Bachelorarbeit

Informatik

Feature-orientierte Programmierung mit Superimposition in C#

Alexander von Rhein

Betreuer:
Dr. Sven Apel

Fakultät für Informatik und Mathematik
Universität Passau

SS 2008

Zusammenfassung
Feature-orientierte Softwareentwicklung hat zum Ziel Softwaresysteme in Module (Features) zu zerlegen. Aus diesen Features können nach Bedarf unterschiedliche Varianten des Softwaresystems generiert werden. In dieser Arbeit wird untersucht wie Feature-orientierte Komposition von Modulen auf Basis der Programmiersprache C# realisiert werden kann. Als spezielles Kompositionsverfahren wird Superimposition auf Grundlage einer Feature-Algebra verwendet.
Weiterhin werden die gewonnenen Ergebnisse genutzt um ein Tool zur Feature-Komposition zu erweitern. Das Tool stellt Feature-orientierte Programmierung für verschiedene Sprachen (Artefaktsprachen) zur Verfügung.

Gliederung

Verzeichnis der Abbildungen

Tabellenverzeichnis

Einleitung

Die wichtigsten Ziele des Software Engineering sind die Software-Qualität zu steigern, die Entwicklungskosten zu senken und die Wartbarkeit/ Erweiterbarkeit von Softwaresystemen zu verbessern. Eine wichtige Vorgehensweise um diese Ziele zu erreichen ist, Software in Modulen zu entwickeln (Dekomposition) und diese in verschiedenen Kombinationen zu fertigen Softwareprodukten zu komponieren. Module verringern die Komplexität der Codebasis und verbessern ihre Lesbarkeit.

Das Prinzip Softwareentwicklung durch Modularität zu vereinfachen wird seit langem in verschiedensten Programmierkonzepten eingesetzt. Beim Einsatz in der Objekt-Orientierung (OOP) ergeben sich jedoch folgende Probleme [TOH+99, KLM+97]:

- Artefakte[1] sind nur begrenzt wiederverwendbar bzw. erfordern Anpassungen im Zielprogramm.
- Die Lesbarkeit von Softwarelösungen kann durch komplexe Interaktionen zwischen Modulen erschwert werden.
- Wartungsarbeiten und Erweiterungen erfordern tiefgreifende Änderungen im gesamten Programm.
- Erweiterbarkeit wird erschwert, weil sich ein Belang auf viele Softwareartefakte/ Module verteilen kann („Separation of Concerns“[2]) und darum schwer zu erkennen ist.

Diese Probleme sind keineswegs spezifisch für OOP. Durch Objekt-Orientierung werden sie sogar teilweise vermieden. Das Prinzip der „Seperation of Concerns“ kann aber durch Objekt-orientierte Programmierung alleine nicht unterstützt werden. Die Feature-orientierte Programmierung bietet die Möglichkeit diesen Problemen zu begegnen und sie teilweise zu lösen. Da die Feature-orientierte Programmierung gut mit der Objekt-orientierten Programmierung harmonisiert, kann man die Stärken beider Ansätze gemeinsam nutzen.

In dieser Arbeit wird die Unterstützung von Feature-orientierter Programmierung durch Superimposition der Programmiersprache C# untersucht. Weiterhin wird anhand von Fallbeispielen geprüft ob Feature-Komposition in C# praktikabel ist beziehungsweise wo sich Probleme ergeben.

Als Grundlage für die Entwicklung der Kompositionssoftware wird der FST-Composer[3] verwendet. Dieses Tool wurde an der Universität Passau von Dr. Sven Apel entwickelt [AL08] und erlaubt es Softwarefragmente, die in Feature-Modulen vorliegen, zu komponieren. Im Zuge dieser Arbeit wird es um die neue Artefaktsprache C# erweitert.

Die Feature-Algebra [ALM+08] wird genutzt um eine theoretische Grundlage für diese Erweiterung zu schaffen. Diese Algebra wird im entsprechenden Grundlagenkapitel beschrieben. Auf Basis der Algebra wird formal untersucht wie C#-Programmkonstrukte in den FSTComposer zu integrieren sind.

Feature-orientierte Programmierung ermöglicht es eine Menge von verwandten Produkten zu entwickeln, die gemeinsame Komponenten verwenden. Die Unter-

[1] Verwendete Fachbegriffe werden in den betreffenden Grundlagenkapiteln definiert.

[2] Unter anderem in [TOH+99]

[3] http://www.infosun.fim.uni-passau.de/cl/staff/apel/FSTComposer/

schiede zwischen den Produkten werden in Features verwaltet. Die benötigten Features werden dann zum Beispiel vom FSTComposer zu einem Produkt komponiert. Durch dieses Prinzip wird die Codereplikation reduziert und die verschiedenen Varianten des Endprodukts nehmen nur soviel Speicherplatzbedarf in Anspruch wie die wirklich benötigten Features brauchen. Die verschiedenen Produkte einer Softwareproduktlinie können erstellt werden ohne dass Anpassungen im Quellcode nötig sind.

Eine Besonderheit des FSTComposer ist, dass er Kompositionsmechanismen für verschiedene Artefakttypen (z. B. XML und Java) in einem Tool vereint. Dadurch wird das Prinzip der Uniformität erfüllt, wonach jeder Typ von Softwareartefakt verfeinert werden kann (frei nach [BSR03]). Das Prinzip wird in dieser Arbeit weiter verfolgt, indem C# als weitere Artefaktsprache in den FSTComposer integriert wird.

Da Java und C# an einigen Stellen sehr ähnlich sind, werden alle nötigen Regeln für C# analog zu den in Java existierenden Regeln formuliert. Dadurch wird der Aufwand den Nutzer zur Einarbeitung in das Tool aufbringen müssen minimiert. Es gibt jedoch in C# viele Mechanismen die kein entsprechendes Pendant in Java haben. Da auch für diese Sprachkonstrukte Kompositionsregeln entwickelt werden, erhalten wir neue Erkenntnisse über die Anwendbarkeit von Feature-Orientierung im Allgemeinen (Diese Mechanismen sind auch für andere, künftige Sprachen anwendbar).

Während der Entwicklung dieser Arbeit liegt ein Fokus auf der syntaktischen Korrektheit des generierten Softwareprodukts. Das bedeutet, vorausgesetzt, dass die eingegebenen Features korrekt sind, so ist (mit wenigen später beschriebenen Ausnahmen) das Softwareprodukt korrekt.

Zur Evaluierung des erweiterten FSTComposer wird in dieser Arbeit eine Produktlinie, welche in Java vorliegt, nach C# portiert. Da dies einen bedeutenden Teil des Evaluierungsschrittes in Anspruch nimmt, wird sich ein Kapitel mit der Portierung der Produktlinie nach C# beschäftigen.

Die eigentliche Evaluierung des C#-Teils vom FSTComposer besteht dann darin, einige mögliche Produkte der C#-Produktlinie zu bilden und diese mit ihren Java-Äquivalenten zu vergleichen. Dabei sind besonders die Laufzeit, die Unterschiede in den Ausgaben (bzw. die Erklärung dieser) und strukturelle Unterschiede im generierten Sourcecode interessant.

Das Ziel, aus korrekten Features ein garantiert korrektes Softwareprodukt zu erzeugen, konnte weitgehend erreicht werden. Aber an manchen Punkten konnte dies nicht umgesetzt werden. Diese Punkte werden in der Arbeit diskutiert. Weiterhin wurde erkannt, dass die entwickelten Kompositionsalgorithmen nicht in ihrer vollen Mächtigkeit von den zur Evaluierung verwendeten Produktlinien genutzt wurden. Die praktische Verwendung des Tools wird zeigen ob einfachere Kompositionsmechanismen für die Programmierung ausreichen würden.

0.1 Struktur der Arbeit

Grundlagen

In diesem Kapitel werden die Grundlagen des Themengebietes erläutert. Diese Grundlagen umfassen:

- die Feature-orientierte Programmierung
- relevante Sprachkonstrukte der Programmiersprache C#

- das Programm FSTComposer sowie zugehörige Konzepte und Begriffe
- die Feature-Algebra

Außerdem werden verwandte Themengebiete kurz vorgestellt.

Theoretische Untersuchung von C# als Artefaktsprache

Im theoretischen Teil der Arbeit werden anhand der Feature-Algebra die möglichen Kompositionsmethoden für C# als Artefaktsprache entwickelt. Mit Hilfe der Feature-Algebra wird dann eine theoretische Grundlage für die Komposition von Features in der Artefaktsprache C# entwickelt. Zu diesem Zweck wird diskutiert, wie die relevanten Elemente der C# Syntax komponiert werden können.

Implementierung

In diesem Kapitel wird die Architektur des FSTComposer dargestellt. Auf dieser Grundlage werden zunächst die notwendigen Erweiterungen am Kernprogramm beschrieben. Weiterhin wird aufgrund der Erkenntnisse des theoretischen Teils eine Teilmenge der Sprache C# definiert, die vom FSTComposer erkannt wird und für die Feature-Komposition implementiert wird.

Evaluierung

Zur Evaluierung der Software werden drei Testproduktlinien verwendet.

Die erste Produktlinie enthält nur zwei Features, deren Elemente jeweils spezielle Sprachkonstrukte/ Kombinationen abdecken. Diese Produktlinie wurde während der Implementierung entwickelt um frühzeitig Fehler zu erkennen.

Als zweite Produktlinie wird die „Graph Product Line“ (GPL)[4] verwendet. Diese liegt bereits in Java vor und ist ein nicht-triviales Programm (~3000 LOC). Diese Produktlinie wurde nach C# portiert. Sie wird genutzt um zu zeigen, dass der C#-Teil des FSTComposer auf realistischen Projekten performant funktioniert.

Die dritte Produktlinie ist ein Webservice, der auch WSDL (Webservice Specification Definition Language) Dateien enthält. Mit diesem Beispiel wird gezeigt, dass der FSTComposer auch Features mit gemischten Artefakttypen (inkl. C#) komponieren kann.

1. Grundlagen

1.1 Feature-Orientierung

1.1.1 Begriffe

Feature

Features sind der zentrale Bestandteil der Feature-orientierten Programmierung. Die Bedeutung eines Features lässt sich am besten an einem Beispiel aus der

[4]Entwickelt in [LB01]

Automobilindustrie erklären. Im modernen Fahrzeugbau kann der Kunde sein Fahrzeug aus vielen Komponenten zusammenstellen. Dabei ist es möglich die Innenausstattung (Leder, Radio, Navigationssystem, u. v. m.), die äußere Form (Cabrio, Limousine) und Lackierung zu wählen. Diese Auswahlmöglichkeiten werden auch als Features des Fahrzeugs betrachtet.

Features stellen Gemeinsamkeiten und Unterschiede zwischen den Produkten dar. Sie entsprechen einer Anforderung, die vom Kunden an das Produkt gestellt wird und erlauben es das Produkt zu konfigurieren. Das bedeutet ein Autokäufer kann sich aus einer Anzahl von Features ein Auto zusammenstellen, das seinen individuellen Ansprüchen gerecht wird. Features werden dabei genutzt, um nicht den Überblick über die enorm große Anzahl von Kombinationsmöglichkeiten zu verlieren.

Der so beschriebene Feature-Begriff wird in der Softwareentwicklung bereits eingesetzt. Im IEEE Standard „IEEE Std 829-1983 [5]“ [IEEE94] wird der Begriff Feature definiert als ein differenzierendes Merkmal eines Software-Moduls (z. B. Performanz, Portabilität oder Funktionalität). Dabei werden Features meist in der Konzeption bzw. im Pflichtenheft festgelegt und dann in der Design- und Implementierungsphase umgesetzt. In der Feature-orientierten Programmierung werden Features von Anfang an als eigenständige Module konzipiert und implementiert. Diese Module können durchaus aus verschiedenen nicht zusammenhängenden Programmteilen bestehen. Sie sind jedoch konzeptionell zusammengehörig bzw. realisieren dasselbe Feature. Auf Abhängigkeiten, die zwischen Features entstehen können, wird im Kapitel zur Feature-orientierten Programmierung [2.1.2] eingegangen. Wie Features konkret repräsentiert werden, wird im Kapitel [2.4] beschrieben.

Die Entwicklung von Features wird durch so genanntes Domain Engineering [CE05] vereinfacht. Ein Feature befindet sich immer im Kontext einer Domäne. Eine Domäne ist eine Menge von gegenwärtigen und zukünftigen Anwendungen mit einer Menge von gemeinsamen Funktionen und Daten [KCH+90]. Im Domain Engineering werden Informationen über diese spezielle Domäne gesammelt und aufbereitet. Mit Feature-orientierter Domänen Analyse (FODA) können Features aus einer Domäne extrahiert werden. In der Studie [KCH+90] werden Features (in FODA) als Eigenschaften eines Systems, die direkt den Endnutzer betreffen definiert. Der Endnutzer muss dabei nicht unbedingt menschlich sein. Es kann auch ein weiteres System sein, dass mit Systemen in der Domäne interagiert.

Belang

Zusätzlich zum Featurebegriff wird noch der Begriff des *Belangs* definiert. Jede Problemstellung, die für einen Stakeholder des Softwaresystems (Kunde, Programmierer, Designer, ...) von Interesse ist, ist ein Belang. Diese Definition impliziert, dass jedes Feature ein Belang ist. Es gibt jedoch Belange die keine Features darstellen. Die Menge der Features ist also eine Teilmenge der Menge der Belange.

Was genau als Feature oder Belang betrachtet wird hängt von der jeweiligen Problemdomäne ab. Bei einer sehr zeitkritischen Software könnte zum Beispiel die Verwendung eines besonders performanten Algorithmuses ein Feature darstellen. Falls die Software aber keine (problematischen) Zeitrestriktionen erfüllen muss, ist die Entscheidung, welcher Algorithmus verwendet wird, ein Belang. Der

Belang betrifft in diesem Fall nur den konkreten Programmierer / das Programmierteam.

Der Belang erleichtert es zu bestimmen für welche Programmteile es sich lohnt sie als separates Feature zu implementieren. Im Allgemeinen sollten nur Programmteile, die nach obiger Definition ein Feature sind, in separate Feature-Module gekapselt werden. Falls der Programmteil nur ein Belang ist lohnt sich dieser Aufwand nicht. An dieser Stelle muss man berücksichtigen, dass jedes zusätzliche Feature-Modul die Architektur komplexer macht.

Produktlinie

Als letzte Definition wird die *Produktlinie* eingebracht. Eine Software-Produktlinie ist eine Menge von Softwaresystemen, die eine gemeinsame, verwaltete Menge von Features haben, die die spezifischen Bedürfnisse eines bestimmten Marktsegments oder einer Aufgabe erfüllen und die von einer gemeinsamen Menge von Grundmodulen in einer vorgeschriebenen Weise entwickelt werden [CN07].

Als anschauliches Beispiel wird an dieser Stelle wieder der Fahrzeugbau herangezogen. Eine Produktlinie ist in diesem Beispiel eine Limousine eines bestimmten Autoherstellers, die in verschiedenen Ausführungen bestellt werden kann. Die Limousine als abstraktes Konzept, mit der Angabe welche Konfigurationsentscheidungen (d. h. Features) nötig sind um eine fertiges Auto festzulegen, wird als Produktlinie bezeichnet.

Auf Software übertragen ist eine Produktlinie eine Menge von Programmen, die auf ein gemeinsames Marktsegment zugeschnitten sind und mit dem Ziel der Wiederverwendung von gemeinsamen Softwareartefakten entwickelt wurden [MPP]. Ein Beispiel für bekannte Software-Produktlinien sind Office-Suiten, die mehrere Aufgaben wie Editieren von Textdokumente, Tabellenkalkulation und Präsentationsentwicklung unterstützen. Diese Produktlinien basieren auf einem gemeinsamen Kern und können je nach Wunsch des Benutzers individuell zusammengestellt werden.

1.1.2 Feature-orientierte Programmierung

Die Feature-orientierte Programmierung stellt eine relativ neue Technik zur Softwareentwicklung dar. In dieser werden die (abstrakten) Features in *Featuremodulen* implementiert. Diese Featuremodule werden dann mit Tool- und Sprachunterstützung zu einem Softwareprodukt zusammengefügt.

Features kapseln bestimmte Teile des Funktionsumfangs der Software. Der Nutzer kann entscheiden, ob er diese Features nutzen möchte oder eine schlankere, schnellere Software bevorzugt. Die vom Benutzer gewählte Menge von Features wird im Folgenden als *Feature-Auswahl* bezeichnet.

Das Konzept ist verwandt mit der Vererbung in Objekt-orientierten Sprachen. Durch Vererbung wird die Flexibilität von Software stark erhöht. Es ergeben sich jedoch auch Probleme, die im Folgenden angesprochen werden, um die Feature-Orientierung als Lösungsansatz vorzustellen.

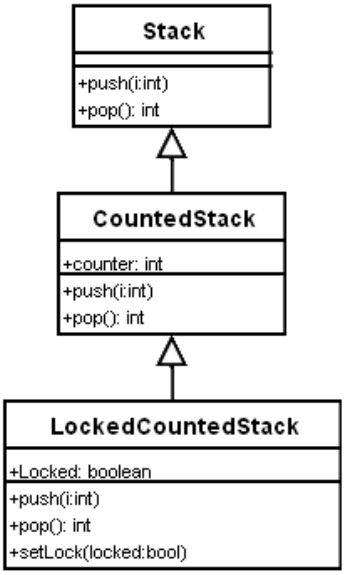
Stellen wir uns einen Stack[5] vor, der die folgenden Features enthält[6]:

[5]Zur Vereinfachung kann der Stack nur Integer speichern.

[6]Das Beispiel wurde aus [PRE97] übernommen.

- **Stack**, stellt `push(..)` und `pop()` zur Verfügung. Dies ist das „Basis-Feature“, das in jeder Feature-Auswahl enthalten sein muss.
- **Counter**, zählt wie viele Elemente der Stack hat
- **Lock**, ermöglicht es den Stack für die Bearbeitung zu sperren
- **Undo**, implementiert eine Funktion die den Zustand des Stacks vor der letzten Modifikation wiederherstellt

Um diese Features durch Vererbung zu implementieren, könnte man eine Klasse CountedStack definieren, die von der Stack-Klasse erbt [Idee aus PRE97]. Mit den anderen Features würde man entsprechend verfahren. In der folgenden Abbildung wird dieser Ansatz gezeigt.




```
1  class Stack {
2      public virtual void push(int i) {/*...*/}
3      public virtual int pop() {/*...*/}
4  }
5  class CountedStack : Stack {
6      public int counter;
7      public override void push(int i) {
8          base(i);
9          counter++;
10     }
11     public override int pop() {
12         counter--;
13         return base();
14     }
15 }
16 class LockedCountedStack : CountedStack {
17     bool locked;
18     public override void push(int i) {
19         if (!locked)
20             base(i);
21     }
22     public override int pop() {
23         if (locked) throw new Exception();
24         return base();
25     }
26     public void setlock(bool locked) {
27         this.locked = locked;
28     }
29 }
```

Abb. 1: Stack-Implementierung in C# durch Vererbung

Weil man Features variabel komponieren möchte (z. B. einen Stack mit Lock und Undo, einen Stack mit Lock, ...), ist die Vererbungshierarchie zu unflexibel. Wenn man die komplette Hierarchie (fünf Klassen die auf einander aufbauen) implementiert hat, kann man nicht einfach eine Klasse aus der Hierarchie entfernen, falls das Feature nicht mehr benötigt wird.

Man könnte alle Feature-Kombinationen separat implementieren, was aber sehr aufwendig wäre. Die Zahl der möglichen Kombinationen wächst exponentiell mit der Zahl der möglichen Features. Für k Features sind $2^{(k-1)}$ Kombinationen möglich. Durch Ausnutzung von Abhängigkeiten zwischen Features und Elimination von sinnlosen Kombinationsmöglichkeiten wird diese Zahl etwas gemindert, aber für Produktlinien mit 100 oder mehr Features ist sie immer noch extrem groß. Für das Stack Beispiel sind maximal $2^{(4-1)}=8$ Kombinationen möglich. Es sind aber noch viele weitere Features denkbar (zum Beispiel eine `top`-Funktion).

