehende Werkzeuge zur Entwicklung wie die Visualisierung von Aspekten weiter zu verwenden.

Ziel ist es also dem Programmierer zu erlauben, Aspekte unabhängig von der Entscheidung zu schreiben, ob der Aspekt statisch oder dynamisch gewoben wird. Als Folge dieses Ansatzes kann diese Entscheidung auf den Konfigurationszeitpunkt der Anwendung verschoben weben.

In dieser Arbeit wird dieser Ansatz in der Domäne der Sprache AspectC++ verfolgt und umgesetzt. Dabei wurde eine Infrastruktur entworfen, die es erlaubt, Erweiterungsmodule in der Sprache AspectC++ zu implementieren, die zur Laufzeit geladen werden können.

3.6 Zusammenfassung

Aspektorientierte Techniken können sowohl statisch zur Übersetzungszeit als auch dynamisch zur Laufzeit der Anwendung wirken. Für statisches Weben in C++ Programmen kann dies mit dem Werkzeug `ac++` erfolgen. Aspekte können zur Laufzeit gewoben werden, um Anwendungen noch während ihres Betriebs in ihrem Verhalten zu ändern oder um (potentiell quer schneidende) Erweiterungen einzubringen. Dabei stellen Aspekte, die zur Laufzeit gewoben werden, immer eine Form von Erweiterungsmodul dar.

Im speziellen Fall des *dynamischen* Webens in C++ Programmen ist die Umsetzung von solchen Aspektmodulen schwierig, weil statische Typinformation, die zur Übersetzung von diesen Modulen nötig ist, nicht automatisch vorhanden ist. Es müssen also Einschränkungen bei der Benutzung von Aspektmodulen gemacht werden. Dabei stellt sich heraus, dass es nötig ist, *denselben* Aspekt in einer Erweiterungskette sowohl statisch als auch dynamisch zu Weben. Aus diesem Grund bietet es sich an, für beide Art von Aspekten *dieselbe* Sprache zu benutzen.

4 Infrastruktur

Wie im vorigen Kapiteln angedeutet wurde, ist für die Umsetzung von aspektorientierten Sprachmerkmalen zur Laufzeit mehr Infrastruktur nötig. Diese Infrastruktur wurde bereits im Vorfeld dieser Arbeit z.B. in [SPLG+06] skizziert und andiskutiert. Dieser Arbeit implementiert sie als prototypisch und wird an bestehenden Anwendungen in Kapitel 10 evaluiert.

Dieses Kapitel gibt zunächst einen Überblick über die einzelnen Komponenten die nötig sind, um Aspekte zur Laufzeit weben zu können. Anschließend werden diese einzeln vorgestellt und ihre Zusammenarbeit erörtert.

4.1 Überblick

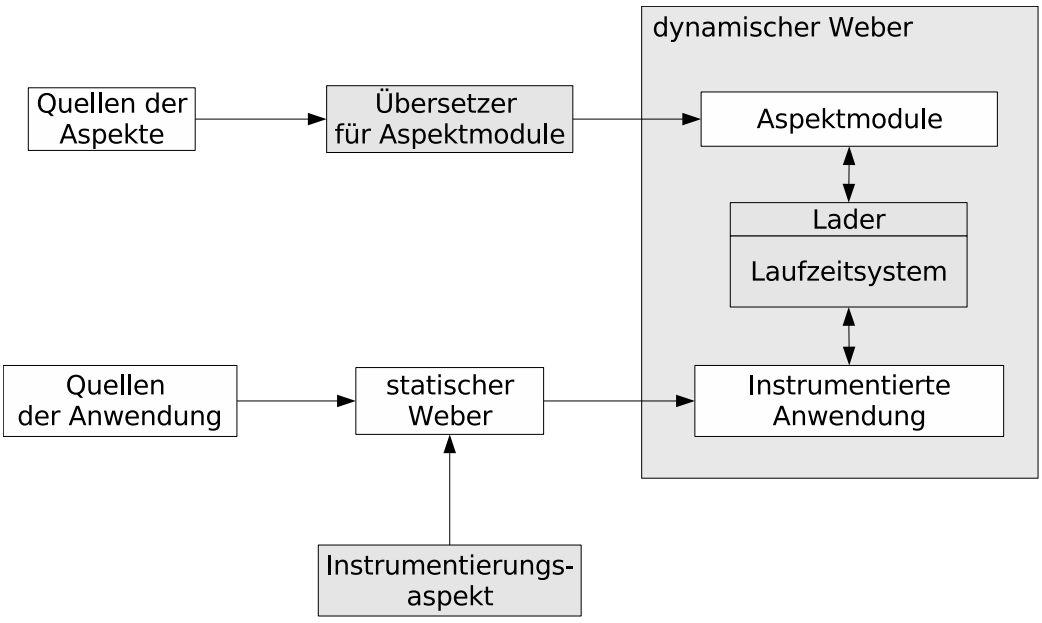


Abbildung 11: Architektur der entwickelten Infrastruktur zum dynamischem Weben

Abbildung 11 zeigt den schematischen Aufbau der in dieser Arbeit vorgestellten und entwickelten Infrastruktur zum dynamischem Weben in C++ Programmen. Innerhalb dieser Infrastruktur ist es vorgesehen, dass Aspekte sowohl statisch als auch dynamische gewoben werden. Die Abbildung zeigt wie die einzelnen Komponenten eines Softwaresystems zusammenspielen. Im

weiteren Verlauf dieses Kapitels werden die einzelnen Komponenten aus Abbildung 11 näher erläutert.

Teile der Infrastruktur konnten entweder aus vorhergehenden Arbeiten übernommen werden, oder werden vom Entwickler gestellt. Die im Rahmen dieser Arbeit entwickelten Komponenten sind in der Abbildung 11 mit grau unterlegten Kästen markiert.

4.2 Instrumentierung mittels statischem Weben

Um Aspekte zur Laufzeit weben zu können, ist es nötig, die Anwendung so zu manipulieren, dass im Programmablauf geprüft wird, ob ein Aspekt aktiviert werden soll.

Techniken wie diese sind eher bei interpretierten Sprachen und Laufzeitumgebungen üblich. In Sprachen wie Python oder Perl ist es ohne Aufwand möglich, globale Symboltabellen zu modifizieren und so das Verhalten zur Laufzeit zu verändern. Implementierungen für aspektorientierte Programmierung in Perl[1] oder in Python[2] benötigen daher keine Spracherweiterung, sondern können als *native* Bibliothek implementiert werden. Beide Implementierungen unterscheiden nicht zwischen statischem und dynamischem Weben. Advice wird durch einen gewöhnlichem Methodenaufruf aktiviert und deaktiviert. Während also in interpretierten Sprachen die Aktivierung von Advice in Form von Bibliotheken realisiert werden kann, existiert für C++ z.B. AspectC++ als Spracherweiterung zur Umsetzung von quer schneidenden Belangen zur Übersetzungszeit.

In Unterabschnitt 3.4.2 wird die Notwendigkeit einer Laufzeitunterstützung zur Manipulation des Programmflusses der Anwendung erläutert. Die Umsetzung dieser Laufzeitunterstützung kann dabei entweder an die Sprachebene oder an die Binärebene gebunden werden. Wie sich in Unterabschnitt 3.1 herausstellt, kommt für die Domäne für `C++` Anwendungen nur die Bindung auf Sprachebene in Frage.

Aus diesem Grund bietet es sich an, die Manipulation der Anwendung in Form einer statischen Instrumentierung umzusetzen. Dies stellt einen quer schneidenden Belang dar, der mittels eines statischen Aspektwebers in die Anwendung zur Übersetzungszeit gewoben wird. Dieser statische Aspektweber musste nicht neu entwickelt werden, sondern es kann für diese Aufgabe das Werkzeug `ac++`, das in Unterabschnitt 3.2 vorgestellt wurde, unverändert

[1] http://search.cpan.org/ eilara/Aspect-0.11/lib/Aspect.pm
[2] http://www.cs.tut.fi/ ask/aspects/aspects.html

übernommen werden. Die Umsetzung der Instrumentierung der Anwendung kann also in der Sprache AspectC++ geschehen.

Dieser (statische) Instrumentierungsaspekt wird konfigurierbar entworfen und kann daher vom Laufzeitsystem als *abstrakter Aspekt* bereitgestellt werden. (siehe auch Unterabschnitt 2.5). Um die Anwendung zu instrumentieren, muss dieser Aspekt instantiiert werden. Dabei ist die Frage, an welchen *join points* der Aspekt instantiiert werden soll, spezifisch für das Softwareprojekt. Häufig macht es wenig Sinn, *alle* potentiellen *join points* zu instrumentieren. Daher ist die Auswahl dieser *join points* für die Instrumentierung Teil der Konfiguration des dynamischen Webers.

4.3 Dynamische Aspekte als dynamische Bibliotheken

Zur Laufzeit geladene Aspekte werden im Rahmen der hier entwickelten Infrastruktur nach Unterabschnitt 3.5 mit derselben Sprache entwickelt, in der auch statische Aspekte beschrieben werden: AspektC++. Zur deren Übersetzung ist jedoch ein anderes Werkzeug als der statische Aspektweber `ac++` nötig.

Unter *UNIX*-artigen Betriebssystemen sowie unter Windows werden Erweiterungen in Form von dynamisch ladbaren Bibliotheken gekapselt. Dies erlaubt es, bereits vorhandene Betriebsystemfunktionalität zu benutzen um neue Symbole, Funktionen oder globale Variablen in ein laufendes Programm einzubringen. Dynamisch gewobene Aspekte stellen immer eine Erweiterung eines Systems dar (siehe auch Unterabschnitt 3.4). Daher ist es notwendig, dass der Übersetzer für solche Aspekte als Ergebnis solche dynamisch ladbare Bibliotheken erzeugt.

4.4 Das Laufzeitsystem und der Aspektlader

Damit Aspekte zur Laufzeit in eine laufende Anwendung gewoben werden können, muss dem Laufzeitsystem bekannt gemacht werden, welches dynamisch ladbare Modul geladen werden soll. Das eigentliche Laden dieses Moduls geschieht wie im vorigem Abschnitt angedeutet über Betriebssystemfunktionen.

Der Lader stößt durch das Laden von dynamischen Bibliotheken als Seiteneffekt sogenannte *Bibliothekskonstruktoren* an. Bibliothekskonstruktoren sind Funktionen, die beim Laden eines dynamisch ladbaren Moduls automatisch aufgerufen werden. Bekanntestes Beispiel für solche Bibliothekskonstrukto-

ren sind Konstruktoren von globalen Objekten. Gängige Übersetzer bieten jedoch Erweiterungen an, um einzelne Funktionen explizit als Bibliothekskonstruktoren zu deklarieren.

Der dynamische Weber besteht in dieser Infrastruktur im Gegensatz zum statischem Weber aus mehreren Komponenten: Der instrumentierten Anwendung, eines dazugebundenen Laufzeitsystems sowie aus übersetzten Aspektmodulen, die von einem speziellem Werkzeug erzeugt wurden.

4.5 Abhängigkeiten

Die einzelnen Komponenten in der Infrastruktur besitzen Abhängigkeiten untereinander. Abhängigkeit bedeutet in diesem Zusammenhang, dass die Änderung einer Komponente, die andere Komponenten als Abhängigkeit hat, Änderungen in davon abhängigen Komponenten nach sich ziehen.

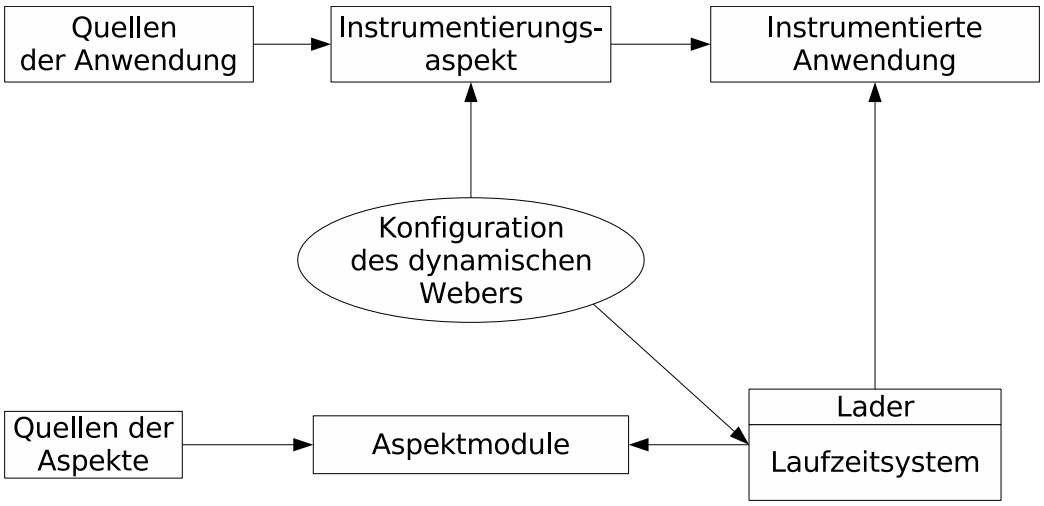


Abbildung 12: Abhängigkeiten der Komponenten

In Abbildung 12 werden die einzelnen Komponenten in Kästen dargestellt. Die im Bild oval dargestellte Konfiguration ist insofern keine entwickelte Komponente, sondern muss für jedes Projekt vom Benutzer erstellt werden. Abhängigkeiten, die bei Änderung in ihrer Komponente weitere Änderungen in abhängigen Komponenten erfordern, sind mit einem Pfeil dargestellt. Dies betrifft das Laufzeitsystem, das während der Entwicklung mehrfach überarbeitet wurde. Änderungen erfordern hierbei eine erneute Instrumentierung der Anwendung und ein Neuübersetzen der dynamischen Aspekte.

Das Laufzeitsystem selbst ist nur von der Konfiguration abhängig. Diese legt das Verhalten und die bereitgestellten Merkmale fest. Dies kann je nach Änderung eine Neuinstrumentierung der Basisanwendung nach sich ziehen. Dies ist genau dann der Fall, wenn die alte Instrumentierung auf Merkmale zugreift, die in der neuen Variante nicht vorhanden sind.

Bei Änderung des Instrumentierungsaspektes, oder bei Änderung der Konfiguration um andere Stellen in der Anwendung zu instrumentieren, ist es nötig, die Anwendung erneut statisch zu weben. Dieser Vorgang ist verhältnismäßig rechenzeitaufwändig, denn statische Aspekte werden *mit* der Anwendung übersetzt.

Abbildung 12 zeigt, dass dynamische Aspektmodule keine anderen Abhängigkeiten ausser ihren Quelltexten haben. Es stellt sich heraus, dass die Übersetzung von solchen Aspektmodulen verhältnismäßig wenig Rechenzeit beansprucht. Im Gegensatz zu Aspekten, die in jede einzelne Übersetzungseinheit des Projekts eingewoben werden, werden dynamische gewobene Aspekte *gegen* das Laufzeitsystem übersetzt.

4.6 Zusammenfassung

Das Weben von Aspekten zur Laufzeit erfordert im Vergleich zum Weben zur Übersetzungszeit erheblichen zusätzlichen Aufwand. Diesem Aufwand wird in dieser Arbeit mit einer Softwareinfrastruktur begegnet, die einzelne Aufgaben, die zur Bewältigung dieser Aufgabe nötig sind, in einzelne Komponenten kapselt. Dieses Kapitel hat aufgezeigt, aus welchem einzelnen Komponenten die Infrastruktur zum dynamischen Weben in C++ Programmen besteht, und wie diese dzusammenspielen. Es stellt sich dabei heraus, dass ein *dynamischer Aspektweber* mehr Aufgaben zu bewältigen hat als ein *statischer Aspektweber*. Daher besteht der dynamische Weber aus mehreren Einzelkomponenten.

Die Anwendung wird dabei von einem *statischem Instrumentierungsaspekt* so manipluliert, dass zur Laufzeit geladene Aspekte den Programmablauf beinflussen können. Diese Aspekte, werden in *dynamisch ladbare Module* übersetzt, die durch den *Aspektlader* aktiviert und deaktiviert werden können. Das *Laufzeitsystem* übernimmt dabei die Verwaltung der geladenenen Aspektmodule und deren Aktivierung. Der dynamische Aspektweber besteht also aus den Komponenten *Aspektmodule*, *Aspektlader* und dem *Laufzeitsystem*.

5 Dynamischer Code-Advice

Zu den Basistechniken zum Weben von Aspekten zur Laufzeit gehört die Möglichkeit, Advice zur Laufzeit zu aktivieren. Dabei wird Infrastruktur bereitgestellt, die es ermöglicht, zur Laufzeit zu prüfen, ob der Programmfluss an *join points* wegen geladener Aspektmodulen geändert werden soll. Die Überprüfung findet im *Laufzeitsystem* statt, das Laden der Aspektmodule selbst mittels des *Aspektladers*. Da diese Überprüfung an *allen* instrumentierten *join points* durchzuführen ist, ergibt sich hier ein enormer Optimierungsbedarf. Aus diesem Grund werden alternative Implementierungen angeboten, die verschiedene Vor- und Nachteile besitzen. Das Laden von dynamischem Advice bietet die Grundlage für die in den nächsten Kapiteln vorgestellten Techniken. Daher kann die Aktivierung von dynamischen Advice als grundlegende Operation im dynamischem Weber betrachtet werden.

Das Grundprinzip von dynamischem Advice ist, den *Advice Code* des dynamischen Aspekts der Anwendung verfügbar zu machen und dafür zu sorgen, dass dieser an den von seinem *Pointcut* beschriebenen *join points* aufgerufen wird. Dabei ergeben sich zwei Problemstellungen: Einerseits muss der dynamische Advice mit den Kontextinformationen versorgt werden, die von der Sprache AspectC++ vorgegeben werden. Außerdem müssen die Aspektmodule an den entsprechenden *join points* den zu aktivierenden dynamischen Advice registrieren. Dieses Kapitel zeigt auf, welche Anforderungen zur Umsetzung von dynamischem Advice die Entwicklung beeinflusst haben.

5.1 Dynamischer und statischer Kontext

Wie in Unterabschnitt 2.6 gezeigt wurde, ist der Zugriff auf Kontextinformation auf den Ausführungskontext von *Advice Code* ein wesentliches Merkmal der Sprache AspectC++. Diese Techniken sind wesentliche Grundlagen für die Entwicklung von nichttrivialem Advice. Man unterscheidet zwischen dynamischem und statischem Kontext. Eine Übersicht über in AspectC++ verfügbarem Kontext wurde in Tabelle 1 auf Seite 10 gegeben.

Unter statischem Kontext versteht man Informationen, die zur Übersetzungszeit von *Advice* feststehen. Dazu gehört unter anderem die die Signatur der Funktion, auf die *Advice* wirkt, die Typen des Rückgabewertes und der Argumente, sowie deren Anzahl. Wie in Unterabschnitt 3.4.3 auf Seite 23 bereits betont wurde, ist es für jede nichttriviale Adviceimplementierung nötig, sie mit statischer Typinformation über ihren Ablaufkontext zu versorgen. Damit diese Art von Kontext im Advice nutzbar wird, ist es nötig, sie in geeigneten

Strukturen zu kapseln. Es muss also eine Art Datenbank geschaffen werden, die statische Typinformationen über den Ausführungskontext an den in der Anwendung instrumentierten *join points* enthält. Aus dieser Datenbank können dann bei Bedarf für dynamischen Advice Strukturen erzeugt werden, die die statische Typinformation geeignet kapselt. Diese Datenbank wird im weiteren Verlauf dieser Arbeit *project repository* genannt.

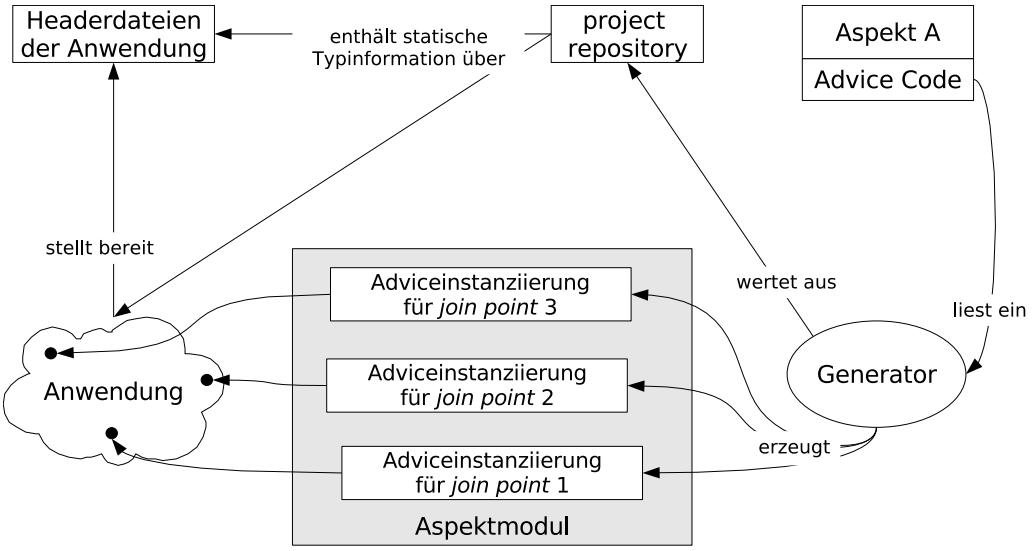


Abbildung 13: Adviceimplementierungen werden pro *join point* erzeugt.

Abbildung 13 zeigt eine Anwendung mit drei instrumentierten *join points*. Die von der Anwendung bereitgestellten Headerdateien dienen sowohl als interne Schnittstelle für die einzelnen Übersetzungseinheiten der Anwendung als auch als exportierte Schnittstelle für Erweiterungsmodule. Das *project repository* enhält dabei sowohl diese (statischen) Typinfomationen, als auch die statischen Kontextinformationen über die instrumentierten *join points* der Anwendung. Das *project repository* wird von einem Generator für Adviceinstanzen ausgewertet, der aus jeder *Advice Code* Definition für jeden *join point* eine Adviceimplementierung erzeugt. Diese Adviceimplementierungen werden im Aspektmodul gekapselt und beim Weben des Aspekts geladen. Der eigentliche Webevorgang besteht nun darin, diese Adviceimplementierungen an die *instrumentierten join points* zu binden.

Das Prinzip der Adviceinstantiierung, wie sie in Abbildung 13 veranschaulicht wird, wird sowohl im statischem als auch im dynamischen Weber gleich umgesetzt. Dabei wird der Programmtext des Aspekts A in eine Klasse transformiert, in der Advice eine parametrisierte Funktion ist. Es werden pro *join*

point Hilfsfunktionen (*Wrapperfunktionen*) erzeugt, die mit einer *join point* spezifischen *Joinpoint-Klasse* instantiiert werden. Damit wird als Generator in Abbildung Abbildung 13 der C++ *template* Mechanismus ausgenutzt.

Dynamischer Kontext ist erst zur Laufzeit bekannt. Insbesondere kann er sich bei jeder Ausführung des *Advice Code* ändern. Das Laufzeitsystem muss also diesen dynamischen Kontext in von der Instrumentierung bereitgestellten Datenstrukturen kapseln und dem dynamisch Advice über eine definierte Schnittstelle (im Fall von AspectC++ die *Joinpoint-API*) Zugriff darauf bieten. Beispiele für dynamischen Kontext wären in diesem Zusammenhang der Zeiger auf die Objekte der aufrufenden (`that()`) oder aufgerufenen (`target()`) Funktion, sowie der Rückgabewert und die Funktionsparameter.

Beim *statischen* Weben wird sowohl der statische als auch der dynamische Kontext vom Aspektweber `ac++` gestellt. Im *dynamischen* Fall ist der Aufwand, den man für den Zugriff auf den dynamischen Kontext erbringen muss, höher. Damit dynamischer Advice auf diese Art von Kontextinformationen zugreifen kann, ist es nötig, den dynamischen Kontext in Datenstrukturen zwischenzuspeichern. Die hierzu nötigen Datenstrukturen sind an die vom statischem Weber `ac++` erzeugten Hilfsstrukturen angelehnt. In `ac++` werden sie jedoch als Optimierung genau so generiert, dass nur solche Hilfstrukturen für die Bereitstellung von Kontext erzeugt werden, die auch tatsächlich vom *Advice Code* aus benötigt werden. Beim dynamischem Weben existiert jedoch dieses Wissen nicht, so dass die Instrumentierung immer den kompletten dynamischen Kontext an das Laufzeitsystem übergeben muss.

Falls im Voraus bekannt ist, dass bestimmte Arten von dynamischen Kontextinformationen von keinem dynamisch geladenen Aspektmodul benötigt werden, bietet es sich an, in der Konfiguration des dynamischen Webers diejenigen Merkmale, die den nicht benötigten dynamischen Kontext bereitstellen, abzuwählen. Dies bringt Geschwindigkeitsvorteile für alle *join points*, unabhängig davon, ob dynamischer Advice geladen ist oder nicht.

5.2 Generic Advice in dynamisch gewobenen Aspekten

Um zu verstehen wie *generic advice* zur Laufzeit umgesetzt werden kann, ist zunächst etwas Hintergrundwissen über die Umsetzung von Advice im statischem Weber `ac++` nötig. Innerhalb des Übersetzers für dynamische Aspektmodule wird der Aspekt zunächst wie beim statischem Weber `ac++` in die Sprache `C++` übersetzt. Dabei werden die Aspekte in Klassen und Advice in Templatefunktionen übersetzt. Die Templatefunktion bekommt dabei als Parameter ein Objekt, das den dynamischen Kontext kapselt und von einem

Typ ist, der den statischen Kontext beinhaltet. Dieses Funktionstemplate wird dabei für jeden *join point* einzeln instantiiert.