In der Objekt-orientierten Programmierung müssen die meisten dieser Kombinationen separat implementiert und getestet werden. Das wären im schlimmsten Fall $2^{(k-1)}$ separate Implementierungen.

In der Feature-Orientierung wird ein Feature wie in dem Beispiel als Modul betrachtet. Diese Module können fast nach Belieben komponiert werden um ein fertiges Programm zu erhalten. Dieses fertige Programm wird im Folgenden *Produkt* genannt. Ein Produkt ist also das (lauffähige) Ergebnis der Komposition von Features. Im oben beschriebenen Beispiel muss für jedes Feature ein Featuremodul, das genau die beschriebene Funktionalität enthält, implementiert werden. Das sind k Featuremodule. Im Vergleich zur klassischen Implementierung (exponentieller Aufwand) hat man so nur linearen Aufwand. Falls man sicher ist, dass nur eine bestimmte Feature-Auswahl benötigt wird lohnt es sich natürlich nicht den zusätzlichen Aufwand zur Dekomposition in Features auf sich zu nehmen. Der so genannte *Break-even-Point* gibt an ab welcher Anzahl von verschiedenen Produkten in einer Produktlinie es sich lohnt mit Features zu arbeiten [CE05 Kapitel 1.3].

Es kann auch Interaktionen zwischen Features geben. So wird zum Beispiel das Feature `Undo`, falls das Feature `Counter` genutzt wird, den Zustand des Counters mit wiederherstellen/ sichern. Es muss also zusätzlicher Code existieren, der in keinem dieser beiden Features enthalten ist (beide Features müssen auch unabhängig funktionieren). Ein Modell für die graphische Darstellung der Beziehungen zwischen Features wird in [CE05 Kapitel 4] vorgestellt.

Durch das Feature-orientierte Konzept können außer Programmfragmenten auch andere Typen von Dokumenten komponiert werden. Die Typen die verwendet werden können werden im Folgenden als *Artefakttypen* bezeichnet. Ein *Artefakt* kann zum Beispiel eine normale Textdatei oder ein XML-Dokument sein. Für alle Artefakttypen muss festgelegt werden wie sie im Einzelnen komponiert werden. Durch diesen Ansatz können auch Dokumentationen, Grammatiken und weitere Artefakttypen an die gewählte Feature-Komposition angepasst werden. Das Kompositionstool („composer Tool“, in unserem Fall der FSTComposer) koordiniert dabei die Komposition der Artefakte und ruft die Artefakttyp-spezifischen Kompositionsalgorithmen auf.

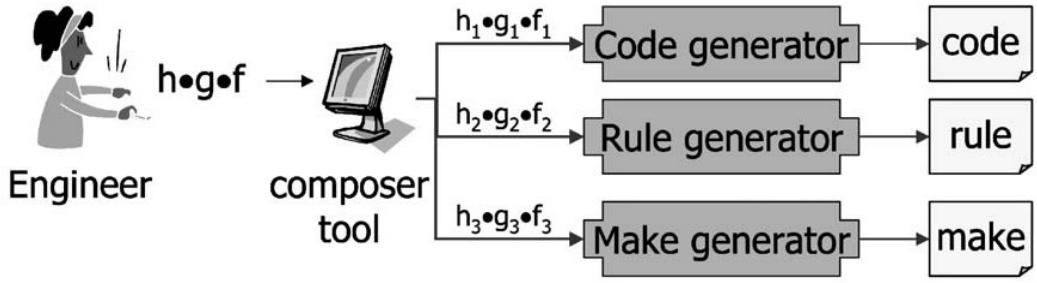


Abb. 2: Kombination von verschiedenen Artefakttypen

Quelle: [BSR03]

Um diese Spezifikation und den Dialog über die Feature-Orientierung im Allgemeinen zu vereinfachen wurde in [ALM+08] eine Feature-Algebra entwickelt. Diese Algebra wird in [2.4] vorgestellt und dann zur Spezifikation des C#-Artefakttyps verwendet.

1.2 Einführung in die Sprache C#

C# ist eine universale, typsichere und Objekt-orientierte Programmiersprache [AA07]. Sie arbeitet wie Java eng mit einem Framework, in diesem Fall der Common Language Runtime[7], zusammen.

C# wurde unter Anderem entwickelt um C und C++ Entwicklern den Umstieg auf diese neue Programmiersprache zu erleichtern [GUN00]. Darum wurden viele sehr mächtige und hardwarenahe Programmierkonzepte, die aus C++ bekannt sind, in C# übernommen. Es ist jedoch ebenso möglich auf einer höheren Abstraktionsebene, wie zum Beispiel in Java, zu programmieren.

C# bietet viele Konzepte, die von einer modernen Objekt-orientierten Programmiersprache zu erwarten sind. Dazu gehören unter anderem:

- automatische Speicherverwaltung (Garbage-Collection)
- Ausnahmebehandlung (Exception-Handling)
- Typsicherheit
- Attribute/ Annotationen

Auf einige dieser Konzepte wird im Folgenden noch weiter eingegangen. Außerdem werden einige Konstrukte vorgestellt, die im Weiteren Verlauf der Arbeit wichtig sein werden. Auf Attribute wird im Kapitel zu verwandten Themengebieten [2.5.2] eingegangen. Dort wird ein Ansatz vorgestellt, der mit Attributen ein ähnliches Ziel verfolgt wie diese Arbeit.

Die Spezifikation der Sprache C# wurde von der ECMA (European association for standardizing information and communication systems) als ECMA-334 [ECMA] veröffentlicht. Dieses Dokument ist im Internet verfügbar. In der ECMA-334 wird die Version 3.0 der Programmiersprache definiert. Zum Zeitpunkt der Erstellung des praktischen Teils dieser Arbeit stand die neueste Version der Programmiersprache (Microsoft Visual C# 2008) leider noch nicht zur Verfügung. Aus diesem Grund beschränkt sich diese Arbeit auf die in der ECMA-334 definierte Version.

Wie oben erwähnt stellt C# viele Konzepte zur Verfügung, die es sehr mächtig machen. Dies bedeutet aber auch, dass die Syntax umfangreicher wird als zum Beispiel bei Java und das der Aufwand sich in diese Konzepte einzuarbeiten steigt. Es werden nun einige Konzepte vorgestellt, in denen sich C# von Java und von C++ unterscheidet. Einige, ebenfalls wichtige, C#-Mechanismen werden an dieser Stelle nicht beschrieben. Hierzu wird auf die Literatur zu C# [AA07, GUN00] verwiesen.

Alle C#-Beispiele in dieser Arbeit wurden mit VisualStudio 2005 von Microsoft entwickelt und getestet. Ebenso wurden die Programme, die zur Evaluierung genutzt wurden, in VisualStudio entwickelt und getestet.

1.2.1 Unterschiede zu Java

Einheitliches Typsystem

In Java gibt es zwei verschiedene Arten von Typen, primitive und referentielle Typen. Diese Typen unterscheiden sich in der Effizienz (primitive Typen sind meist effizienter) und in der Verwendung (primitive Typen können zum Beispiel nicht als Typparameter verwendet werden). Um einen primitiven Typen in einen

[7]Framework der Microsoft-Implementierung von C#

Stack einzufügen muss er immer in den entsprechenden Wrappertyp[8] umgewandelt werden.

Diese Umwandlung entfällt bei C#, weil wirklich alle Typen vom C#-Typ `object` abgeleitet werden [AA07]. Dadurch kann man alle Typen direkt (ohne Wrapper) in einen Stack einfügen.

Erweiterte Methodenkonzepte

In C# werden einige Konzepte eingeführt, die das Entwickeln von Anwendungen beschleunigen und den Code übersichtlicher machen können. Diese Ansätze erweitern den Begriff der Methode aus der reinen Objekt-Orientierung. Als Beispiel hierfür werden „Properties“ vorgestellt. Eine Property ist ein Klassenelement, das wie ein Feld einen Teil der Zustandsmenge der Klasse kapselt. Die Property enthält aber auch Zugriffsfunktionen für diesen Teil der Zustandsmenge. In Java würde man hierfür „Getter“ und „Setter“ schreiben die den Zugriff auf das private Feld verwalten.

Der Mechanismus in C# wird durch das folgende Beispiel verdeutlicht. Die Klasse CountableAccess protokolliert die Lesezugriffe auf eine Information.

```
class CountableAccess {
    public int accessCounter;
    private String innerInformation;
    public String information {
        get {
            accessCounter++;
            return information;
        }
        set {
            information = value;
        }
    }
}
```

Abb. 3: Property Beispiel

Die von der Property verwaltete Information wird in der privaten Variablen `inner Information` gespeichert. Beim Lesezugriff durch die get-Funktion der Property wird der accessCounter erhöht. Das Feld `value` in der set-Methode wird automatisch von C# auf den übergebenen Wert gesetzt. Falls man die set-Methode nicht implementiert wird die Property als schreibgeschützt behandelt.

Bis zu diesem Punkt ist der einzige Vorteil gegenüber der Getter/Setter Struktur in Java, dass die Methoden zusätzlich gekapselt sind. Der eigentliche Vorteil liegt aber in der Verwendung der Properties. Sie sind von jeder Aufrufstelle, die nicht in der Klasse liegt, vollkommen transparent. Das bedeutet an der Aufrufstelle kann nicht unterschieden werden, ob `CountableAccess.information` ein Feld oder eine Property ist.

Daraus ergeben sich auch Vorteile in der Erweiterbarkeit von Programmen. So kann man Felder zuerst als einfaches Feld implementieren und später, wenn Zugriffsfunktionen nötig sind, eine Property schreiben. Jeder Code, der auf das (ehe-

[8]Ein Typ, der nur eine Variable des jeweiligen primitven Typs enthält. Er kapselt den primitiven Typ.

malige) Feld zugreift, ist per Definition immer noch korrekt solange man die Zugriffsmodifier nicht einschränkt.

Partielle Klassen

In C# ist es möglich Klassen mit dem Schlüsselwort `partial` zu definieren. In diesem Fall werden alle Klassendefinitionen mit demselben Namen, die auf einer Strukturebene existieren, zu einer Klasse zusammengefasst. Dieser Mechanismus wird genutzt um zum Beispiel Code, der von Code-Generatoren wie Visual-Studio-Assistenten generiert wurde, von normalem Code zu unterscheiden. Der Entwickler schreibt also seinen Code in einen Bereich, der visuell von dem generierten Code getrennt ist. Dadurch werden Probleme bei der wiederholten Generierung vermieden.

Dieser Ansatz könnte auch genutzt werden um einfache Features zu komponieren. Dazu könnte man einfach alle Klassen als „partial“ definieren und in eine Datei kopieren. Der Compiler fügt dann die verteilen Deklarationen zusammen.

Das funktioniert leider nur bei sehr einfachen Beispielen, weil jedes Programmfragment (Feld, Methode,...) nur einmal in allen partiellen Klassen definiert sein darf. Es könnten also keine Features, die gleiche Elemente enthalten, komponiert werden. Man erreicht also mit diesem Ansatz keine ausreichende Mächtigkeit um mit der Komposition durch Superimposition konkurrieren zu können. Eine detailliertere Diskussion der partiellen Klassen findet sich im Kapitel [3.2.1].

Delegates

Mit Delegates kann man in C# Funktionen als Felder verwenden. Damit sind sie gewissermaßen ein Ersatz für Funktionszeiger in C++. Ein Delegate definiert man mit seinem Rückgabetyp und der Liste seiner Parametertypen. Dann kann man dem Delegate eine existierende Funktion mit der passenden Signatur zuweisen. Delegates können auch als Parameter übergeben werden.

```
1  delegate int Transformer (int i);
2  class Test {
3      static void Main() {
4          Transformer t = Square;
5          int result = t(3);
6          Console.WriteLine(result);
7      }
8      static int Square(int x) { return x * x; }
9  }
```

Abb. 4: Delegate Verwendung

Aus C# 3.0 Seite 105

Dies ist ein kleines Beispiel, um die Syntax des Delegates zu veranschaulichen. Als umfangreicheres Beispiel wurde eine generische Implementierung einer Reduktion entwickelt und in den Anhang (Kapitel [8]) eingefügt. In diesem Beispiel wird auch die Benutzung der Lambda-Funktion gezeigt (siehe unten).

In C# 3.5 können Delegates mittels Lambda-Funktionen belegt werden. So spart man sich die Definition einer Methode, die man ohnehin nur gebraucht hätte

um sie dem Delegate zuzuweisen. Dieser Mechanismus (Lambda-Funktion) kann in dieser Arbeit aber leider nicht berücksichtigt werden[9].

Der Delegate-Mechanismus erfüllt die Bedingung für Funktions-Kontravarianz nach dem Cardelli-Typsystem[10]. Das bedeutet der Rückgabetyp der Funktion die dem Delegate zugewiesen wird, darf einen spezielleren Typ haben als der des Delegate (Rückgabewert-Kovarianz). Bei den Parametertypen verhält es sich genau umgekehrt. Durch diesen Umstand gewinnen die Delegates an Mächtigkeit ohne dafür (Typ-) Sicherheit zu opfern.

Präprozessorausdrücke

C# hat keinen echten Präprozessor wie er in C++ benutzt werden kann. Einige Präprozessordirektiven wie „#include“ oder Textersetzung durch „#define“ werden nicht unterstützt. Durch diese Änderung konnte eine einfachere Kompilierungsstruktur erzielt werden und die Kompilierungsgeschwindigkeit erheblich gesteigert werden [GUN00 Kapitel 27.2]. Es werden aber weiterhin die bekannten Direktiven wie „#if“, „#elif“ und „#endif“ unterstützt. Dazu kommen Direktiven die Ausgaben des Compilers steuern können.

Es wird hier nicht näher auf die Präprozessordirektiven eingegangen, weil sie aufgrund von grammatikalischen Problemen in dieser Arbeit nicht behandeln werden können. Für weitere Informationen wird auf das Buch „C#“ von Eric Gunnerson [GUN00] und die C# Spezifikation [ECMA] verwiesen.

1.2.2 Unterschiede zu C++

C# baut auf C++ auf um Entwicklern, die in C++ erfahren sind, den Umstieg zu erleichtern. Es besteht die Möglichkeit weiterhin auf Bibliotheken (COM oder DLL) zuzugreifen. C# hat jedoch nicht alle Mechanismen der Sprache C++ übernommen. Zum Beispiel gibt es keine Header-Dateien mehr. Die in C# enthaltenen Objekte sind vollständig selbstbeschreibend und können ohne Registrierung eingesetzt werden [GUN00].

Zu den wichtigen Unterschieden gehören sicher auch die Einführung von automatischer Speicherverwaltung und Ausnahmebehandlung. Für zeitkritische Programmteile stellt C# die Möglichkeit zur Verfügung mit dem `unsafe`-Keyword in einem C++-ähnlicheren Stil zu programmieren. Dann kann man auch Funktionszeiger verwenden.

Eine Eigenschaft die in C# aber strukturell ausgeschlossen ist, ist die Mehrfachvererbung von Klassen.

1.3 Feature-Structure-Trees und der FSTComposer

Ein Ansatz um Softwarefragmente zu komponieren ist die *Superimposition*. Bei Komposition durch Superimposition werden Artefakte komponiert, indem ihre Strukturen rekursiv komponiert werden. Dabei wird normalerweise der Name einer (Unter-) Struktur als identifizierendes Kriterium betrachtet. Es werden also z. B. zwei Klassen aus zwei verschiedenen Features komponiert, wenn sie denselben Namen (z. B. „Stack“) haben. Das Ergebnis dieses Kompositionsprozesses wird dann wieder den Namen „Stack“ haben. Aus der Komposition von zwei Fea-

[9]Lamda-Funktionen wurden in der ECMA-334 noch nicht eingeführt.

[10]Typsystem in [CW85], Funktions-Kontravarianz beschrieben in [OOP Kapitel 22]

tures entsteht dabei, wie von der Feature-Orientierung vorgesehen, wieder ein Feature.

Im Folgenden werden bei der Behandlung von diesen Kompositionsprozessen beide Komponenten als *Operanden* der Komposition und das komponierte Feature als *Ergebnis* bezeichnen.

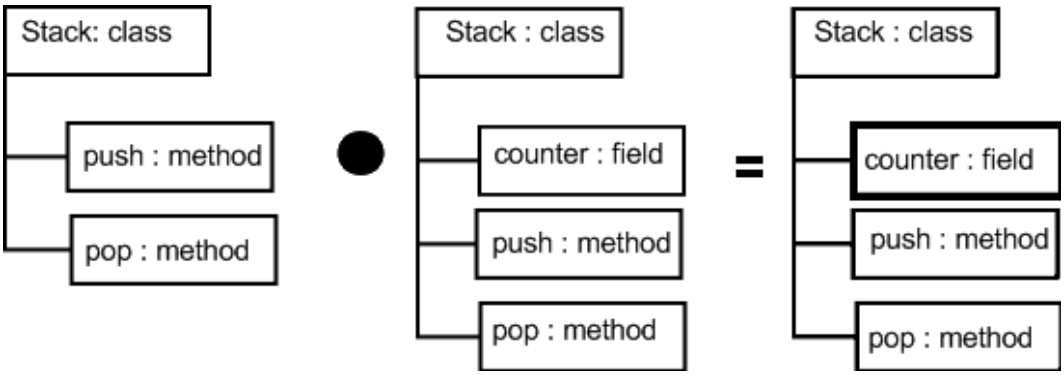


Abb. 5: Stack Komposition

Der FSTComposer beschränkt sich ganz allgemein darauf den Programmcode der zu bearbeitenden Features zu ändern. Dadurch kann der FSTComposer zum Beispiel keine Auswertungen des Programmflusses zur Laufzeit vornehmen (vgl. Aspekt-Orientierung [2.5.1]).
Für die Komposition von Artefakttypen gibt es viele verschiedene Tools, die auf der Superimposition aufbauen [AL08]. Diese Tools können jeweils nur wenige Artefakttypen komponieren und sind darum sehr spezialisiert. Das ist nicht besonders flexibel, weil der Entwickler zwischen Tools wechseln muss oder sogar Artefakte selbst zusammenstellen muss. Der FSTComposer versucht dieses Problem zu beheben indem er die Behandlung von vielen verschiedenen Artefakttypen mit einem einzigen Tool ermöglicht. Der genaue Mechanismus wird im Kapitel zur Erweiterung des FSTComposer um die Artefaktsprache C# (Kapitel [3]) erläutert.

Bei der Komposition von Features baut der FSTComposer auf so genannte Feature-Structure-Trees (FSTs) auf. Ein Feature-Structure-Tree repräsentiert die Struktur des Features. Im folgenden Kapitel wird erläutert was ein Feature-Structure-Tree ist. Danach kann genauer auf die Komposition von FSTs eingegangen werden.

1.3.1 Feature-Structure-Tree

Ein *Feature-Structure-Tree* (FST) ist eine Darstellung der inneren Struktur eines Features. Informationen, die angeben wie die FST-Knoten sprach-spezifisch notiert werden, sind nicht im FST gespeichert. Der FST enthält auch nur bis zu einem gewissen Grad Implementierungsdetails des Features. Das unterscheidet einen FST von einem abstrakten Syntax-Baum (abstract syntax tree, AST). Der Unterschied wird an einem Beispiel verdeutlicht. Das Beispiel implementiert das Feature Stack, das in Kapitel [2.1.2] vorgestellt wurde. Die Ordnung des FST wird durch Einrückung dargestellt.

```
1  using System.Collections;
2  class Stack {
3      ArrayList data = new ArrayList();
4      public object pop() {
5          object ret = data[data.Count];
6          data.RemoveAt(data.Count);
7          return ret;
8      }
9      public void push(object o) {
10         data.Add(o);
11     }
12 }
```

Abb. 6: Feature Stack, Sourcecode in C#

```
CompilationUnit
  UsingList
    Using
  Class
    Field
      FieldType
      Initializer
    Method
      ModifierList
      ReturnType
      InputParameterList
      MethodBody
    Method
      ModifierList
      ReturnType
      InputParameterList
        InputParameterName
          InputParameterType
      MethodBody
```

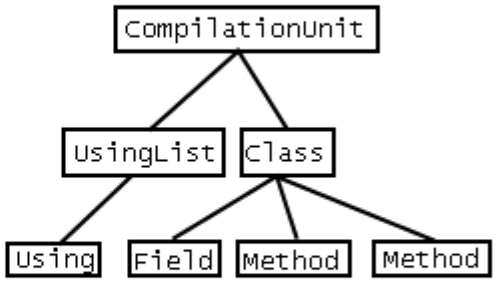


Abb. 7: Feature Stack, FST

Den AST konnte aus Platzgründen nicht einfügen werden, weil der AST bzw. der CST („concrete syntax tree“), den unser C#-Parser für dieses Code-Fragment erzeugt, über 400 Knoten hat. Auf diesen Parser wird in Kapitel [4] genauer eingegangen.