Mit dem Werkzeug zur Erzeugung von statischen Kontextinformationen und Hilfsfunktionen zur Aktivierung von Advice ist es möglich, auch einen Aspekt mit *generic advice*, wie er in Abbildung 2 auf Seite 11 abgebildet ist umzusetzen. Der statische und der dynamische Kontext liegen den generierten Hilfsstrukturen, wie in Unterabschnitt 5.1 beschrieben, vor. Die erzeugte Aspektklasse kann also eine Methode `result()` besitzen, die als Rückgabewert den selben Typ wie die Funktion des *join points* hat, in dessen Kontext der *Advice Code* ausgeführt wird. Dadurch wird bei der Übersetzung dieser Hilfsfunktion die richtige Implementierung des Ausgabeoperators gewählt.

5.3 Advice Container

Eine wichtige Aufgabe des Laufzeitsystems ist es, geladene Aspektmodule zu verwalten. Die Instrumentierung ruft dabei an allen instrumentierten *join points* das Laufzeitsystem auf um zu prüfen, ob ein Aspekt an diesem *join point* aktiv ist. Diese Überprüfung muss immer und an allen instrumentierten Stellen durchgeführt werden. Es bietet sich daher an, diese Überprüfung für häufig auftretende Spezialfälle zu optimieren.

In vielen Anwendungen ist es ausreichend, wenn pro *join point* höchstens ein Aspekt gleichzeitig aktiv ist. Es bietet sich daher an, die Datenstruktur, an der dynamischer Advice registriert wird, den Anforderungen anpassbar zu entwerfen. Diese Datenstruktur wird im Rahmen dieser Arbeit *Advice Container* genannt. Es existieren zwei Arten von *Advice Containern*: einen einfachen, der nur einen dynamischer Advice pro Join Point erlaubt, sowie eine listenbasierte Implementierung ohne diese Limitierung. Es sind weitere *Advice Container* Implementierungen denkbar. Zum Beispiel könnten Datenstrukturen wie Hashes oder Felder zur Speicherung benutzt werden. Eine weitere Erweiterung für mehrfädige Programme könnte sein, dass die *Advice Container* Aspekte für die einzelnen Fäden getrennt verwalten.

In dem in dieser Arbeit entwickeltem dynamischem Weber gibt es zwei Arten von *Advice Containern*: einfache und listenbasierte Advice Container. Die Konfiguration (siehe Abschnitt 4.1) legt fest, welche Implementierung für welchen *join point* verwendet wird.

Der statische Instrumentierungsaspekt sorgt dafür, dass an allen instrumentierten Stellen sogenannte *Monitore* installiert werden. Diese Monitore prüfen, ob an diesem *join point* eine Advicefunktion registriert wurde. Sie ver-

wenden zur Verwaltung registrierter Aspekte die weiter oben beschriebenen *Advice Container*.

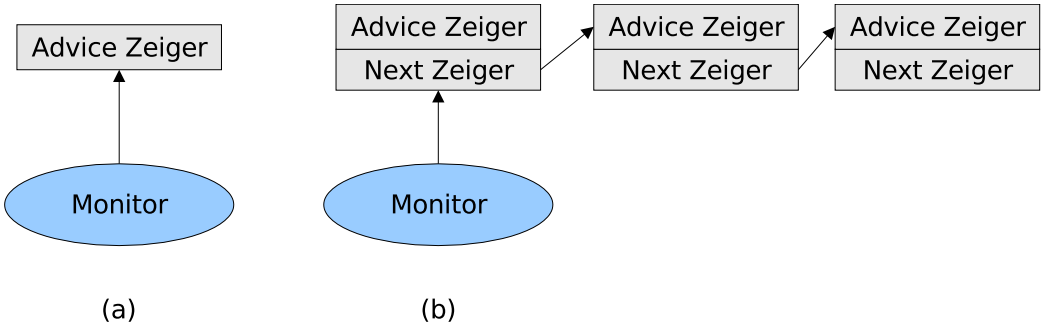


Abbildung 14: (a) Ein *einfacher Advice Container*. (b) Eine listenbasierte Implementierung eines *Advice Containers*

Abbildung 14 (a) zeigt einen *einfachen* Monitor, der nur einen Advice registrieren kann. Die Umsetzung der Überprüfung erfordert konstanten Aufwand. Listenbasierte Advicecontainer machen die Überprüfung im Advicemonitor aufwändiger. So sind pro *join point* mindestens zwei statt einem Zeiger zu reservieren[3]. Der Aufwand zur Überprüfung im Monitor wächst dabei linear mit der Anzahl der registrierten Advicefunktionen. Abbildung 14 (b) veranschaulicht den Aufbau eines listenbasierten *Advice Containers*.

5.4 Aktivierung von Advice

Die Aspektsprache AspectC++ unterscheidet bei *Code Advice* zwischen `call`, `execution`, `construction` und `destruction` Advice. Die Klassifizierung erfolgt mittels der *pointcut* Funktion bei der Deklaration von Advice (vgl. Unterabschnitt 2.3). Die Umsetzung der einzelnen Arten von Advice ist praktisch gleich. Es werden hierzu Hilfsfunktionen (*Advicewrapper*) generiert, die von den bei der Instrumentierung installierten Monitoren aufgerufen werden.

Die Aufgabe dieser Hilfsfunktionen ist es, den *Advice Code* mit sowohl statischen als auch dynamischen Kontextinformationen zu versorgen. Wie im vorigem Abschnitt beschrieben wurde, ist der statische Kontext aus dem *project repository* der instrumentierten Anwendung generierbar. Der dynamische Kontext wird vom Laufzeitsystem gesammelt und der Hilfsfunktion übergeben. Die Hilfsfunktion führt diese beiden Arten von Kontext zusammen und aktiviert den *Advice Code* in geeigneter Art und Weise.

[3]acht anstatt vier Bytes pro *join point* auf 32Bit Architekturen

Für die Gewinnung der Strukturen des statischen Kontextes und die Zusammenführung mit den dynamischen Kontextinformationen aus dem Laufzeitsystem ist ein gesondertes Werkzeug notwendig. Es wertet das *project repository* der instrumentierten Anwendung und die *pointcut* Deklarationen von den dynamisch zu webenden Aspekten aus und erzeugt die benötigten Hilfsfunktionen. Diese Werkzeug ist Teil des Übersetzers für Aspektmodule und wird im Detail in Unterabschnitt 9.2 vorgestellt.

Die Einfügung von diesen Hilfsfunktionen führt an sich noch nicht zur Aktivierung von *dynamischem Advice*. Die Registrierung an den Monitoren erfolgt daher in einer weiteren Hilfsfunktion, die zur Ladezeit des Aspektmodules ausgeführt werden muss. Die genaue Implementierung hiervon ist abhängig von den verwendeten Werkzeugen zur Umsetzung von dynamisch ladbaren Modulen.

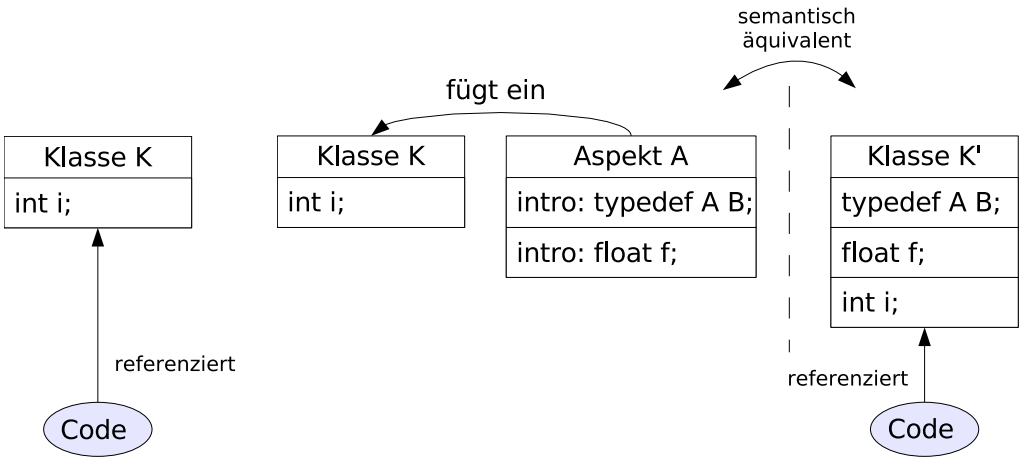
5.5 Zusammenfassung

Dynamischer Advice ist die Basistechnik zum dynamischen Weben von Aspekten zur Laufzeit. Um die Kosten der Instrumentierung der Anwendung zu minimieren, stellt das Laufzeitsystem alternative Implementierungen zur Verwaltung von geladenen Aspekten bereit. Eine weitere Aufgabe des Laufzeitsystems ist es, den *dynamischen* Kontext in geeignete Datenstrukturen zu kapseln. Der *statische* Kontext wird aus einer Typdatenbank, dem *project repository*, gewonnen. Der *dymamische* Kontext wird hingegen vom Laufzeitsystem bereitgestellt. Es wird ein Werkzeug gebraucht, das Programmtexte generiert, beide Arten von Kontext zusammenführt und dem *Advice Code* zur Verfügung stellt. Die eigentliche Aktivierung von dynamischem Advice geschieht mit hilfe von Monitoren, die von einem Instrumentierungsaspekt in den Anwendungscode installiert werden.

6 Dynamische Einfügungen

Ein Sprachmerkmal, das von eher wenigen dynamischen Webern unterstützt wird, sind Einfügungen in bestehende C++ Strukturen. Der statische Weber `ac++` unterstützt die Einfügungen von neuen Klassen, Attributen, Typaliase (`typedefs`), Aufzählungen und neuen Basisklassen. Für eine vollständige Umsetzung des *Single Language Approach* nach Unterabschnitt 3.5 ist es notwendig, diese Einfügungen auch im Rahmen dynamischen Webens zu ermöglichen. Die Sprache AspektC++ erlaubt die Einfügung von unterschiedlichen Sprachelementen. In diesem Abschnitt wird zunächst gezeigt, welche Auswirkung die Einfügung in Typen hat, gefolgt von der Vorstellung der einzelnen Arten von Einfügungen. Anschliessend werden die entwickelten Techniken vorgestellt, mit denen diese technisch zur Laufzeit umgesetzt werden können.

In statisch getypten Sprachen wie `C`/`C++` macht der Übersetzer bei der Programmcodeerzeugung Annahmen über die Struktur von Typen, die im Programmtext referenziert werden. Man kann konzeptionell sagen, dass der Komponentencode zur Übersetzungszeit eine feste *Sicht* auf die von ihm benutzten Typen hat. Abbildung 15a verdeutlicht diesen Sachverhalt. Wird in diesem Programm ein Aspekt gewoben, der Elemente in einen Typ einfügt (Abbildung 15b links), so ist die Auswirkung für das Programm, dass Komponentencode, der diesen Typ verwendet, eine *andere* Sicht auf diesen Typ hat (Abbildung 15b rechts).



(a) Vor dem Weben **(b)**: Nach dem Weben

Abbildung 15: Sicht des Programmtextes auf Klassen mit Einfügungen

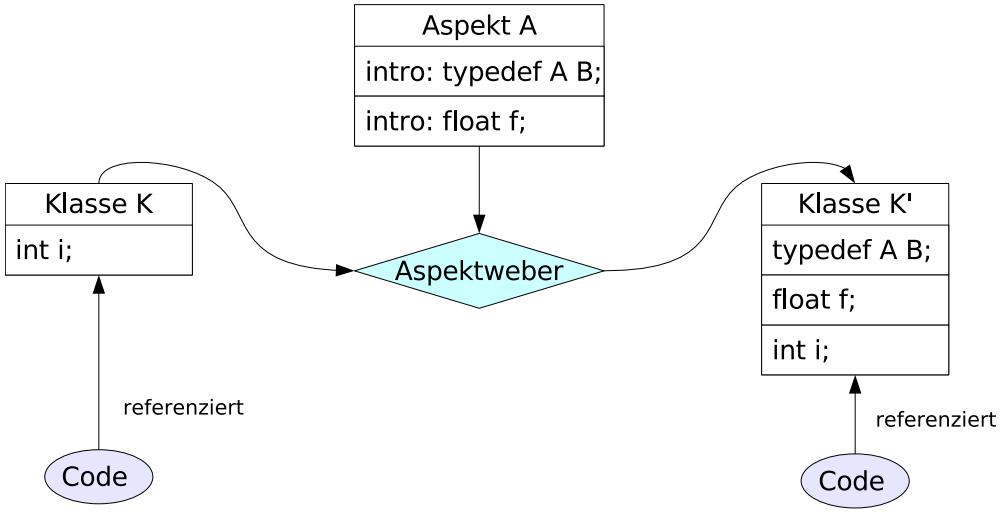


Abbildung 16: Manipulation einer Klasse K durch Einfügungen

Dieser wird Umstand dadurch realisiert, dass der Aspektweber eine Klasse K, die von einem Aspekt A erweitert wird, in eine neue Klasse K' überführt (Abbildung 16). Dies ist ein technisches Detail; der Programmierer hat aus der Sprache AspectC++ keine Möglichkeit, die transformierte Klasse K' direkt zu referenzieren. Sprachlich existiert im Programmtext nur die Klasse K. Eine Einfügung kann je nach der Art des konkret eingefügtem Elements dazu führen, dass die Klassen K und K' eine andere Struktur besitzen. Dies ist genau dann ein Problem, wenn es in einem System verschiedene Arten von Programmcode gibt, bei deren Übersetzung der Übersetzung einmal eine Sicht auf die Klasse K und einmal auf die Klasse K' hat. Beim dynamischen Weben von Aspektmodulen ensteht genau dieser Sachverhalt.

Innerhalb eines Programms darf der Programmcode nur dann wahlweise K oder K' ohne weiteres referenzieren, wenn sichergestellt ist, dass die beiden Klassen *strukturverträglich* sind. Andernfalls würde das Programm mit inkonsistenten Datenstrukturen arbeiten, was undefiniertes Verhalten zur Laufzeit zur Folge hätte. Im Falle von statischem Weben geschieht diese Transformation zur Übersetzungszeit. Dadurch wird sichergestellt, dass es im Programm kein Code existiert, der eine Sicht auf die unmodifizierte Klasse K besitzt, sondern alle Programmteile mit der transformierten Klasse K' arbeiten. Im Falle dynamischen Webens wird ist die durch Aspekteinwirkung modifizierte Sicht nur für Code sichergestellt, der im *selben* (oder später geladenem) Erweiterungsmodul übersetzt wurde. Der Programmcode in der Basisanwendung arbeitet hingegen auf der unmodifizierten Klasse K. Der dynamische Weber hat also Sorge zu tragen, dass für *strukturunverträgliche*

Einfügungen Vorkehrungen getroffen werden, um das oben angesprochene undefinierte Verhalten zu vermeiden.

6.1 Attribute

```
1  #include <iostream>
2
3  struct K {
4    int i;
5  };
6
7  int main() {
8    K array[5];
9    printf ("Size: %d bytes\n", sizeof(array));
10   return 0;
11 }
```

Abbildung 17: Eine einfaches Programm

Um die Problemstellung des Einfügens von Attributen in eine Anwendung zu veranschaulichen, wird die (triviale) Anwendung in Abbildung 17 betrachtet. Sie definiert eine Struktur K, welche ein `int` Attribut enthält. In der Funktion `main()` wird ein Feld mit fünf Elementen angelegt und anschließend die Grösse dieses Feldes ausgegeben. Wird dieses Programm mit einem C++ Übersetzer übersetzt, so wird die Ausgabe (unter der Annahme, dass ein `int` eine Größe von 32bit hat) `20 bytes` sein. Wird jedoch das Programm zusammen mit dem Aspekt aus Abbildung 18 mit einem AspectC++ Übersetzer übersetzt, so ist die Ausgabe `40 bytes`. Man sieht deutlich, dass die Einfügung des Aspekts A die Struktur der Klasse K ändert. Dies hat Auswirkungen auf jeden Programmcode, der die Struktur K verwendet.

```
1  struct K;
2  aspect A {
3    advice "K" : slice struct {
4      int j;
5    };
6  };
```

Abbildung 18: Einfügung eines Attributs mittels eines Aspekts

Falls der Aspekt statisch zur Übersetzungszeit der Anwendung gewoben wird ist dies kein weiteres Problem. Hier ist sichergestellt, dass kein Programmcode existiert, der mit der *Sicht vor* der Webertransformation übersetzt wurde. Falls der Aspekt jedoch als Erweiterungsmodul zur Laufzeit in ein System

eingebracht wird, besteht die Anwendung aus Programmcode aus der Anwendung, der mit der Sicht *vor* der Webertransformation und Programmcode aus dem Erweiterungsmodul, der mit der Sicht *nach* der Webertransformation übersetzt wurde. Die Klassen K und K' aus Abbildung 16 auf Seite 16 sind in diesem Fall also *struktur**un**gleich*.

Um ein definiertes Verhalten der Anwendung auch nach dem Laden des Aspektmoduls sicherzustellen, existieren grundsätzlich zwei Ansätze. Einerseits könnte die Laufzeitumgebung das Programm *geeignet* modifizieren. Da Anwendungen in der Sprache `C++` für gewöhnlich nicht innerhalb einer virtuellen Machiene ausgeführt werden, ist dieser Ansatz sehr aufwändig und darüber hinaus implementierungsabhängig. Es wird daher ein Ansatz gewählt, der zur Übersetzungszeit der Erweiterungsmodule dafür sorgt, dass Module, die *dynamisch* eingefügte Attribute benutzen, nicht direkt auf Strukturen und Typen arbeiten, in die diese Attribute *statisch* eingefügt worden sind. Stattdessen muss der Zugriff auf geeignete Hilfsstrukturen transformiert werden.

6.1.1 Instantiierung der eingefügten Attribute

Bei dynamischen Einfügungen von Attributen stellt sich außerdem die Frage, wann die eingefügten Attribute instantiiert werden sollen, falls Instanzen von Klassen, in die eingefügt wurde, bereits existieren. Dies ist insbesondere dann kritisch, wenn Attribute eingefügt werden, deren Konstruktoren Seiteneffekte mit sich bringen. Der Instanziierungszeitpunkt der Objekte, in die eingefügt wird, kommt nicht in Frage, denn zu diesem Zeitpunkt ist der Aspekt, und damit die Attribute noch nicht im Programm vorhanden. Es bleiben also noch zwei Alternativen: Zum Ladezeitpunkt des Aspekts (Semantik *„immediate“*), oder zum Zeitpunkt des ersten Zugriffs auf diesen Attribut (Semantik *„lazy“*).

Die *„immediate“* Semantik bedeutet, dass die Lebenszeit von Attributen in Instanzen, in die dynamisch eingefügt wurde, zum Ladezeitpunkt des Aspekts beginnt, und entweder mit der Destruktion *ihrer* Instanz oder des Aspekts endet. Hierzu müssen zum Ladezeitpunkt des dynamischen Aspektes alle Instanzen von denjenigen Klassen bekannt sein, in die der dynamische Aspekt ein Attribut einfügt. Es ist also eine Datenstruktur nötig, die Kenntnis von sämtlichen Konstruktionen und Destruktionen von Objekten und Aspekten im System hat. Diese kann dann die Konstruktion und Destruktion von dynamisch eingefügten Attributen vornehmen.

Die *„lazy“* Semantik hingegen besagt, dass die Lebenszeit von eingefügten Attributen mit dem *ersten Zugriff* auf diese beginnt. Dadurch ist es nicht

mehr nötig, (zum Ladezeitpunkt des Aspektmoduls) die Existenz von *allen* Instanzen im System zu kennen. Eingefügte Attribute können ja nur von Programmcode aus dem Erweiterungsmodul referenziert werden. Dadurch ist sichergestellt, dass zu jedem Zugriff auf diese Attribute ihr dazugehöriger Code vorhanden ist. Die Lebenszeit der Attribute kann zu zwei Zeitpunkten enden. Der erste Zeitpunkt ist der Zeitpunkt, zu dem die Instanz in die sie eingefügt wurden, entfernt wird. Der andere Zeitpunkt ist der Entladezeitpunkt des Aspektmoduls, an dem sämtliche noch nicht entfernte Instanzen von Attributen aus den Instanzen der Klassen, in die eingefügt wurde, entfernt werden müssen.

6.1.2 Introduction Manager

Der erste Ansatz zur Realisierung von eingefügten Attributen auf Basis von Codetransformation ist es, einen *Introduction Manager* einzuführen, bei dem alle Aspekte ihre Einfügungen registrieren. Das Problem hierbei ist die Effizienz. Einerseits sind dann zum Zugriff auf diese eingefügte Attribute jeweils zwei Indirektionen nötig: Einmal im Manager für das jeweilige Objekt, und einmal für den einfügenden Aspekt, wenn man Einfügungen von mehreren Aspekten erlaubt.

Abbildung 19: Dynamische Einfügungen von Attributen werden bei einem *Introduction Manager* registriert. (1) Bei jeder Instantiierung eines Objektes, in das potentiell dynamisch neue Attribute eingefügt werden können, meldet sich dieses Objekt beim *Introduction Manager* an. (2) Wird ein dynamischer Aspekt aktiv, teilt dieser dem *Introduction Manager* mit, welche Attribute in welche Klassen nun eingefügt wurden. Der Zugriff auf diese Attribute erfolgt immer über diesen *Introduction Manager*.

Der Vorteil bei diesem Verfahren ist, dass das an den Originalklassen keine Veränderungen nötig sind. Dies erlaubt dynamisches Weben in Teile von Bibliotheken, die von bestehende Anwendungen benutzt werden. Dies ist insbesondere dann wichtig, wenn die Anwendung nur in Binärform vorliegt und ein erneutes Übersetzen nicht in Frage kommt.

Der *Introduction Manager* ist geeignet, um sowohl die *immediate* als auch die *lazy* Semantik zu realisieren. Da der Aspektlader alle dynamischen Einfügun-

gen bei ihm registriert, kann dieser zum Ladezeitpunkt des Aspekts prüfen, welche Instanzen von Klassen, in die dynamisch eingefügt wurde, existieren. Die eingefügten Attribute werden je nach gewünschter Semantik entweder sofort oder beim ersten Zugriff instantiiert.

Um Attribute in Objekten zu instantiieren, nachdem der Aspekt geladen wurde, ist es nötig, dynamisch *construction* Advice in die Klassen zu weben, um dieses Ereignis beim *Introduction Manager* zu signalisieren. Genauso muss *destruction* Advice gewoben werden, damit eingefügte Attribute zusammen mit der Objektinstanz entfernt werden können. Beim Entladen der Aspektmodule können alle (noch nicht entfernten) eingefügten Attribute problemlos entfernt werden, da sowohl alle Instanzen als auch der Aspekt, welcher die Einfügungen gemacht hat, bekannt sind.

6.1.3 Introduction Pointer

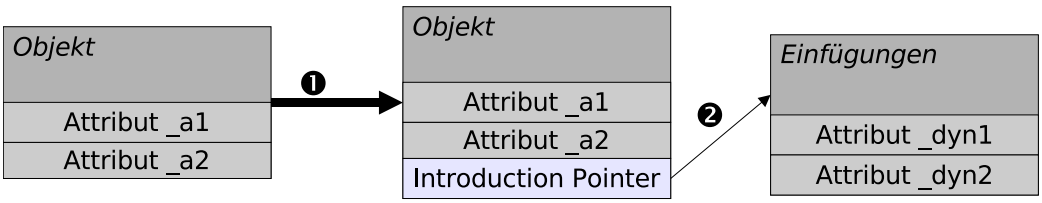


Abbildung 20: Dynamische Einfügungen von Attributen mittels *Introduction Pointer*. Es wird in jede Klasse, in die zur Laufzeit Attribute eingefügt werden können ein Zeiger *statisch* eingefügt. Dieser zeigt dann auf eine Hilfsstruktur, die die eingefügten Attribute hält.

Der zweite Ansatz erweitert bestehende Klassen um einen *Introduction Pointer*. Dabei wird ein Zeiger in Klassen eingefügt, an dem Erweiterungsobjekte angefügt werden. Diese Erweiterungsklassen ergeben sich aus allen *dynamischen* Einfügungen aus dem *Aspektmodul*. Daraus folgt, dass für jedes neue Erweiterungsmodul eine neue Erweiterungsklasse erzeugt wird. Der Aufwand für den Zugriff auf ein *dynamisch* eingefügtes Attribut ist damit linear zur Anzahl der geladenen Aspektmodule. Der Erweiterungszeiger wird im Rahmen des statischen Instrumentierungsaspekts mittels *construction Advice* initialisiert. Sowohl der Erweiterungszeiger als auch dessen Initialisierung werden *statisch* in die Basisanwendung gewoben.

Um auf dynamisch eingefügte Attribute in Objektinstanzen zugreifen zu können, die bereits vor dem Zeitpunkt des Ladens des Aspektmoduls existieren,

ist es nötig, dass das Laufzeitsystem für jeden Zugriff auf diese Attribute eine Überprüfung vornimmt, ob die Attribute bereits konstruiert wurden. Falls (noch) nicht, werden sie als *Seiteneffekt* des Zugriffs konstruiert. Dies entspricht der Semantik *lazy*. Die Semantik *immediate* kann mit diesem Ansatz nicht realisiert werden.

Um nach dem Laden eines Aspektmoduls dessen eingefügten Attribute in neu erstellten Objektinstanzen zu initialisieren bzw. bei Zerstörung der Objektinstanzen die dynamisch eingefügten Attribute zu entfernen, ist es nötig, diese Ereignisse mit Hilfe eines Konstruktors bzw. eines Destruktors im Erweiterungszeiger dem Laufzeitsystem zu signalisieren. Wenn dem Laufzeitsystem signalisiert wird, dass eine Objektinstanz entfernt wurde, muss die Hilfsstruktur aus Abbildung 20 entfernt werden. Dies wird dadurch erreicht, dass diese Hilfsstrukturen von einer gemeinsamen Basisklasse mit virtuellem Destruktor erben. Dadurch erzeugt der Übersetzer Programmcode, der bei der Destruktion die Attribute aus den abgeleiteten Klassen ebenfalls entfernt. Das Laufzeitsystem braucht also nicht die Struktur der Hilfsstrukturen kennen, sondern es reicht, die gemeinsame Basisklasse vorzugeben. Falls die Signalisierung von einem Erweiterungszeiger stammt ist dies problemlos durchführbar, weil dieser auf genau diese Hilfsstruktur zeigt. Falls jedoch ein Aspektmodul entladen wird müssen **alle** verbleibenden Hilfsstrukturen aus dem System entfernt werden. Um dies zu realisieren bietet es sich an, **alle** Hilfsstrukturen eines Erweiterungsmoduls in einer doppelt verketteten Liste zu halten. Dadurch ist sichergestellt, dass die sowohl das Einfügen als auch das Entfernen eines Hilfsstrukturenelements mit konstantem Aufwand durchgeführt werden kann.

6.1.4 Zugriff auf eingefügte Attribute

Unabhängig von der Verwendung des *Introduction Managers* oder des *Introduction Pointer* Modells bleibt der Zugriff auf diese problematisch. Dank des *Single Language Approach* (siehe Unterabschnitt 3.5) können Aspektmodule Einfügungen auf der Sprachebene ganz normal verwenden, obwohl die Einfügungen in Wirklichkeit nicht in den Strukturen vorhanden sein können. Aus diesem Grund müssen die Erweiterungsmodule mit einem Werkzeug so modifiziert werden, dass sie geeignete Hilfsfunktionalitiät aus einer Laufzeitumgebung benutzen. Dieser Unterabschnitt beschreibt nun Lösungsansätze für dieses Problem.

Das Problem Zugriffe auf Attribute zu erkennen ist an sich quer schneidend, womit eine aspektorientierte Lösung in AspectC++ in Betracht kommt. So

wäre ein Aspekt denkbar, der alle Zugriffe auf *dynamisch* eingefügte Attribute durch geeignete Aufrufe in das Laufzeitsystem ersetzt. Leider bietet die Sprache AspectC++ keine sprachlichen Mittel zur Realisierung eines solchen Aspekts.

In AspectJ gäbe es für diesen Zweck *get-* und *set- join points*. Leider existiert in AspectC++ kein vergleichbares Sprachmerkmal. Tatsächlich würden derartige *join points* in AspectC++ auch nicht das Problem der Erkennung von allen möglichen Zugriffen vollständig lösen, denn C++ erlaubt sehr viele Möglichkeiten zur Erzeugung von Aliasen. Wenn das für C++ Anwendungen übliche Ausführungsmodell nicht geändert werden soll, ist es unpraktikabel *alle* Arten von Aliasen zu erkennen.[4] Die Infrastruktur muss daher eine andere Lösung anbieten.

Um beim üblichen Ausführungsmodell für C++ Anwendungen zu bleiben, wird im Rahmen dieser Arbeit eine Lösung entwickelt, die auf der Basis von Quelltexttransformation arbeitet. Um Einfügungen mit Hilfe von `ac++` zu realisieren, liegt es also nahe, die von `ac++` transformierten Quellen von einem Postprozessor nachträglich *korrigiert* werden. Die Aufgabe dieser Korrektur ist es, die durch die Transformation verursachte *Strukturunverträglichkeit* in den einzelnen Datenstrukturen zu korrigieren, indem einerseits die eingefügten Attribute entfernt werden, und andererseits gleichzeitig Zugriffe auf *dynamisch* eingefügte Attribute in den Erweiterungsmodulen durch Aufrufe von geeigneten Hilfsfunktionen aus der Laufzeitumgebung ersetzt werden.

6.1.5 Postprozessors für dynamische Einfügungen

Der Postprozessor besitzt also zwei Aufgaben: Die Entfernung von dynamisch eingefügten Attributen, sowie die Ersetzung der Zugriffe auf diese durch Aufrufe in die Laufzeitumgebung.

Dabei muss darauf geachtet werden, dass nur dynamische Einfügungen entfernt werden, die in dem aktuell zu übersetzendem Modul dynamisch eingefügt wurden. Der Grund dafür soll nun etwas veranschaulicht werden. Abbildung 21 zeigt die schrittweise dynamische Erweiterung eines Softwareprojekts. Die Einfügungen von Aspekten werden aufgeteilt in Einfügungen in Klassen, die im aktuellen Erweiterungsmodul eingefügt werden, und in Einfügungen, die in Ebene $n-1$ eingefügt werden. Erste geschehen *statisch*, letztere *dynamisch*. Damit der Postprozessor erkennen kann, welche eingefügten Attribute zu entfernen sind, wird der betroffene Typ im *project repository*

[4] vgl. http://aspectc.org/index.php?id=12&tx_faq_faq=8

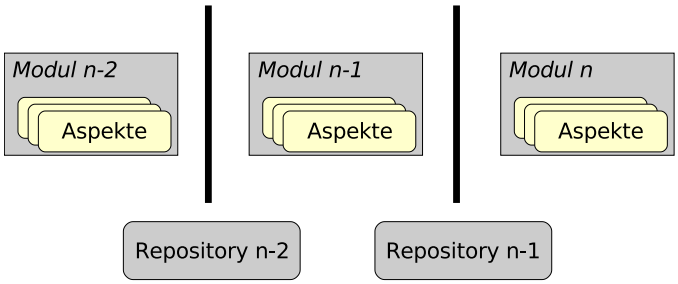


Abbildung 21: Ein Softwaresystem wird schrittweise mittels dynamisch geladenen Modulen erweitert. Bei jeder Erweiterung entsteht ein neues *project repository* für diese Modulebene

der Ebene $n-1$ nachgeschlagen. Ist er dort bekannt, so weiß er, dass diese Einfügung *dynamisch* geschieht und damit diese Einfügung *korrigiert* werden muss. Ist der Typ dort nicht bekannt, so kann davon ausgegangen werden, dass der Typ im aktuellen Modul neu eingeführt wurde, und die Einfügung *statisch* erfolgen kann.

Um den Postprozessor effizient zu implementieren, wird vermieden, die Erweiterungsmodule syntaktisch und semantisch zu analysieren. Diese syntaktische und semantische Analyse ist bereits von dem statischem Weber durchgeführt worden und wird idealerweise als Zwischenergebnis weiterverwendet. Tatsächlich ist es für den statischen Aspektweber `ac++` technisch mit wenig Aufwand möglich, diese Einfügungen sowie die Zugriffe auf diese zu erkennen zu markieren. Der Postprozessor liest dann den annotierten Programmtext zeilenweise ein, und ersetzt dabei die *Zugriffe* auf *dynamisch* eingefügte Attribute durch Aufrufe in das Laufzeitsystem.

Liest der Postprozessor eine Markierung für einen Zugriff auf einen *dynamisch* eingefügtes Attribut ein, so muss geprüft werden, ob diese Einfügung *dynamisch* oder *statisch* geschieht. Im Falle einer dynamischen Einfügung muss der Zugriff durch einen Aufruf in das Laufzeitsystem ersetzt werden, der die Einfügung über die jeweils verwendete Hilfsstruktur realisiert. Im Falle einer *statischen* Einfügung braucht keine Modifikation geschehen.

6.1.6 Kollisionsauflösung

Wenn zwei Aspekte ein Attribut mit gleichem Namen in dieselbe Klasse einfügen entsteht eine Kollision. Im *statischen* Fall prüft `ac++` dies und verbietet dies mit einer Fehlermeldung. Dies gilt im Prinzip auch im *dynamischen* Fall. In der Tat können zwei Aspekte Attribute mit gleichem Namen ein-

fügen, wenn sichergestellt ist, dass die zwei Aspekte nicht in verschiedenen Aspektmodulen enthalten sind.

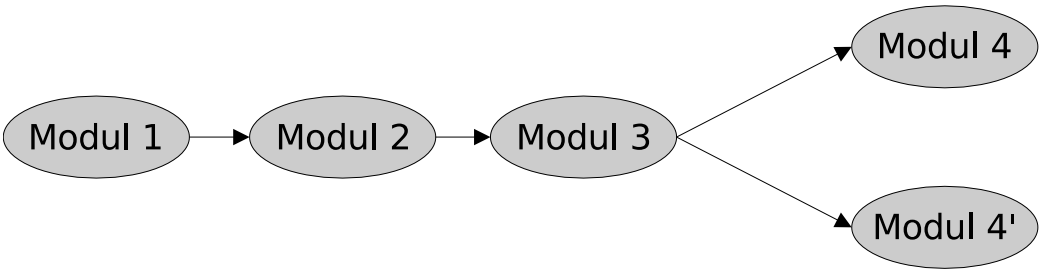


Abbildung 22: Sei Modul 1 die instrumentierten Basisanwendung. Es werden nach einander Modul 2 und 3 nachgeladen. Modul 4 und 4' können als *Geschwistermodule* nur wahlweise geladen und entladen werden.

Abbildung 22 zeigt eine schrittweise dynamische Erweiterung einer laufenden Anwendung. Modul 4 und 4' können durchaus jeweils einen (dynamischen) Aspekt enthalten, die in dieselbe Klasse ein Attribut gleichen namens einfügen, wenn das Laufzeitsystem, genauer der Aspektlader, oder der Benutzer sicherstellt, dass diese *Geschwistermodule* nicht *gleichzeitig* aktiviert und geladen werden können. Im Allgemeinen können also Aspekte im dynamischen Fall Attribute gleichen Namens einfügen, wenn Kollisionen nur in Geschwistermodulen stattfinden, d.h. der Pfad von Ebene 0 auf Ebene n kollisionsfrei ist.

Um sicherzustellen, dass Geschwistermodule nicht *gleichzeitig* geladen werden können, wird der Aspektlader so verändert, dass er im Laufzeitsystem eine Epochennummer mitzählt, beginnend bei 0. Jedes geladene Erweiterungsmodul besitzt eine Modulnummer, entsprechend wie in Abbildung 22. Beim Laden eines Moduls wird der Epochenzähler inkrementiert, beim Entladen dekrementiert. Der Lader stellt sicher, dass das zu ladende Modul immer eine Modulkennung der aktuellen Epoche +1 hat. Damit ist gewährleistet, dass Geschwistermodule nicht gleichzeitig aktiv sein können. Gleichzeitig wird erlaubt diese Implementierung, dass sie nacheinander geladen und entladen werden können.

Ein häufiger Anwendungsfall von Geschwistermodulen ist z.B. eine Reihe von Erweiterungsmodule zum Nachvollziehen des Programmablaufs (engl. *Tracing*), die auf eine bestimmtes Teilsystem spezialisiert sind. Diese können dann z.B. schrittweise verfeinert werden, bis ein dynamischer Aspekt gefunden ist, dessen Ausgabe aussagekräftig genug ist.

6.2 Aufzählungen und Typaliase

Abbildung 23 zeigt eine Klasse K, in die ein Aspekt A ein `enum` und ein `typedef` einfügt. Dies kann in der Sprache AspectC++ mit dem Codebeispiel aus Abbildung 24 realisiert werden. Man sieht deutlich, dass der Programmcode aus der Funktion `introduced()` auf Elemente der transformierten Struktur K' zugreifen kann, obwohl im Programmtext lediglich die Struktur K referenziert wird. Seine *Sicht* auf K entspricht also der Struktur K'.

Sei der Programmtext aus Abbildung 24 ein zur Laufzeit zu ladendes Erweiterungsmodul mit den Definition der Klasse K in Zeile 11 in der Basisanwendung. In diesem Fall wird der Code der Basisanwendung eine *Sicht* der Klasse K haben, wie sie in Abbildung 23 links abgebildet ist. Da das Erweiterungsmodul zum Zeitpunkt der Entwicklung nocht nicht *bekannt* sein kann, können hier keine anderen als in der Basisanwendung deklarierten Elemente referenziert werden. Wird das Erweiterungsmodul nun zur Laufzeit gewoben, ist seine *Sicht* auf die Klasse K also die von K', so wie in Abbildung 23 rechts. Dieses Modul kann ohne weitere Anpassungen am Programmtext des Erweiterungsmoduls übersetzt werden, weil die Strukturen K und K' *strukturverträglich* sind.

6.3 Methoden, statische Klassenvariablen und -funktionen

Weitere *strukturverträgliche* Einfügungen sind Einfügungen von nicht virtuellen Methoden, statischen Klassenvariablen sowie von statischen Klassenfunktionen dar. Sie erweitern zwar die Struktur von Klassen, und sind damit erst einmal als kritisch einzustufen. In der Tat ist es jedoch so, dass die Transformation von Klassen mit dieser Art von Einfügungen durch einen statischen Aspektweber, wie sie in Abbildung 16 auf Seite 38 veranschaulicht wird, dazu führt, dass die vom Übersetzer erzeugten Strukturen *strukturverträglich* sind.

Der Übersetzer beachtet die neuen Elemente nur für den Programmtext, der diese neuen Elemente benutzt. Von Operatoren wie dem `sizeof()` Operator, werden die Sprachelemente nicht virtuelle Methoden, statische Klassenvariablen und Klassenfunktionen nicht beachtet. Technisch ist es so, dass der C++ Übersetzer für diese Art von Sprachelementen lediglich neue Symbole erzeugt, die von anderen Übersetzungseinheiten, wie nachgeladenen Aspektmodulen, referenziert werden können. Es ist daher unkritisch, diese Symbole für Datentypen aus der Basisanwendung in Aspektmodulen *nachzuliefern*.

Ein gemeinsame Eigenschaft von *einfachen* Einfügungen ist es, dass sie alleine

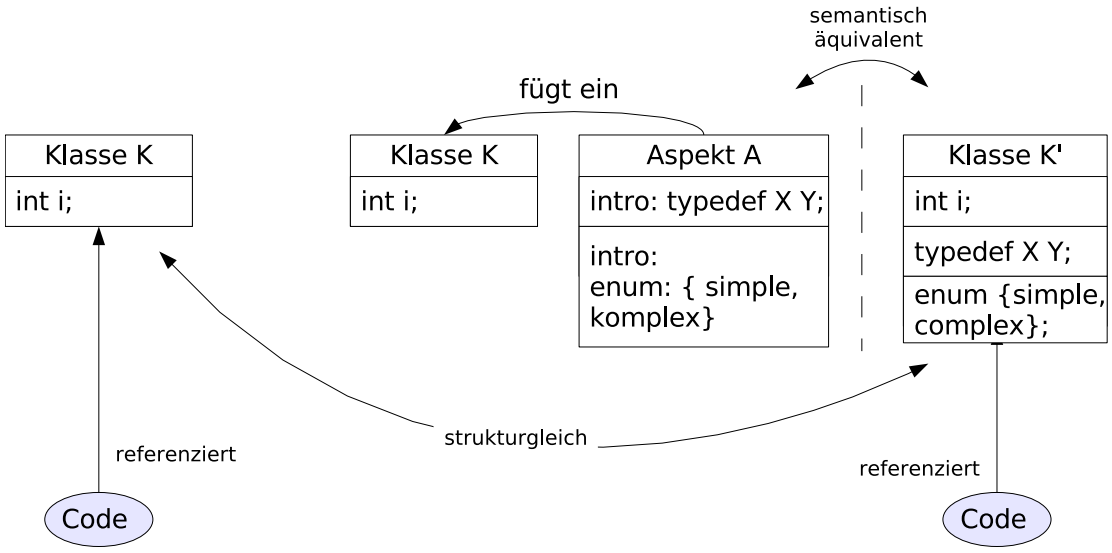


Abbildung 23: Einfügung von Aufzählungen und Typaliasen

```
1  struct X {};
2  struct K;
3
4  aspect A {
5    advice "K" : slice struct {
6      typedef X Y;
7      enum { simple, complex };
8    };
9  };
10
11 struct K {
12   int i;
13 };
14
15 int introduced() {
16   K::Y x;
17   K k;
18   k.i = K::complex;
19 }
```

Abbildung 24: AspectC++ Programm zur Abbildung 23

noch keinerlei semantische Auswirkungen auf das laufende Programm haben. Es wird lediglich dem dynamischem Binder neue Symbole bekannt gemacht, die von Aspektmodulen aufgelöst und referenziert werden können. Diesen Umstand nutzt übrigens auch der Aspektlader aus, um dynamischen Advice in Form von neu eingefügten Symbolen (Funktionen) am Laufzeitsystem in der Basisanwendung zu registrieren.

6.4 Virtuelle Methoden

Ein wesentliches Sprachkonzept von objektorientierten Programmiersprachen ist der Begriff des *Polymorphismus*. In Sprachen wie C++ versteht man darunter die Fähigkeit von Objekten (von möglicherweise unterschiedlichen Klassen) auf Methodenaufrufe mit demselben Namen unterschiedlich zu behandeln. Der Programmierer braucht dabei den genauen Typ des Objekts, dessen Methode er aufruft, nicht im Voraus zu kennen. Stattdessen wird der Methodenaufruf vom Laufzeitsystem während der Programmausführung aufgelöst. Die dazu nötige Technik nennt man auch *spätes* oder *dynamisches* Binden (engl. *late/dynamic binding*).

In der Sprache C++ werden nur *virtuelle* Methoden dynamisch gebunden. Virtuelle Methoden sind Methoden, für die C++ Programme zur Laufzeit prüfen, welche Methodenimplementierung für ein konkretes Objekt benutzt werden soll. Dies ist dann nicht eindeutig, wenn in einer abgeleiteten Klasse eine virtuelle Methode überschrieben wurde. Man bezeichnet diesen Vorgang auch als dynamisches Binden.

Der Codeausschnitt aus Abbildung 25 demonstriert wie virtuelle Methoden in `C++` verwendet werden: `B` sei eine von `A` abgeleitete Klasse. `A` definiert eine Methode `v()`, die in der abgeleiteten Klasse überschrieben wird. In der Funktion `main()` wird eine Instanz der Klasse `B` mit einer Zeigervariablen des Typs `A*` referenziert. Der Übersetzer kann im Allgemeinen nicht erkennen, dass bei einem Methodenaufrufs an einer Zeigervariablen auf `A*` die Methode aus der Klasse `B` aufgerufen werden soll. Stattdessen erkennt er, dass die Methode `v()` *virtuell* ist, und erzeugt Programmcode für eine Überprüfung zur Laufzeit, welche Methode tatsächlich aufgerufen werden soll. Diese Überprüfung wird im Folgenden *dynamic dispatch* genannt.

Zur Implementierung des *dynamic dispatch* benutzen C++-Übersetzer eine so genannte *virtuelle Funktionstabelle* oder auch *v-table* genannt, die für alle *polymorphe* Klassen angelegt wird. Polymorphe Klassen sind Klassen, die mindestens eine virtuelle Methode haben. Abbildung 26 veranschaulicht die Funktionsweise der *v-table* für das Programm aus Abbildung 25. Jede In-

```
1   struct A {
2     virtual void v() {
3       printf("A");
4     }
5     void n() {} // nicht virtuelle Methode
6   };
7
8   struct B : public A {
9     void v() { // implizit virtuell
10      printf("B");
11    }
12  };
13
14  int main() {
15    A *a = new B;
16    a->v(); // gibt "B" aus
17  }
```

Abbildung 25: Virtuelle Methoden in C++

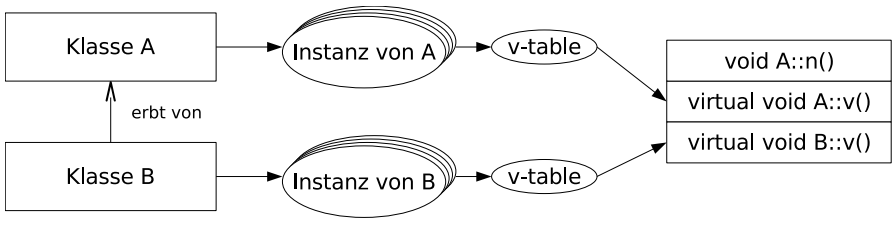


Abbildung 26: Zusammenhang von Instanzen, v-tables und Klassenfunktionsimplementierungen für das Programm aus Abbildung 25

stanz einer polymorphen Klasse besitzt dabei eine *v-table*. Sie verweist auf die Funktionsimplementierungen, die zu *ihrer* Klasse gehören. Dadurch kann der Übersetzer dafür sorgen, dass zur Laufzeit ein Methodenaufruf einer Instanz immer die *richtige* Implementierung wählt.

Problemstellung

Werden nun virtuelle Methoden in Klassen *dynamisch* eingefügt, ist zu beachten, dass diese Art von Einfügungen Auswirkungen auf die virtuelle Funktionstabelle der betroffenen Klassen haben können. Wenn wie bei den bisher vorgestellten Lösungsansätzen das für C++ übliche Ausführungsmodell nicht geändert werden soll, ergeben sich prinzipiell verschiedene Möglichkeiten zur Umsetzung der dynamischen Einfügung von virtuellen Methoden. Eine Möglichkeit besteht darin, bestehende *v-tables* zur Laufzeit zu Manipulieren. Dieser Ansatz wurde für die Sprache C++ in [Che03] durchgeführt. Ein großes Problem an diesem Ansatz ist die Tatsache, dass sie von der konkreten Implementierung des C++ Übersetzers abhängig ist. Es geht dabei der Vorteil der Portabilität verloren. Aus diesem Grund wird eine andere Lösung erarbeitet, welche auf Basis von Quellcodetransformation arbeitet.

Unkritisch sind Einfügungen von Klassen mit virtuellen Methoden. Diese können sogar von bestehenden polymorphen oder nicht polymorphen Klassen erben. Dadurch wird keine bereits bestehende virtuelle Funktionstabelle *geändert*, sondern *nur* in Instanzen von neuen Klassen erweitert.

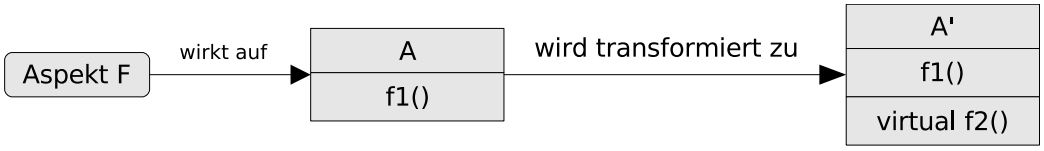


Abbildung 27: Der Aspekt `F` erweitert die Klasse `A` um eine virtuelle Methode `f2()`. Die dabei entstehende Klasse `A'` wird dabei *polymorph*.

Problematischer ist das Einfügen von virtuellen Methoden in bestehende Klassen. Abhängig davon, ob die Klasse bereits *polymorph* war oder nicht, würde die Einfügung von virtuellen Methoden die virtuelle Funktionstabelle entweder anlegen oder erweitern. Dieser Umstand muss bei der Übersetzung von dynamischen Aspekten analog zu den dynamischen Einfügungen von Attributen behandelt werden.

6.4.1 Virtuelle Funtkionstabellen

Um den *dynamic dispatch* zur Laufzeit durchführen zu können, ohne die virtuelle Funktionstabelle der Klasse `A` zu verändern, muss eine neue Art von *dynamic dispatch* eingeführt werden, der nicht mit der herkömmlichen, sondern mit einer *anderen* Hilfsstruktur arbeitet. Diese andere Hilfsstruktur wird im Folgenden *dynamic v-table* genannt.

Die *dynamic v-table* ist eine Tabelle von Funktionszeigern. Sie existiert für jede Klasse, in die *dynamisch* virtuelle Methoden eingefügt werden. Sie wird für jedes Objekt dieser Klassen erzeugt. In dieser Tabelle werden Funktionszeiger auf die Methodenimplementierungen derjenigen Klasse notiert, von dessen Typ das Objekt ist. Bei jedem Aufruf der virtuellen Methode muss in der *dynamic v-table* des gegebenen Objekts nachgeschlagen werden, auf welche Implementierung der Funktionszeiger in dieser Tabelle zeigt. Das Verfahren ist also analog zum weiter oben beschriebenen statischem *dynamic dispatch*.

Diese *dynamic v-tables* werden ähnlich wie die Hilfsstrukturen für eingefügte Attribute pro Klasse generiert (siehe Unterabschnitt 6.1.3). Diese werden per *dynamic introduction* (siehe auch Kapitel 6), in jede Klasse, in die dynamisch virtuelle Methoden eingefügt werden, eingefügt. Alle Stellen, an denen der *dynamic dispatch* für dynamisch eingefügte Methoden durchgeführt wird, werden so angepasst, dass diese Hilfsstrukturen anstatt der statischen *v-tables* berücksichtigt werden.

Abbildung 28 zeigt einen komplizierteren Fall einer Einfügung einer virtuellen Methode. Hier wird die Methode `M::f2()` durch den Aspekt A, der in die Basisklasse `K` die Methode `virtual f2()` einfügt, *virtualisiert*. Da die Deklaration von virtuellen Methoden transitiv in der Vererbungshierarchie ist, muss hier zusätzlich im Code zum Aufruf der Funktion aus der *Zwischenklasse* `L::f2()` geprüft werden, ob die Instanz nicht von einer von `L` abgeleiteten Klasse, wie z.B. die Klasse `M`, ist. In diesem Fall muss an dieser Stelle zusätzlich ein *dynamic dispatch* wie weiter oben beschrieben durchgeführt werden. Die Klasse `L` benötigt hingegen keine weitere Behandlung, da sie keine Methoden der Basisklasse(n) überschreibt.