Jeder FST-Knoten hat einen Typen, der von einem gemeinsamen FST-Knoten-Typ abgeleitet wird. Diese Typen bestimmen welche Bedeutung der Knoten in der Featurestruktur hat und welche Unterknoten (-Typen) er haben kann. Im FST wird wie man sieht die Implementierung des Methodenkörpers nicht im Detail analysiert. Der gesamte Methodenkörper wird als String übernommen und als Name des Knotens „MethodBody“ gespeichert. Außerdem werden Identifier, die aus mehreren Elementen (durch „.“ getrennt) bestehen, nicht getrennt. Da sich die größte Komplexität eines AST im Rumpf der Methoden befindet, kann durch diesen Ansatz eine starke Vereinfachung des Baumes erreicht werden. Die genaue Kenntnis des Methodenrumpfes ist nicht nötig. Bei einem Identifier ist die detaillierte Auflösung nicht nötig, weil in unserem Ansatz den Elementen eines Identifiers ohnehin keine spezielle Bedeutung zukommen würde.

Das sind die wesentlichen Unterschiede zum AST. Ein AST speichert alle syntaktischen Informationen, die man aus dem Code ziehen kann. Der FST speichert

nur die Informationen, die für die Feature-Komposition nötig sind. Alle weiteren Informationen (z. B. Methoderümpfe) werden als Strings in den Terminalknoten gespeichert. Wie die Komposition abläuft wird im nächsten Kapitel erläutert.

Komposition von FSTs

Die Komposition durch Superimposition läuft im FSTComposer in vier Schritten ab.

- Einlesen der Feature-Auswahl
- Einlesen aller Dateien der Features als FSTs
- Komposition der FSTs durch Superimposition
- Schreiben der komponierten FSTs in das Ausgabeverzeichnis

Zum Ausführen der Komposition sind also zwei wesentliche Informationen notwendig. Es muss bekannt sein wo die Features gespeichert sind und welche Features in dem aktuellen Produkt enthalten sein sollen.

Die Auswahl, welche Features zu dem Produkt gehören, wird in einer Datei gespeichert. Diese Implementierung der abstrakten Feature-Auswahl wird im Folgenden als *Feature-Auswahl* bezeichnet. Der FSTComposer akzeptiert für diesen Zweck Dateien mit der Endung „.expression“. Die Dateien enthalten die Namen der Features mit einem Leerzeichen als Trennzeichen. Dabei ist die Reihenfolge der Features relevant. Der Grund dafür wird im Kapitel zur Feature-Algebra diskutiert [2.4].

Da unterschiedliche Artefakttypen behandelt werden können und nicht alle Elemente eines Features in einer Datei gespeichert werden können, wird ein Feature auf Dateiebene durch einen Ordner dargestellt.

Die Komposition wird rekursiv auf den FSTs ausgeführt. Die Entscheidung, welche Unterbäume komponiert werden, wird durch den Vergleich ihrer Namen und Typen getroffen (vgl. Abb.3 Stack Komposition). Wie die Typen im Detail komponiert werden, muss für jeden Knotentyp separat definiert werden. Diese Definitionen werden für die C#-Knotentypen im Kapitel [3.2] diskutiert.

1.4 Feature-Algebra

In den vorangegangenen Kapiteln wurde erläutert was ein Feature ist und wie Features prinzipiell komponiert werden. Für die theoretische Behandlung dieses Kompositionsmechanismus wird eine formale Grundlage benötigt, die es erlaubt die Interaktionen von Features auszudrücken. Diese Grundlage ist die Feature-Algebra [ALM+08], die im Folgenden vorgestellt wird. Sie baut auf der Komposition durch Superimposition auf, wie beschrieben im Grundlagenkapitel zu FST und FSTComposer.

Die Feature-Algebra definiert drei Operatoren, die zur Featurekomposition genutzt werden können. Diese Operatoren werden in den nächsten Kapiteln beschrieben. Zunächst wird die Algebra geschlossen vorgestellt.

$$
\begin{aligned}
FA ::= Grundmenge &: Features = Introductions \cup Modifications \\
Operationen &: \bullet : Features \times Features \rightarrow Features \\
&\quad \oplus : Introductions \times Introductions \rightarrow Introductions \\
&\quad \odot : Modifications \times Introductions \rightarrow Introductions \\
&\quad \odot : Modifications \times Modifications \rightarrow Modifications
\end{aligned}
$$

Die Introduktionssumme ($\oplus$), die Modifikationskomposition ($\odot$) und die Modifikationsanwendung ($\odot$) sind jeweils Einschränkungen des Kompositionsoperators ($\bullet$) auf eine bestimmte Grundmenge.

1.4.1 Komposition

Der Kompositionsoperator wird definiert als

$\bullet : F \times F \to F$

wobei F die Menge der Features ist.

In der folgenden Abbildung entspricht jeder Baumknoten einer Introduktion. Der gesamte Baum entspricht einem Feature. Falls ein Knoten Unterknoten enthalten kann, ist er ein *Nichtterminal*. Knoten die keine Unterknoten haben können bezeichnet man als *Terminalknoten*.

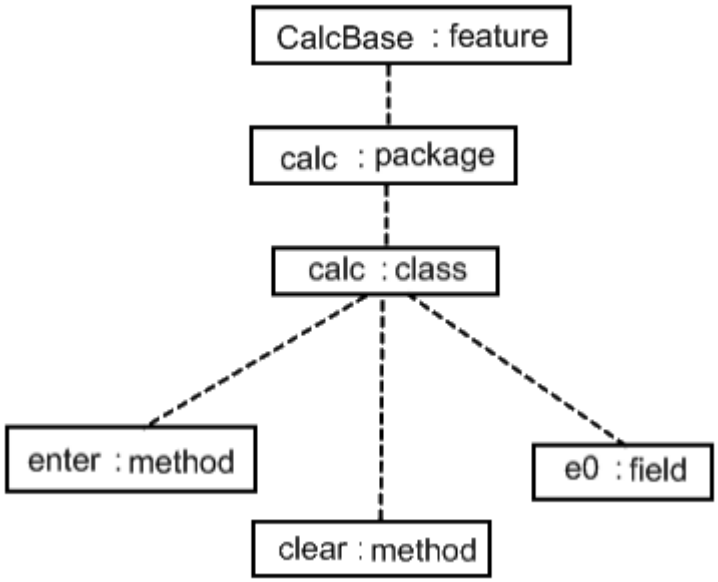


Abb. 8: FST des Features CalcBase

Quelle: frei nach Apel/ Feature-Algebra

In diesem Beispiel sind die Knotentypen „feature“, „package“ und „class“ Nichtterminale. Die Unterbäume der Nichtterminale bestimmen die Struktur des Artefakts. Weil diese Struktur durch den FST sichtbar gemacht wird, kann sie auch durch den FSTComposer behandelt und verändert werden. Die Terminalknoten („method“, „field“) enthalten zwar auch eine Implementierung (z. B. Methodenrumpf), deren Struktur ist aber mit unserem FST nicht sichtbar. Diese Informationen (z. B. Methodenrümpfe) werden nur als Strings in den Terminalknoten gespeichert.

Die Komposition von Features läuft rekursiv ab. Das bedeutet zwei kompatible Nichtterminale werden komponiert, indem ihre Kindknoten komponiert werden. Für die Komposition von kompatiblen Knoten müssen für jeden Knotentypen Regeln gegeben sein. Die rekursive Komposition terminiert bei Terminalknoten.

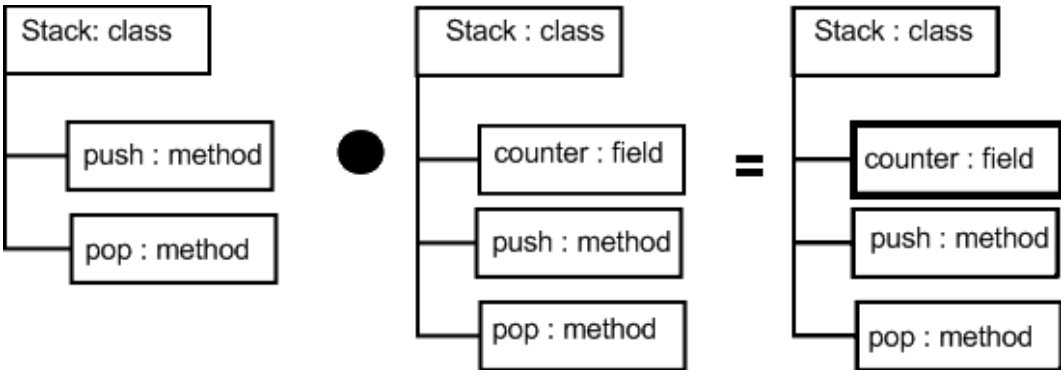


Abb. 9: Stack Komposition

1.4.2 Introduktion

Die Definition der Introduktion und die Beispiele sind übernommen aus [ALM+08].

Eine Introduktion entspricht dem Unterschied zwischen zwei Terminalknoten. Sie ist ein Teil eines Features. Beispiele für Introduktionen sind eine Methode, ein Feld aber auch größere Strukturen wie eine Klasse oder ein Interface. Die Notation von Introduktionen wird nun an einem Beispiel gezeigt. Introduktionen werden über ihren Pfad im FST identifiziert. Der FST in der Abbildung „FST des Features CalcBase„ (Abb.8) enthält folgende Introduktionen:

- *calc* (Package)
- *calc.calc* (Klasse)
- *calc.calc.enter* (Methode)
- *calc.calc.clear* (Methode)
- *calc.calc.e0* (Feld)

Zur Komposition von Introduktionen definiert die Feature-Algebra die Introduktionssumme

$\oplus : I \times I \rightarrow I$

Der FST aus dem obigen Beispiel kann also durch die Summe dargestellt werden:

$CalcBase = calc \oplus calc.calc \oplus calc.calc.enter \oplus calc.calc.clear \oplus calc.calc.e0$

Durch die Definition wird deutlich, dass die Introduktionssumme auf einer Teilmenge des Kompositionsoperators operiert. Die Introduktionssumme ist ein speziellerer Operator als der Kompositionsoperator.

1.4.3 Modifikation

Die Modifikationsanwendung wird definiert als $\odot : M \times I \rightarrow I$ wobei M die Menge der Modifikationen ist. Eine Modifikation besteht aus zwei Teilen: Die Definition, wo sie Änderungen vornimmt („Query“), und die Definition der Änderungen („Change“). In einer Query können mehrere Stellen einer Introduktion gleichzeitig ausgewählt werden.

Die Behandlung der Modifikation läuft in zwei Schritten ab. Als erstes werden die von der Query definierten Stellen in der Introduktion identifiziert. An diesen Stellen werden dann die Änderungen, die im Change definiert sind, vorgenommen.

Mit der Modifikation lassen sich leicht Änderungen vornehmen, bei denen an vielen Stellen im Programmcode dasselbe eingefügt oder geändert werden soll. Die Modifikation und die Introduktion ergänzen sich zu einer Feature-Algebra mit der man effizient Änderungen definieren kann.

Als weiteren Operator stellt die Feature-Algebra eine Überladung der Modifikationsanwendung bereit, mit der man nach dem gleichen Muster Modifikationen komponieren kann. $\odot : M \times M \to M$

Für ausführliche Beispiele zu den Operatoren wird auf das Paper zur Feature-Algebra [ALM+08] verwiesen.

1.4.4 Algebraische Gesetze der Feature-Algebra

Die Introduktionssumme $\oplus$ über der Menge der Introduktionen I definiert einen nicht-kommutativen, idempotenten Monoid $(I, \oplus, \xi)$ [ALM+08]. Das neutrale Element des Monoiden ξ ist der leere FST. In dem Monoiden gelten die Assoziativität $(i \oplus j) \oplus k = i \oplus (j \oplus k)$, die Identität $\xi \oplus i = i \oplus \xi = i$, die Idempotenz $i \oplus j \oplus i = j \oplus i$ und die Nicht-Kommutativität[11].

Die Nicht-Kommutativität $i \oplus j \neq j \oplus i$ der Introduktionen hat größere Auswirkungen auf die Verwendung der Feature-Algebra. Unter Anderem muss deshalb die Reihenfolge der Features in einer Feature-Auswahl [2.1.2] berücksichtigt werden. Bei der Entwicklung der C#-Artefaktsprache in Kapitel 3 wird darauf geachtet, das die Komposition für möglichst viele Introduktions-Typen kommutativ gehalten wird. Dadurch wird die Entscheidung welche Reihenfolge bei der Feature-Auswahl zu wählen ist vereinfacht. Bei der Introduktion von Methoden wird von anderen Feature-orientierten Sprachen und speziell von der Java-Artefaktsprache des FSTComposers ein nicht-kommutativer Ansatz gewählt. In Kapitel [3.2.6] werden verschiedene Möglichkeiten diese Komposition zu realisieren diskutiert.

1.5 Verwandte Ansätze

1.5.1 Aspekt-Orientierung

Der Aspekt-orientierte Ansatz ist mit der Feature-Orientierung am nächsten verwandt und es ist wohl auch der bekannteste Ansatz. Die Aspekt-Orientierung ist aus der Metaobjektprogrammierung entstanden. Sie erlaubt es, einzelne Strukturen im Code (Klassen, Felder, Methoden, ...) zu identifizieren und zu ändern. Die Idee hinter der Aspekt-Orientierung ist nun alle Meta-Objekt-Änderungen, die für die Implementierung eines Features notwendig sind, in ein Modul, einen „Aspekt“ zu kapseln.

Aus dem beschriebenen Ansatz ergeben sich die zwei dominierenden Verarbeitungsschritte der Aspekt-Orientierung:

- Identifikation der zu ändernden Stellen („Wo?“)
- Definition der Änderungen („Was?“)

Die Stelle, an der geändert werden soll, nennt man *„Joinpoint“* und die Änderung selbst wird *„Advice“* genannt. Im Feature `Counter` des Stack-Beispiels [2.1.2] wäre es möglich die Methode „push“ (Joinpoint) um den Code zur Inkrementierung der Variablen `counter` (Advice) zu erweitern. Die bekannte Aspekt-orientierte Sprache AspectJ erlaubt es auch Strukturen zu identifizieren, die sich

[11] Alle algebraischen Eigenschaften wurden aus [ALM+08] übernommen.

erst zur Laufzeit ergeben. So kann man zum Beispiel einen Joinpoint definieren, der nur gilt solange man sich im Programmfluss einer bestimmten Methode befindet. Diese Bedingung kann nur zur Laufzeit ausgewertet werden. Daraus ergeben sich zwei verschiedene Zeitpunkte an denen eine Aspekt-orientierte Sprache ansetzen muss:

- Compilezeit um statische Änderungen am Code auszuführen
- Laufzeit um die angesprochene dynamische Bedingung zu prüfen

Wie oben erwähnt kann der FSTComposer keinen Einfluss nach der Compilezeit ausüben. Der zweite Punkt kann also nicht berücksichtigt werden. Damit ist die Aspekt-Orientierung mächtiger als unser Ansatz. Es sind aber eine Vielzahl von zusätzlichen Schlüsselwörtern und eine sehr komplizierte Syntax nötig um diese Mächtigkeit zu erreichen.

Das so genannte Unbewusstseinsprinzip fordert in der Aspekt-Orientierung, dass die zu erweiternden Programme möglichst nicht für die Erweiterung mit Aspekten angepasst werden müssen. Die Basisprogramme würden also auch ohne Aspekte funktionieren. Das entspricht in etwa einer minimalen Feature-Auswahl in der Feature-orientierten Programmierung.

Eine ausführlichere Einführung in die Aspekt-orientierte Programmierung findet sich in [KLM+97]. Es existiert eine Implementierung des Aspekt-orientierten Ansatzes für C# von der Iowa State University of Science and Technology. Diese Aspekt-orientierte Sprache hat den Namen „EOS“[12].

Im Folgenden Kapitel wird ein Ansatz angesprochen, der versucht Aspekt-Orientierung mit C#-eigenen Sprachkonstrukten zu erreichen.

1.5.2 Attribute Programming

Das Attribute-Konstrukt in C# erlaubt es Klassen Meta-Informationen zu geben. Außerdem ist es möglich eigene Attribut-Klassen zu definieren. Man kann so einer existierenden Methode Code (in Form eines Attributs) hinzufügen. Da ein Attribut eine Klasse ist, wird am Anfang der Methode die Konstruktorfunktion der Attributsklasse aufgerufen.

Dieser Ansatz erfordert schon für kleine (sinnvolle) Beispiele viel Code und darum wird nur auf die existierenden Projekte zu dem Thema zu verweisen.

Weil man nur den Konstruktor des Attributs aufrufen kann, und diesen Aufruf auch nur zu Beginn der erweiterten Methode einfügen kann, ist dieser einfache Attribute Programming Ansatz der Aspekt-Orientierung weit unterlegen. Ein weiteres Problem liegt darin, dass die zu erweiternde Klasse geändert werden muss. Das Unbewusstseinsprinzip der Aspekt-Orientierung kann also nicht erfüllt werden.

Projekte und Tutorials zu diesem Thema findet man unter den folgenden Quellen:

- Tutorial zu Attribute-Programming und EOS
  http://www.geocities.com/aspectdotnet/AOP2.htm
- PostSharp http://www.postsharp.org/
- EOS http://www.cs.virginia.edu/~eos/doc/

[12]Online verfügbar unter http://www.cs.iastate.edu/~eos/

2. Untersuchung von C# als Artefaktsprache

In diesem Kapitel wird untersucht, ob und wie die Programmiersprache C# als Artefaktsprache für die Komposition durch Superimposition genutzt werden kann. Dafür wird zunächst die Feature-Algebra erweitert und dann auf dieser Grundlage die einzelnen syntaktischen Elemente von C# untersucht.

2.1 Erweiterung der Feature-Algebra

Um die Feature-Algebra für die Untersuchung von C# nutzen zu können, wird noch eine genauere Regelung für die Voraussetzungen zur Komposition von Operanden benötigt. In der Feature-Algebra wird beschrieben wie die Komposition von FSTs abläuft. Die Voraussetzungen, die zwei Eingabe-Knoten der FSTs erfüllen müssen um komponiert zu werden, sind bisher nicht formal definiert.

Dieser Umstand tritt besonders bei der Summe von Introduktionen auf. Bisher wird in der vorgestellten Feature-Algebra davon ausgegangen, dass zwei Knoten immer komponiert werden wenn ihre Namen (Pfade im FST) und Typen übereinstimmen. Diese Bedingung reicht für die Komposition von einigen Knotentypen des C#-FST nicht aus.

Im folgenden Kapitel wird beschrieben welche Relationen zwischen zwei Operanden gelten müssen, damit sie sicher komponiert werden können.

Kompatibilität von Knoten

Für die Prüfung, ob zwei Knoten kompatibel sind, werden zwei Äquivalenzrelationen definiert. Die Funktionen geben jeweils an, ob sich die Argumente (Operanden) in derselben Äquivalenzklasse befinden. Ein Operand entspricht dabei einem beliebigen Knoten eines FSTs.

Die erste dieser Relationen ist die Relation *equals* [13]. Mit ihr wird festgestellt, ob zwei Eingabe-Knoten komponiert werden *sollten*. Im Normalfall sollten zwei Knoten komponiert werden falls sie, aufgrund von Beschränkungen der Artefaktsprache, nicht gemeinsam im Ergebnis existieren können. Es werden später noch weitere, stärkere Beschränkungen für einzelne Knotentypen definiert.

$$equals : F \times F \rightarrow \begin{cases} true, & \text{Knoten können nicht im selben Kontext exisiteren} \\ false, & \text{sonst} \end{cases}$$

Falls zwei Operanden in derselben *equals* -Äquivalenzklasse liegen, müssen sie vom FSTComposer entweder komponiert werden, einer der Knoten muss ignoriert werden oder der Prozess muss abgebrochen werden. Wenn beide Operanden in das Ergebnis übernommen würden, wäre das Ergebnis syntaktisch inkorrekt. In den meisten Fällen sind Operanden gleich, wenn ihre Namen übereinstimmen. Zum Beispiel können zwei Knoten der Typen „field“ und „method“ mit demselben Namen nicht auf einer Ebene des FST existieren. Es muss also auch möglich sein, dass Knoten unterschiedlichen Typs bezüglich *equals* gleich sind.

[13] Im Folgenden die *equals* Relation auch umgangssprachlich als die Gleichheit von Knoten bezeichnet. Die *composable* Relation wird im gleichen Sinn als Komponierbarkeit bezeichnet.

$$composable : F \times F \rightarrow \begin{cases} true, & \text{falls die Knoten kombiniert werden können} \\ false, & \text{sonst} \end{cases}$$

Die Relation *composable* ist stärker als die Relation *equals*. Es muss also gelten $\forall a, b : composable(a, b) \Rightarrow equals(a, b)$. Aus diesem Grund wird zu Beginn jeder *composable* Relation geprüft ob *equals* gilt, falls nicht gibt *composable* *false* zurück. Im Folgenden wird also bei Anwendung der Relation *composable* davon ausgegangen, das die Parameter gleich (bzgl. *equals*) sind.

Wenn zwei Knoten gleich sind, bedeutet das noch nicht unbedingt, dass sie auch komponiert werden können. Ein Beispiel für diese Situation: Zwei Klassen (die bzgl. *equals* gleich sind) sollen komponiert werden. Die Klassen haben unterschiedliche Supertypen. Sie stehen in keiner Vererbungshierarchie und die verwendete Sprache kennt keine Mehrfachvererbung.

Da die Klassen in der Relation *equals* stehen, wird *equals* auf den Supertypen aufgerufen. Der Aufruf von *equals* auf den Supertypen muss *true* ergeben, weil jede Klasse nur einen Supertyp haben darf (vgl. Definition von *equals*). Nun muss noch die Komponierbarkeit der Supertypen geprüft werden. Dieser Vergleich muss *false* zurückgeben, weil die Supertypen nicht komponierbar sind. In diesem Fall ($equals = true \wedge composable = false$) wird die gesamte Komposition abgebrochen und eine Fehlermeldung ausgegeben. Es genügt in diesem Fall nicht nur die fehlgeschlagene Komposition zu beenden und mit dem restlichen FST fortzusetzen. Wenn die Komposition von FST-Knoten fehlschlägt ist das ein Anzeichen für eine grundsätzliche Inkorrektheit bzw. Inkompatibilität der Eingabe-Features. Diese Situation muss an den Entwickler gemeldet werden. Die Komposition darf erst dann erfolgreich sein, wenn jede einzelne Teil-Komposition erfolgreich ist.

Insgesamt ergeben sich also folgende Möglichkeiten:

equals	*composable*	
false		Knoten können parallel existieren, beide Knoten werden in das Ergebnis übernommen
true	*false*	Knoten können nicht komponiert werden, Fehlermeldung
true	*true*	Knoten werden mit dem jeweiligen Kompositionsalgorithmus komponiert

Tab. 1: Auswertung der Relationen equals und composable

Die konkreten Kriterien der Relationen *equals* und *composable* müssen für jeden Knotentyp des FST separat festgelegt werden. Diese werden im nächsten Kapitel diskutiert.

2.2 Introduktionen in C#

Im Grundlagenkapitel zur Feature-Algebra wurde die Introduktionssumme definiert. Nun wird diese Operation genutzt um zu untersuchen wie Introduktionen der Artefaktsprache C# komponiert werden können. Vor der Komposition von zwei Introduktionen wird geprüft, ob die beiden Introduktionen gleich

(*equals*) und komponierbar (*composable*) sind. Dann werden die Introduktionen nach einem Kompositionsalgorithmus komponiert.

Diese drei Mechanismen (*equals* , *composable* und der Kompositionsalgorithmus) werden in diesem Kapitel für die einzelnen Elemente der C# Syntax (nach ECMA-334) definiert. Bei einigen Elementente werden mehrere Alternativen vorgestellt und diskutiert.

2.2.1 Klassen

Relation: equals In C# werden Klassen durch ihren Namen und die Anzahl ihrer Typparameter identifiziert. Folgendes Codefragment ist also nicht korrekt:

```
namespace test {
    class A { }
    class A { }
    class B { }
    class B <T> { }
}
```

Abb. 10:Inkorrekte Klassendefinitionen

Die „B“ -Klassen sind jedoch korrekt definiert, weil sie unterschiedlich viele Typparameter haben. Die Inkorrektheit der „A“-Klassen kann man in C# beheben, indem beide Klassen das Schlüsselwort `partial` erhalten. Dann betrachtet der C# Compiler alle Klassen, die wie oben beschrieben identisch sind (Name/ Typparameter-Anzahl), als eine Klassendefinition. Diese identischen Klassen dürfen aber keine doppelten Member enthalten. Um dies zu veranschaulichen wird ein weiteres Beispiel gegeben. Die Definitionen von „A“ sind wieder inkorrekt, die von „B“ korrekt:

```
namespace test {
    partial class A { int i; }
    partial class A { int i; }
    partial class B { int j; }
    partial class B { int k; }
}
```

Abb. 11:Partielle Klassendefinitionen

Durch die Inkorrektheit der „A“-Klassen in dieser Abbildung wird deutlich, dass die *equals* Relation für Klassen-Introduktionen den Namen der Klassen und die Anzahl ihrer Typparameter vergleichen muss. Falls diese Eigenschaften übereinstimmen müssen die Klassen-Introduktionen komponiert werden.
Ein Problem entsteht nun durch das `partial`-Schlüsselwort. In der Feature-Algebra-Notation wird dies verdeutlicht:

$$feature1 = partialClassB \oplus partialClassB$$
$$feature2 = partialClassB \oplus partialClassB.intMyInteger$$

Wenn diese Features komponiert werden, enthält der entstehende FST zwei *partialClassB* , die jeweils einen Integer mit dem Namen „MyInteger“ enthalten. Das ist nach Abbildung 11 verboten. Falls dies erlaubt wäre, könnte aus zwei in sich korrekten Features ein inkorrektes Feature entstehen. Das soll unbedingt vermeiden.
Die Ursache dieses Problems ist, dass die der Name „partialClassB“ mehrfach auf einer Ebene des FST existiert. Dieses Problem ist in der Entwicklung des FST-Composer schon aufgetreten, und C# erfüllt mit dem `partial`-Modifier nicht die Auflagen an eine Artefaktsprache der Feature-Algebra [ALM+08].
Aus diesen Gründen wird festgelegt, dass das Schlüsselwort `partial` von der vom FSTComposer akzeptierten Untermenge der C#-Sprache ausgeschlossen wird.
Die Relation *equals* wird wie folgt festgelegt:

$$equals_{Class}(x, y) \in ClassIntroduktion \times ClassIntroduktion$$
$$\rightarrow x.Path = y.Path \wedge x.Typeparameters.count = y.Typeparameters.count$$

Relation: composable Die Eigenschaften einer Klasse, die notwendig sind um die Komponierbarkeit zu bestimmen, befinden sich sämtlich im „Kopf“ der Klasse. Damit ist der Bereich vor der ersten geschweiften Klammer gemeint, nach der der Klassenkörper folgt. In diesem Bereich sind die folgenden Angaben zu finden:

- eine Liste mit Attributen
- eine Liste mit Typparametern
- eine Liste mit Supertypen (in C# wird nicht syntaktisch zwischen Superklasse und Superinterface unterschieden)
- eine Liste mit Modifiern
- eine Liste mit Typparameter-Einschränkungen
- der Name der Klasse

Um zu überprüfen ob zwei Klassenintroduktionen komponierbar sind, wird *composable* auf den Introduktionen, die den Klassenintroduktionen untergeordnet sind, aufgerufen. Das bedeutet, dass die oben genannten Listen jeweils mit ihrem Äquivalent in der anderen Klassenintroduktion verglichen werden. Wenn einer dieser *composable* Vergleiche *false* ergibt, ist die Komposition gescheitert.

$$composable_{Class}(x, y) \in ClassIntroduktion \times ClassIntroduktion$$
$$\rightarrow Composable_{Attributes}(x.Attributes, y.Attributes)$$
$$\wedge Composable_{Typeparameters}(x.Typeparameters, y.Typeparameters)$$
$$\wedge \ldots$$

Dieses Schema der *composable* Relation, in dem man die Prüfung an untergeordnete Introduktionen delegiert, ist das Standard-Schema für die *composable* Relation. Falls keine weiteren Prüfungen gemacht werden müssen, wird immer dieses Schema angewandt.

Komposition Klassenintroduktionen werden komponiert, indem die in der FST-Hierarchie untergeordneten Introduktionen komponiert werden. Die komponierte Klassenintroduktion erhält denselben Klassennamen wie beide komponierte

Introduktionen (diese Namen müssen ja gleich sein, weil die Klassen in *equals* Relation stehen). Außerdem werden die komponierten untergeordneten Introduktionen an die neue Klassenintroduktion angehängt.

2.2.2 Supertyp-Listen

Da in C# bei der Deklaration von z. B. Klassen nicht zwischen Superklassen und Superinterfaces unterschieden wird, enthält eine Liste von Supertypen beide Möglichkeiten.

Da C# keine Mehrfachvererbung kennt, enthält jede Supertyp-Liste maximal eine Superklasse, und beliebig viele Superinterfaces. Es kann jedoch ohne zusätzliche Mittel nicht aufgrund der Deklaration festgestellt werden, ob ein Element der Liste eine Klasse oder ein Interface ist.

Ein Beispiel für eine Deklaration einer Klasse mit Superklasse und Superinterface:

```
1  class test : Stack, IEnumerable {
2      // Stack ist eine Klasse
3      // IEnumerable ist ein Interface
4  }
```

Abb. 12:Klassendeklaration mit Supertypen

Die Reihenfolge der Elemente bei Deklaration einer Supertyp-Liste ist beliebig und die Konvention Interface-Namen mit einem „I“ zu beginnen ist nicht zwingend. Die einzige Möglichkeit um herauszufinden ob mehrere Superklassen verwendet wurden ist also den gesamten Code nach der betreffenden Deklaration zu durchsuchen. Diese Code-Basis umfasst im Zweifelsfall auch das gesamte C#-Framework. Dieser Umstand wird bei der Definition der *composable* Relation wichtig. Bei der Deklaration von Interfaces sind natürlich keine Klassen als Supertypen erlaubt.

Relation: equals An jeder Stelle des FST darf nur eine Supertyp-Liste existieren. Darum müssen Supertyp-Listen immer gleich sein, falls die Pfade ihrer Introduktionen übereinstimmen.

$$equals_{SupertypeList}(x, y) \in SupertypeList \times SupertypeList \rightarrow (x.Path = y.Path)$$

Relation: composable Wir gehen davon aus, das man nicht feststellen kann, ob ein Element der Supertyp-Liste eine Klasse oder ein Interface ist (siehe oben).

In diesem Fall bleiben zwei mögliche Vorgehensweisen:

- Nur identische Supertyp-Listen zulassen (Reihenfolge kann sich unterscheiden)
- Alle Supertyp-Listen zulassen und bei der Komposition die Obermenge bilden

Die erste Alternative ist sehr restriktiv, weil es nicht erlaubt ist, einer Klasse einen neuen Typen (auch kein neues Interface) hinzuzufügen. Dafür ist diese Alternative auch sehr sicher, weil man garantieren kann, dass die Deklaration in Bezug auf die Supertyp-Listen korrekt ist, wenn beide Introduktionen korrekt sind.

Die *composable* Relation für diese Alternative ist wie folgt:

$$composable_{SupertypeList}(x, y) \in SupertypeList^2 \rightarrow (x \subseteq y) \wedge (y \subseteq x)$$

Die zweite Alternative erlaubt es der Deklaration einer Klasse neue Supertypen hinzuzufügen. So kann man etwa in einem Feature neue Methoden hinzufügen und diese Methoden über ein zugehöriges Interface bekanntgeben.

Die Gefahr dieser Alternative besteht darin, dass man auch zwei Klassen mit gleichem Namen aber unterschiedlichen Superklassen komponieren würde. Das Ergebnis hätte dann zwei Superklassen und wäre nicht korrekt (Mehrfachvererbung).

Die *composable* Relation für die zweite Alternative ist

$$composable_{SupertypeList}(x, y) \in SupertypeList \times SupertypeList \rightarrow true$$

weil keine Prüfungen gemacht werden können.

Komposition Die Komposition der Supertyplisten erfolgt je nach gewähltem *composable* Ansatz. Bei der ersten Alternative wird eine der beiden Introduktionen übernommen (sie sind identisch). Bei der zweiten Alternative wird die Vereinigungsmenge der beiden Supertyp-Listen berechnet. Diese Menge stellt das Ergebnis der Komposition dar.

2.2.3 Interfaces und Structs

Die Relationen und Kompositionsalgorithmen für Interfaces und Structs werden exakt von den Klassen übernommen. Dies ist möglich, weil ein Interface genau dieselben Eigenschaften hat wie eine Klasse. Ein Struct hat keine Supertypen (Structs erben immer direkt von `object`). Die Relationen für Structs entsprechen also den Relationen von Klassen vermindert um die Supertyp-Listen.

2.2.4 Modifier-Listen

In C# gibt es insgesamt 15 Modifier. Diese gliedern wir in 3 Teilmengen: *Zugriffsmodifier*, *Parametermodifier* und *sonstige Modifier*.

Zugriffsmodifier werden genutzt um die Zugriffsrechte auf Sprachkonstrukte zu regeln. Die Menge der Zugriffsmodifier umfasst in C# die folgenden Schlüsselworte: `private`, `public`, `protected` und `internal`.

Mit Parametermodifier kann man steuern wie Methodenparameter übergeben werden (z. B. byReference). Die Schlüsselwörter sind `ref` und `out`.

Alle weiteren Modifier sind in der Menge der sonstigen Modifier zusammengefasst. Das sind `new`, `abstract`, `sealed`, `static`, `readonly`, `volatile`, `override` und `extern`.

Relation: equals Da auf jeder Ebene des FST nur eine Modifier-Liste erlaubt ist, müssen alle Modifer-Listen-Introduktionen die im Pfad übereinstimmen die equals Relation erfüllen. Weil eine Liste keinen Namen hat wird der String „<NoName>“ als Name angenommen. Der Pfad einer Modifier-Liste besteht also aus dem den Namen der FST-Knoten die von der Feature-Baum-Wurzel zur Liste führen und dem String „<NoName>“. Die *equals* Relation für Modifier-Listen wird demnach wie folgt definiert:

$$equals_{ModifierList}(x, y) \in ModifierList \times ModifierList \rightarrow (x.Path = y.Path)$$

Relation: composable Für die Entscheidung ob zwei Modifierlisten komponierbar sind, spielt es eine wichtige Rolle ob die Modifier, die in den Listen enthalten sind zusammengenommen eine gültige Modifier-Liste ergeben. Diese Entscheidung wird erschwert, weil in C# viele Modifier-Komposition in verschiedenen Situationen nicht erlaubt sind. Die Reihenfolge der Modifier ist in jedem Fall irrelevant. Im Folgenden werden also die Modifier-Listen als ungeordnete Mengen betrachtet.

Die sichere Entscheidung ist immer, dass nur Modifier-Listen die komplett identisch sind komponierbar sind. Das wäre aber eine sehr restriktive Argumentation, die auch dem Prinzip der Feature-Orientierung nicht gerecht werden würde. Diese restriktive Formulierung wird nur für die Menge der sonstigen Modifier und die Menge der Parametermodifier genutzt.

Die Begründung für diese Entscheidung ist bei Parametermodifiern einfach: Die C# Spezifikation erlaubt nur einen der beiden Modifier [ECMA Kap.17.5.1]. Falls in einer der beiden zu komponierenden Modifierlisten kein Parametermodifier enthalten ist, würde das Einfügen des Modifiers zu einer empfindlichen Änderung des Programmverhaltens im restlichen Code dieses Features führen. Da wir diese eventuell unbeabsichtigten Auswirkungen vermeiden wollen, nutzen wir die restriktive Formulierung der *composable* Relation für Parametermodifier.

Für die sonstigen Modifier wird diese Entscheidung mit den extrem komplexen Wechselwirkungen zwischen den Modifiern, die sich auch ändern, je nach dem wo die Modifier angewandt werden (bei Klassen, Methoden, Interfaces, ...), begründet. Die Praxis wird zeigen welche Kompositionsmöglichkeiten für Modifier aus dieser Menge möglich und hilfreich sind. Hier wäre eine Möglichkeit für weitere Arbeiten gegeben. Diese könnten sich eventuell am Beispiel der Zugriffsmodifier orientieren, das im folgenden Absatz gegeben wird.

Zugriffsmodifier umfassen eine vergleichbar kleine und überschaubare Menge von Modifiern, die auch jedem Leser leicht verständlich sein sollte. Auf den Zugriffsmodifiern kann eine Hierarchie definiert werden, die von dem restriktivsten Modifier (`private`) zum offensten Modifier (`public`) reicht.

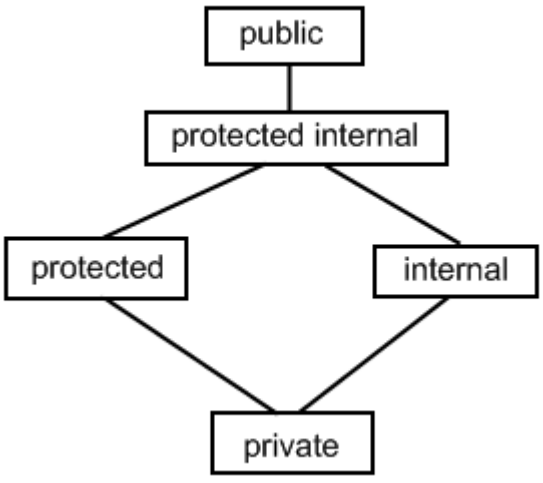


Abb. 13:Zugriffsmodifier-Hierarchie

Die Modifier `protected` und `internal` existieren sowohl in Komposition als auch einzeln. Der Effekt von `protected internal` ist genau die Summe der Effekte von `protected` und `internal`. Die Einzelheiten sind in der C#-Spezifikation nachzulesen [ECMA].

Weitere Kombinationen von Zugriffsmodifiern sind nicht erlaubt. Es ist also zum Beispiel nicht möglich eine Methode als `private internal` zu definieren.

Aufgrund dieser Erkenntnisse wird eine Funktion definiert, die zwei gegebene Zugriffsmodifier als Argumente nimmt, und einen Zugriffsmodifier zurück gibt. Diese Funktion wird im nächsten Abschnitt definiert.

Zunächst wird die *composable* Relation für Modifier-Listen definiert:

$$composable_{ModifierList}(x, y) \in ModifierList \times ModifierList \rightarrow$$
$$(x \cap PARMETER\,MODIFIERS) = (y \cap PARMETER\,MODIFIERS)$$
$$\wedge\ (x \cap OTHER\,MODIFIERS) = (y \cap OTHER\,MODIFIERS)$$

Zugriffsmodifier werden in dieser Definition nicht erwähnt, weil die oben genannte Funktion für alle Komposition von Zugriffsmodifiern definiert ist. Es müssen also keine Einschränkungen bezüglich der Zugriffsmodifier gemacht werden.