Zur Realisierung des Falls, dass eine Methode in einer abgeleiteten Klasse durch die Einfügung eines Aspekts zur Laufzeit *polymorph* wird, muss der vom Übersetzer generierte *dynamic dispatch* nachgebildet werden. Hierzu wird wird bei der Übersetzung des dynamischen Aspektmoduls ein Hilfsaspekt erzeugt, der mit *around advice* an den nötigen Stellen prüft, ob ein dynamischer Aspekt diese Methode virtualisiert hat und gegebenenfalls einen

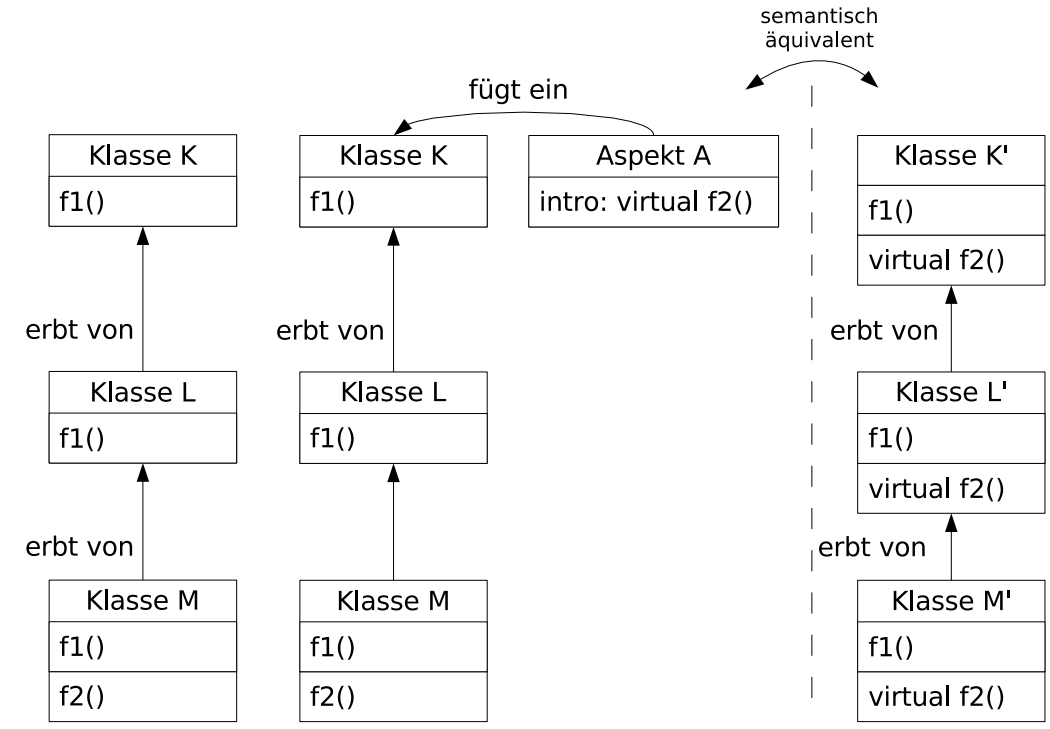


Abbildung 28: Sicht des Programmcodes auf Klassen mit Einfügungen

dynamischen *dynamic dispatch* durchführt.

6.4.2 Dynamisch statische virtuelle Funktionstabellen

Keine Implementierung eines *dynamic dispatch* in der Sprache C++ ist ähnlich effizient und portabel einsetzbar wie die, die vom C++ Übersetzer selbst erzeugt wird. Daher bietet es sich an, die eigentliche Implementierung des dynamischen *dynamic dispatch* vom Übersetzer erledigen zu lassen. Dieser Ansatz der Implementierung von dynamischen virtuellen Funktionstabellen greift auf die Tatsache zurück, dass Einfügungen von neuen Klassen mit virtuellen Methoden problemlos möglich sind. Es wird daher für jede Klasse im System jeweils eine *Schattenklasse* erstellt. Diese Schattenklassen bilden die Vererbungshierarchie ihrer entsprechenden Klassen nach. Abbildung 29 veranschaulicht die Hierarchie von Schattenklassen.

Jede Instanz einer Klasse, in die virtuelle Methoden eingewoben werden,

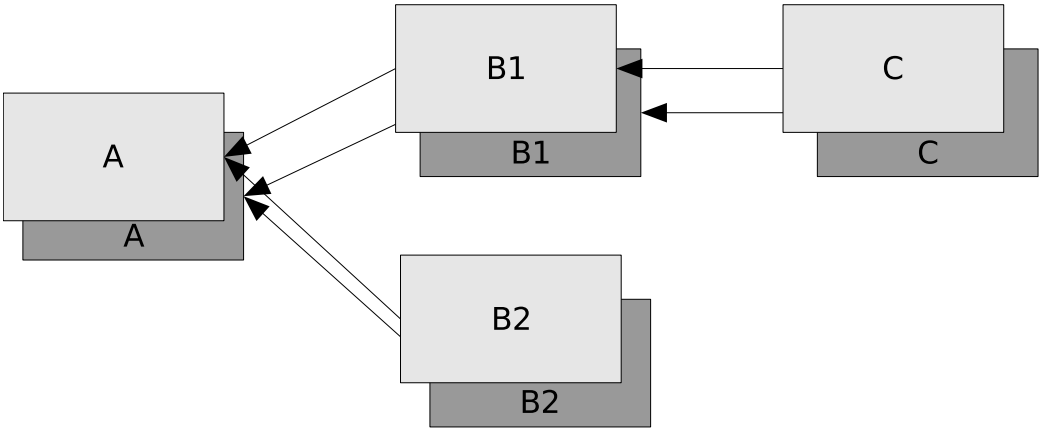


Abbildung 29: Jede Klasse der Klassenhierarchie der Anwendung (Klassen im Vordergrund) wird durch eine generierte *Schattenklasse* ergänzt.

hält eine Referenz auf ein Schattenklassenobjekt ihres entsprechenden Typs. Dazu wird in die allgemeinste (tiefste) Basisklasse ihrer Klassenhierarchie ein sogenannter *Schattenklassenzeiger* eingefügt. Abbildung 30 veranschaulicht eine Klassenhierarchie von zwei Klassen `A` und `B`, in die ein solcher Schattenklassenzeiger eingefügt wurde. Dabei erbt die Klasse `B` den Zeiger `A_shdw *shdw_ptr`, besitzt aber eine Instanz auf ein Schattenobjekt vom Typ `B_shdw`. Dies ist möglich, weil die Hierarchie der Schattenklassen die Hierarchie ihrer entsprechenden Klassen nachbilden (siehe Abbildung 29).

Die Einfügung des Schattenklassenzeigers kann als *dynamische* Einfügung eines Attribts wie in Unterabschnitt 6.1 beschrieben, durchgeführt werden. Es ist dabei nicht relevant, ob die Einfügung mit der Semantik *immediate* oder *lazy* geschieht. Wichtig ist, dass jede Instanz einer Klasse eine Instanz **ihrer** Schattenklasse besitzt.

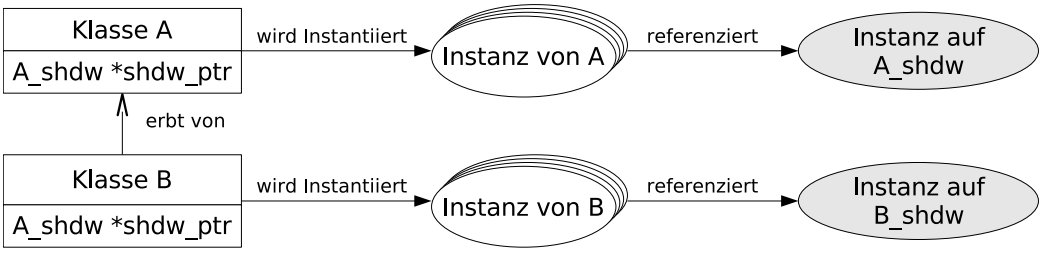


Abbildung 30: Objekte und ihre Schattenobjekte

Die eingefügte Schattenklasse deklariert alle virtuellen Methoden mit dersel-

ben Signatur und gleichem Rückgabetyp wie die Methoden aus ihrer entsprechenden Klasse. Diese *Schattenmethode* ruft lediglich ihre entsprechende Methode *ihrer* entsprechenden Klasse auf. Dieses Prinzip wird in Abbildung 31 veranschaulicht.

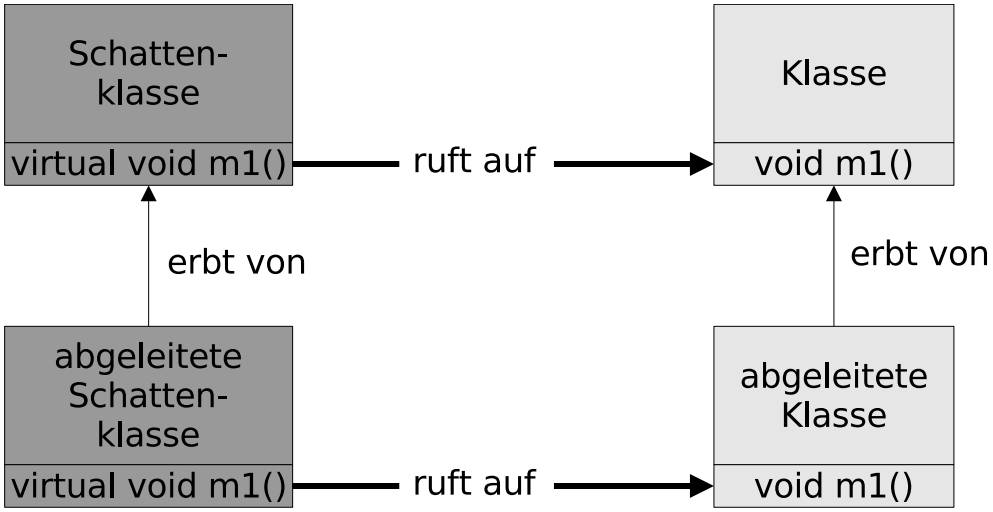


Abbildung 31: Die Schattenklasse enthält für jede *virtualisierte* Methode eine Methode mit gleicher Signatur, die die dazugehörige Methode aus der „Originalklasse“ aufruft.

Nun kann ein dynamischer *dynamic dispatch* wie folgt realisiert werden: Jeder Aufruf der dynamisch eingefügten virtuellen Methode `m1` wird nun durch einen Aufruf auf die dazugehörige Schattenmethode `m1` ersetzt. Es ist sichergestellt, dass in diesem Attribut immer eine Referenz auf ein zu diesem Objekt passenden *Schattenobjekt* existiert (s.o.). Damit wird beim Aufruf der virtuellen Methode des *Schattenobjekts* ein vom C++ Übersetzer erzeugter statischer *dynamic dispatch* durchgeführt, und die *richtige Schattenmethode* gewählt. Diese Schattenmethode ruft dann wie oben beschrieben die Originalmethode auf.

Zur Übersetzung von Aspektmodulen, welche Aspekte beinhalten, die virtuelle Methoden in die Anwendung einfügen, wird ähnlich wie bei der dynamischen Einfügung von Attributen in Unterabschnitt 6.1 der statische Aspektweber `ac++` benutzt. Dabei ist zu beachten, dass die Einfügung von virtuellen Methoden die *v-tables* von bestehenden Klassen ändern oder eine nicht polymorphe Klasse unter Umständen polymorph machen kann. In Abbildung 28 wird die Tatsache verdeutlicht, dass die Wirkung des Aspekts A dazu führt, dass die Klassen M und M' nicht *strukturverträglich* sind, da M polymorph und M' nicht ist.

Damit der (C++) Übersetzer bei Übersetzung des Programmtextes aus dem Aspektmodul dieselbe *Sicht* auf die Strukturen der Anwendung wie bei der Übersetzung der Basisanwendung hat, ist es nötig, die vom Aspektweber `ac++` transformierten Klassen durch einen Postprozessor, wie er in Unterabschnitt 6.1.5 vorgestellt wurde, zu korrigieren. Zur Korrektur der Strukturungleichheit wird der Methodendeklarator `virtual` durch den Postprozessor wieder entfernt. Somit ist sichergestellt, dass nach dem Weben die virtuelle Funktionstabelle identisch bleibt, bzw. nicht angelegt wird, falls die Klasse zuvor nicht polymorph war.

Mit dieser Realisierung des dynamischen *dynamic dispatch* muss der Postprozessor nun alle Aufrufe der eingefügten virtuellen bzw. der *virtualisierten* Methoden anpassen. Für diese Anpassung existieren nun verschiedene Lösungsansätze:

1. Die Aufrufe werden wie bei Einfügungen von Attributen (siehe Unterabschnitt 6.1.3) vom statischem Weber erkannt und markiert; ein Postprozessor ersetzt dann die Aufrufe auf die *Schattenmethoden*.

2. Es wird ein Aspekt generiert, der die Aufrufe mit *around call advice* durch Aufrufe auf die *Schattenmethoden* ersetzt. Dieser Aspekt muss dann von einem *Präprozessor* instantiiert werden.

3. Es wird ein Aspekt generiert, der vor der Ausführung der eingefügten Methode mit *around execution advice* den dynamischen *dynamic dispatch* durchführt. Auch dieser Aspekt muss mit einem *Präprozessor* instantiiert werden.

4. Die dynamisch eingefügten virtuellen Methoden werden nicht unter dem Originalnamen eingefügt. Zusätzlich wird mit dem Originalnamen eine weitere nicht statische Methode eingefügt, die den dynamischen *dynamic dispatch* ausführt.

Die Varianten 1 und 2 sind semantisch äquivalent. In der Tat wird diese Variante vom statischen Weber ähnlich wie in Variante 2 umgesetzt. Ebenfalls semantisch äquivalent sind die Varianten 3 und 4. Auch hier erzeugt der statische Weber vergleichbaren Code. Die Frage läuft also darauf hinaus, ob der dynamische *dynamic dispatch* auf der Seite des Funktionsaufrufers oder auf Seite der aufgerufenen Funktion ausgeführt werden soll.

In C++ ist es möglich, den *dynamic dispatch* beim Methodenaufruf durch Angabe des voll qualifizierten Namens der Funktion zu umgehen. Damit dies

auch für dynamisch gewobene virtuelle Methoden möglich ist, wird der dynamische *dynamic dispatch* auf *Aufruferseite* durchgeführt. Dadurch ist es für den Generator für Adviceinstanzen möglich, die Art des Methodenaufrufs zu analyisieren. Handelt es sich um einen Methodenaufruf mit voll qualifiziertem Namen, so wird an diesem *call join point* kein dynamischer Advice zur Durchführung des dynamischen *dynamic dispatch* gewoben. Damit kann die Frage, ob der dynamische *dynamic dispatch* mit `call` oder `execution` Advice in einem Programm gewoben werden soll, beantwortet werden. Es muss `call` Advice verwendet werden, weil mit `execution` Advice die Analyse der Art des Aufrufs, zumindest mit den momentan verfügbaren Mitteln, nicht möglich ist.

6.5 Virtualisierung von Methoden

Unter *ad hoc Polymorphismus* versteht man üblicherweise das Überladen von Funktionen oder Methoden. Überladene Funktionen sind solche, die denselben Bezeichner haben. Die Sprache `C++` erlaubt dies genau dann, wenn ihre Parameterlisten typmäßig eindeutig unterscheidbar sind. Um zu bestimmen, welche Funktion aus einer Menge von Funktionen mit gleichem Bezeichner aufgerufen werden soll, führt der C++ Übersetzer einen Vorgang durch, der allgemein unter dem Namen *overload resolution* bekannt ist.

Hierbei werden alle Funktionen mit dem entsprechendem Bezeichner in eine Menge aufgenommen. Aus dieser Menge werden zunächst alle Funktionen entfernt, die eine andere Anzahl an Funktionsparameter besitzen als der Funktionsaufruf. Diese resultierende Menge wird im Folgenden als *Kandidatenmenge* bezeichnet. Aus dieser *Kandidatenmenge* wird nun versucht mittels Typeumwandlungsoperatoren *kompatible* Methodenaufrufe zu bestimmen. Dabei entsteht ein *ranking*, also eine Bewertung, die zum gegebenen Funktionsaufruf am Besten passt. Die Methode mit dem besten *Ranking* geht als Sieger hervor und wird ausgewählt. Ist kein Sieger bestimmbar gibt der C++ Übersetzer eine Fehlermeldung aus.

Dynamische Einfügungen von Methoden sind problematisch, wenn ihre Einfügung dazu führt, dass die eingefügte Methode bei einem oder mehreren Methodenaufrufen in die *Kandidatenmenge* aufgenommen wird und bei der *overload resolution* eine besseres *ranking* erhält, als alle anderen Kandidaten aus dieser Menge. Ein analoges Problem besteht bei der Einfügung von `enum`s oder `typedef`s in abgeleiteten Klassen, wenn dadurch Deklarationen aus Basisklassen überdeckt werden.

Es stellt sich also heraus, dass das Weben von Aspekten unter bestimm-

ten Umständen die Semantik des Programms ändert. Im Kontext von dynamischem Weben von Aspekten zur Laufzeit bedeutet dies, dass ein laufendes Program durch das Laden eines Aspektmoduls möglicherweise *invalidiert* wird. Dies ist genau dann der Fall, wenn ein Aspekt auf ein Programm so wirkt, dass Teile des Programms ihre Semantik ändern, die nicht vom dynamischem Weber verändert werden (können). Ein ähnliches Problem tritt auf, wenn Methoden einer abgeleiteten Klasse, die in der Anwendung nicht virtuell waren, durch Einfügungen in einer virtuellen Methode in eine Basisklasse *virtualisiert* werden. In der Sprache `C++` ist die Eigenschaft, dass eine Klasse virtuell ist, über die Klassenhierarchie hinweg transitiv. So ist im Codebeispiel aus Abbildung 25 in Zeile 6 die Methode `v` virtuell, obwohl das Schlüsselwort `virtual` *nicht* im Quelltext steht. Vielmehr ist die Methode dennoch virtuell, weil in der Basisklasse die Methode `v` in der Zeile 2 als virtuell deklariert wurde. In diesem Fall erzeugt der C++ Übersetzer je nach dem, ob statisch oder dynamisch gewoben, Code für den *dynamic dispatch*.

In Unterabschnitt 6.4 wurde festgestellt, dass die *Virtualisierung* der Methode rückgängig zu machen ist. Mit den in dieser Arbeit vorgestellten Techniken klappt dies jedoch nur für die Teile der Anwendung, die geeignet instrumentiert wurden. Nur für diese Stellen können vom Übersetzer für Aspektmodule Hilfsaspekte erzeugt werden, die die Semantik des Programms *korrigieren* können.

6.6 Basisklassen

Die Sprache AspectC++ erlaubt es, Klassenfragmente zu definieren, die von einer Basisklasse erben. Die Einfügung von Basisklassen, die Attribute besitzen, führt zwangsläufig dazu, dass sich die Struktur der Klasse ändert. Die bei der mit dem statischem Weber erzeugte transformierte Klasse ist also *strukturunverträglich*. Dies hat zur Folge, dass der C++ Übersetzer bei der Übersetzung der Basisanwendung eine andere Sicht auf Klassen hat wie bei der Übersetzung der Programmteile im Aspektmodul, die die Klasse benutzen, in die eine Basisklasse eingefügt wird.

Diese Art von *dynamischen* Einfügungen von Attributen kann analog zu Unterabschnitt 6.1) umgesetzt werden. Dabei werden die vom statischem Weber transformierten Klassen so korrigiert, dass die Vererbungsbeziehung rückgängig gemacht wird und die Attribute der neuen Basisklasse *dynamisch* in die abgeleitete Klasse eingefügt wird. Ebenfalls problemlos korrigierbar sind eingefügte nicht virtuelle Methoden. Da diese nicht virtuell sind, sind diese nur über ihren vollen Namen aufrufbar, falls sie von abgeleiteten Klassen

überschrieben werden. Falls nicht, so ist sichergestellt, dass Aufrufe dieser Methoden nur aus dem aktuellem Erweiterungsmodul auftreten können. Der Programmcode aus der Basisanwendung hat keine Kenntnis von dieser Methoden und kann sie daher auch nicht aufrufen. Dadurch können sie auch nur vom aktuellen oder späteren Erweiterungsmodulen genutzt werden.

Problematischer sind Einfügungen von *virtuellen* Methoden durch Basisklasseneinfügungen. Diese verändern grundsätzlich die virtuellen Funktionstabellen von abgeleiteten Klassen. Analog zum Unterabschnitt 6.4 entsteht das Problem, dass der C++-Übersetzer zur Übersetzungszeit feststellen muss, ob ein Funktionsaufruf *virtuell* ist oder nicht, um entscheiden zu können, ob ein *dynamic dispatch* nötig ist. Die Lösung hierfür geschieht analog wie in Unterabschnitt 6.4.

6.7 Konflikte mit dem *Single Language* Ansatz

Im Verlauf dieses Kapitels werden die Einfügung von verschiedenen Arten von Sprachelementen vorgestellt. Es stellt sich dabei heraus, dass je nach Art des Sprachelements, unterschiedliche Lösungsansätze benötigt werden. Die Lösungsansätze bauen dabei teilweise aufeinander auf. Man stellt fest, dass es insgesamt drei (nicht überlappungsfreie) Klassen von Einfügungen gibt:

1. Einfache Einfügungen
2. Einfügungen mit Seiteneffekten auf der Codeebene
3. Einfügungen mit Seiteneffekten auf der Sprachebene

Zu den *einfachen* Einfügungen gehören die Sprachelemente:

- Aufzählungen (mittels `enum`)
- Typealiase (mittels `typedef`)
- nicht virtuelle Funktionen
- statische Klassenvariablen
- statische Klassenfunktionen

Sie haben gemeinsam, dass beim Weben des Aspekts eine *strukturverträgliche* Transformation der betroffenen Klassen geschieht. Deshalb ist es möglich, diese Klasse von Sprachelementen mit dem bereits existierenden Aspektweber `ac++` auch für zur Laufzeit geladene Aspektmodule zu weben.

Zu den Sprachelementen, deren Einfügung Seiteneffekte auf *Codeebene* verursachen gehören:

- Attribute
- virtuelle Funktionen
- Basisklassen

In der in dieser Arbeit entwickelten Infrastruktur werden Aspekte, die Sprachelemete von dieser Klasse einfügen, ebenfalls mit dem statischem Weber in die entsprechenden Klassen eingewoben. Die beim Weben verwendete Transformation hat zur Folge, dass die Klassen, in die die Einfügung vorgenommen wurden, *strukturunverträglich* verändert werden. Damit der dynamische Weber zur Laufzeit das Programm nicht korrigieren muss, hat die Infrastruktur in diesem Fall dafür Sorge zu tragen, dass der C++ Übersetzer bei der Übersetzung von Aspektmodulen die *selbe Sicht* auf die betroffenen Typen benutzt. Dazu wird der gewobene Programmtext nach der Transformation durch einen Postprozessor entsprechend korrigiert.

Es hat sich im Verlauf dieses Kapitels jedoch herausgestellt, dass alle Arten von Sprachelemente, die in Klassen eingefügt werden, Seiteneffekte auf Sprachebene verursachen können. Dies ist dann der Fall, wenn die Deklaration einer Einfügung in einer abgeleiteten Klasse die Deklaration eines Sprachelements mit dem selben Namen *überdeckt*. In diesem Fall ändert sich die Semantik der Stellen im Programmtext, die diese Deklarationen verwenden. Wenn das Aspektmodul zur Laufzeit geladen wird und die durch das dynamische Weben von Aspekten die *Semantikänderung* des laufenden Programms nicht durchgeführt werden kann, darf der Aspekt nicht geladen werden, weil dadurch das Programm dann semantisch falsch würde.

Der *Single Language* Ansatz, der in Unterabschnitt 3.5 vorgestellt wurde, geht von der Annahme aus, dass alle Aspekte unabhängig von ihrer Art und Weise wie sie gewoben werden, die gleichen sprachlichen Auswirkungen beibehalten. Ziel ist es einen beliebigen Aspekt, der durch die Sprache AspectC++ beschrieben wird, sowohl *dynamisch* als auch *statisch* weben zu können. Dies ist für Einfügungen mit Seiteneffekten auf Sprachebene jedoch mit den in dieser Arbeit entwickelten Techniken nicht möglich. Aus diesem Grund ist es

also nötig, dass sowohl der statische als auch der dynamische Aspektweber diese Art von Einfügungen erkennen und eine entsprechende Fehlermeldung liefert.

Theoretisch wäre es möglich, mit viel Aufwand bei der Instrumentierung dafür zu sorgen, dass diese Art von Einfügungen zur Laufzeit möglich werden. Das würde bedeuten, Funktionalität wie die oben beschriebene *overload resolution* im Laufzeitsystem nachzubilden. Dann könnte festgestellt werden, welche Methodenaufrufe per *dynamic Advice* mit einem *dynamic dispatch* ersetzt werden müssten. Dies setzt eine geeignete, potentiell sehr aufwändige Instrumentierung voraus. Der Aspektlader muss darüber hinaus beim Laden des Aspekts prüfen, ob die Basisanwendung für die Einfügung *geeignet* instrumentiert ist und gegebenenfalls das Laden des Aspekts mit einer Fehlermeldung abbrechen. Man nähert sich also immer mehr einer interpretierten Ausführung von C++-Programmen an. Praktisch macht diese dieser Lösungsansatz viele wünschenswerte Merkmale der Sprache C++ zunichte.

Diese Anomalie bedeutet jedoch nicht, dass der *Single Language* Ansatz vollständig nicht umsetzbar ist, sondern, dass er nicht vollständig durchführbar ist. Daher kann die Implementierung eines *dynamischen* Webers für die Sprache AspectC++ nicht den vollen Sprachumfang umsetzen, sondern nur eine Teilmenge. Problematische Sprachmittel sollten daher vom sowohl vom statischem als auch vom dynamischem Weber als Fehler erkannt werden.

6.8 Zusammenfassung

Dieses Kapitel hat die verschiedenen Möglichkeiten für Einfügungen der Sprache AspectC++ vorgestellt. Dabei wurden die einzelnen Varianten klassifiziert und für jede Klasse von Einfügungsarten Lösungsansätze und Implementierungsansätze vorgestellt.

Es stellt sich heraus, dass Aspekte mit dynamischen Einfügungen, je nach sematischer Auswirkung zunehmend mehr Anforderungen an das Laufzeitsystem und die Infrastruktur stellen. Es muss daher kritisch hinterfragt werden ob es wirklich sinnvoll ist, alle Sprachmittel aus AspectC++ auch für *dynamisches* Weben zu erlauben. Wenn keine Änderung am für C++ Anwendungen üblichen Ausführungsmodell gewünscht sind, ist es sinnvoll, sich auf Sprachmerkmale zu beschränken, die nicht dazu führen, dass die Basisanwendung beim Weben zur Laufzeit semantisch invalidiert wird.

7 Sonstige AspectC++ Sprachmerkmale

In dieser Arbeit wurde bisher die Umsetzung der Sprachmerkmale *dynamischer Code Advice*, *generic advice*, und *Introductions* vorgestellt. Damit sind bereits viele sinnvolle Aspektimplementierungen möglich. Um eine vollständige Implementierung des *Single Language* Ansatzes zu realisieren, ist es nötig, die restlichen AspectC++ Sprachmerkmalen zu betrachten.

7.1 Pointcut-Funktionen

In der Sprache AspectC++ werden *pointcut*- Funktionen benutzt, die *pointcuts* in andere *pointcuts* zu überführen. Dabei unterscheidet man zwischen *Code*- und *Name*-Pointcuts. Die am häufigsten verwendeten *pointcut*- Funktionen sind:

- `call`
- `execution`
- `construction` und
- `destruction`

Die Umsetzung dieser *pointcut*- Funktionen wurde bereits in Kapitel 5 beschrieben. Sie stellen fundamentale *pointcut*- Funktionen dar, die *Name*-Pointcuts in *Code*-Pointcuts überführen. Darüber hinaus existieren noch weitere *pointcut*- Funktionen, deren Umsetzung in einer dynamischer AOP Implementierung in diesem Abschnitt diskutiert werden soll.

7.2 Statisch evaluierbare *pointcut*- Funktionen

Die *pointcut*- Funktion `within()` erzeugt die Menge von *join points*, die sich syntaktisch *innerhalb* eines benannten *pointcut* befindet. Die Eigenschaft, dass diese *pointcut*- Funktion nur von der syntaktischen Struktur des Programms abhängt, ermöglicht die Umsetzung im Falle *dynamischen* Webens zur Übersetzungszeit des dynamischen Aspektmoduls. Zu diesem Zeitpunkt sind sowohl die Quelltexte der Anwendung, als auch aller bisherigen Erweiterungsmodule bekannt. Damit können im Generator für Adviceinstanzen alle *join points* evaluiert werden und es braucht daher zur Laufzeit keine weitere Evaluierung des *pointcuts* stattfinden.

Die *pointcut*- Funktionen `base()` und `derived()` verhalten sich ähnlich wie `within()`. Auch bei diesen Funktion steht zur Übersetzungszeit fest, welches Ergebnis die Auswertung haben wird. Damit kann man bei diesen drei *pointcut*- Funktionen von *statischen pointcut Funktionen* sprechen, da alle drei statisch zur Übersetzungszeit feststehen. Die Umsetzung dieser Klasse von *pointcut*- Funtkionen erfolgt damit vollständig im Generator für Adviceinstanzen (siehe Unterabschnitte 5.1 und 5.2).

7.3 Dynamisch evaluierbare *pointcut*- Funktionen

7.3.1 Dynamische Typprüfung

Zu den dynamischen *pointcut*- Funktionen gehören `that()` und `target()`. Sie überprüfen in der Regel *dynamisch* (also zur Laufzeit) den Objekttyp, in dem ein Advice ausgeführt wird.

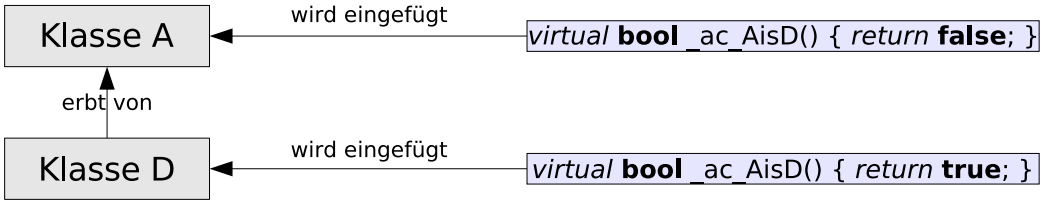


Abbildung 32: In eine Klassenhierarchie werden *virtuelle* Methoden eingefügt, um zur *Laufzeit* zu prüfen, ob der Advice von einem Objekt der abgeleiteten oder der Basisklasse ausgeführt wird.

Der statische Weber `ac++` nutzt zur Umsetzung dieser *pointcut*- Funktionen den *dynamic dispatch* von C++ (siehe Unterabschnitt 6.4). Dazu wird jeweils eine virtuelle Methode in die Basisklasse sowie in die zu prüfende abgeleitete Klasse eingefügt. Die Methode, die in die Basisklasse eingefügt wird, gibt dabei `false` zurück, wohingegen die Methode aus der abgeleiteten Klasse `true` zurück gibt. Für den Aspekt aus Abbildung 33 veranschaulicht Abbildung 32 die eingefügten Methoden.

Bei der Ausführung des Advice wird zur Prüfung der Instanz die eingefügte Methode `_ac_AisD()` aufgerufen. Wenn das Objekt, in dessen Kontext der Advice ausgeführt wird, eine Instanz der der Basisklasse A ist, ergibt die Prüfung den Wahrheitswert `false`. Ist das Objekt hingegen eine Instanz der abgeleiteten Klasse D, ergibt die Prüfung den Wert `true`. Bei der Ausführung der eingefügten virtuellen Methoden sorgt der *dynamic dispatch* dafür, dass

```
1 aspect that_aspect {
2   advice execution("% ...::doStuff()") && that("D") : after() {
3     printf("advice called");
4   }
5 };
```

Abbildung 33: Aspekt mit der *pointcut*- Funktion `that()`.

```
1 int callera() {
2   ::AC::CFlow<testcflow,0> trigger0;
3   ::new (&result) int (__exec_old_callera());
4   return (int &)result;
5 }
```

Abbildung 34: Markierung an einem *join point* für `cflow()`.

in den virtuellen Methodentabellen geprüft wird, welche Methode ausgeführt werden soll. Anhand des Rückgabewertes der ausgewählten Methode kann nun geprüft werden, ob das gegebene Objekt, auf den der Advice wirken soll, vom Typ `D` ist oder nicht.

Die Umsetzung dieser *pointcut*- Funktion in dynamisch gewobenen Aspekten geschieht analog zum statischem Fall. Dabei werden die virtuellen Methoden *dynamisch* in die betroffenen Klassen eingefügt (siehe Unterabschnitt 6.4). Bei der Generierung der *Wrapperfunktion*, die den Advice instantiiert, wird zusätlich ein Test durchgeführt, der mittels der dynamisch eingefügten virtuellen Methoden prüft, von welchem Typ das Objekt ist, in dessen Kontext der Advice ausgeführt wird. Je nach Ausgang dieses Tests wird die Ausführung des *Advice Code* durchgeführt oder verhindert.

7.3.2 Dynamische Prüfung des Kontrollflusses

Mit der *pointcut*- Funktion `cflow()` kann der Programmierer die dynamische Ausführung in einer Anwendung so einfangen, dass der *Advice Code* nur bei Durchschreitung von bestimmten Stellen im Programm aktiviert wird. Dazu wird an der zu überprüfenden Stelle eine Markierung eingefügt, die das Durchschreiten anzeigt. Der *Advicewrapper* von Advice mit `cflow()` prüft diese Markierung und ruft je nachdem, ob sie gesetzt ist oder nicht, den *Advice Code* auf.

`ac++` instrumentiert den Programmtext also an zwei Stellen. So wird an dem *join point*, der durch die *cflow pointcut*- Funktion beschrieben wird, eine Markierung gesetzt. Dazu erzeugt `ac++` Programmtext, wie er in Abbildung 34

```
1 int callee() {
2   AC::ResultBuffer< int > result;
3   if(::AC::CFlow<testcflow,0>::active ()) {
4     AC::invoke_testcflow_testcflow_a0_before ();
5   }
6   ::new (&result) int (__exec_old_callee());
7   return (int &)result;
8 }
```

Abbildung 35: *Advicewrapper*, der eine `cflow()` Bedingung überprüft.

zu sehen ist. Das `cflow-` Objekt in Zeile 2 besitzt dabei einen Zähler, der im Konstruktor inkrementiert und im Destruktor dekrementiert wird. Dadurch kann auch die ein verschachtelter Aufruf korrekt erkannt werden. Diese Markierung kann in einem Aspektmodul mit dynamisch gewobenem *around*-Advice realisiert werden, welcher vom Generator für Adviceinstanzen zusätzlich erzeugt wird. Die andere Stelle ist in den *Advicewrappern*, in denen der Kontrollfluss geprüft wird. Hier wird getestet, ob die Markierung (`trigger0` im Codebeispiel) gesetzt ist. Im statischen Fall erzeugt der statische Aspektweber `ac++` ein Programmfragment, das wie in Abbildung 35 aussieht. Im dynamischem Fall wird die Überprüfung wie im Unterabschnitt 7.3.1 vom Generator für Adviceinstanzen zusätzlich in den entsprechenden *Advicewrappern* eingefügt.

Damit ist die Umsetzung der der *pointcut*- Funktion `cflow()` zur Laufzeit prinzipiell möglich. Es besteht jedoch noch ein semantisches Problem zum Ladezeitpunkt des Aspektmoduls: Es kann nicht erkannt werden, ob sich der Kontrollfluss des laufenden Programms innerhalb des mit `cflow()` zu überprüfenden *pointcuts* befindet, da die Markierung ja erst nach durchschreiten des *join points* mit der Markierung eingefügt wurde.

Zur Lösung dieses Problems gibt es prinzipiell zwei Ansätze. Der eine Ansatz ist es, die Markierung bereits durch den statischen Weber einzufügen. Damit wird jederzeit und an allen *join points* im Programm überprüft, wo sich der Kontrollfluss befindet. Dies hätte jedoch erhebliche Laufzeiteinbußen zur Folge. Der andere Ansatz besteht darin, das Weben des dynamischen Aspekts solange zu verzögern, bis der Kontrollfluss den zu überprüfenden *pointcut* verlassen hat. Dies kann mit einem dynamischem *Hilfsadvice* realisiert werden, der das Laufzeitsystem den Zeitpunkt des Austritts aus dem zu überprüfendem *pointcut* signalisiert. Dazu bestimmt der Übersetzer für dynamische Aspektmodule die *join points*, an denen dieser *Hilfsadvice* den Austritt aus dem zu prüfendem *pointcut* dem Laufzeitsystem signalisieren soll.

Der Aspektlader muss beim Laden eines Aspektmoduls also zusätzlich erkennen, dass das Aspektmodul die *pointcut*- Funktion `cflow()` benutzt und ob sich der Kontrollfluss des laufenden Programms innerhalb des zu prüfenden *pointcuts* befindet. In diesem Fall wird zunächst nur der Hilfsaspekt installiert und das Weben der Aspekte aus dem Aspektmodul bis zum Zeitpunkt der Signalisierung verzögert. Um zu vermeiden, dass der Aspektlader das Aspektmodul nie laden kann, bietet es sich an, dass der Ladevorgang entweder nach einer gewissen Zeitspanne oder vom Benutzer abgebrochen werden kann.

Der Algorithmus zur Prüfung, ob sich ein laufendes Programm innerhalb eines bestimmten *pointcuts* befindet, kann dem Lader im Aspektmodul mitgeliefert werden. Er sollte dabei unter zur Hilfenahme des Kontrollflusses des Programms, des aktuellen Instruktionszeigers und der Symboltabelle der Funktionen im Programm die Entscheidung für den Aspektlader treffen können. Eine effiziente Implementierung zur Generierung eines solchen Algorithmus konnte jedoch aus Zeitgründen nicht im Rahmen dieser Arbeit erarbeitet werden.

7.4 Adviceordnung

Ein weiteres ungelöstes Problem stellt die Ordnung der Ausführungsreihenfolge von Advicefunktionen dar, die am selben *join point* gewoben werden. AspectC++ erlaubt es Advice zu definieren, welcher die Ordnung von anderen Deklarationen von Advice festlegt. Im statischem Fall wird die Reihenfolge vom statischen Aspektweber `ac++` aufgelöst und die *Advicewrapper* so erzeugt, dass die vorgegebene Reihenfolge eingehalten wird. Dies ist problemlos möglich, weil alle Aspekte zur Übersetzungszeit der Anwendung bekannt sind.

Wenn man im dynamischen Fall nur zur Laufzeit gewobene Aspekte betrachtet, ist die Reihenfolge bei Verwendung von *einfachen* Advice Containern (siehe Unterabschnitt 5.3) kein Problem, da hier pro *join point* nur ein *Advicewrapper* möglich ist. Bei Verwendung einer listenbasierten Implementierung kann die Reihenfolge, in der die *Advicewrapper* ausgeführt werden sollen, durch sortiertes Einfügen sichergestellt werden.

Problematisch ist der Fall, wenn in einem System die Reihenfolge zwischen Aspekten mit dynamisch und statisch gewobenem Advice mittels zur Laufzeit gewobener Adviceordnung gelöst werden soll. Eine Lösung dieses Problems ist nur durch Änderung der erzeugten Codemuster des statischen Webers `ac++` möglich. Dieses Problem wurde im Rahmen dieser Arbeit nicht weiter

```
1  #include <iostream>
2
3  aspect Context {
4    advice execution("% print(...)") && args (s) : after(char *s) {
5      std::cerr << s << std::endl;
6    }
7  };
8
9  int print (char *s) {
10   std::cout << s << std::endl;
11 };
12
13 int main () {
14   print("contextvariablestest");
15 }
```

Abbildung 36: Bindung eines Arguments an eine *pointcut*- Variable.

untersucht. Es kann jedoch davon ausgegangen werden, dass diese Anforderungen in realen Anwendungen weniger häufig auftreten.

7.5 Kontextvariablen

Die Sprache AspectC++ sieht vor, dass die *pointcut*- Funktionen `that()`, `target()`, `result()` und `args()` dazu verwendet werden können, um diese Kontextinformation in Form einer Kontextvariablen zu binden. Die Deklaration der Variablen erfolgt dabei in der Argumentliste von `after()`, `before` oder `around()`. Die Verwendung von diesem Sprachmerkmal wird exemplarisch an der *pointcut*- Funktion `args()` in Abbildung 36 demonstriert. Der Aspekt sorgt dafür, dass die Zeichenkette, welche der Funktion `print()` als Argument `char *s` übergeben wird, zusätzlich auf der Standardfehlerausgabe ausgegeben wird.

Die Umsetzung dieses Sprachmerkmals geschieht dadurch, dass die Instantiierungsfunktion, welche die als Funktionstemplate übersetzte Advicefunktion als (zusätzliches) Argument übergibt. Da die Instantiierungsfunktion, die vom Übersetzer für dynamische Aspektmodule (siehe auch 5.1 auf Seite 31) erzeugt wird, die Aufgabe hat, diese Kontextinformation zur verfügung zu stellen, kann der Funktionsaufruf problemlos um die Kontextvariable erweitert werden.

7.6 Zusammenfassung

Diese Arbeit tritt den Versuch an, auf der Basis der Sprache AspectC++, Aspekte sowohl statisch als auch zur Laufzeit weben zu können. Für eine vollständige Umsetzung des *Singe Language* Ansatzes ist es nötig, den vollen Sprachumfang der Sprache AspectC++ im dynamischen Weber zu implementieren. Die letzten beiden Kapitel haben gezeigt, wie dies für die Sprachmerkmale *Code Advice* und *Introductions* gelöst werden kann. Dieses Kapitel zeigt Lösungsansätze zur möglichen Implementierung der restlichen Sprachmerkmale auf.

AspektC++ erlaubt es, mittels *pointcut*- Funktionen *join point* Mengen sowohl zur Übersetzungszeit als auch zur Laufzeit der Anwendung zu manipulieren. Dieses Kapitel zeigt, wie dynamischer Advice (Kapitel 5) geschrieben werden kann, der zur Evaluierung seiner *join points pointcut*- Funktionen benutzt. Dabei stellt sich heraus, dass dies für unterschiedliche *pointcut* Funktionen unterschiedlich viel Aufwand nötig ist. Es wurden hierbei für die Fälle dynamische *pointcut*- Funktionen sowie für die *pointcut*- Funktion `cflow` Lösungsmöglichkeiten präsentiert.

8 Eine Familie von Aspektwebern

Die vorherigen Kapitel haben aufgezeigt, wie die einzelnen Sprachmerkmale der aspektorientierten Programmierung, die für statisches Weben vom Aspektweber `ac++` implementiert werden, zur Laufzeit umgesetzt wird. Dabei wurde bereits klar, dass die Umsetzungen der verschiedenen Sprachmerkmale unterschiedlichen hohen Bedarf an Betriebsmitteln haben.

Die Umsetzungen vieler Sprachmerkmale verursachen einen deutlichen Anstieg an Bedarf von Betriebsmittelressourcen. Um diesen Bedarf zu reduzieren bietet es sich an, die Infrastruktur generisch und konfigurierbar zu gestalten, um für möglichst viele Anwendungsszenarien den Bedarf an Betriebsmittelressourcen dadurch zu minimieren, dass nicht benötigte Sprachmerkmale nicht umgesetzt werden.

Zur Unterstützung der Konfigurierbarkeit wird in dieser Arbeit das Prinzip der familienbasierten Softwareentwicklung [BPKS04, Cza04] verwendet. In diesem Zusammenhang stellt die entwickelte Infrastruktur eine Vielzahl von dynamischen Webervarianten bereit, die an die konkrete Anwendung angepasst wird. Der Softwarearchitekt legt also bereits beim Entwurf fest, mit welchen aspektorientierten Sprachmerkmalen die Anwendung erweitert werden kann, um Betriebsmittelbedarf zu minimieren. Dieses Kapitel stellt die Domäne dynamischer Aspektweber vor. Dabei werden die einzelnen Anforderungen vor allem im Hinblick auf ihren Betriebsmittelbedarf analysiert.

8.1 Generierungsprozess

Im wesentlichen gibt es in der in dieser Arbeit entwickelten Infrastruktur zwei Komponenten, die konfiguriert werden: Die Instrumentierung der Anwendung mit Hilfe eines statischen Aspekts sowie das Laufzeitsystem. Diese Komponenten werden mit Hilfe eines Werkzeugs unter Beachtung einer gegebenen Konfiguration generiert. Die Abbildung 37 zeigt den Generierungsprozess dieser zwei Komponenten.

Als *Konfigurationswerkzeug* wird im Rahmen dieser Arbeit `pure::variants` [Beu03] eingesetzt, das auch in anderen Projekten (z.B. [Bla05], ...) erfolgreich eingesetzt wurde. Es handelt sich dabei um eine kommerzielle Erweiterung der Entwicklungsumgebung „Eclipse“,[5] die von IBM unter einer freien Softwarelizenz veröffentlicht wurde. Bei `pure::variants` besteht ein Projekt aus mehreren Teilen (siehe auch Abbildung 38):

[5] http://www.eclipse.org

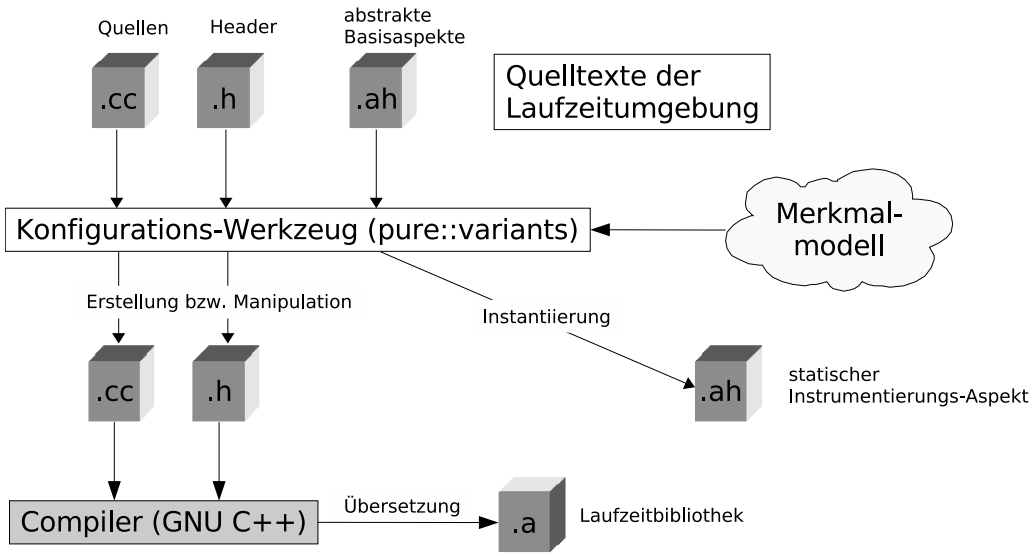


Abbildung 37: Die Bibliothek, sowie der statische Instrumentierungsaspekt sind Zwischenartefakte, die unter Zuhilfenahme eines *Konfigurationwerkzeuges* aus den Quelltexten der Laufzeitumgebung erstellt werden.

Merkmalmodell: Dient der Verwaltung und Auswahl der Merkmale im System auf grafischer Basis. Es können dabei auch Beziehungen wie Restriktionen, Konflikte und Abhängigkeiten ausgedrückt werden

Komponentenmodell: Verbindet konkrete Softwareelemente wie Makro-Definitionen, Klassen, Aspekte, Typaliase mit bestimmten Merkmalen.

Konfigurationsraum: Entsteht durch die Verbindung von *Merkmalmodell* und *Komponentenmodell* durch Konfigurierung, also Bestimmung einer konkreten Variante.

Bei der *Auswertung* des Konfigurationsraumes erstellt, kopiert oder verändert `pure::variants` die Quelltextdateien aus dem Projekt und stellt diese für die nachfolgende Übersetzung geeignet zusammen. Dadurch entstehen Komponenten für eine Systemvariante.

Als Resultat des Generierungsprozesses entstehen im Fall des dynamischen Aspektwebers zwei Zwischenartefakte: Die Laufzeitbibliothek und der statische Instrumentierungsaspekt. In der Laufzeitbibliothek werden Datenstrukturen und Methoden mitgeliefert, die zur Bereitstellung von Kontext sowie zur Ausführung von *dynamischem* Advice (siehe Kapitel 5) und Zugriff auf

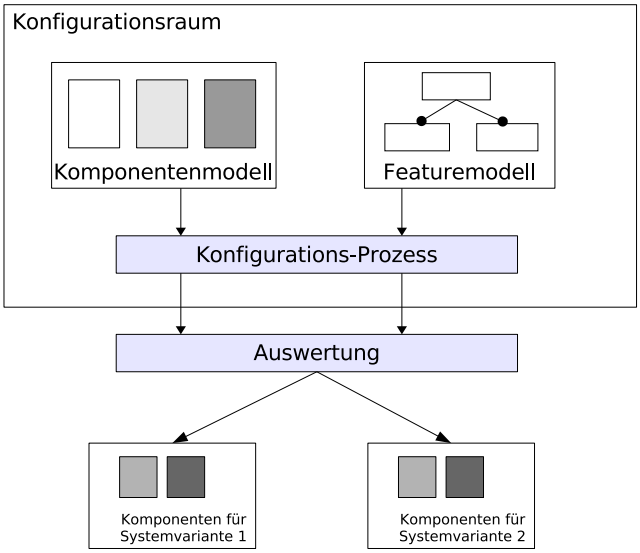


Abbildung 38: In `pure::variants` werden Merkmalmodell und Komponentenmodell getrennt voneinander festgelegt. Zusammen bilden sie den Konfigurationsraum. Durch die Auswertung der Konfiguration stellt das Werkzeug die Komponenten für die gewählte Variante zusammen.

dynamische Einfügungen (siehe Kapitel 6) nötig sind. Die eigentlichen Instanzen der dynamischen Aspekte werden nicht zur Anwendung gebunden, sondern dynamisch zur Laufzeit geladen. Der statische Instrumentierungsaspekt *erbt* von einem allgemeinem Aspekt und definiert rein virtuelle *pointcuts* (siehe Unterabschnitt 2.4). Daher wir der konkrete Aspekt zur Instrumentierung der Anwendung vom Konfigurationswerkzeug `pure::variants` generiert.

8.2 Kosten für Kontext

In [Tar05] konnte gezeigt werden, dass der Löwenanteil der Betriebsmittel für die Bereitstellung von Kontext, genauer: für dynamischen Kontext, beansprucht wird. Statischer Kontext bedeutet statische Typinformation über den Ausführungskontext von *Advice*. Dieser wird in C++ Programmen vom Übersetzer so ausgewertet, dass diese in den erzeugten Programme keine Rolle mehr spielen und damit insbesondere keine Betriebsmittel belegen. Zusätzliche Betriebsmittel entstehen in den von `ac++` transformierten Programmen durch Hilfsstrukturen, die Zugriff auf *dynamischen* Kontext (siehe auch Unterabschnitt 5.1) ermöglichen.