Komposition Der Kompositionsalgorithmus ist nun relativ einfach. Es wird davon ausgegangen, dass die Listen der Parametermodifier und der sonstigen Modifier in den Operanden-Introduktionen komplett identisch sind (vgl. Definition von $composable_{ModifierList}$). Sie werden also ohne weitere Betrachtung in die Ergebnis-Introduktion übernommen.

Bei den Zugriffsmodifiern wissen wir, dass von beiden Komponenten nur je ein Modifier zu erwarten ist (sonst wäre das Feature inkorrekt weil die C#-Spezifikation nur einen Zugriffsmodifier erlaubt). `protected internal` zählt in diesem Fall als ein Modifier.

Nun kann die oben angesprochene Funktion definiert werden. Sie hat folgende Signatur:

$$compose_{AccessModifier}: AccessModifier \times AccessModifier \rightarrow AccessModifier$$

Sie wird in Form einer Tabelle notiert, weil eine mathematisch ausführliche Notation zu unübersichtlich wäre. Außerdem wird die Kommutativität der Funktion genutzt, um Platz zu sparen.

Operand x	Operand y	Ergebnis
public	public, protected internal, protected, internal, private	public
protected internal	protected internal, protected, internal, private	protected internal
internal	protected	protected internal
internal	internal, private	internal
protected	protected, private	protected
private	private	private

Tab. 2: Definition der Komposition für Modifier-Listen

Diese Funktion gibt immer den Zugriffsmodifier zurück, der mehr „erlaubt“. Damit erwirken wir, dass man zum Beispiel die Definition von einer privaten Klasse in eine public Klasse ändern kann. Dafür muss man nur die Introduktion der privaten Klasse mit einer Introduktion einer äquivalenten Klasse (*equals*), die den public-Modifier hat, komponieren.

Da nicht alle Modifier für jede C# -Struktur erlaubt sind, muss überprüft werden ob für jede C#-Struktur, die `protected` und `internal` erlaubt auch `protected internal` möglich ist. Falls das nicht der Fall ist, kann die obige Funktion zu inkorrekten Programmen führen. Laut C#-Spezifikation ist dies aber der Fall.

Man hätte die Funktion auch so definieren können, dass immer der restriktivere Modifier zurückgegeben wird. Die oben angegebene Definition hat den Vorteil, dass sie dem Entwickler weit mehr Flexibilität gibt. Eine nachträgliche Einschränkung von Modifiern wäre in den meisten Fällen nicht sinnvoll.

2.2.5 Felder

Ein Feld ist Teil der (Daten-) Struktur einer Klasse. Felder werden mit einem Typ deklariert und können eine Instanz dieses Typs speichern. In C# ist es auch möglich dem Feld bei der Deklaration schon einen Initalwert zuzuweisen. Außerdem können Felder mit Modifiern deklariert werden um ihr Verhalten zu beeinflussen. Die verschiedenen Möglichkeiten Felder zu deklarieren werden in der folgenden Abbildung angegeben:

```
int i; // einfache Deklaration
int x = 10; //Dekl. mit Initialisierung
private string str; // Dekl. mit Modifier
// Mehrfach-Deklaration
int a, b = 8, c;
```

Abb. 14: Feld-Deklarationen

Bei der Behandlung von Mehrfach-Deklarationen entstehen einige Schwierigkeiten. Darum wird zunächst nur auf die übrigen Deklarationsmöglichkeiten eingegangen. Die Mehrfach-Deklarationen werden dann im Abschnitt „Probleme“ behandelt.

Relation: equals Felder werden in C# über ihren Namen identifiziert. Die *equals* Relation prüft also auf Gleichheit der Namen (Identifier) der Felder.

Alle anderen Eigenschaften eines Feldes, wie Typ oder Initialisierung haben keinen Einfluss auf diese Relation.

Die *equals* Relation für Feld-Deklarationen sieht also aus wie folgt:

$$equals_{Field}(x, y) \in Field \times Field \rightarrow (x.Path = y.Path)$$

Relation: composable Ob zwei gleiche (*equals*) Feld-Introduktionen auch komponiert werden, können hängt von ihren Typen und den Initialisierungen ab.

Wir gehen zunächst auf die Typen ein. Die Typen der Feld-Introduktionen werden im folgenden Abschnitt TypA und TypB genannt. Falls TypA und TypB denselben Typen repräsentieren können die Introduktionen aufgrund der Typen komponiert werden. Wenn sich TypA und TypB unterscheiden, gibt es zwei Möglichkeiten. Entweder es gibt keine Subtypenbeziehung zwischen den Typen oder einer der Typen ist ein Subtyp des anderen. Im ersten Fall kann keine Komposition erfolgen, weil nicht entschieden werden könnte, welcher der Typen übernommen wird. Für das Ergebnis könnte also nicht garantiert werden, dass es weiterhin typkorrekt ist (ausgehend davon, dass beide Operanden typkorrekt sind).

Falls die Typen in einer Subtypenbeziehung stehen, kann man den spezielleren Typ in das Ergebnisfeature übernehmen. Damit stehen alle Variablen und Methoden des Obertyps weiterhin (nach einem Cast) zur Verfügung. Durch Upcasts (die vom Compiler automatisch eingefügt werden) könnte also Typkorrektheit garantiert werden.

Falls nicht festgestellt werden kann, ob zwischen TypA und TypB eine Subtypenbeziehung besteht, hat man keine Wahl. Wenn sich die Typen unterscheiden, muss man die Komposition abbrechen.

Kommen wir nun zu den Initialisierungen der Introduktionen. Falls keine oder nur eine der Feld-Introduktionen Initialisierungen haben, gibt es kein Problem. Die eine Introduktion kann in das Ergebnis übernommen werden. Diese Vorgehensweise ist kompatibel mit der oben beschriebenen Ausnutzung einer Subtypenbeziehung. Man kann in der Initialisierung auch einen Obertypen des deklarierten Typen zuweisen.

Falls jedoch beide Introduktionen Initialisierungen haben, können sie nur übernommen werden, wenn sie identisch sind. Ein Feld kann nicht mit zwei verschiedenen Initalwerten deklariert werden. Eine Alternative wäre hier die Reihenfolge der Features in der Feature-Auswahl zu berücksichtigen, und eines der Features zu priorisieren. Die Initialisierung der priorisierten Introduktion würde dann die andere Initialisierung überschreiben. Es wird, soweit möglich, von einer Priorisierung von Features absehen, weil dadurch der Umgang mit der Artefaktsprache zusätzlich erschwert wird.

Aufgrund der obigen Argumentation werden folgende Alternativen für die *composable* Relation präsentiert:

- Falls die Subtypenbeziehungen berücksichtigt werden können

$$composable_{Field}(x, y) \in Field \times Field \rightarrow$$
$$(x \in Supertypes(y) \vee y \in Supertypes(x))$$
$$\wedge \neg(hasInit(x) \wedge hasInit(y))$$

- Falls keine Subtypenbeziehungen berücksichtigt werden können
$$composable_{Field}(x, y) \in Field \times Field \rightarrow$$
$$(x.Type = y.Type)$$
$$\wedge \neg (hasInit(x) \wedge hasInit(y))$$

Komposition Die Komposition von Feld-Introduktionen ist sehr einfach. Da die Introduktionen komponierbar sind, ist bekannt welcher Typ und (optional) welche Initialisierung die Ergebnisintroduktion haben muss. Der Name steht ist ebenfalls schon festgelegt, da beide Introduktionen denselben Namen haben müssen um bezüglich *equals* gleich zu sein. Zur Komposition müssen also nur noch diese Informationen in zu einer neuen Feld-Introduktion zusammengefasst werden. Diese Introduktion ist dann das Ergebnis.

Probleme Die Behandlung der Mehrfach-Deklaration wurde in diesem Abschnitt bisher noch nicht behandelt. Der bisherige Kompositionsalgorithmus funktioniert nicht mehr sobald man Mehrfach-Deklarationen zulässt. In einem AST[14] wird eine Deklaration, die mehrere Felder gleichzeitig deklariert, als ein Knoten dargestellt. Dieser Knoten hat Unterknoten, die die jeweiligen Felder darstellen. Diese Deklarationsmöglichkeit ist logisch äquivalent zur einzelnen Deklaration der Felder. Darum darf ein Feld, welches bereits in einer Mehrfach-Deklaration auftaucht, nicht noch einmal deklariert werden.

Mit einem FST ist das aber schlecht aufzulösen. Man müsste für jedes Element einer Mehrfachdeklaration auch in der nächsthöheren Hierarchieebene nach einem entsprechenden Knoten suchen. Das führt zu einigem zusätzlichen Aufwand. Das folgende Beispiel demonstriert die Interaktionen von einfachen und Mehrfachdeklarationen:

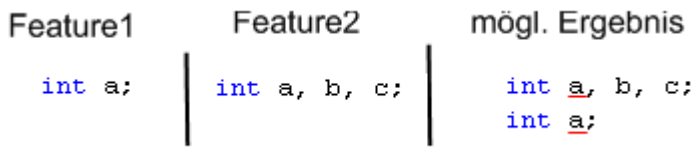


Abb. 15:Konflikt bei Mehrfachdeklarationen

Um diesen Konflikt zu erkennen muss man bei der Behandlung der Mehrfachdeklarations-Liste in Feature2 auf der höheren Ebene die Deklaration in Feature1 erkennen. Dieser Mechanismus ist in der Feature-Algebra nicht vorgesehen. Eine einfache Lösung dieses Problems ist es vor der Behandlung alle Mehrfachdeklarationen in einfache Deklarationen umzuwandeln.

2.2.6 Methoden

Die Komposition von Methoden stellt den wichtigsten Teil der Feature-orientierten Komposition dar, weil so neue Funktionalität in bestehende Programmabläufe eingefügt werden kann. Es werden verschiedene Ansätze für die Komposition von Methoden entwickelt und verglichen.

[14]Im CST ist jede Deklaration eine (potentielle) Mehrfachdeklaration. Dies kann jedoch aufgrund der unten genannten Argumentation nicht für den FST übernommen werden.

Im folgenden Code-Ausschnitt wird ein Beispiel für die Komposition von Methoden mit dem FSTComposer in Java gegeben.

Feature1

```
int test() {
  System.out.println("Feature1");
  return 1;
}
```

Feature2

```
int test() {
  System.out.println("Feature2");
  return original();
}
```

Ergebnis

```
int test_Feature1() {
  System.out.println("Feature1");
  return 1;
}
int test() {
  System.out.println("Feature2");
  return test_Feature1();
}
```

Abb. 16:Methodenkomposition in Java

Für die Steuerung der Komposition wurde ein neues Schlüsselwort (`original`) eingeführt. Dieses Schlüsselwort wird durch den Aufruf der überschriebenen Methode ersetzt. Dies ist aber nicht die einzige Möglichkeit Methoden zu komponieren. Im Folgenden Abschnitt werden verschiedene Alternativen diskutiert. Zunächst werden aber die *equals* - und *composable* Relationen für Methoden definiert. Da sie nur aus C#-Sprachspezifikationen abgeleitet werden, hat man hier nur wenig Alternativen.

Relation: equals Methoden werden in C# durch ihren Namen, und die Anzahl und Typen ihrer Parameter identifiziert. Anhand dieser Eigenschaften müssen also die Kriterien für die *equals* Relation festgelegt werden. Der Rückgabetyp einer Methode ist in der C#-Spezifikation ausdrücklich von ihrer Signatur ausgeschlossen [ECMA Kapitel 10.6]. Er darf also für die *equals* Relation nicht betrachtet werden.

Es wäre denkbar, dass man auch Methoden komponiert, die unterschiedliche Anzahlen von Parametern haben. Man könnte zum Beispiel eine Methode, die einen Integer erwartet durch eine Methode, die einen Integer und einen String erwartet erweitern. Durch diesen Ansatz würde man aber die Überladung von Methoden durch C# behindern. Außerdem hätte das negative Auswirkungen auf die Korrektheit von Code im ersten Feature. Aus diesen Gründen wird von dieser Lösung abgesehen.

Zwei Methoden sind im Sinne von *equals* gleich, wenn sie denselben Namen haben und ihre Parameterlisten übereinstimmen.

$$equals_{Method}(x, y) \in Method \times Method \rightarrow (x.Path = y.Path)$$
$$\wedge\, y.InputParamters = x.InputParameters$$
$$\wedge\, y.TypeParamters.count = x.TypeParameters.count$$

Für diese Definition muss noch die Gleichheit von Parameterlisten definiert werden. Folgende Bedingungen werden an die Parameterlisten gestellt:

- Reihenfolge der Typen muss übereinstimmen
- Länge der Listen muss übereinstimmen

Die Namen der Parameter sind nicht in der Signatur einer Methode enthalten. Sie dürfen also nicht in der *equals* Relation genutzt werden.

In C# können Methoden Typparameter zugewiesen bekommen. Die Signatur einer Methode enthält die Anzahl ihrer Typparameter. Es können also zwei Methoden im selben Namensraum existieren, die sich nur in der Anzahl ihrer Typparameter unterscheiden. Darum muss für die Gleichheit von Typparameter-Listen nur folgende Bedingung gelten:

- Länge der Typparameter-Listen stimmen überein

Relation: composable Damit zwei gleiche Methoden, komponiert werden können müssen sie noch einige zusätzliche Bedingungen erfüllen. Dazu gehört unter anderem, dass ihre Parameternamen gleich sind. Diese Bedingung ist nicht in der C#-Spezifikation begründet sondern wird gestellt um die Kompositionsalgorithmen leichter nachvollziehbar zu gestalten. Es ergibt sich daraus keine wesentliche Einschränkung für den Entwickler. Zusammen mit der *equals* Relation bedeutet das, dass nur Methoden mit exakt identischen Parameterlisten komponierbar sind. Durch diese restriktive Formulierung werden Probleme mit `ParameterArrays`[15] vermieden.

Weiterhin wird gefordert, dass die Rückgabewerte der Methoden identisch sind. Hier wäre es auch möglich die Typhierarchie zu nutzen oder die Umwandlung der Typen dem Entwickler (in der überschreibenden Methode) zu überlassen.

Weiterhin müssen die Typparameter und Typparameter-Einschränkungen der Methoden exakt identisch sein. Die Begründung für diese Bedingung wird im Kapitel über Typparameter [3.2.7] gegeben.

Daraus folgt, dass die *composable* Relation für Methoden wie folgt definiert sein muss:

$$
\begin{aligned}
&composable_{Method}(x, y) \in Method \times Method \rightarrow (x.Path = y.Path) \\
&\quad \wedge y.ReturnType = x.ReturnType \\
&\quad \wedge y.InputParamters.Names = x.InputParameters.Names \\
&\quad \wedge y.TypeParamters = x.TypeParameters \\
&\quad \wedge y.TypeParamterConstraints = x.TypeParameterConstraints
\end{aligned}
$$

Je nach verwendetem Kompositionsalgorithmus werden weitere Bedingungen nötig. Diese werden an entsprechender Stelle als Erweiterung dieser Definition formuliert.

In den folgenden Abschnitten werden die verschiedenen Alternativen für die Kompositionsalgorithmen von Methoden vorgestellt.

Ansatz 1: einfaches Überschreiben

Dieser Ansatz realisiert die Komposition von Methoden-Introduktionen, indem die zweite Introduktion einfach die erste ersetzt. Der Code der ersten Introduktion ist danach nicht mehr verfügbar. Der Vorteil von diesem Ansatz liegt ganz offensichtlich in seiner Einfachheit. Die Nachteile sind aber sehr schwerwiegend.

Der Ansatz priorisiert das zweite Feature. Darum ist die Introduktionssumme nicht mehr kommutativ, falls dieser Ansatz gewählt würde. Das würde auch eine eventuelle Kommutativität des Kompositionsoperators verhindern.

[15][ECMA S.24]: Parameter Arrays dürfen nur als letztes Element einer Parameterliste auftreten.

Ein weiterer Nachteil ist, dass der gesamte Code der ersten Methoden-Introduktion verloren geht. Dies würde zu erhöhter Code-Replikation führen, weil Entwickler den benötigten Code in alle Features kopieren müssen. Damit würde ein Teil des Nutzens der verhindert.

Insgesamt ist es dieser Ansatz nicht wert, verfolgt zu werden. Er gibt aber einen guten Einblick in die Möglichkeiten der Methodenkomposition.

Ansatz 2: Konkatenation

Um den Code aus der ersten Methoden-Introduktion nicht zu verlieren, kann man den Code aus beiden Methoden hintereinander in eine neue Methode kopieren. Voraussetzung hierfür ist, dass die Parameternamen der Methoden-Introduktionen übereinstimmen, oder die Referenzen im Code entsprechend angepasst wurden.

Dieser Ansatz ist in einigen Varianten denkbar:

- Code in einen gemeinsamen Codeblock kopieren
- zwei getrennte Codeblöcke (in einem Methodenrumpf)
- eine dritte Methode die beide originale Methoden aufruft.

Bei der ersten Variante bekommt man eventuell Probleme mit Variablennamen, die in beiden Methoden verwendet werden. Dieses Problem wird durch die Variante mit getrennten Blöcken vermieden. Bei der dritten Variante ist es erforderlich die beiden ursprünglichen Methoden umzubenennen. Sie könnten zum Beispiel in „`Methoden-Name_Feature-Name`“ umbenannt werden. So ist relativ sicher, dass die Namen eindeutig und noch nicht verwendet sind.

Bei jeder dieser drei Varianten entstehen zwei Probleme. Zum einen wird ein Feature priorisiert (siehe Ansatz 1). Zum Anderen wird bei einer Methode mit Rückgabewert das `return`-Statement der ersten Methode die Ausführung der zweiten Methode verhindern. Dieses Problem wird in der folgenden Abbildung veranschaulicht.

Feature1

```
public int add(int x, int y) {
    System.Console.Out.WriteLine("Feature1");
    return x + y;
}
```

Feature2

```
public int add(int x, int y) {
    System.Console.Out.WriteLine("Feature2");
    System.Console.Out.WriteLine
        (x + " " + y + " = " + (x+y));
    return x + y;
}
```

Ergebnis

```
public int add(int x, int y) {
    {
        System.Console.Out.WriteLine("Feature1");
        return x + y;
    }
    {
        System.Console.Out.WriteLine("Feature2");
        System.Console.Out.WriteLine
            (x + " " + y + " = " + (x+y));
        return x + y;
    }
}
```

Abb. 17:Methoden Konkatenation

Der gesamte zweite Block des Ergebnisses wird nicht mehr ausgeführt, weil die Methode schon durch das `return`-Statement des ersten Blocks beendet wurde. Wenn man dieses Problem (zum Beispiel mit der dritten Variante) umgeht, muss man sich immer noch entscheiden welchen der beiden Rückgabewerte man zurück gibt. Dieser Ansatz ist wenig flexibel und wirft dennoch enorme Probleme auf.

Ansatz 3: original Schlüsselwort

Im FSTComposer wird in der Artefaktsprache Java Methodenkomposition durch das neue Schlüsselwort `original` realisiert. Vor der Erläuterung dieses Mechanismus sollte man bemerken, dass es grundsätzlich zu vermeiden ist neue Schlüsselwörter in eine Programmiersprache einzuführen. Die Sprachdesigner von C# werden wohl kaum Rücksicht auf unsere Entwicklung nehmen, wenn sie in der nächsten Version der Sprache auch ein `original`– Schlüsselwort einführen wollen. Dieses neue Schlüsselwort wäre dann zunächst mit dem FSTComposer nicht nutzbar.

Das Schlüsselwort wird aufgrund seiner Verwendung von der C#-Grammatik als Methodenaufruf erkannt. Darum muss die Grammatik nicht erweitert werden.

Dies hat den Vorteil die Features (die eventuell `original` enthalten) weiterhin von Tools, die auf der C#-Spezifikation basieren, geparst werden können.