Weil im statischem Fall die Codeerzeugung zur Instantiierung von Advice

```
aspect Tracer {
  advice TracePoints() : before() {
    printf("running %s\n", JoinPoint::signature());
  }
};
```

Abbildung 39: Einfacher Aspekt zur Programmablaufverfolgung

mit dem Zeitpunkt des Webens der Aspekte zusammenfällt, ist es ac++ möglich die erzeugten Hilfsstrukturen zum Webezeitpunkt zu optimieren. Dabei analysiert ac++ den *Advice Code* um wertet dabei aus, auf welche Arten von dynamischen Kontext konkret zugegriffen wird. Mittels dieser Information kann dann entschieden werden, welche Hilfsstrukturen erzeugt werden müssen. Dadurch hängt der Bedarf an Betriebsmitteln vom konkreten *Advice* ab.

Im dynamischem Fall unterscheidet sich der Zeitpunkt der Codeerzeugung zur Instantiierung von dynamischem Advice mit dem Zeitpunkt, an dem die Anwendung instrumentiert wird. Damit ist die gerade beschriebene Optimierungsmöglichkeit nicht mehr möglich, weil keine Kenntnis über den tatsächlich verwendeten Kontext existiert. Es müssen also *auf Verdacht* alle Arten von Kontext vorbereitet werden, was den Betriebsmittelbedarf des instrumentierten Programms deutlich steigert. Dabei ist zu bedenken, dass die Betriebsmittel an allen *join point shadows* bereitgestellt werden müssen; selbst dann, wenn kein Advice aktiv ist.

Das Merkmalmodell für die Bereitstellung von Kontext kann mit Hilfe des Merkmaldiagramms aus Abbildung 40 dargestellt werden. Es besteht zwar die Möglichkeit auf dynamischem Kontext komplett zu verzichten; ein Verzicht auf statischen Kontext ist nicht vorgesehen, da der statische Kontext aus dem *project repository* erstellt werden kann. Bei der Übersetzung der *dynamischen* Aspektmodule wird kein Programmcode benötigt, der zur Laufzeit Betriebsmittel für den Zugriff belegt.

Einfache Aspekte zur Programmablaufverfolgung wie in Abbildung 39 kommen unter Umständen mit rein statischem Kontext aus. In diesem Fall werden zur Laufzeit keine Hilfsstrukturen zum Zugriff auf dynamischem Kontext benötigt, da lediglich auf *statischen* Kontext zugegriffen wird. Die Beschränkung auf statischen Kontext hat zur Konsequenz, dass *around*-Advice nicht mehr umsetzbar ist. Ohne dynamischen Kontext ist es für *around*-Advice nicht möglich, den Programmfluss fortzusetzen. Es bleibt damit nur noch die Möglichkeit, die Ausführung von *pointcuts* abzuschalten, was nur in Einzelfällen Sinn macht.

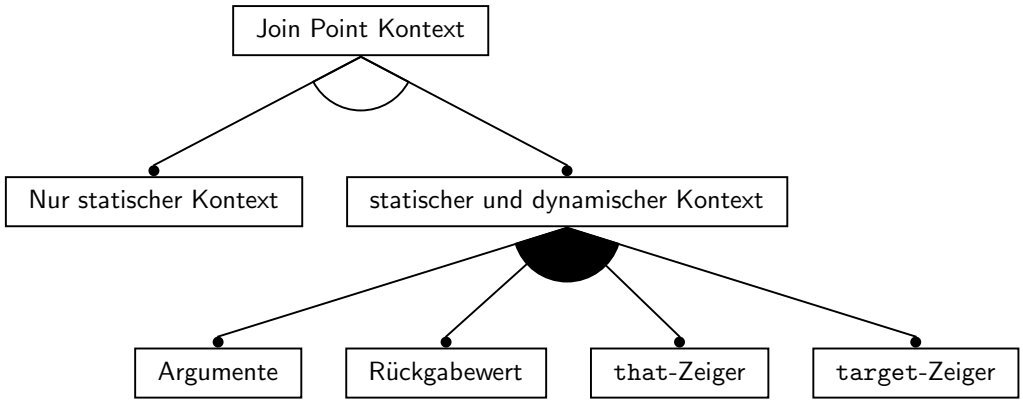


Abbildung 40: Merkmaldiagramm für bereitstellbaren Kontext

Kontextart	Kosten in Maschinenworte
statische Typinformation	0
Argumente	1 pro Argument
Rückgabewert	1
`that`	1
`target`	1

Tabelle 2: Kosten für dynamischen Kontext

Tabelle 2 zeigt, wie viel Bedarf an Stapelspeicher für die Bereitstellung der einzelnen Merkmale entsteht. Der dynamische Kontext ist hierbei eine Datenstruktur, die nur aus Zeigern besteht. Diese Zeiger verweisen dabei auf Puffer, die vom statischen Weber `ac++` bereitgestellt werden. Daher sind die Zahlen als zusätzlicher Bedarf an Betriebsmitteln zu den Kosten [Tar05] des statischen Webers zu verstehen.

8.3 Instrumentierung

Eine wichtige Fragestellung ist die Festlegung, welche Teile der Anwendung instrumentiert werden soll. Um die Anwendung maximal dynamisch im Verhalten änderbar zu gestalten ist es notwendig, *alle* möglichen *join points* der Anwendung zu instrumentieren. Dies bedeutet jedoch auch maximalen zusätzlichen Aufwand, der von der instrumentierten Anwendung zu erbringen ist. Aus diesen Grund bietet es sich an, den *pointcut*, welcher die zu instrumentierenden *join points* festlegt, als Teil der Konfigurierung aufzunehmen.

Es bietet sich also an, die Auswahl der *join points*, die von Erweiterungsmodulen dynamisch gewoben werden können, als konfigurierbares Merkmal im Merkmalmodell zu definieren. Das Werkzeug `pure::variants` erlaubt es, Merkmalen Attribute zuzuordnen. Im Falle von *pointcuts* geschieht dies dadurch, dass die *match expression* ein Attribut des Merkmals *instrumentierte join points* ist. Abbildung 41 zeigt das Merkmaldiagramm für die Festlegung die Instrumentierung der *join points*.

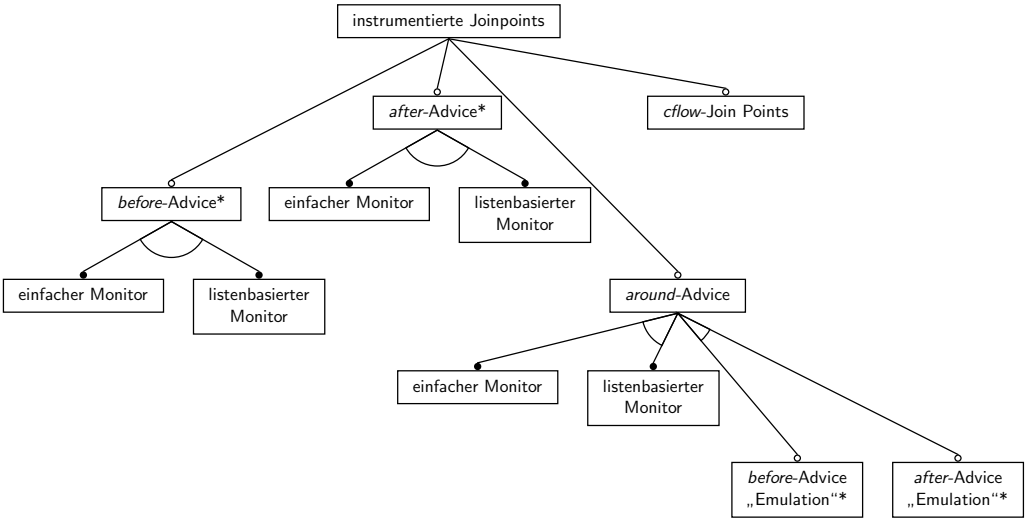


Abbildung 41: Das Merkmalmodell bestimmt welche *join points* auf welche weise instrumentiert werden. Dabei legt der Benutzer für jeweils *before*, *after*, und *around* einen *Pointcut* Ausdruck fest. Der Benutzer muss sich zwischen „echter“ und „emuliertem“ *before*- und *after*-Advice entscheiden. Mit * markierte Merkmale machen gleichzeitig gewählt keinen Sinn.

Die Semantik von *around*-Advice erlaubt es, die Funktion von *before*- und *after*-Advice zu implementieren. Beim *statischen* Weben von Aspekten ist dies im allgemeinen nicht empfehlenswert, da für *around*-Advice etwas mehr Betriebsmittel durch zusätzlichen Kontext[6] in Anspruch genommen wird. Im Falle von *dynamischem* Weben sind die Kosten jedoch noch vergleichbarer, falls für *before*- und *after*- Advice dynamischer Kontext bereitgestellt werden muss (siehe Unterabschnitt 8.2).

Unter *Emulation* von *before*- und *after*-Advice mittels *around*- Advice ver-

[6]Der zusätzliche Aufwand ist verhältnismäßig gering. Es wird ein sogenanntes *action object* angelegt, welches initialisiert werden muss. Der Aufbau dieses Objekts ist in der Sprachdefinition von AspectC++ nicht spezifiziert und daher auch im Rahmen dieser Arbeit nicht weiter untersucht

steht man die Erzeugung von *around*- Advice, der sich semantisch gleich verhält. Dies Technik erspart mehrfache Instrumentierung desselben *join points*, falls mehrere Advicearten gewoben werden sollen. Bei *Emulation* von *before*-/*after*-Advice muss sichergestellt sein, dass die *pointcuts* für die *native* und die *emulierte* Umsetzung im Merkmalmodell schnittfrei sind. Andernfalls würde Advice, welcher dynamisch an *join points* aus der Schnittmenge gewoben wird, mehrfach ausgeführt werden.

Wie in Unterabschnitt 5.4 bereits angedeutet wurde, ist es möglich noch weitere Betriebsmittel zur Laufzeit einzusparen, indem man sich auf einen einzelnen Advice pro *join point* beschränkt. Daher beinhaltet das Merkmalmodell für jeden *join point* die Wahl der Monitorimplementierung.

8.4 Zusammenfassung

Die Vorbereitung von Anwendungen auf dynamische Erweiterungen zur Laufzeit bedeutet im Rahmen dieser Arbeit, dass die Anwendungen statisch instrumentiert werden müssen, damit dynamischer Advice Aspekte an die instrumentierten *join points* binden kann. Diese Instrumentierung schlägt sich in erhöhtem Betriebsmittelgrundbedarf nieder, der auch dann zu erbringen ist, wenn keine Erweiterungen aktiv sind. Ein weiterer Kostenpunkt entsteht durch die Bereitstellung von Kontextinformationen, auf die der *Advice Code* zur Laufzeit zugreifen kann.

Um diese zusätzlichen Kosten zu senken, kann die Infrastruktur an die Anwendung durch statische Konfiguration angepasst werden. Die statische Konfiguration geschieht im Rahmen dieser Arbeit mit Hilfe des Konfigurationswerkzeuges `pure::variants`, das die Instrumentierung der Anwendung sowie den Umfang des dynamischen Kontextes auf das benötigte Maß zuschneidet. Als Ergebnis des Konfigurationsschrittes erhält man eine maßgeschneiderte Laufzeitbibliothek zur Umsetzung der in den letzten Kapiteln vorgestellten Techniken sowie einen an die Anwendungsbedürfnisse angepassten Aspekt zur Instrumentierung der Anwendung.

9 Werkzeuge für das Weben zur Laufzeit

Aspektorientierte Programmierung bedeutet nicht nur das Einführen von neuen Sprachkonzepten, sondern auch den Einsatz von neuen Werkzeugen zur Übersetzung, Integration und Entwicklung von Aspekten. Während die Werkzeugkette zur Entwicklung von objektorientierten Programmen weit verbreitet und gut bekannt ist, erfordert die Entwicklung von aspektorientierten Anwendungen die Integration von neuen Werkzeugen, die in die Abläufe zur Entwicklung und Wartung von Software eingebunden werden müssen. Dieses Kapitel stellt daher die einzelnen Werkzeuge vor, die bei der Entwicklung von Programmen, in die Aspekte sowohl *statisch* als auch *dynamisch* gewoben werden. Anschließend werden die im Rahmen dieser Arbeit entwickelten zusätzlichen Werkzeuge und damit die Komponenten der familienbasierten Infrastruktur vorgestellt, die für den Einsatz von dynamischen Aspekten notwendig sind.

9.1 Der statische Weber ac++

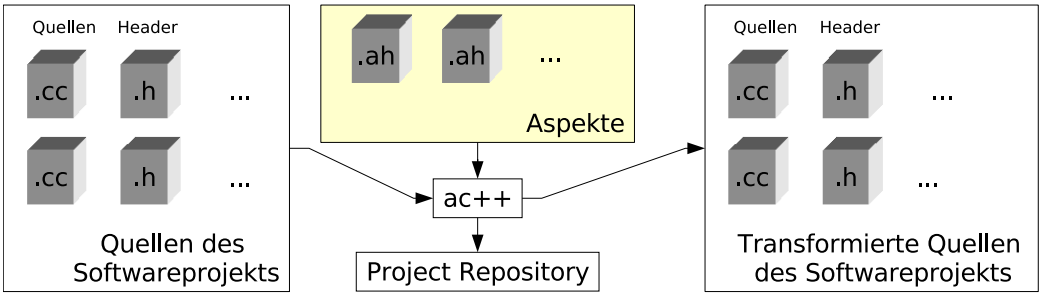


Abbildung 42: Bei der Entwicklung von aspektorientierten C++ Programmen ist der statische Weber ac++ von zentraler Bedeutung. ac++ transformiert dabei die Quelltexte des Softwareprojekts zu C++ Quelltexten, die von einem C++ Übersetzer weiterverarbeitet werden.

C++ Programme, in die Aspekte nur statisch gewoben werden, lassen sich mit einem einzigem weiterem Werkzeug übersetzen: Dem statischem Weber ac++. Abbildung 42 zeigt den Prozess zum statischem Weben von Aspekten. Normalerweise wird ac++ im sogenannten STU (***s**ingle **t**ranslation **u**nit*) Modus betrieben. Dabei transformiert ac++ jede Übersetzungseinheit einzeln, und speichert globale Informationen wie Programmstrukturen und *join points* in ein *project repository* in Form einer XML-Datei. Dieses *project repository* wird von ac++ intern gebraucht, um *join point* Kennungen über Grenzen

von Übersetzungseinheiten hinweg stabil zu halten. Es wird aber auch von anderen Werkzeugen wie zum Beispiel dem Werkzeug ACDT,[7] einer Erweiterung für die Entwicklungsumgebung Eclipse für AspectC++ Programme, benutzt, um Aspekte und deren Wirkung in AspectC++ Programmen im Quelltexteditor zu visualisieren.

Wie in Unterabschnitt 4.2 bereits angesprochen wurde, geschieht die Instrumentierung der Anwendung mit Hilfe eines konfigurierbaren (siehe Unterabschnitt 8.3) statischen Aspekts. Der in Abbildung 42 skizzierte Prozess ist als Teilprozess der Infrastruktur für dynamisches Weben zu verstehen.

9.2 Umsetzung von dynamischem Advice

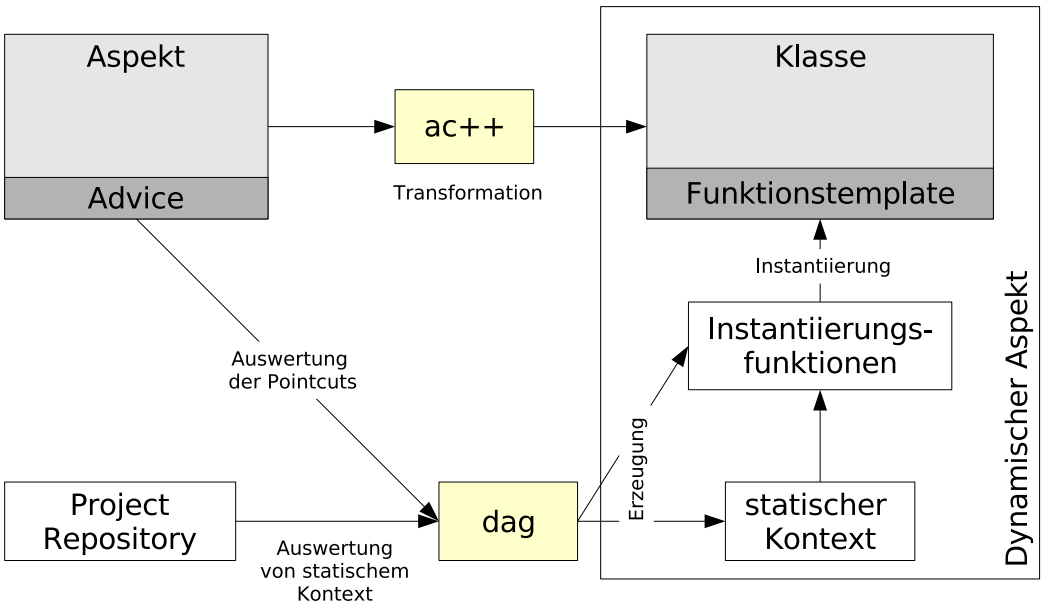


Abbildung 43: Bei der Umsetzung von dynamischem *Advice* spielen die Werkzeuge ac++ und dag zusammen. ac++ stellt dabei die den Aspekt in Form von transformierten Klassen und Advice in Form von Funktionstemplates bereit. dag wertet mit Hilfe des *project repository* die *pointcuts* aus und erzeugt daraus statischen Kontext, der zur Instantiierung der von ac++ erzeugten Templates benutzt wird.

Das Prinzip zur Übersetzung von dynamischem Advice in die Sprache C++ wurde bereits in Kapitel 5 vorgestellt. Dabei liegt der *Advice Code* im wesentlichen in der Sprache C++ vor. AspectC++ erweitert hier die C++ Syn-

[7] http://acdt.aspectc.org/

tax um die Joinpoint-API (siehe Unterabschnitt 2.6), die eine Programmierschnittstelle für den Zugriff auf Kontext (siehe Unterabschnitt 5.1) bietet.

Abbildung 43 illustriert das Zusammenspiel der Werkzeuge `ac++` und `dag`. Dabei ist `dag` (***d**ynamic **a**dvice **g**enerator*) das Werkzeug zur Generierung von dynamischem Advice. `ac++` übersetzt dabei Aspekte in C++ Klassen, wobei *Code Advice* in Funktionstemplates übersetzt wird. Diese Funktionstemplates werden mit einem (ebenfalls von `ac++` generierten) *join point* spezifischem Typ parametrisiert. Dieser Typ implementiert den statischen Teil der Joinpoint-API. Das Prinzip der Instantiierung von Advice wurde bereits in Unterabschnitt 5.1, Seite 31 erläutert. Zum *dynamischem* Weben können die von `ac++` erzeugten Aspektklassen mit den als Funktionstemplate übersetztem *Advicefunktionen* direkt benutzt werden. `dag` bestimmt anhand des *project repository* die *join points* der *pointcuts* der Aspekte. Mit dieser Information können nun (analog wie beim statischem Weber `ac++`) Typen generiert werden, die den statischen Kontext bereitstellen. Die Rolle des Generators aus Abbildung 13 auf Seite 32 übernimmt also das Werkzeug `dag`, das für jeden *join point*, einen *Advicewrapper* generiert, der mit Hilfe des C++ *template* Mechanismus die Typen mit dem *statischen* Kontext instantiiert und die von `ac++` erstellten Advicefunktionen zusammen mit dem von dem Laufzeitsystem bereitgestelltem *dynamischen* Kontext aufruft.

Der statische Kontext wird komplett aus dem *project repository* gewonnen und in Form von *join point* spezifischen Typen gekapselt. Der Codeausschnitt aus Abbildung 44 zeigt einen von `dag` erstellten Typ. Dieser Typ erweitert dabei den dynamischem Kontext durch Ableitung (Zeile 6). Um Typen aus der Anwendung auflösbar zu machen, müssen geeignete `#include` Anweisungen erzeugt werden (Zeile 3 und 4). In Zeile 28 wird die Signatur als statischer Kontext aufgenommen. Da diese Definition in einem Template geschieht, belegt die Zeichenkette nur dort Betriebsmittel, wo das Template instantiiert wird. Im Rahmen der hier vorgestellten Infrastruktur geschieht dies also im Aspektmodul.

Der Umfang des dynamischen Kontext ist, wie in Unterabschnitt 8.2 beschrieben, *projektweit* konfigurierbar. Diese Konfiguration bezieht sich im Wesentlichen auf die Klasse `DynamicJoinPoint`. Die Klasse `DynamicJoinPoint` ist ein Funktionstemplate, das den dynamischen Kontext kapselt. Es wird mit der *join point id* parametrisiert, und als Parameter `DynamicContext` dem *template* `StaticContext` (s.o.) übergeben. Diese Klasse stellt dem *Advice Code* eine Schnittstelle für den Zugriff auf den dynamischen Kontext bereit. Sie wird von der Instrumentierung initialisiert. Da die `StaticContext`-Strukturen keine Variablendefinitionen beinhalten, ist eine Erweiterung des

```
 1  template <int JPID, int REPOID, class DynamicContext>
 2  struct StaticContext {};
 3  #include "HttpRequest.h"
 4  #include "http.h"
 5  template <class DynamicContext>
 6  struct StaticContext <1223, 0, DynamicContext> : public DynamicContext {
 7    typedef int Result;
 8    typedef void That;
 9    typedef void Target;
10    static const int JPID = 1223;
11    static unsigned int id() { return 1223; }
12    static const AC::JPType JPTYPE = (AC::JPType)8;
13    static AC::JPType jptype () { return JPTYPE; }
14    enum { ARGS = 1 };
15    static unsigned int args() { return ARGS; };
16    template <int I, int DUMMY = 0> struct Arg {
17      typedef void Type;
18      typedef void ReferredType;
19    };
20    template <int DUMMY> struct Arg<0, DUMMY> {
21      typedef const char *Type;
22      typedef const char *ReferredType;
23    };
24    using DynamicContext::arg;
25    template <int I> typename Arg<I>::ReferredType *arg () {
26      return (typename Arg<I>::ReferredType*)arg (I);
27    }
28    static const char *signature ()
29    { return "int HttpRequest::parseHeader(const char *)"; }
30  };
```

Abbildung 44: Von dag erzeugter statischer Kontext

dynamischen Kontextes durch Ableitung problemlos möglich. Dieser *zusammengesetzte* Typ stellt dadurch die komplette Joinpoint-API so bereit, wie sie von der Sprachdefinition AspectC++ festgelegt wird.

Abbildung 45 zeigt einen (einfachen) Aspekt zur Programmablaufverfolgung in der Sprache AspectC++. Der darin enthaltene Advice nutzt dabei lediglich statischen Kontext. `ac++` übersetzt dann diesen Aspekt in die Klasse, die in Abbildung 46 abgebildet ist. Der statische Kontext wird dabei von der Klasse `JoinPoint`, der dynamische Kontext von dem übergebenen `tjp` Objekt gekapselt. Die von `dag` erzeugte Hilfsfunktion zur Instantiierung des *Advice* in Abbildung 47 erzeugt ein zu der in der AspectC++ Sprachdefinition beschriebenen „`tjp`-Struktur“ kompatible „`djp`-Struktur“, die uniformen Zugriff auf sowohl statischen als auch dynamischen Kontext erlaubt, wie er von der Sprachdefinition der Sprache AspectC++ festgelegt wird. Der dynamische Kontext an sich wird dieser Funktion vom Laufzeitsystem als Parameter übergeben.

```
1  #include <iostream>
2  aspect Tracer {
3    advice tracepoints() : after() {
4      printf("running at joinpoint %d for %s\n",
5           JoinPoint::id(), JoinPoint::signature());
6    }
7  };
```

Abbildung 45: Einfacher Aspekt zur Programmablaufverfolgung

```
1  class Tracer {
2  public:
3    static Tracer *aspectof () {
4      static Hello __instance;
5      return &__instance;
6    }
7
8     template <class JoinPoint>
9     invoke_hello_a0_after (JoinPoint *tjp) {
10      printf("running at joinpoint %d for %s\n",
11           JoinPoint::id(), JoinPoint::signature());
12    }
13 };
```

Abbildung 46: Erzeugte Klasse und Instantiierung von dynamischem Advice

```
1   void __dacwrapper_0_CheckForBrokenClients_a0_after(void *djp) {
2    typedef StaticContext<1223, 0, DynamicJoinPoint> DJP;
3    Hello::aspectof()->__a0_after<DJP>((DJP*) djp);
4  }
```

Abbildung 47: Hilfsfunktion zur Instantiierung von dynamischem Advice

9.3 Umsetzung von dynamischen Einfügungen

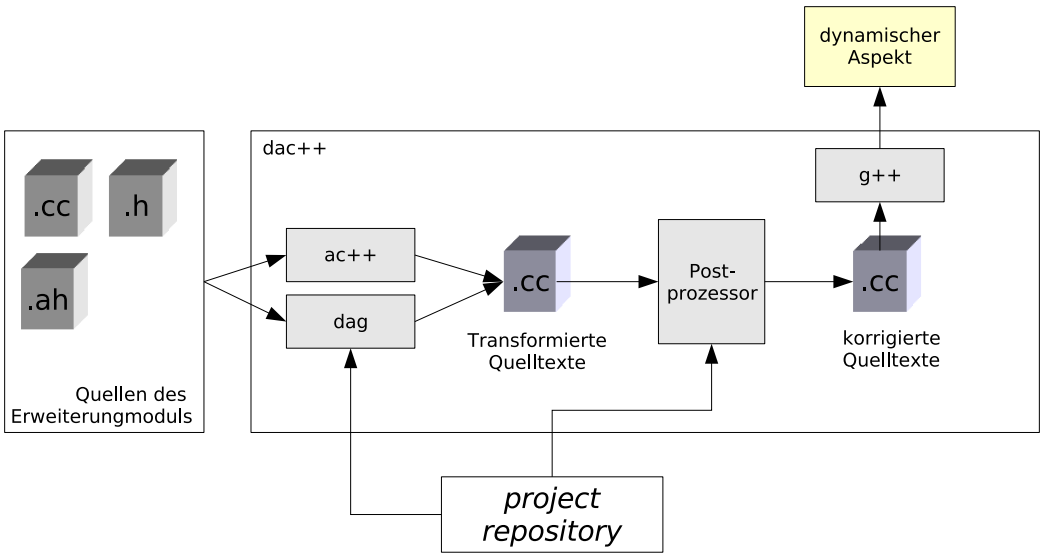


Abbildung 48: Die Struktur der dynamischen Infrastruktur dac++

In Kapitel 6.1 wurden die Schwierigkeiten bei der Umsetzung von dynamischen Einfügungen von Attributen vorgestellt. Im Rahmen dieser Arbeit wird dabei die *Introduction Pointer* Variante aus Unterabschnitt 6.1.3 implementiert. Zur Umsetzung des Zugriffs und der *Korrektur* der eingefügten Attribute spielen verschiedene Komponenten zusammen. Die Abbildung 48 zeigt deren Zusammenspiel, und ist als Verfeinerung von Abbildung 43 zu verstehen. Aspekte mit dynamischen Einfügungen und dynamischem Advice werden vom statischem Aspektweber `ac++` in C++ Quelltexte transformiert. Dabei wird `ac++` in einem speziellem Modus betrieben, der alle dynamischen Einfügungen mit einer Markierung in Form eines C++ Kommentars versieht. Ebenfalls markiert werden Zugriffe auf eingefügte Attribute.