Bei Verwendung des `original`-Ansatzes muss genau eine der Introduktionen das Schlüsselwort im Methodenrumpf enthalten. Dieses Schlüsselwort wird durch den Kompositionsalgorithmus durch den Aufruf der anderen Introduktion ersetzt. So kann man genau steuern an welcher Stelle die andere Introduktion aufgerufen werden soll. Die Introduktion, die das Schlüsselwort nicht enthält, muss nach dem oben beschriebenen Prinzip umbenannt werden. Als Beispiel soll die Abbildung „Methodenkomposition in Java“ (Abb. 16) am Beginn dieses Kapitels dienen.

Es sind noch einige Varianten dieses Ansatzes denkbar. Bisher wurde keine Introduktion priorisiert, weil beide Methoden das Schlüsselwort enthalten dürfen. Es wäre aber möglich das Schlüsselwort etwa nur in der zweiten Introduktion zu erlauben. Damit hätte man die zweite Introduktion priorisiert. Es ergibt sich aber ein bedeutender Vorteil. Wenn das Schlüsselwort in der zweiten Introduktion nicht auftaucht, ergibt sich ohne Änderung des Algorithmus, dass die erste Methode nicht aufgerufen wird[16]. Man könnte also den „Ansatz 1“, die Überschreibung der Methode, integrieren. Diese Möglichkeit hat man nicht, wenn das Schlüsselwort in beiden Introduktionen erlaubt ist, weil man nicht entscheiden kann welche Methode verwendet wird, falls keine das Schlüsselwort enthält. Man erkauft sich also mit dieser Variante des `original`-Ansatzes Flexibilität und gibt dafür die Kommutativität von Methoden-Introduktionen auf.

Aufgrund der Argumentation und weil dieser Ansatz von der Java-Artefaktsprache verwendet wird ist die letzte Variante der günstigste Kompositionsalgorithmus für Methoden-Introduktionen. Weil sich bei Verwendung dieser Variante die Artefaktsprachen Java und C# ähnlich verhalten, ist der FSTComposer-spezifische Lernaufwand sehr gering.

2.2.7 Typparameter

Typparameter werden in C# bei der Deklaration von Klassen und Methoden verwendet[17]. Weil an jeder Stelle des FST nur eine Typparameter-Liste existieren darf, vergleicht die *equals* Relation wie schon bei den Modifiern nur den Pfad. Weil Listen keinen Namen haben wird der Name „<NO_NAME>“ vergeben.

$$equals_{TypeparameterList}(x, y) \in TypeparameterList^2 \rightarrow (x.Path = y.Path)$$

Die *composable* Relation hängt sehr davon ab wie restriktiv man die Komposition formuliert. Man könnte zulassen, dass zwei teilweise verschiedene Typparameter-Listen komponiert werden, indem sie in eine Liste kopiert werden und doppelte Elemente entfernt werden. So hätte man in jedem Fall wieder eine gültige Typparameter-Liste, weil die Reihenfolge der Typparameter irrelevant ist.

Falls man jedoch mit einem Feature zum Beispiel an eine Methode neue Typparameter anfügt, sind alle Aufrufe dieser Methode im anderen Feature ungültig, weil die Typparameter Teil der Methodensignatur sind. Dieses Verhalten ist zu vermeiden, weil aus zwei in sich korrekten Features ein inkorrektes Feature resul-

[16]Die erste Methode muss in diesem Fall nicht in das Ergebnis eingefügt werden

[17]Die Typparameterangaben beim Aufruf von Methoden/ Instantiierung von Klassen werden nicht behandelt, weil sie im FST nicht als eigene Knoten auftauchen.

tieren würde. Die Korrektheit bzw. die Existenz von solchen Aufrufen kann anhand des FST nur schlecht überprüft werden.

Wir definieren also, restriktiv, dass nur Typparameter-Listen komponiert werden können die identisch sind. Die Reihenfolge der Parameter ist wichtig, weil bei der Instantiierung einer typparametrisierten Methode oder Klasse die Typparameter in dieser Reihenfolge angegeben werden müssen.

$$composable_{TypeparameterList}(x, y) \in TypeparameterList \times TypeparameterList \rightarrow (x \subseteq y) \wedge (y \subseteq x)$$

Der Kompositionsalgorithmus für Typparameter-Listen erübrigt sich damit, weil die Listen identisch sind. Es kann eine beliebige der beiden Introduktionen in das Ergebnis übernommen werden.

Falls es dem Entwickler überlassen werden soll, die Korrektheit des resultierenden Codes sicherzustellen, kann man eine Erweiterung der Typparameter zulassen. Es könnte zum Beispiel erlaubt sein neue Typparameter zu einer Typparameterliste hinzuzufügen. Im Code müssten in diesem Fall (vom Entwickler) alle Referenzen auf die erweiterte Klasse oder Methode angepasst werden. Dies ermöglicht einige neue Möglichkeiten in der Gestaltung von Features. Es würde die C#-Programmierung mit dem FSTComposer flexibler machen. Aufgrund der obigen Argumentation haben wir uns aber gegen diese Möglichkeit entschieden.

2.2.8 Typparameter-Einschränkungen

Die Einschränkung von Typparametern wird in C# durch das `where`-Schlüsselwort vorgenommen.

```
class Dictionary<KeyType, ElementType> where KeyType: IComparable {}
```

Abb. 18: Typparameter-Einschränkungen

ECMA-334

Die *equals* Relation muss mit der bekannten Begründung (nur höchstens einmal pro FST-Ebene erlaubt) festgelegt werden:

$$equals_{TypeparameterConstraintsList}(x, y) \in TypeparameterConstrainsList \times TypeparameterConstrainsList \rightarrow (x.Path = y.Path)$$

Die Reihenfolge der Listenelemente ist wiederum irrelevant. Die Liste wird also als ungeordnete Menge betrachtet. Die Möglichkeiten für Kompositionsalgorithmen sind dieselben wie bei den Typparametern im letzten Kapitel. Entweder die Liste darf erweitert werden oder es können nur identische Listen komponiert werden. Für die Argumentation wird die Dictionary-Deklaration aus der obigen Abbildung verwendet. Wenn in einem Feature1 die Deklaration ohne die Typparameter-Einschränkung enthalten ist, kann in diesem Feature auch ein Objekt das `IComparable` nicht implementiert eingefügt werden. Durch die Erweiterung der Deklaration um die Einschränkung in Feature2 kann dieses Objekt nicht mehr eingefügt werden. Der Code wird also inkorrekt.

Aus diesem Grund werden wieder nur identische (bis auf die Reihenfolge) Listen zugelassen.

$$composable_{TypeparameterConstraintsList}(x, y) \in$$
$$TypeparameterConstraintsList \times TypeparameterConstraintsList$$
$$\rightarrow (x \subseteq y) \wedge (y \subseteq x)$$

Der Kompositionsalgorithmus für identische Listen ist trivial und wurde schon mehrfach beschrieben.

Es wäre auch bei Typparameter-Einschränkungen möglich dem Entwickler mehr Flexibilität zu erlauben. So könnte man die Spezialisierung von Einschränkungen erlauben. Falls in einem Feature ein Typparameter auf einen Obertyp eingeschränkt wird, könnte man erlauben, dass er durch die Kombination auf einen Untertyp eingeschränkt wird. Dafür bräuchte man eine Möglichkeit die Klassenhierarchie zu prüfen um die Ober-/ Untertyp Beziehung festzustellen. Der statische Typ von eingeschränkten Typparametern ist auf die Einschränkung festgelegt (ohne Einschränkung ist er `object`). Falls man also die Einschränkung spezialisiert, ändert man auch den Typ, der in beiden Features verwendet wird. Das kann z. B. bei unsauberer[18] Überladung von Methoden zu unerwartetem Verhalten führen. Aufgrund dieser Überlegungen wird im FSTComposer zunächst der oben definierte Kompositionsalgorithmus verwendet.

2.2.9 Konstruktoren

Im Wesentlichen sind Konstruktoren in C# einfache Methoden. Es wird also versucht die Relationen und den vierten Kompositionsalgorithmus von Methoden zu übernehmen. Es gibt nur drei Unterschiede zwischen Methoden und Konstruktoren:

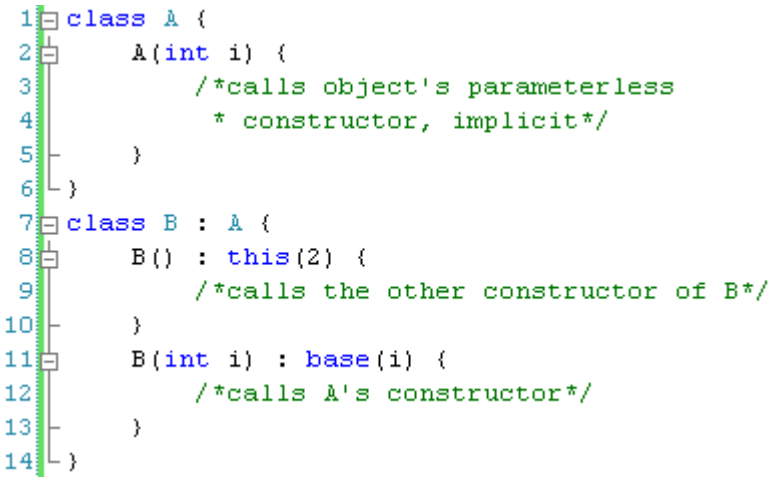
- Konstruktoren haben keinen Rückgabetyp
- Konstruktoren erlauben keine Typparameter und -Einschränkungen
- Konstruktoren können definieren welcher Konstruktor zu Beginn des Konstruktors ausgeführt werden soll[19]

Die ersten beiden Punkte stellen keine Schwierigkeit dar. Die fehlenden Eigenschaften fallen einfach in allen Relationen die für Methoden definiert wurden weg.

Wie deklariert werden kann welcher Konstruktor als Initialisierung dienen soll zeigt die folgende Abbildung:

[18] Falls die überladende Methode nicht die Spezifikation der überladenen Methode erfüllt.

[19] Die so definierte Aufrufhierarchie endet im parameterlosen Konstruktor von `object`



```
class A {
    A(int i) {
        /*calls object's parameterless
         * constructor, implicit*/
    }
}
class B : A {
    B() : this(2) {
        /*calls the other constructor of B*/
    }
    B(int i) : base(i) {
        /*calls A's constructor*/
    }
}
```

Abb. 19:Konstruktor-Initialisierung

Konstruktoren können genau wie Methoden behandelt werden, mit der zusätzlichen Einschränkung, dass die Initialisierungen von zu komponierenden Konstruktoren übereinstimmen müssen.

In der Artefaktsprache Java war diese Möglichkeit nicht gegeben. In Java werden Konstruktoren immer mit dem Kompositionsalgorithmus 1 „einfaches Überschreiben“ komponiert. Der Grund dafür ist, dass in der ersten Zeile eines Java-Konstruktors (also im Konstruktor-Rumpf) die Initialisierung vorgenommen werden kann.

In dem Kapitel zu Methoden wurde bemerkt, dass der Kompositionsalgorithmus 3 den Algorithmus 1 „enthält“. Bei Verwendung des Algorithmus 3 können also Konstruktoren immer noch wie in Java geschrieben werden. Es ist aber auch möglich das `original`-Schlüsselwort zu verwenden. An dieser Stelle ist die Komposition in C# also flexibler als in Java und gleichzeitig kompatibel zu der Komposition in Java. Die Relationen werden also analog zu den Relationen für Methoden definiert. Außerdem sollte der Kompositionsalgorithmus 3 verwendet werden.

$$equals_{Constructor}(x, y) \in Constructor \times Constructor \rightarrow (x.Path = y.Path) \\ \wedge\, y.InputParamters = x.InputParameters$$

$$composable_{Constructor}(x, y) \in Constructor \times Constructor \rightarrow (x.Path = y.Path) \\ \wedge\, y.InputParamters.Names = x.InputParameters.Names \\ \wedge\, y.Initialisation = x.Initialisation$$

Zu der Komposition mit dem vierten Kompositionsalgorithmus ist noch zu bemerken, dass die Konstruktor-Introduktion, die das `original`-Schlüsselwort enthalten hat, im Ergebnis wieder ein Konstruktor ist. Die andere Introduktion kann jedoch kein Konstruktor sein, weil man Konstruktoren nicht umbenennen kann. Der zweite Konstruktor muss also in eine Methode (mit anderem Namen) umgewandelt werden. Diese Methode wird anstelle des `original`-Schlüsselwortes im Ergebnis-Konstruktor aufgerufen.

2.2.10 Enums

Enums werden in C# über ihren Namen identifiziert. Die *equals* Relation muss also prüfen ob diese Eigenschaft übereinstimmt.

$$equals_{Enum}(x, y) \in Enum \times Enum \rightarrow (x.Path = y.Path)$$

Den Elementen des Enums werden numerische Werte zugewiesen. Dies kann implizit vom Compiler oder explizit in der Deklaration erfolgen. Außerdem kann man den Typen der Werte (`int`, `byte`, ...) festlegen. Ein deklariertes Enum sieht dann aus wie folgt:

```
1  enum directions : byte
2  { Left, Right1 = 2, Right2 = 2,
3      Top = 5, Bottom }
4  /* Values:
5  * Left = 1
6  * Right1 = Right2 = 2
7  * Top = 5
8  * Bottom = 6
9  */
```

Abb. 20:Enum Deklaration

Es kann auch mehreren Elementen derselbe Wert zugewiesen werden. Sie sind dann gleich. In Deklarationen von Enums können noch einige weitere Festlegungen getroffen werden [ECMA, AA07]. Diese Festlegungen sind aber in diesem Zusammenhang nicht relevant.

Zu komponierende Enums müssen mit demselben Typ deklariert sein. Falls kein Typ angegeben ist, wird implizit `int` angenommen [ECMA].

Die Komposition von zwei Operanden-Enums wird im Wesentlichen durch Zusammenfügen der Enum-Elemente durchgeführt. Dabei werden mehrfach vorkommende Elemente entfernt. Durch diese Vorgehensweise können zwei Probleme auftreten. Falls ein Enum-Element (über den Namen identifiziert) in den Operanden verschiedene Werte zugewiesen bekommt, können die Operanden nicht komponiert werden. Die Komposition muss abgebrochen werden.

Nun kann die *composable* Relation definiert werden.

$$composable_{Enum}(x, y) \in Enum \times Enum \rightarrow (x.Path = y.Path) \wedge (y.Type = x.Type)$$
$$\wedge \forall ex \in x \; \forall ey \in y$$
$$(((ex.Name = ey.Name) \wedge ex.hasNum \wedge ey.hasNum) \Rightarrow (ex.Num = ey.Num))$$

Das zweite Problem entsteht wenn der Entwickler im restlichen Programm auf die Nummerierung der Enum-Elemente zugreift. Die implizit nummerierten Elemente haben eventuell nach der Komposition eine andere Nummer als davor. Dieses Problem kann vom FSTComposer nicht gelöst werden. Der Entwickler muss das bei der Konzeption des Programms berücksichtigen.

Die folgende Abbildung zeigt die Auswirkungen dieses Problems.

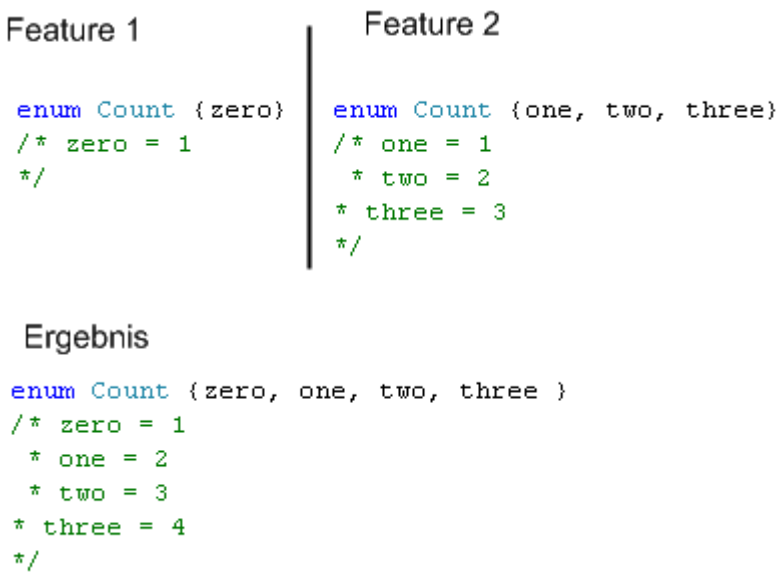


Abb. 21: Geänderte Enum Nummerierung

Die Nummerierung der Elemente `one`, `two` und `three` aus dem Feature 2 konnte nicht beibehalten werden.

2.2.11 using-Anweisungen

Das `using`-Schlüsselwort wird in C# genutzt um Elemente aus anderen Klassen und Namespaces in der eigenen Klasse oder dem eigenen Namespace bekannt zu machen. Im Gegensatz zu Deklarationen werden durch `using` keine neuen Elemente definiert.

Das `using`-Schlüsselwort funktioniert im Wesentlichen wie `import` in Java. Man kann aber auch mehrere Elemente in einer `using`-Anweisung zusammenfassen und neue „Aliase“ für Elemente vergeben. Ein Element, das mit Alias importiert wurde, kann im betreffenden Sichtbarkeitsbereich durch das Alias referenziert werden.

Durch `using`-Konstrukte, die mehrere Elemente bekannt machen, entstehen dieselben Probleme wie bei mehrfachen Feld-Deklarationen. Die einzige mögliche Lösung ist auch hier vor der Komposition alle Mehrfachanweisungen in einzelne Anweisungen umzuwandeln.

In Bezug auf die Aliase können bei Operanden mit gleichen Namen[20] verschiedene Fälle auftreten:

[20]Die Operanden importieren dasselbe „fremde“ Element oder denselben Namespace

Situation	Ergebnis
Beide Operanden haben kein Alias	Kein Alias
Genau ein Operand hat ein Alias	Beide Operanden werden übernommen
Beide Operanden haben das gleiche Alias	Einer der Operanden wird übernommen
Beide Operanden haben verschiedene Aliase	Beide Operanden werden übernommen

Tab. 3: Fallunterscheidungen bei der Kombination von using-Anweisungen

Diese Behandlung ist notwendig, weil in beiden Features die Aliase genutzt werden können. Im Ergebnis müssen also alle Aliase verfügbar sein, die in den Features vorhanden sind.

Die Operanden müssen nur komponiert werden falls genau ein Operand ein Alias hat oder beide verschiedene Aliase haben. Die *equals* Relation wird so definiert, dass sie andernfalls *false* ergibt. Die Operanden werden dann nicht komponiert, sondern automatisch beide in das Ergebnis übernommen.

$$equals_{using}(x, y) \in using \times using \rightarrow (x.Path = y.Path) \wedge (x.Alias = y.Alias)$$

Nachdem *equals* zu *true* ausgewertet hat, ist klar die Operanden dasselbe Alias haben. Darum sind sie identisch und können immer komponiert werden, indem einer der Operanden in das Ergebnis übernommen wird.

Die *composable* Relation hat also immer *true* als Ergebnis.

$$composable_{using}(x, y) \in using \times using \rightarrow true$$

2.2.12 Properties

Eine Property in C# ist eine Komposition aus einem Feld und einer (oder zwei) Methoden (im Folgenden „Accessors“). Weil eine Property von außerhalb der Klasse, in der sie deklariert ist, wie ein normales Feld wirkt, ist es nicht verwunderlich, dass sie exakt so identifiziert wird wie ein Feld. Die Definition der *equals* Relation kann von den Feldern übernommen werden.

$$equals_{property}(x, y) \in property \times property \rightarrow x.Path = y.Path$$

Die *composable* Relation kann ebenfalls von den Feldern übernommen werden. Hier ist nur die Relation ohne Berücksichtigung der Vererbungshierarchie möglich. Die Relation mit Vererbungshierarchie könnte zu Problemen in den Accessors führen[21]. Außerdem haben Properties keine Initialisierungen. Darum kann diese Eigenschaft aus der Relation entfernt werden.

$$composable_{Property}(x, y) \in property \times property \rightarrow (x.Type = y.Type)$$

[21]Die Accessors könnte evtl. einen falschen Typ zurückgeben oder als Parameter erwarten.