Der *Postprozessor*, der in Unterabschnitt 6.1.4 auf Seite 43 vorgestellt wurde, entfernt bzw. ersetzt dann die von `ac++` markierten Stellen im Programmtext mit Aufrufen in das Laufzeitsystem. Er schlägt dabei im *project repository* nach, ob die Klasse, in die eingefügt wurde, bereits eingetragen ist. Falls sie (noch) nicht bekannt ist folgt daraus, dass die Klasse im selben Erweiterungsmodul eingebracht wurde. Das heißt, dass diese Einfügung nicht *dynamisch* sondern *statisch* geschieht. Für *statische* Einfügungen sind keine Anpassungen wie für *dynamische* nötig, sondern können wie beim herkömmlichem statischen Weben vorgenommen werden.

9.4 Aspektlader

Damit ein Aspektmodul zur Laufzeit geladen werden kann, muss die Anwendung Systemfunktionen benutzen, um den Ladevorgang zu initiieren. Dies geschieht in der Komponente Aspektlader, der statisch zur Anwendung dazugebunden wird. Die konkrete Implementierung des Nachladens von Erweiterungsmodulen ist abhängig von der verwendeten Betriebssystemumgebung. In dieser Arbeit wurde ein GNU/Linux System verwendet, das die Betriebssystemaufrufe `dlopen()` und POSIX Signalbehandlung bereitstellt.

Die Laufzeitumgebung wurde in Form einer statischen Bibliothek mit dem Namen `libdacrt.a` entwickelt. Sie besitzt eine *Konstruktorfunktion*, eine Eigenheit der GNU Umgebung, die vor der Funktion `main()` ausgeführt wird. In dieser Konstruktorfunktion wird ein Signalhandler installiert, der bei Aktivierung einen *FIFO* öffnet und auf ein Kommando wartet. Das Kommando zum Laden von dynamischen Aspektmodulen beinhaltet den Pfad zu einem Aspektmodul, der in den *FIFO* geschrieben wird. Das Kommando zum Entladen bewirkt, dass die Laufzeitumgebung den jeweils zuletzt geladenen Aspekt entfernt.

9.5 Zusammenfassung

Während in den vorigen Kapiteln die Konzepte und Lösungsansätze vorgestellt wurden findet in diesem Kapitel die Vorstellung der im Rahmen dieser Arbeit entwickelten Werkzeuge statt. Hierbei wurde der statische Weber `ac++` aus dem AspectC++ Projekt (fast) unverändert übernommen.

Ziel dieser Arbeit ist es, auf Basis der Sprache AspectC++ die Möglichkeit zu schaffen, Aspekte zur Laufzeit zu weben. Dazu müssen Konzepte und Werkzeuge geschaffen werden, die nachladbare Aspektmodule übersetzen und Laden können. Dabei wird auf vorhandene Werkzeuge wie `ac++` sowie den Übersetzer `g++` zurückgegriffen. Neu entwickelt werden musste ein Werkzeug zur Generierung von Aspektinstanzen. Das in dieser Arbeit dazu entwickelte Werkzeug heißt `dag`.

Zur Umsetzung von Einfügungen ist es weiterhin nötig, dass die transformierten C++ Quelltexte *korrigiert* werden. Diese Aufgabe übernimmt der *Postprozessor*. Das eigentliche Laden von Aspektmodulen ist exemplarisch mit Hilfe von GNU/Linux Eigenheiten und Unix Signalbehandlung implementiert.

10 Evaluation

Wie in den einleitenden Kapiteln betont wird, erlaubt die aspektorientierte Programmierung die Kapselung von quer schneidenden Belangen, die weder mit strukturierter noch mit funktionaler oder mit objektorientierter möglich sind. Diese Technik ermöglicht also eine neue Dimension von programmiersprachliche Abstraktion. Jede neue Abstraktion bringt jedoch neuen Bedarf an Betriebsmitteln mit sich. Die in dieser Arbeit eingeführte dynamische Umsetzung von AOP muss daher zeigen, dass der Mehraufwand gemessen an den neu geschaffenen Möglichkeiten gerechtfertigt ist.

Dieses Kapitel analysiert daher zunächst den Grundbedarf an Betriebsmitteln bei Verwendung der Infrastruktur `dac++` in Form von *Microbenchmarks*. Anschließend wird beispielhaft für die Softwareprojekte *ACE* und *Squid* gezeigt, welche Anwendungsgebiete sich für dynamische AOP auftun.

10.1 Microbenchmarks

Um den Grundbedarf an Betriebsmitteln von `dac++` zu senken wird die Infrastruktur konfigurierbar entworfen. Aus diesem Grund ist es notwendig, bei der Evaluation möglichst viele Konfigurationen zu betrachten. Die Konfigurationsmöglichkeiten wurden bereits in Kapitel 8 vorgestellt und andiskutiert. Die Verwendung eines dynamischen Webers in einer Anwendung verursacht zusätzlichen Bedarf an den Betriebsmitteln Stapelspeicher, Laufzeit und Codegröße. Diese Betriebsmittel werden mittels eines synthetisierten Minimalbeispiels bestimmt. Abbildung 49 zeigt die Funktion, die in diesem Unterabschnitt als Messgrundlage verwendet wurde.

Die Messungen der Laufzeit sowie des Bedarfs an Stapelspeicher wurden auf einem Pentium-M Notebook mit 1,4 GHz Taktfrequenz durchgeführt. Dabei wurde als Betriebssystem Debian GNU/Linux 4.0 mit GCC 4.1.2 und ac++ 1.0pre3 verwendet. Die Messmethode macht sich einen Messaspekt zu nutze, der in einer früheren Arbeit [Tar05] entwickelt wurde. Hierbei wird die zu messende Funktion (Abbildung 49) mehrfach ausgeführt und jeweils unmittelbar vor und nach dem Aufruf dieser Funktion das `RDTSC` Register aus der CPU ausgelesen. Die Differenz stellt die Laufzeit der Funktion in CPU Zyklen dar. Die Messung des Bedarfs an Stapelspeicher benötigt die Kooperation der Anwendung: Sie muss im Funktionsrumpf den Inhalt des *%esp* in eine globale Variable sichern. Der Messaspekt (Abbildung 50) vergleicht diesen Wert mit dem Inhalt des *%esp* Registers unmittelbar vor dem Funktionsaufruf.

```
1     int testfunc(int argc, char **argv) {
2       // save current stack
3       asm volatile ("movl %%esp, %0\n" : "=m" (sp):);
4       return 0;
5     }
```

Abbildung 49: Minimale, leere Funktion, die den Messungen zu Grunde liegt

```
1  #define RDTSC(x) __asm__ volatile ( "cpuid\n rdtsc\n" : "=A" (x));
2  aspect measure {
3    static const int COUNT = 500;
4    static const int RUNS = 500;
5    static const int SKIP = 10;
6
7    pointcut measurepoint() = call ("% testfunc(...)");
8    advice measurepoint() : order ("measure", !"measure");
9    advice measurepoint () : around () {
10     unsigned long long ticks_start, ticks_end;
11     unsigned long long results[COUNT];
12     unsigned overhead;
13     int skip = SKIP;
14
15     sched_yield(); // get a fresh timeslice from the scheduler
16     asm volatile ("movl %%esp, %0\n" : "=m" (overhead):); // save current
           stack
17
18     while(skip--) // warm up
19       tjp->proceed();
20
21     for (int j = 0; j < COUNT; j++ ) {
22       RDTSC(ticks_start);
23       for (int i = 0; i < RUNS; i++) {
24         tjp->proceed();
25       }
26       RDTSC(ticks_end);
27       results[j] = ticks_end - ticks_start;
28     };
29
30     std::sort(results, results + COUNT);
31     printf("%s running %d times.\nStack: %d\nMedian: %d\n",
32          tjp->signature(), COUNT * RUNS,
33          (unsigned) (overhead) - (unsigned) (sp),
34          results[COUNT/2] / RUNS );
35   }
36 };
```

Abbildung 50: Der verwendete Messaspekt

	Laufzeit	Stack	Funktionsgröße	Laufzeit	Stack	Programmgröße	Laufzeit	Stack	Programmgröße
Adviceart	before			after			around		
Ohne Instrumentierung	11	0	13	11	0	13	11	0	13
Kein Kontext	20	48	26	22	40	34	-	-	-
arguments	25	48	48	28	40	56	43	64	108
result	23	48	36	24	40	44	42	64	108
arguments + result	27	46	54	30	56	62	46	64	111
arguments + result + that + target	30	46	68	33	56	76	47	80	125

Tabelle 3: Microbenchmarks für Instrumentierung mit „einfachem“ Advice Container

Eine wichtige Entscheidung bei Konfiguration des Laufzeitsystems ist die Wahl der an den instrumentierten *join points* installierten Monitoren. In Unterabschnitt 5.4 wurde zwei alternative Implementierungen vorgestellt. Es werden zunächst die Ergebnisse für die Messungen mit der Variante *einfacher Monitor* durchgeführt. Tabelle 3 zeigt die Ergebnisse bei Verwendung der Übersetzeroption `-Os`. Diese Übersetzeroption wird nicht allgemein für die Übersetzung von Anwendungsprogrammen empfohlen. In diesem Fall wird sie als Hilfestellung für den Übersetzer gewählt, damit der Programmcode für die von `ac++` und `dag` (siehe Kapitel 9) erzeugten Hilfsstrukturen besser optimiert wird. Im Bereich eingebetteter Systeme hingegen ist die gewählte Übersetzeroption durchaus üblich.

Die Spalte *Stack* zeigt die Zahlen für den Bedarf an Stapelspeicher gemessen in Bytes. Man könnte für *before*- und *after*- Advice dieselben Messergebnisse erwarten, da ja die gleichen Hilfsstrukturen verwendet werden. Sie unterscheiden sich dennoch um wenige Bytes, weil sich für den Übersetzer aufgrund einer anderen Reihenfolge der Instruktionen andere Optimierungsmöglichkeiten ergeben. Man stellt gerade in der Spalte für *around*-Advice ein gestuftes Wachstum des Bedarfs fest. Der Grund liegt daran, dass der Übersetzer bei der Reservierung des Stapelspeichers darauf achtet, dass der *Stackframe* der Funktion auf einer *geraden* Adresse anfängt. Die Intelarchitektur erlaubt zwar den Zugriff auf beliebige Adressen, beim Zugriff auf *ungerade* Adresse bedeutet dies jedoch Laufzeitnachteile für das Programm. Bei *around*- Advice ist der Bedarf an Stapelspeicher wesentlich höher. Dies liegt

	Laufzeit	Stack	Programmgröße	Laufzeit	Stack	Programmgröße	Laufzeit	Stack	Programmgröße
Adviceart	before			after			around		
Ohne Instrumentierung	11	0	13	11	0	13	11	0	13
Kein Kontext	20	48	26	22	40	26	-	-	-
arguments	25	48	48	28	40	48	43	64	108
result	22	48	36	24	40	36	42	64	108
arguments + result	27	64	54	30	56	54	46	64	111
arguments + result + that + target	30	64	68	33	56	68	47	80	125

Tabelle 4: Microbenchmarks für Instrumentierung mit *listenbasierten* Advice Container

daran, dass `ac++` in diesem Fall größere Hilfsstrukturen anlegen muss. `dac++` legt an zusätzlichen Kosten hier lediglich eine weitere Referenz auf das *action object* an.

Die Spalte *Laufzeit* gibt die Laufzeiten der Funktion aus Abbildung 49 in CPU-Zyklen an. Die einzelnen Laufzeiten unterscheiden sich zwischen den einzelnen Varianten auf der verwendeten Testmaschine ein wenig. Der Grund für die unterschiedlichen Laufzeiten bei *before*- und *after*- Advice liegt in der Tatsache, dass zwar die gleichen Assembleranweisungen ausgeführt werden, aber in unterschiedlicher Reihenfolge. Dies kann Auswirkungen auf die Auslastung der *Pipelines* in der CPU haben, was durchaus Schwankungen in der Größenordnung von ein paar Taktzyklen bedeuten kann. Für *around*- Advice muss eine zusätzliche Struktur[8] angelegt und als dynamischer Kontext bereitgestellt werden, was sich in etwas höheren Laufzeiten niederschlägt

Die Größen der instrumentierten Funktion unterscheiden sich zwischen *before*- und *after*- Advice bei gleichem bereitgestellten Kontext wieder aufgrund unterschiedlicher Optimierungsmöglichkeiten um wenige Bytes. Für *around*- Advice ist der zusätzliche Aufwand für Initialisierung des *action object* deutlich sichtbar.

Abbildung 4 zeigt die Ergebnisse des gleichen Versuchaufbaus bei Verwendung von listenbasierten Monitoren. Unterschiede sind lediglich in der Programmgröße bemerkbar. Dies liegt daran, dass eine *listenbasierte* Implemen-

[8] das *action object*

tierung von *Advice Containern* etwas mehr Programmcode beansprucht. Der zusätzliche Bedarf der Betriebsmittel *Stapelspeicher* und *Laufzeit* bei der Instrumentierung der Anwendung ist also unabhängig von der Wahl der Monitore.

10.2 Das ACE Framework

Das *ADAPTIVE Communication Environment* (ACE) ist ein objektorientiertes, freies Framework zur Unterstützung bei der Entwicklung von portabler Middleware. Es bietet Klassen für Interprozesskommunikation, Fadenverwaltung, effizienter Speicherverwaltung, Signalverarbeitung, Nebenläufigkeit und Synchronisation. Es wurde von Douglas C. Schmidt et. al. zu Forschungszwecken entworfen (z.B.: [SSRB00, HMS98])

Eine bekannte und von denselben Entwicklern entworfene und entwickelte Middleware ist hierbei *„The ACE ORB“* (TAO) [SLM98, Sch98, PSCS01]. TAO ist eine CORBA Implementierung, die das ACE Framework benutzt. Dabei handelt es sich um eine dem auf durchgängig objektorientierte und speziell auf Echtzeitanforderungen optimierte Implementierung der CORBA Spezifikation nach der OMG. Es wird sowohl zu Forschungszwecken als auch in produktiven Umgebungen eingesetzt.

10.2.1 Instrumentierung

Im Rahmen dieser Evaluation wurde die Bibliothek ACE mit einem statischem Instrumentierungsaspekt übersetzt, wie er in Abbildung 51 abgedruckt ist. Dieser Aspekt kann mit einem Konfigurationswerkzeug wie das in Unterabschnitt 8.1 vorgestellt `pure::variants` erzeugt werden, da er keine eigentliche Implementierungsdetails beinhaltet, sondern nur gewählte *Merkmale* festlegt. In Zeile 6 wird festgelegt, dass die angegebenen *join points* mit *before*- Advice instrumentiert werden. Die Zeilen 9 und 10 legen die *pointcut expressions* fest, an denen Advice zur Laufzeit gewoben werden kann. In diesem Beispiel wurde von der Möglichkeit gebrauch gemacht, nicht *alle* möglichen *join points* aus der Bibliothek ACE zu instrumentieren; die Zeile 12 nimmt Operatoren aus der Menge der instrumentierten *join points* aus.

Mit diesem Aspekt wurden insgesamt 2680 *execution join points* instrumentiert. Dabei wächst die Größe der statischen Bibliothek `libACE.a` von 17460758 auf 41734866 Bytes, also um etwa 140%. Der Grund für diesen drastischen Anstieg liegt darin, dass in die Bibliothek der Instrumentierungsaspekt statisch eingewoben wird. Dadurch ensteht bei der Übersetzung

```
1  #ifndef __pv___instrumentExe__
2  #define __pv___instrumentExe__
3
4  #include "static_instrumentation.ah"
5
6  aspect static_instrumentation : public InstrumentBefore {
7
8
9   pointcut virtual dyn_advice_without_context() = call("");
10  pointcut virtual dyn_advice_with_context() = execution(
11      "% ACE_%::%(...)" &&
12      ! "% ...::operator =(...)" && ! "% ...::operator ==(...)" );
13 };
14
15 #endif //__instrumentExe__
```

Abbildung 51: Instrumentierungsaspekt

```
1   #include <iostream>
2   aspect Tracer {
3     advice execution("% ...::ACE_Event%::%(...)") : before () {
4       std::cout << "Tracer: " << JoinPoint::signature() << std::endl;
5     }
6   };
```

Abbildung 52: Aspekt zur Verfolgung des Programmablaufs

der einzelnen Übersetzungseinheiten sehr viel zusätzlicher redundanter Programmcode. Beim Binden dieser Bibliothek mit einer Anwendung ist zu erwarten, dass viel Programmcode vom Binder entfernt wird.

10.2.2 Generische Programmverfolgung in AspectC++

Gerade für Entwickler, die sich in eine bestehende Software neu einarbeiten müssen, ist es wichtig, Programmabläufe der Software zu verstehen. AOP bietet hier Möglichkeiten zur Darstellung des Kontrollflusses eines laufenden Programms. Dazu kann wie der in Abbildung 52 abgebildete Aspekt verwendet werden.

Dieser Aspekt gibt an allen *join points* des *pointcuts* in Zeile 3 die Signatur der Funktion, in dessen Kontext der Advice ausgeführt wird. Zur Änderung des *pointcuts* muss im Falle von *statischem* Weben jedes Mal die ganze Anwendung gewoben werden. Auf einem 2 Prozessor Dual Core Opteron 280 System mit jeweils 2400 Mhz benötigt der Bauvorgang der Bibliothek ACE etwa 20 Minuten.

Falls der Aspekt *dynamisch* gewoben wird, muss nur für die Instrumentierung einmal *statisch* gewoben werden. Der eigentliche Aspekt zur Programmverfolgung kann anschließend ohne die Bibliothek neu zu übersetzen *dynamisch* gewoben werden. Die Übersetzung des dafür benötigten Aspektmoduls benötigt weniger als 5 Sekunden auf dem gleichem System. *Dynamisches* Weben bringt senkt in diesem Fall die Zeitdauer für den Entwicklungsmikrozyklus *Editieren - Übersetzen - Testen* also drastisch.

Der gerade vorgestellte Aspekt soll nun um die Ausgabe der Funktionsparameter verbessert werden. Dabei soll der Aspekt so generisch wie möglich entworfen werden, um möglichst viele Typen ausgeben zu können. Die meisten Typen in `C++` überladen zur Ausgabe den `operator<<`. Daher soll dieser verwendet werden, falls einer für diesen Typ existiert. Andernfalls soll die Adresse des Objekts ausgegeben werden. Dabei kann im Aspektmodul für Datentypen, die den Ausgabeoperator nicht implementieren, ein solcher mitgeliefert werden. Um zu erkennen, ob ein Typ einen solchen Ausgabeoperator implementiert, wird ein Trick der Sprache C++ ausgenutzt. Das folgende Programmfragment führt die Erkennung zur Übersetzungszeit durch:

```
1  struct X { X (...) {} };
2
3  typedef char magically_sized_type[43564367];
4  magically_sized_type &operator<< (ostream &os, X);
5
6  template <typename T> struct detector {
7    enum { RET = (sizeof (cout << T()) == sizeof (magically_sized_type)) };
8  };
```

In Zeile 1 wird eine Klasse definiert, die aus jedem Typ heraus konstruiert werden kann. Dazu wird ein allgemeiner Konstruktor mit einer Ellipse als Parameterliste verwendet, was dazu führt, dass dieser Konstruktor in der *overload resultion* (siehe Unterabschnitt 6.5 auf Seite 57) ein sehr schlechtes *ranking* bekommt. In den nächsten Zeilen wird ein Typ mit einer sehr ungewöhnlichen Größe definiert, die (hoffentlich) kein realer Typ erreichen wird. Der eigentliche Detektor wird in Zeile 6 definiert. Er versucht einen Ausgabeoperator aufzulösen, von dem die Größe seines Rückgabetyps mit der Größe des soeben definierten Typs mit *ungewöhnlicher* Größe verglichen wird. Es gibt nun zwei Möglichkeiten: Die erste Möglichkeit ist, dass für diesen Typ ein Ausgabeoperator aufgelöst werden kann. In diesem Fall gibt der `sizeof()` Operator einen Wert zurück, der (hoffentlich) nicht 43564367 ist was dazu führt, dass das `enum` *RET* auf 0 gesetzt wird. Andernfalls wird der Ausgabeoperator aus Zeile 1 ausgewählt, was dazu führt, dass das `enum` *RET* den Wert 1 erhält. Zur eigentlichen Ausgabe eines Arguments wird ein *template meta program* benutzt. Es benutzt dabei den Ausgabeoperator des

parametrisierten Arguments `arg`, um dessen Inhalt auf der Standardausgabe auszugeben. Es hat den folgenden Aufbau:

```
1   template <typename T, int> struct Printer {
2   static void print (ostream &out, const T &arg) {
3     out << arg << endl;
4   }
5  };
6
7  template <typename T> struct Printer<T,1> {
8   static void print (ostream &out, const T &arg) {
9     out << "Unprintable type, addr: " << &arg
10        << "size: " << sizeof (T) << endl;
11  }
12 };
13
14 template <typename T> void print (ostream &out, const T &arg) {
15  Printer<T, detector<T>::RET>::print (out, arg);
16 }
```

Durch *partielle Spezialisierung* wird eine Fallunterscheidung durchgeführt, ob der oben vorgestellte *detector* eine Klasse mit oder ohne Ausgabeoperator erkannt hat. Dieses *generische* Programm zur Ausgabe von beliebigen Datentypen wird nun von einem weiterem *template meta program* benutzt, das die AspectC++ Joinpoint-API (siehe Unterabschnitt 2.6) benutzt, um alle Argumente der Funktion des *join points* rekursiv auszugeben:

```
1  template <int I> struct ArgPrinter {
2   template <class JP> static inline void work (JP &tjp) {
3     ArgPrinter<I - 1>::work (tjp);
4     cout << "Arg " << I << ": ";
5     print(cout, *tjp.template arg<I - 1> ());
6     cout << endl;
7   }
8  };
9
10 template <> struct ArgPrinter<0> {
11  template <class JP> static inline void work (JP &tjp) {}
12 };
```

Mit diesen Hilfsmitteln kann nun ein *generischer* Aspekt geschrieben werden, der alle Funktionsparameter der Reihe nach auf der Standardausgabe (`cout`) ausgibt:

```
1  aspect DynamicTracer {
2   template <class JP> void print_args (JP &tjp) {
3     ArgPrinter<JP::ARGS>::work (tjp);
4   }
5   advice execution("% ...::ACE_Event%::%(...)") : before() {
6     cout << "now running: " << JoinPoint::signature() << endl;
7     tjp->arg<0>();
8     print_args (*tjp);
9   }
10 };
```

Textsegment	Datensegment	BSS-Segment	Gesamtgröße
26473	292	168	26933

Tabelle 5: Größen für das generische Aspektmodul zur Programmverfolgung

Diese Implementierung der Programmablaufverfolgung hat verglichen mit der bereits in ACE eingebauten Version den Vorteil, dass hier die Funktionsparameter ausgegeben werden.

10.2.3 Ergebnisse

Dieser Unterabschnitt hat einen generischen Aspekt zur Programmablaufverfolgung gezeigt, der anhand von mitgelieferten ACE Beispielen getestet wurde. Der Aspekt ist generisch, weil er unabhängig von den jeweiligen Typen, die Funktionsparameter ausgeben kann. Dies funktioniert nur deshalb, weil Advice für jeden *join point* einzeln instantiiert wird. Instantiierung bedeutet, dass für jeden *join point* neu Programmcode erzeugt wird, der sich jedoch in der Wahl des Ausgabeoperators unterscheidet.

Für einen Aspekt, wie er im vorigem Abschnitt vorgestellt wurde, werden für das Beispiel `examples/Reactor/FIFO` beispielsweise 37 *join points* instrumentiert. Dabei erreicht der dynamische Aspekt Segmentgrößen, die in Tabelle 5 abgebildet werden. Die dazu erzeugten Adviceinstanzen belegen dabei Codegrößen im Bereich von 53 Bytes bis 416 Bytes, je nach verwendetem Ausgabeoperator. Hier wird ein interessanter Effekt der verwendeten Techniken zum Weben von Aspekten sichtbar: Die generierten *Adviceinstanzen* haben an ihren statischen Ausführungskontext angepasste Codegrößen.