Komposition Die Schwierigkeit der Komposition von Properties wird erst deutlich, wenn man versucht die Property-Methoden (im Folgenden „Accessors“) zu komponieren. Accessors haben viel weniger Eigenschaften als normale Methoden. Es existieren keine beliebigen Namen (nur `get` und `set`), keine Typparameter und keine formalen Parameter. Accessors können allerdings mit einem Zugriffsmodifier deklariert werden. So könnte etwa die set-Methode private und die get-Methode public gemacht werden. Die Accessor-Zugriffsmodifier müssen restriktiver sein als die der zugehörigen Property.

Zur Komposition von zwei Property-Introduktionen werden zunächst die „Header“ der Introduktionen komponiert. Diese Komposition kann auf die Komposition von Feldern zurückgeführt werden und läuft ebenso ab.

Weiterhin müssen die Accessors komponiert werden. Eine Property kann maximal einen get-Accessor und einen set-Accessor haben. Falls einer der Accessors nur in einer der Introduktionen vorhanden ist, wird er ohne Änderung in das Ergebnis übernommen.

Falls ein Accessor-Typ (get oder set) in beiden Introduktionen vorhanden ist, werden die Accessors komponiert. Dafür wird eine Variante des von Methoden bekannten Kompositionsalgorithmus 3 verwendet. Der Accessor, der kein `original` enthält, wird in eine normale Methode der enthaltenden Klasse umgewandelt. Diese Methode erhält den Zugriffsmodifier des Accessors[22] und alle Nicht-Zugriffsmodifier der Property. Die restlichen Eigenschaften der zu erzeugenden Methode (Rückgabetyp und Parameter) sind in der folgenden Tabelle aufgeführt:

Accessor-Typ	Rückgabetyp	Parameter
`get`	<Property-Typ>	
`set`	`void`	<Property-Typ> `value`

Tab. 4: Eigenschaften von komponierten Property-Accessors

Das `original`-Schlüsselwort im nicht umgewandelten Accessor muss durch den Aufruf der neu erzeugten Methode ersetzt werden.

Problem

Für jede Property werden automatisch zwei Signaturen in der enthaltenden Struktur reserviert. Dazu ein Zitat aus der ECMA-334 [ECMA]:

> 17.2.7.1 Member names reserved for properties
>
> For a property P (§17.6) of type T, the following signatures are reserved:
>
> ```
> T get_P();
> void set_P(T value);
> ```

Abb. 22: Für Properties reservierte Namen

ECMA-334 S. 276

[22] Falls der Accessor keinen Zugriffsmodifier hat, erhält die Methode den Zugriffsmodifier der Property.

Aus diesem Grund kann es zu Konflikten kommen, wenn eine Property mit z. B. einer Klasse komponiert wird, in der bereits eine der Signaturen vorhanden ist. In diesem Fall wären beide Features in sich korrekt. Das Ergebnisfeature würde jedoch die oben genannte Bedingung verletzen. Dieser Konflikt kann vom FSTComposer nicht einfach erkannt werden. Es wird dem Entwickler überlassen solche Situationen zu vermeiden.

2.2.13 Delegates

Delegates stellen eine Art Platzhalter für Methoden dar. Ihre Deklaration enthält dieselben Informationen wie eine Methodendeklaration. Delegates können jedoch nicht wie Methoden überladen werden. Darum können sie nur über ihren Namen und die Zahl der Typparameter identifiziert werden. Die formalen Parameter kommen also in der *equals* Relation nicht vor.

$$equals_{Delegate}(x, y) \in Delegate \times Delegate \rightarrow (x.Path = y.Path) \\ \wedge y.TypeParameters.count = x.TypeParameters.count$$

Die *composable* Relation stimmt im Wesentlichen mit der Relation für Methoden überein. Es kann jedoch eine kleine Lockerung vorgenommen werden. Die Namen der Input-Parameter müssen nicht unbedingt übereinstimmen (die Typen-Reihenfolge der Input-Parameter jedoch schon). Der Grund hierfür ist, dass die Parameter-Namen von der implementierenden Methode nicht übernommen werden müssen. Dies wird in der folgenden Abbildung gezeigt.

```
delegate int IntegerTransformation(int i);
void increment(int x) {
    return x + 1;
}
void test() {
    IntegerTransformation a = increment;
    /* Assignment is valid
     * though the delegate's
     * ParameterName ("i") differs from
     * the method's ("x")
     */
    a(1); // = 2
}
```

Abb. 23:Delegate Parameter-Namen

$$composable_{Delegate}(x, y) \in Delegate \times Delegate \rightarrow \\ y.InputParameters = x.InputParameters \quad (\text{Typen-Reihenfolge}) \\ \wedge y.TypeParamters = x.TypeParameters \\ \wedge y.TypeParamterConstraints = x.TypeParameterConstraints$$

Delegates enthalten (im Gegensatz zu Klassen-Methoden) keine Implementierung[23]. Dadurch wird der Kompositionsalgorithmus ungleich einfacher. Es wurde

[23]Lambda-Funktionen werden nicht unterstützt, weil in der verwendeten Spezifikation [ECMA] noch nicht eingeführt wurden.

bereits sichergestellt, dass die Delegate-Introduktionen in ihren Parameter-Typen, in der Reihenfolge ihrer Parameter-Typen und im Rückgabetypen übereinstimmen. Es müssen also nur noch eine der Introduktionen übernommen werden und die Modifierlisten komponiert werden. An dieser Stelle ist es wichtig, das C# keine benannten Argumente erlaubt, wie zum Beispiel Visual Basic.NET[24]. Für die Identifikation eines benannten Arguments würde dann der Name in der Parameterliste des Delegate verwendet, was zu Problemen beim Aufruf des Delegate führen würde. In C# werden Parameter ausschließlich über ihre Reihenfolge zugeordnet. Darum ist der Name des Delegate-Parameters nicht wichtig.

2.2.14 Weitere Konstrukte

In diesem Kapitel wird die Behandlung von weitergehenden C# Sprachkonstrukten diskutiert. Die meisten dieser Konstrukte lassen sich durch andere, bereits beschriebenen Konstrukte ersetzen. Sie fügen der Sprache also keine weitere Mächtigkeit hinzu. Dieser „syntaktische Zucker“ macht das Programmieren aber einfacher und hilft Komplexität und damit Fehler zu vermeiden. Auf eine detaillierte Beischreibung der Sprachkonstrukte muss hier aus Platzgründen verzichtet werden. Für weitere Informationen werden [ECMA] und [AA07] empfohlen.

Indexer

Ein Indexer ermöglicht es einem Objekt einen Array-ähnlichen Zugriff bereitzustellen. Die Deklaration von Indexern ist sehr ähnlich zu Property-Deklarationen. Die Unterschiede sind lediglich das Indexer keinen Namen haben (bzw. der Name ist immer `this`) und das Indexer eine Parameterliste haben können. Daraus lassen sich auch die Relationen für Indexer ableiten. Die *equals* Relation kann exakt von den Properties übernommen werden. Die *composable* Relation muss so erweitert werden, dass die Parameterlisten übereinstimmen.

$$composable_{Indexer}(x, y) \in Indexer \times Indexer \rightarrow (x.Type = y.Type) \wedge y.ParameterList = x.ParameterList$$

Bei der Komposition ist zu beachten, dass die Indexer-Methodenparameter bei den neu entstehenden Methoden als Parameter deklariert werden müssen. Der übrige Kompositionsalgorithmus ist identisch zu dem der Properties.

Events

Ein Event ermöglicht es einem Objekt Benachrichtigungen an andere Objekte bereitzustellen. Die Empfänger müssen bei dem Event eine Methode (eines bestimmten Delegate-Typs) registrieren. Wenn das Ereignis eintritt, werden alle registrierten Methoden ausgeführt. Durch diese Konstruktion kann das Observer Pattern aus [GHJ+95 Seite 293ff] sehr einfach implementiert werden.

Ein Beispiel für die Verwendung von Events findet sich in [ECMA S.36]. Die Deklaration eines Events ist syntaktisch identisch zur Deklaration eines Feldes mit zusätzlichem Schlüsselwort `event`. Alle Relationen und Kompositionsalgorithmus können von den Feldern übernommen werden.

[24] [VB9 Kapitel 11.8, Seite 214, Beispiel auf Seite 54]

Operatoren

Operatordeklarationen definieren die Bedeutung eins Operators, der auf die betreffende Struktur (Klasse, ...) angewandt werden kann. In der Deklaration von Operatoren hat der Entwickler viele Freiheiten. Es können unäre, binäre und Konversionsoperatoren definiert werden.

Um zwei Operatoren komponieren zu können müssen die Deklarationen der Operator-"Köpfe" (Typ des Operators, Parameter) identisch sein. Die restliche Komposition ist auf die Komposition von Methoden zurückzuführen.

Exceptions

In C# ist es genau wie in Java möglich Ausnahmebehandlung zu verwenden. Die in einer Methodendeklaration angegebenen Exceptions können analog zu formalen Methodenparametern behandelt werden.

Finalizer (früher Destruktoren)

In früheren Versionen der C#-Spezifikation wurde der Begriff Destructor verwendet. In der ECMA-334 wird dieser durch Finalizer ersetzt. Ein Finalizer ist eine spezielle Methode, die nur vom Garbage-Collector aufgerufen wird. Weil ein Finalizer nur eine (stark eingeschränkte) Methode ist, können alle Relationen der Methoden auf Finalizer übertragen werden. Einige Eigenschaften die Finalizer nicht haben (Namen, Parameter, Modifier, ...) müssen aus den Relationen entfernt werden.

2.2.15 Generalisierung der Equals-Relationen

In [3.1] wurde bemerkt das auch z. B. Klassen und Interface-Introduktionen, die auf derselben FST-Ebene auftauchen, bezüglich *equals* gleich sein müssen, falls sie denselben Namen und dieselbe Typparameter-Anzahl haben. Dies lässt sich verallgemeinerten *equals* Relationen ausdrücken.

Die von diesem Umstand betroffenen Konstrukte lassen sich in zwei Gruppen teilen. Die erste Gruppe umfasst alle *Typdeklarationen* (Klassen, Interfaces, Enums, Structs und Delegates). Die zweite Gruppe umfasst die *Memberdeklarationen* die in manchen Typen auftauchen können (Feld, Methode, Property).

Bei Betrachtung der *equals* Relationen der Typdeklarationen fällt auf, dass die Relationen fast identisch sind. Weil Enums keine Typparameter haben, ist *Typparameters.count* bei Enums immer 0.

$$equals_{TypeDeclaration}(x, y) \in TypeDeclaration \times TypeDeclaration \rightarrow x.Path = y.Path \\ \wedge x.Typeparameters.count = y.Typeparameters.count$$

Diese Relation wird nun für alle Typdeklarationen gemeinsam genutzt um zu erzwingen, dass zwei Typdeklarationen, die sich in Namen und Typparameter-Anzahl nicht unterscheiden, *equals* erfüllen. Die *composable* Relation muss natürlich die Knotentypen vergleichen und bei unterschiedlichen Knotentypen *false* ergeben.

Bei den Memberdeklarationen wird diese Verallgemeinerung durch die Überladung von Methoden erschwert. Felder und Properties überprüfen in der *equals* Relation nur den Namen/ Pfad. Methoden prüfen zusätzlich noch die formalen- und Typparameter. Die allgemeine Relation ist also:

$$equals_{MemberDeclaration}(x, y) \in MemberDeclaration \times MemberDeclaration \rightarrow$$
$$x.Path = y.Path$$
$$\wedge((x, y) \in Method^2)$$
$$\Rightarrow (x.Typeparams = y.Typeparams \wedge x.InputParams = y.InputParams)$$

Durch die Ersetzung der spezialisierten *equals* Relationen durch diese verallgemeinerten wird sichergestellt, dass Namen im Ergebnisquellcode nicht illegal mehrfach verwendet werden.

Es kann nicht sichergestellt werden, dass `using`-Anweisungen oder deren Aliase von Typnamen überlagert werden. Zum Beispiel kann eine Klassendeklaration aus einem Feature eine `using`-Anweisung in einem anderen Feature überlagern (verstecken) falls diese Features kombiniert werden. Die Behandlung dieses Umstandes muss dem Entwickler überlassen werden.

2.3 Modifikation

Modifikation kann in C# genau so umgesetzt werden wie es in der theoretischen Beschreibung der Algebra dargestellt wurde (Kapitel [2.4.3]). Es muss eine Möglichkeit geschaffen werden Elemente eines FST auszuwählen („Query“). Dies kann zum Beispiel über eine XML-Datenstruktur realisiert werden. In dieser werden die Typen und Namen der Elemente eines FST-Pfades ausgewählt. Dabei kann festgelegt werden welches Pfadelement wie geändert werden soll („Change“).

Der Query-Teil einer Modifikation enthält also den Namen und Knotentypen der ausgewählten Knoten sowie eventuelle Unterknoten. Dabei können Namen und Typen auch mit regulären Ausdrücken angegeben werden. Dadurch ist es auch möglich mehrere Knotentypen auszuwählen (reguläre Ausdrücke erlauben einen „oder“-Operator).

Der Change-Teil einer Modifikation muss definieren welcher Teil des Knotens ersetzt oder geändert werden soll. Dabei können eventuell auch ganze Unterbäume geändert werden. Einfache Modifikationen wie die die Änderung eines Methodennamens sollten direkt in der Modifikation-Datei spezifiziert werden (eventuell direkt bei der Query). Für umfassende Änderungen bei der ganze neue Unterbäume eingeführt werden kann ein Verweis auf eine externe Definition genutzt werden.

3. Implementierung

In diesem Kapitel wird zunächst beschrieben wie der FSTComposer vor Beginn dieser Arbeit funktioniert hat und welche Änderungen im Rahmen dieser Arbeit vorgenommen wurden. Außerdem wird auf die Implementierung der im vorangegangenen Kapitel gewonnenen Ergebnisse eingegangen. Der FSTComposer wurde erweitert, um die Komposition von Introduktionen in der Artefaktsprache C# zu unterstützen. Die Modifikation von C#-FSTs wurde aufgrund des beschränkten Umfangs dieser Arbeit nicht implementiert. Welche Schritte dafür nötig wären wurde bereits im Kapitel [3.3] erläutert.

3.1 Ausgangszustand des FSTComposer (Version 0.2)

Zum einfacheren Verständnis des FSTComposer wird zunächst der grobe Ablauf einer Komposition erklärt. Im Anschluss wird darauf eingegangen, wie dieser Ablauf implementiert wurde und wie der FSTComposer erweitert werden kann.

Zur Komposition von Features sind drei Schritte nötig: Umwandlung der als Dateien vorliegenden Features in FSTs, Komposition der FSTs und Umwandlung der FSTs in Dateien der Artefaktsprache.

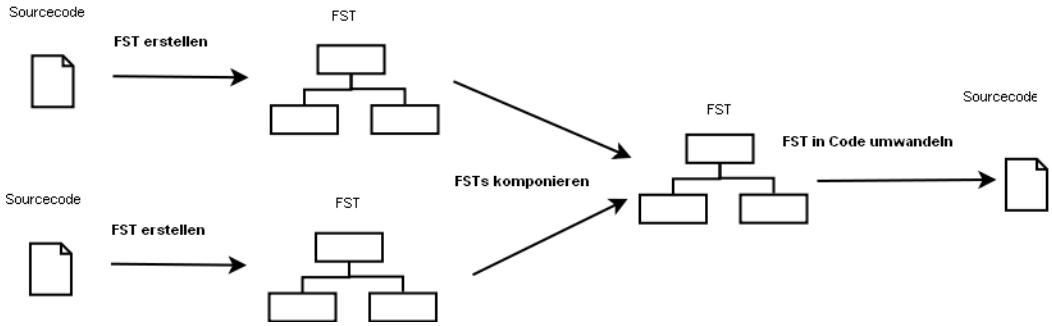


Abb. 24: Funktionsprinzip FSTComposer

Dieser Ablauf wurde im FSTComposer implementiert. Die hierfür wesentlichen Klassen werden im folgenden (schematischen) UML-Diagramm dargestellt (Abb. 25).

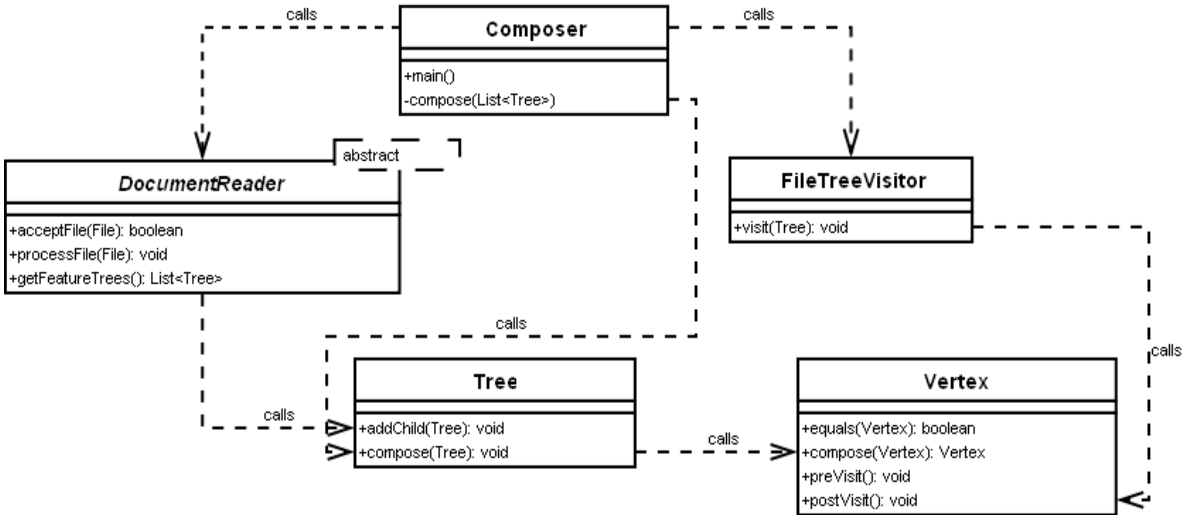


Abb. 25: Vereinfachtes UML-Diagramm

Der Programmablauf wird hier aus Platzgründen nur kurz erläutert. Zunächst ruft die Hauptklasse (`Composer`) alle registrierten `DocumentReader` auf (für jede Artefaktsprache existiert ein `DocumentReader`). Diese `DocumentReader` erstellen FSTs aus den ausgewählten Dateien. Die Datenstruktur der FSTs ist in den Klassen `Tree` und `Vertex` gespeichert.

Im nächsten Schritt werden die gesammelten FSTs komponiert. Die zentrale Methode zur rekursiven Komposition ist in der Klasse `Tree` enthalten. Sie ruft jeweils die Methoden `equals` und `compose` in `Vertex` auf. Diese Methoden realisieren die in der Feature-Algebra definierten Relationen. Falls *composable*

zu *true* auswertet, gibt `compose` den neuen, komponierten `Vertex` zurück, andernfalls einen Fehler.

Am Ende des Kompositionsprozesses wurden alle FSTs in einen FST komponiert. Dieser FST muss wieder in eine Datei der Artefaktsprache umgewandelt werden. Zu diesem Zweck wird der Baum mit dem `FileTreeVisitor` rekursiv abgelaufen und jeder `Vertex` wird vor dem Aufruf seiner Unterknoten (`preVisit`) und danach (`postVisit`) aufgerufen. In diesen Methoden werden syntaktische Informationen in die Ergebnisdatei eingefügt. Da Terminalknoten des FST keine Unterknoten haben, geben sie einfach ihren Inhalt (etwa den Namen eines Enum-Elements) in der `preVisit`-Methode aus.

3.2 Erweiterungen am FSTComposer (Kernprogramm)

Zur Vorbereitung der Integration der Artefaktsprache C# wurde der letzte Schritt aus der obigen Abbildung [Abb. 24] (FST in Code umwandeln) erweitert. Mit der bisherigen Implementierung konnte bei jedem Knoten nur an zwei Stellen in den Ausgabeprozess eingegriffen werden (vor und nach der Ausgabe der Unterknoten). Außerdem konnte kein Einfluss auf die Reihenfolge der Unterknoten genommen werden.