Für den Fall, dass ein Typ keinen geeigneten Ausgabeoperator besitzt, kann dieser mit dem Aspektmodul mitgeliefert werden. Dadurch ist es möglich, auch Typen auszugeben, die zum Zeitpunkt des Entwurfs der Basisanwendung nicht darauf vorbereitet wurden. Der dafür erforderliche zusätzliche Programmcode belegt dabei nur Platz im Aspektmodul, nicht in der Basisanwendung.

Name des Advicewrappers	Größe in Bytes
__dacwrapper_0_DynamicTracer_a0_before(void*)	53
__dacwrapper_2_DynamicTracer_a0_before(void*)	53
__dacwrapper_11_DynamicTracer_a0_before(void*)	53
__dacwrapper_15_DynamicTracer_a0_before(void*)	53
__dacwrapper_16_DynamicTracer_a0_before(void*)	53
__dacwrapper_20_DynamicTracer_a0_before(void*)	53
__dacwrapper_21_DynamicTracer_a0_before(void*)	53
__dacwrapper_22_DynamicTracer_a0_before(void*)	53
__dacwrapper_23_DynamicTracer_a0_before(void*)	53
__dacwrapper_24_DynamicTracer_a0_before(void*)	53
__dacwrapper_25_DynamicTracer_a0_before(void*)	53
__dacwrapper_28_DynamicTracer_a0_before(void*)	53
__dacwrapper_29_DynamicTracer_a0_before(void*)	53
__dacwrapper_31_DynamicTracer_a0_before(void*)	53
__dacwrapper_33_DynamicTracer_a0_before(void*)	53
__dacwrapper_34_DynamicTracer_a0_before(void*)	53
__dacwrapper_35_DynamicTracer_a0_before(void*)	53
__dacwrapper_36_DynamicTracer_a0_before(void*)	53
__dacwrapper_1_DynamicTracer_a0_before(void*)	149
__dacwrapper_3_DynamicTracer_a0_before(void*)	149
__dacwrapper_4_DynamicTracer_a0_before(void*)	149
__dacwrapper_5_DynamicTracer_a0_before(void*)	149
__dacwrapper_6_DynamicTracer_a0_before(void*)	149
__dacwrapper_8_DynamicTracer_a0_before(void*)	149
__dacwrapper_12_DynamicTracer_a0_before(void*)	149
__dacwrapper_13_DynamicTracer_a0_before(void*)	149
__dacwrapper_14_DynamicTracer_a0_before(void*)	149
__dacwrapper_17_DynamicTracer_a0_before(void*)	149
__dacwrapper_26_DynamicTracer_a0_before(void*)	149
__dacwrapper_27_DynamicTracer_a0_before(void*)	149
__dacwrapper_30_DynamicTracer_a0_before(void*)	149
__dacwrapper_7_DynamicTracer_a0_before(void*)	238
__dacwrapper_9_DynamicTracer_a0_before(void*)	238
__dacwrapper_19_DynamicTracer_a0_before(void*)	238
__dacwrapper_32_DynamicTracer_a0_before(void*)	238
__dacwrapper_10_DynamicTracer_a0_before(void*)	327
__dacwrapper_18_DynamicTracer_a0_before(void*)	416

Tabelle 6: Größen der *Advicewrapper* des generischen Programmablaufverfolgers

10.3 Der Proxyserver Squid

Ein großer Anteil an Datenverkehr im Internet wird über das HTTP Protokoll abgewickelt. Um Transportkosten und Verzögerungen zu vermeiden werden sogenannte *HTTP Proxys* eingesetzt, die übertragene Objekte in einem Zwischenspeicher einlagern und bei nochmaliger Anfrage wiedergeben. Dabei braucht nicht das komplette Objekt erneut übertragen zu werden. [ASA+95] nennt als maximal erreichbare Treffrate in realen Szenarien von etwa 40% bis 50%.

Ein aus dem Umfeld freier Software entsprungener bekannter Proxyserver heißt Squid[9]. Er ist aus dem Harvest-Projekt entstanden, das bis zur Version 1.4 von Peter B. Danzig und Duane Wessels entwickelt wurde. Harvest wurde ab Version 2.0 von Danzig kommerziell weiterentwickelt, während Wessels aus den gleichen Harvest-Quellen den freien Squid entwickelte. Er wird speziell im Umfeld unixartiger Betriebssysteme eingesetzt.

Squid wurde bis zur Version 2 objektorientiert, aber in der Programmiersprache `C` entwickelt. Die Version 3 stellt eine Portierung in die Sprache `C++` dar. Er besitzt eine statisch modularisierte Struktur, die zur Übersetzungszeit festgelegt ist. Der gängige Ansatz um Squid um Funktionalität zu erweitern liegt darin ein weiteres Modul zu implementieren. Diese Erweiterungen müssen jedoch zusammen mit der Basisanwendung übersetzt werden; eine Erweiterung zur Laufzeit ist nicht vorgesehen.

In diesem Unterabschnitt wird aufgezeigt, wie mittels dynamischer AOP ein unerwartetes Problem analysiert und behoben werden kann. Dabei soll es nicht notwendig sein, dass die Anwendung neu gestartet wird. Es wird daher eine (konstruierte) Problemstellung vorgestellt und die einzelnen Schritte, die zur Problembehebung führen, beschrieben.

Instrumentierung

Zur Vorbereitung für das dynamische Weben von Aspekten zur Laufzeit muss die Basisanwendung instrumentiert werden. Es werden alle `execution-` *Join Points* mit *after-* Monitoren instrumentiert. Im Falle der Version `3.0 pre4` entspricht dies 3099 instrumentierter Funktionen. Im Familienmodell wird das Merkmal `result` gewählt um *Code Advice* den Rückgabewert als dynamischen Kontext bereitzustellen. Dabei steigt die Programmgröße bedingt durch die Wahl der Konfiguration (siehe Kapitel 8) von 1,7 MB auf 2,2 MB.

[9] http://squid-cache.org

10.3.1 Generische Programmverfolgung

Basierend auf dieser Version von Squid können nun Aspekte, die in der Programmiersprache AspectC++ geschrieben sind, zur Laufzeit gewoben werden. Als erstes Beispiel wird der in Unterabschnitt 10.2.2 vorgestellte generische Aspekt zur Programmablaufverfolgung eingesetzt. Bei statischem Weben müsste dieser Aspekt von `ac++` zusammen mit der Anwendung übersetzt werden, was etwa 17 Minuten auf dem verwendetem Entwicklungsrechner beanspruchen würde. Die Übersetzung *desselben* Aspekts in ein Aspektmodul benötigt hingegen lediglich unter 5 Sekunden. Dies macht es sehr angenehm den Aspekt zu verändern und neu zu übersetzen, um z.B. Menge der zu überwachenden *join points* einzugrenzen oder zu erweitern, bis die für den Programmierer interessante Stelle gefunden ist.

In einem fiktivem Beispiel wird der Proxyserver Squid zum Zwischenlagern von Webseiten in einer Firma verwendet. Ein Benutzer meldet ein Problem beim Herunterladen großer Dateien. Noch während Squid läuft, wird ein Aspekt dynamisch gewoben, welcher den HTTP Nachrichtenaustausch und die deren internen Abläufe innerhalb von Squid detailliert anzeigen kann:

```
1 #include <iostream>
2 aspect HTTPTracer {
3   advice execution("% ... Http%::%(...)") : after() {
4     cout << "trace: " << JoinPoint::signature() << endl;
5     ArgPrinter<JP::ARGS>::work (tjp);
6   }
7 };
```

Der Pointcut `execution("% ... Http%::%(...)")` dieses Aspekts trifft auf insgesamt 165 von 3099 instrumentierten *join points* zu. Er gibt alle Argumente der zu überwachenden Funktionen mit Hilfe des in Unterabschnitt 10.2.2 vorgestellten *template meta program* aus.

Das Laufzeitsystem wird über ein Unix-Signal über den neuen Aspekt informiert und der Programmierer kann nun direkt den Fluss den HTTP Protokolls verfolgen. Der Benutzer, der das Problem gemeldet hatte, meldet sich mit einer genauen Zeitangabe wieder, was hilft, die genaue Stelle des Problems im Programmfluss zu lokalisieren:

```
trace: void HttpRequest :: initHTTP(_method_t ,proto ...
Arg 1: 1
Arg 2: 1
Arg 3: /releases/edgy/ubuntu-6.10-dvd-i386.iso
trace: int HttpRequest :: parseHeader (const char *)
Arg 1: Range: bytes=17904205-
```

```
1   aspect CheckForBrokenWget {
2     advice "HttpRequest " : slice class {
3       bool _clBroken;
4     public:
5       bool clientIsBroken () const { return _clBroken; }
6       void clientIsBroken (const char *s) {
7         _clBroken = strstr(s, "Wget /1.10.2");
8       }
9     };
10
11    advice execution("% HttpRequest :: parseHeader ...") :
12    after() {
13      tjp->that()-> clientIsBroken (*tjp->arg <0 >());
14    }
15
16    advice execution("bool HttpStateData :: decide ...") :
17    after() {
18      HttpRequest *request = *tjp->arg <0 >();
19      if (request -> clientIsBroken ())
20        *tjp->result () = false;
21      }
22  };
```

Abbildung 53: Dynamischer Aspekt zur Problemvermeidung

```
User-Agent: Wget /1.10.2
Accept: */*
Host: cdimages.ubuntu.com
```

Diese Ausgabe verrät, dass der Benutzer das Programm `wget` benutzt, das einen *range request* durchführt, um die Übertragung einer teilweise geladenen Datei fortzuführen. Es stellt sich in diesem Szenario heraus, dass es sich um einem Mangel im Programm `wget` handelt[10].

10.3.2 Dynamisch geladene Fehlerbehebung

Nachdem das Problem lokalisiert werden konnte, wird ein Aspekt entwickelt, der das Problem behebt, ohne dass der Proxyserver neu gestartet werden muss. Das Listing ist in Abbildung 53 abgedruckt. Der erste Teil ist die Einfügung enines booleschen Wertes sowie von zwei Methoden, die in die Klasse `HttpRequest` eingefügt werden. Die erste der beiden Methoden ist eine Zugriffsfunktion für das eingefügte Attribut, während die zweite Methode zur Erkennung des *defekten* Browsers dient. Das Studium der Quellen und

[10] In der Tat funktioniert wget Version 1.10.2 einwandfrei. Hier wird ein hypothetisches Szenario konstruiert.

Aspektmodul	Textsegment	Datensegment	BSS-Segment	Summe
`HttpTracer`	110734	268	736	111738
`CheckForWget`	4459	276	68	4903

Tabelle 7: Codegrößen von dynamisch geladenen Aspekten

der Ausgaben unseres Programmverfolgungsaspekts aus dem vorigem Ausschnitt ergibt, dass die Methode `HttpRequest::parseHeader()` bei jedem Empfang einer HTTP Nachricht aufgerufen wird. Diese Methode wird mit Advice in Zeile 11 so instrumentiert, dass sie die zuvor eingefügte Methode `HttpRequest::clientIsBroken()` zur Überprüfung des Browsers aufruft. Später im Kontrollfluss von Squid wird entschieden, ob es sich um eine partielle Übertragung handelt, oder die Datei komplett übertragen wird. Dies geschieht im Advice aus Zeile 16, welcher das eingefügte Attribut überprüft und die Entscheidung entsprechend manipuliert. Dies vermeidet das Problem indem für diesen Klienten keine partiellen Übertragungen mehr durchgeführt werden. Nach einem Test mit einer Testinstanz von Squid, kann der dynamisch gewobene Aspekt eingesetzt werden. Während der ganzen Entwicklung musste das Produktionssystem kein einziges mal angehalten werden. Derselbe Aspekt kann in die nächste Version von Squid statisch eingewoben werden um bessere Performanz und kleinere Codegröße zu erreichen.

10.3.3 Codegröße und Performanz

Wie bereits erwähnt, steigt durch die Instrumentierung von 3099 statischen *join points* die Codegröße von Squid von 1,7 auf 2,2 MB. Die aktuelle Implementierung des statischen Webers läßt jedoch noch Raum für Verbesserung, so dass erwartet werden kann, dass die Codegröße von instrumentierte Programmen in Zukunft verbessert werden kann. Neben der Basisanwendung Squid, tragen auch die dynamisch gewobenen Aspekte zur Codegröße bei. Tabelle 7 gibt die Größen der Aspektmodule `HttpTracer` und `CheckForBrokenWget`.

Das `HttpTracer` Aspektmodul ist wesentlich größer als das Aspektmodul zur Fehlerumgehung. Der Grund dafür liegt darin, dass das ersetere Aspektmodul die Anwendung an 165 *join points* verändert. Dabei wird für jeden *join point* Programmcode erzeugt. Darüber hinaus enthält die Struktur mit statischer Kontextinformation die Funktionssignaturen als Zeichenkette, die im Textsegment abgelegt wird. Das Aspektmodul zur Fehlerumgehung hingegen hat eine vernachlässigbare Codegröße, da nur zwei *join points* manipuliert

Optimierungsziel	gewöhnliches Attribut	dyn. eingef. Attribut
`-O0`	5	34
`-O1`	4	10
`-O2`	3	6

Tabelle 8: Zugriffszeiten auf ein dynamisch eingefügtes Attribut

werden.

Der Aspekt zur Fehlervermeidung sieht aus wie ein gewöhnliches Programm in der Sprache AspectC++. Er macht eine dynamische Einfügung des Attributs `_clBroken`. Dies impliziert die Verwendung von etwas komplizierten Strukturen, die als *Introduction Pointer* in Unterabschnitt 6.1.3 vorgestellt wurden. Aus diesem Grund sind etwas höhere Zugriffszeiten für das dynamisch eingefügte Attribut zu erwarten. Tabelle 8 fasst die Ergebnisse der Messungen bei verschiedenen Optimierzielen zusammen.

Für die Messungen wurden die Attribute 32000 mal in einer Schleife verwendet, die Laufzeit mittels des Kommandos `RDTSC` erfasst, aufsummiert und das Ergebnis wieder durch 32000 dividiert. Die Werte zeigen dass geeignete Optimierung im Übersetzer signifikante Auswirkungen auf die Zugriffszeiten hat. Jedoch ist selbst mit der Übersetzeroption `-O2` bedingt durch die Indirektion mit doppelten Zugriffskosten zu rechnen. Diese zusätzlichen Kosten für den Zugriff auf das dynamisch eingefügte Attribut ist für diesen speziellen Aspekt sind jedoch im Gesamtkontext von Squid kaum messbar.

10.4 Zusammenfassung

Die Möglichkeiten, die sich durch zur Laufzeit gewobene Aspekte ergeben, sind enorm. Durch die Bereitstellung von statischen Kontextinformationen ist es möglich, beeindruckende *template meta* Programme zu entwickeln, die auch bei größeren Projekten in Sekunden neu übersetzt werden können. Der vorgestellte Aspekt zur Programmablaufverfolgung kann helfen wertvolle Einarbeitungszeit, die zum Verständnis vonfremder Software aufgebracht werden muss, einzusparen. Weitere Einsatzmöglichkeiten existieren für die Instrumentierung von langlaufen Prgrammen wie dem vorgestelltem Proxyserver Squid. Dynamisches Weben ermöglicht es, Fehler in Software zu beheben, ohne dass sie neu gestartet werden muß. Diese Technik ist in Sprachen mit dynamischer Typprüfung schon seit längerem möglich, im Umfeld von C++ Programmen jedoch bemerkenswert.

11 Zusammenfassung und Ausblick

Für viele Softwareprojekte ist `C++` die einzige in Frage kommende Programmiersprache. `C++` bietet alle objektorientierten Sprachmerkmale zur Dekomposition von funktionalen Belangen gepaart mit hoher Ausführungsgeschwindigkeit und geringem Betriebsmittelbedarf. Trotz der weiten Verbreitung gibt es eher wenige Projekte, die sich mit der Umsetzung von aspektorientierten Techniken im C++ Umfeld beschäftigen.

Im C++ Umfeld existieren derzeit vor allem Aspektweber, die Aspekte statisch in Programme weben. Dabei kann der Weber `ac++` v.a. mit der Implementierung der die Sprache `C++` erweiternden Sprache AspectC++ trumpfen, die wenig Einarbeitungszeit und gute Werkzeugunterstützung bietet. Durch aspektorientierte Programmierung können viele typische quer schneidende Probleme mit *Aspekten* elegant gelöst werden. Viele Softwareprojekte nutzen diese Möglichkeiten erfolgreich zur Verbesserung der statischen Konfigurierbarkeit.

Aspekte haben dabei immer einen *global* wirkenden Charakter. Dieser ist in dynamisch getypten Sprachen wesentlich einfacher umzusetzen, weil statisch getypte Sprachen wie `C++` die Informationen über Typen vollständig wegoptimieren. Ohne statische Typinformation sind jedoch nur triviale Aspektimplementierungen möglich. Dieser Sachverhalt macht die Umsetzung von Aspekten zur Laufzeit kompliziert. Der Lösungsansatz, statische Typinformation in eigenständige Typen im Basisprogramm zu kapseln, sollte eher gemieden werden, weil dadurch viele Optimierungen und damit die Vorteile von statisch getypten Sprachen zunichte gemacht werden. Da diese Art von Information nur für den Bau von Erweiterungsmodulen, aber nicht für den Betrieb benötigt werden, ist der Lösungsansatz daher der Weg über eine separate Projektdatenbank, die die benötigte Typinformation getrennt vom Basisprogramm vorhält.

Die Implementierung von dynamischem Weben in C++ ist in dieser Arbeit prototypisch durchgeführt. Eine der grundlegenden Entwurfsentscheidungen war, für sowohl statisches als auch dynamisches Weben *dieselbe* Sprache, AspectC++, für die Beschreibung von Aspekten zu nehmen. Dabei war zu untersuchen, inwieweit sich dabei die von der Sprache AspectC++ bereitgestellten Sprachmerkmale in zur Laufzeit nachgeladenen Aspektmodule umsetzen lassen. Das Experiment dieser Arbeit liegt also in der Umsetzung des *Single Language* Ansatzes.

Dabei stellt sich heraus, dass sich nicht alle Sprachmerkmale mit einem dynamischem Weber umsetzen lassen. Gerade Einfügungen machen durch ihre

Seiteneffekte die vollständige Umsetzung der Sprache AspectC++ im dynamischem Fall nicht praktikabel. Man stellt aber fest, dass für viele Anwendungsfälle eine vollständige Umsetzung auch gar nicht nötig ist. In den Unterabschnitten 10.2 und 10.3 wurden Aspekte entwickelt, die auch zur Laufzeit gewoben in realen Systemen sinnvoll eingesetzt werden können. Mit einer Teilmenge der von AspectC++ bereitgestellten Sprachmerkmale ist der *Single Language* Ansatz also durchaus sinnvoll durchführbar!

Die derzeit nicht unterstützten Sprachmerkmale sind im Einzelnen:

- Einfügungen mit Seiteneffekten auf Sprachebene (Unterabschnitt 6.7)
- Dynamisch evaluierbare Kontextfunktionen (Unterabschnitt 7.3)
- Ordnung von Advice (Unterabschnitt 7.4)
- Kontextvariablen (Unterabschnitt 7.5)

Für diese Sprachmerkmale wurden Lösungsansätze gegeben, die in nachfolgenden Arbeiten realisiert werden können. Dabei stellen vor allem die Einfügungen mit Seiteneffekten auf Sprachebene, sowie die Unterstützung der *pointcut*- Funktion `cflow()` besondere Herrausforderungen dar. Um diese auch im dynamischem Fall zu implementieren, sollte dazu in der Sprachdefinition der Sprache AspectC++ die genauen Auswirkungen dieser Sprachmerkmale überdacht werden, wenn sie in einer Anwendung zur Laufzeit angewendet werden.

Die Implementierung der Infrastruktur zum dynamischen Weben wurde dabei (teilweise) in der Sprache AspektC++ selbst realisiert. Dabei wird für die eigentliche Übersetzung von AspektC++ Programmen auf das bereits bestehende Werkzeug `ac++` zurück gegriffen. Dies wird durch die Erkenntnis ermöglicht, dass Advice für jeden *join point* einzeln instantiiert wird und diese Instantiierungen in Erweiterungsmodule ausgelagert werden können. An zusätzlichen Werkzeugen mussten also lediglich ein Generator für die Adviceinstanzen, der Postprozessor, ein Lader für Aspektmodule sowie eine Laufzeitumgebung entwickelt werden. Dabei teilt sich der Generator Programmtexte mit dem statischem Weber `ac++`. Es bietet sich daher an, die Quelltexte zu konsolidieren und die dynamische Weberinfrastruktur weiter in das AspektC++ Projekt zu integrieren.

Insgesamt hat dieses Experiment gezeigt, dass man mit Hilfe von dynamischem Weben von Aspekten spürbare Verbesserungen bei der Entwicklung

von Software erzielt werden kann. Aspektorientierte Programmierung ist also eine sinnvolle Ergänzung für die Wartung und Erweiterung bestehender Projekte. Interessant für den Benutzer der Sprache AspectC++ mag die Tatsache sein, dass auch große Anwendungen, wie das Framework ACE oder der Proxyserver Squid in der Version 3, erfolgreich mit Aspekten gewoben werden können. Man kann diese Arbeit daher auch als eine Art (weitere) Reifeprüfung von AspectC++ ansehen, die durchaus überzeugend gemeistert wurde. Dem Einsatz von aspektorientierten Techniken in weiteren Projekten sollte also nichts mehr im Wege stehen.

Literatur

[AE04] Sufyan Almajali and Tzilla Elrad. Dynamic Aspect Oriented C++ for Upgrading without Restarting. In *Proceedings of the Conference on Advances in Internet Technologies and Applications, with special emphasis on E-Education, E-Enterprise, E-Manufacturing, E-Mobility, and related issues*, Purdue, USA, July 2004.

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[ASA+95] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching proxies: limitations and potentials. In *Proceedings of the 4th International WWW Conference*, 1995.

[Beu03] Danilo Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.

[Bla05] Georg Blaschke. Eine skalierbare Speicher- und Objektverwaltung für eingebettete Systeme. Master's thesis, Friedrich Alexander Universität, Erlangen, 2005.

[BPKS04] G. Böckle, P.Knauber, K.Pohl, and K. Schmid. *Software-Produktlinien.* dpunkt.verlag, 2004.

[CBE+00] Constantinos A. Constantinides, Atef Bader, Tzilla H. Elrad, P. Netinant, and Mohamed E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys*, 32(1es):41, 2000.

[Che03] Yan Chen. Aspect-oriented programming (aop): Dynamic weaving for C++. Master's thesis, Vrije Universiteit Brussel and École des Mines de Nantes, August 2003.

[Cza04] Krzysztof Czarnecki. Overview of generative software development. In Jean-Pierre Banatre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Proceedings of the Unconventional Programming Paradigms International Workshop (UPP '04)*, volume 3566 of *Lecture Notes in Computer Science*, pages 326–341, Le Mont Saint Michel, France, September 2004. Springer-Verlag.

[DFL+05] R. Douence, T. Fritz, N. Loriant, J. M. Menaud, M. S. Devillechaise, and M. Suedholt. An expressive aspect language for system applications with Arachne. In Peri Tarr, editor, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 27–38, Chicago, Illinois, March 2005. ACM.

[FF00] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced SoC (OOPSLA '00)*, October 2000.

[GS05] Wasif Gilani and Olaf Spinczyk. Dynamic aspect weaver family for family-based adaptable systems. In *NetObjectDays (NODe '05)*, Lecture Notes in Informatics, pages 94–109, Erfurt, Germany, September 2005. German Society of Informatics.

[HMS98] James C. Hu, Sumedh Mungee, and Douglas C. Schmidt. Techniques for developing and measuring high performance web servers over high speed atm networks. In *Proceedings of the 17th IEEE Conference on Computer Communications (IEEE infocom '98)*, pages 1222–1231, San Francisco, USA, April 1998. IEEE Computer Society Press.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, June 2001.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[LBS04] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative*

Programming and Component Engineering (GPCE '04), volume 3286 of *Lecture Notes in Computer Science*, pages 55–74. Springer-Verlag, October 2004.

[PSCS01] Irfan Pyarali, Marina Spivak, Ron Cytron, and Douglas C. Schmidt. Evaluating and optimizing thread pool strategies for Real-Time CORBA. In *Proceedings of the 2001 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES '01)*, pages 214–222, Snow Bird, Utah, 2001. ACM.

[Sch98] Douglas C. Schmidt. Evaluating architectures for multi-threaded CORBA Object Request Brokers. *Communications of the ACM*, 41(10), 1998.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific '02)*, pages 53–60, Sydney, Australia, February 2002.

[SL06] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. In *Journal on Knowledge-Based Systems, Special Issue on Creative Software Design*. Elsevier North-Holland, Inc., 2006. (to appear).

[SLM98] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4), April 1998.

[SLU05] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP extension for C++. *Software Developers Journal*, (5):68–76, May 2005.

[SPLG+06] Wolfgang Schröder-Preikschat, Daniel Lohmann, Wasif Gilani, Fabian Scheler, and Olaf Spinczyk. Static and dynamic weaving in system software with AspectC++. In Yvonne Coady, Jeff Gray, and Raymond Klefstad, editors, *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS '06) - Mini-Track on Adaptive and Evolvable Software Systems*. IEEE Computer Society Press, 2006.

[SSRB00] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[Tar05] Reinhard Tartler. Analyse des Betriebsmittelbedarfs von Aspectc++ Programmen, 2005. Studienarbeit.