Dies ist zum Beispiel bei Namespaces sehr wichtig. Zur Vereinfachung wird hier angenommen, dass Namespaces nur `using`-Anweisungen und Klassen enthalten können. Die C#-Spezifikation schreibt vor, dass alle `using`-Anweisungen vor den Klassendeklarationen stehen müssen. Falls man nun einem Namespace-FST, der bereits eine Klasse enthält, eine `using`-Anweisung hinzufügt, steht sie im FST hinter der Klasse. Sie wird also bei einem einfachen Baumdurchlauf auch nach der Klasse ausgegeben. Das würde zu einem inkorrekten Ergebnisnamespace führen.

Um dieses Problem zu lösen wurden die Kind-Knoten jedes FST-Knotens (jedes `Tree`-Objekts) in Gruppen geordnet. So erhält zum Beispiel ein Namespace-Knoten eine Gruppe `using`-Anweisungen und eine Gruppe mit sonstigen Deklarationen (Klassen, Interfaces, ...). Die Gruppen sind geordnet und jede Gruppe steht für eine Position in der Deklaration (in Artefaktsprachen-Syntax) des FST-Knotens. So kann jedes Element der Deklaration an der richtigen Position (in die richtige Gruppe) eingefügt werden.

Durch diese Erweiterung ist die Struktur der FSTs übersichtlicher und besser zu dokumentieren. Außerdem kann ein Knoten nach der Behandlung jeder Unterknoten-Gruppe in den Ausgabeprozess eingreifen (nicht wie bisher nur an zwei Zeitpunkten). Für diese Arbeit wurde die Zahl der Gruppen auf zehn begrenzt. Es ist aber ohne weiteres denkbar die Zahl zu erhöhen oder dynamisch zur Laufzeit den Anforderungen anzupassen.

Als Konsequenz aus den entwickelten Relationen *equals* und *composable* wurde außerdem eine neue Exception (`NotComposableException`) eingeführt. Diese Exception tritt auf wenn der Fall $equals = true \wedge composable = false$ eintritt (Kapitel [2.4]).

3.3 Erweiterung des FSTComposer um C# als neue Artefaktsprache

In diesem Kapitel wird beschrieben wie C# als neue Artefaktsprache in den FSTComposer integriert wurde. Zu Beginn wird die Artefaktsprache definiert. Die weitere Gliederung des Kapitels orientiert sich an den Arbeitsschritten des FSTComposer. Diese Struktur wurde in der Abbildung 24 dargestellt.

3.3.1 Definition der akzeptierten C# Artefaktsprache

Aufgrund der Untersuchungen in Kapitel 3 werden einige Einschränkungen an der Artefaktsprache nötig. Es kann nicht die gesamte Sprache C# als Eingabesprache akzeptiert werden. Die Menge der vom FSTComposer akzeptierten C#-Programme wird im Folgenden definiert. Als Ausgangsmenge wird die Menge der korrekten C#-Programme verwendet. Diese Menge wird weiterhin eingeschränkt und ergänzt:

- `original` darf wie in [3.2.6] definiert verwendet werden. Es wird hiermit als neues Schlüsselwort eingeführt. `original` darf im Quelltext nicht als Identifier oder Teil eines Identifiers verwendet werden.[25] Der Grund für diese Einschränkung wird in [4.3.3] gegeben.
- Das Schlüsselwort `partial` wird ausgeschlossen.

Außerdem werden einige C#-Sprachkonstrukte in dieser Version des FSTComposer nicht implementiert. Die folgenden Konstrukte werden also von der Menge ausgeschlossen.

- Indexer
- Events
- Operatoren
- Exceptions
- Finalizer
- Präprozessordirektiven

Aufgrund von Problemen mit dem C#-Parser (im nächsten Kapitel beschrieben) können die Namen „i“ und „in“ nicht geparst werden. Der Parser versucht sie als das Schlüsselwort „int“ zu erkennen und bricht mit einem Fehler ab. Außerdem werden die Zeichen „>>“ als der Shift-Operator erkannt. Bei der Verwendung zur Definition von geschachtelten Typparametern muss ein Leerzeichen zur Trennung eingefügt werden.

Es gibt noch weitere, einzelne C#-Sprachelemente, die von der Grammatik nicht erkannt werden (z. B. der `??`-Operator). Der Fokus dieser Arbeit liegt jedoch nicht darauf die gesamte Sprache C# abzudecken, sondern die für die Programmierung und Featurekomposition wesentlichen Elemente zu behandeln. Besonders die Sprachelemente, die innerhalb einer Methode auftreten können (z. B. der `??`-Operator), sind für die Featurekomposition uninteressant, weil der Methodenrumpf nicht detailliert betrachtet wird.

3.3.2 Erstellen des Feature-Structure-Tree

Der erste Schritt des Programmablaufs des FSTComposer ist die Umwandlung der C#-Dateien (.cs) in FSTs. Dies wird mit Hilfe eines Parsers realisiert. Der Par-

[25] Zum Beispiel als Name einer Variable oder Methode, Siehe [4.3.3]

ser liest die C#-Datei ein und generiert daraus einen AST. Dieser AST wird dann vom FSTComposer in einen einfacheren FST umgewandelt. Um den Parser zu erhalten muss die Grammatik der Eingabesprache (C#) formal definiert werden. Aus dieser Definition kann ein Parsergenerator den Parser generieren.

Zu Beginn dieser Arbeit gab es bereits eine C#-Grammatik für den Parsergenerator JavaCC[26]. Diese wurde vom Projekt CIDE [KAE07] zur Verfügung gestellt. Sie orientiert sich im Wesentlichen am Syntax-Kapitel der C#-Spezifikation. Im Rahmen dieser Arbeit wurde die Grammatik verbessert und um einige fehlende Sprachkonstrukte erweitert.

Zu den Erweiterungen gehören die Einführung von Typparametern, von Typparameter-Einschränkungen und des Delegate-Konstrukts. Außerdem wurden einige Fehler in der Grammatik korrigiert.

Nachdem der generierte Parser eine C#-Datei eingelesen hat gibt er einen AST zurück. Wie im Kapitel zu FSTs beschrieben muss dieser noch in einen FST umgewandelt werden. Das geschieht mit der Klasse `TreeConverter` des FSTComposer. In diesem Schritt wurde also aus allen (C#-)Dateien der ausgewählten Features ein FST erzeugt.

Zusätzlich zu Dateien der Artefaktsprachen muss der FST auch die Ordnerstruktur des Features und den Featurenamen speichern. Zu diesem Zweck gibt es zwei besondere Knotentypen. Der Typ `FeatureVertex` wird als oberster Knoten jedes FST verwendet. Er enthält den Namen des Features. Dieser wird zum Beispiel bei der Kombination von Methoden [2.3.6] benötigt. Der Typ `FolderVertex` wird genutzt um die innere Ordnerstruktur des FST abzubilden. Dabei werden immer Ordner gleichen Namens komponiert. Die FSTs der Artefaktsprachen-Dateien werden in die Ordnerstruktur-FSTs eingefügt.

3.3.3 Komposition der FST-Knoten

Die Komposition der im vorhergehenden Schritt erstellten FSTs läuft nach dem im Kapitel zum FSTComposer [2.3] beschriebenen Verfahren ab. Hierfür werden die Algorithmen verwendet, die im Kapitel zur Untersuchung von C# als Artefaktsprache entwickelt wurden. Die Relationen *equals* und *composable* werden jeweils in den FST-Knotentyp-Klassen implementiert. Die Relation *equals* entspricht dabei der aus Java bekannten `equals` Funktion. Die Relation *composable* wurde in die Funktion `compose` integriert. Diese Funktion gibt den Ergebnis-Knoten zurück, falls die Knoten komponierbar sind. Falls sie nicht komponierbar sind, wirft diese Methode eine `NotComposableException`. Diese Exception wird mit Angabe der Stelle im Featurecode, die sie ausgelöst hat, an den Benutzer ausgegeben.

Die Implementierung der Knoten konnte wie in der Untersuchung beschrieben vorgenommen werden. Die beschriebenen Konstrukte wurden dabei in auf mehrere FST-Knotentypen aufgeteilt. Jeder Knotentyp wurde durch eine Klasse implementiert. So wurde zum Beispiel die Implementierung des Feld-Konstrukts in Kapitel [3.2.5] auf die FST-Knotentypen `FieldVertex`, `FieldInitialisationVertex` und `FieldTypeVertex` aufgeteilt. Eine komplette Liste der FST-Knotentypen mit Zuordnung zu den untersuchten Konstrukten findet sich im Anhang.

[26]https://javacc.dev.java.net/

Um die generalisierten *equals* Relationen aus Kapitel [3.2.15] zu realisieren, wurden zwei abstrakte `Vertex`-Klassen eingeführt (`TypeDeclarationVertex` und `MemberDeclarationVertex`). In diesen Klassen wurden die generalisierten Relationen implementiert. Die entsprechenden `Vertex`-Klassen erben von diesen abstrakten Implementierungen (z. B. `ClassVertex` von `TypeDeclarationVertex`).

Unterschiede zur Beschreibung in der Untersuchung entstanden bei der Implementierung der Delegates. Im Kapitel zu Delegates in der Untersuchung [3.2.13] wurde diskutiert, dass die unterschiedlichen Namen von formalen Parametern zugelassen werden können. Diese Möglichkeit wurde aufgrund geringen Nutzens und den erforderlichen Änderungen nicht ausgenutzt.

Für die Komposition von Methoden wurde der in [3.2.6] vorgestellte Ansatz 3 (Einführung des `original`-Schlüsselwortes) gewählt. Bei der Umsetzung dieses Ansatzes ergab sich ein schwerwiegendes Problem.

Umsetzung des Methoden-Kompositionsalgorithmus

Für die Durchführung der Methodenkomposition nach Kompositionsalgorithmus 3 muss in einer Methode das `original`-Schlüsselwort durch den Aufruf der neu erzeugten Methode ersetzt werden. Dabei wird davon ausgegangen, dass hinter dem Schlüsselwort bereits die richtigen Parameter stehen.

```
int  test(string str) {
    int ret = original(str);
    bool original;
}
```

Abb. 26:Komposition mit original

Die Felddeklaration (`bool original`) sollte möglichst nicht verändert werden. Es wäre also nötig den Methodenrumpf nach Auftreten des Schlüsselwortes mit nachfolgenden Typparametern und formalen Parametern zu durchsuchen. Die Parameterkonfiguration der aufgerufenen Methode soll dabei der Parameterkonfiguration der eigenen Methode entsprechen. Das angesprochene Problem liegt nun in der Erkennung des Schlüsselwortes mit nachfolgendem Typparameter/ formalen Parameter. Da Typparameter möglich sind, kann man nicht nach `original(` suchen. Weil Typparameter verschachtelt sein können, stellen sie eine kontextfreie Grammatik (Chomsky Typ 2) dar. Diese lassen sich nicht von regulären Ausdrücken erkennen. Man kann also nicht mit einem einfachen Suchen/Ersetzen-Ausdruck alle `original`-Methodenaufrufe im Methodenrumpf erkennen und ersetzen. Dieses Problem könnte man lösen, indem man einen weiteren Parser für den Methodenrumpf implementiert bzw. die Methodenrümpfe im FST genauer auflöst. Diese Lösungen sind im Umfang dieser Arbeit nicht praktikabel. Darum wurde entschieden das Schlüsselwort nur an den Stellen wo die überschriebene Methode aufgerufen wird zu erlauben. Bezeichner die `original` enthalten werden nicht korrekt behandelt. Durch diese Vereinfachung kann man einfach alle `original`-Vorkommen durch den (neuen) Methodennamen ersetzen. Der Entwickler muss darauf achten, dass die angegebenen Typ- und formalen Parameter einen korrekten Aufruf ergeben bzw. dass die aufgerufene Methode verfügbar ist.

3.3.4 Überführung eines FST(C#) in Sourcecode

Nachdem die Features komponiert wurden, existiert nur noch ein Ergebnis-FST. Dieser FST muss wieder in Sourcecode übersetzt werden. Bei der Übersetzung muss beachtet werden wie die Elemente des FST im Sourcecode syntaktisch ausgedrückt werden. Diese Regeln sind in den jeweiligen FST-Knotentypen gespeichert.

Um einen FST auszugeben werden die `visit`-Methoden der Knoten genutzt. Die Knoten werden nach ihrer Gruppierung aufgerufen. Nach jeder Gruppe von Kind-Knoten wird die Methode visitAfterGroup des Vaterknotens aufgerufen. So kann der Vaterknoten flexibel Syntaxelemente einfügen.

In der folgenden Abbildung wird ein FST gezeigt, der eine Klasse repräsentiert. Die FST-Struktur unterhalb der Kinder der Klasse wird ausgeblendet. Dieser FST wird in Sourcecode umgewandelt. Dabei müssen einige syntaktische Elemente eingefügt werden (`class`, der Klassenname, `{` und `}`). Die weiteren Syntaxelemente wie die spitzen Klammern oder der Doppelpunkt werden von den untergeordneten Knoten behandelt.

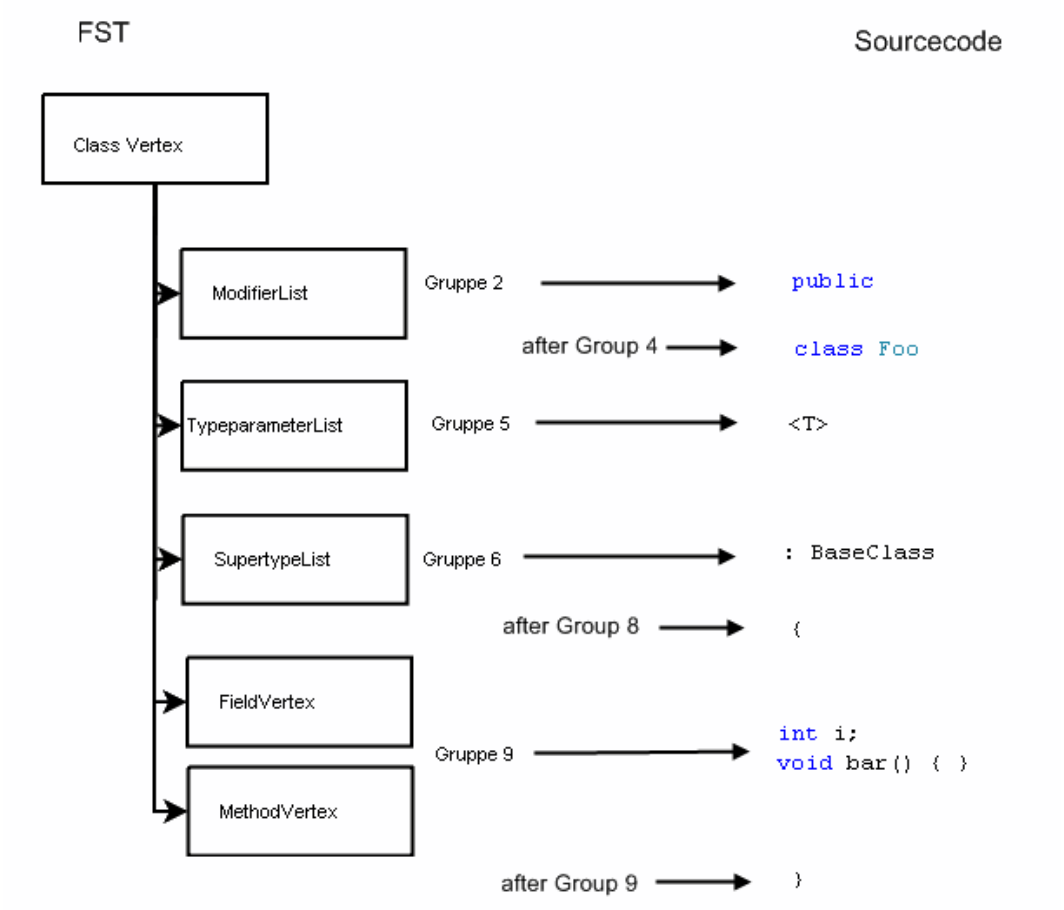


Abb. 27: Übersetzung eines FST in Code

Eine besondere Behandlung ist für die Typen `FeatureVertex`, `Folder-Vertex` und `CompilationUnitVertex` nötig. `FeatureVertex` und `FolderVertex` repräsentieren jeweils einen Ordner, also wird bei der Ausgabe dieser FST-Knoten ein Ordner mit dem entsprechenden Namen erzeugt.

`CompilationUnitVertex` ist einer der C#-spezifischen Knotentypen er repräsentiert eine .cs-Datei. Bei der Ausgabe dieses Knotens muss also eine neue Datei erzeugt werden. Der Pfad der Datei wird durch die im FST übergeordneten `Folder`- und `Featurevertex`-Knoten bestimmt. In diese Datei werden dann die Ausgaben der untergeordneten FST-Bäume geschrieben.

4. Evaluierung

Die Evaluierung der Erweiterung des FSTComposer wurde in drei Schritten vorgenommen. Zunächst wurde während der Implementierung eine sehr kleine Produktlinie geschrieben. Diese wurde genutzt um die implementierten Algorithmen direkt zu testen. Weiterhin wurden existierende Java-Produktlinien nach C# übersetzt und als Grundlage für Tests genutzt. In diese Produktlinien wurden dann Typparameter[27] eingeführt. Als dritter Schritt wurde mit dem MS Visual Studio 2005 ein kleiner Webservice entwickelt, der zusätzlich zu den C#-Dateien auch die WSDL-Spezifikation (in XML) des Webservices enthält. Diese wurde je nach Feature-Auswahl zusammengestellt. Mit diesem Beispiel wurde gezeigt, dass der FSTComposer auf unterschiedlichen Artefakttypen arbeiten kann. Die Schritte werden in den folgenden Kapiteln einzeln behandelt.

4.1 Test der Implementierung an kleinen Beispielklassen

In diesem Test wurde für einige Knoten der C#-FSTs Klassen geschrieben, die spezielle Situationen der Kompositionsalgorithmen abdeckt. Zum Beispiel wurde so die komplexeren Fälle der Kompositionsalgorithmen für Properties und für Modifier-Listen getestet. Für jeden dieser FST-Knoten existieren zwei „.cs“-Dateien die vom FSTComposer komponiert werden. Die Dateien werden in zwei Features („Feature1“ und „Feature2“) organisiert. Zum Test werden diese beiden Features ausgewählt und mit dem FSTComposer komponiert. Nach der Komposition muss überprüft werden ob das Ergebnis die erwarteten Sourcecode enthält. Außerdem wurden Features entwickelt bei deren Komposition ein Fehler vom Kompositionsalgorithmus erwartet wird. So können einige Fehlerzustände herbeigeführt und überprüft werden.

Mit diesem Test können viele Spezialfälle abgedeckt werden, die sonst vermutlich in keinem realistischen Programm auftreten würden. Außerdem konnten die implementierten Algorithmen so schon während der Implementierung mit sehr einfachen Codestrukturen getestet werden. Mit diesem Test kann jedoch nicht überprüft werden ob zum Beispiel die Laufzeit des FSTComposer bei großen Produktlinien stark steigt oder ob auch mehr als zwei Features komponiert werden können.

[27] Typparameter sind in der Java-Artefaktsprache nicht verfügbar.

4.2 Test mit realistischem Programm – GPL

Nachdem die Implementierung weitgehend abgeschlossen war, wurde die existierende Produktlinie GPL[28], die in Java vorliegt, nach C# übersetzt. Die C#-Version der Produktlinie wurde dann komponiert. Die komponierten C#-Programme wurden mit den Ergebnissen der Java-Produktlinie verglichen. Da es sich bei der Produktlinie um ein Konsolenprogramm handelt, kann man die Ausgaben der Produkte sehr gut vergleichen. Der Testprozess wird in der folgenden Abbildung veranschaulicht.

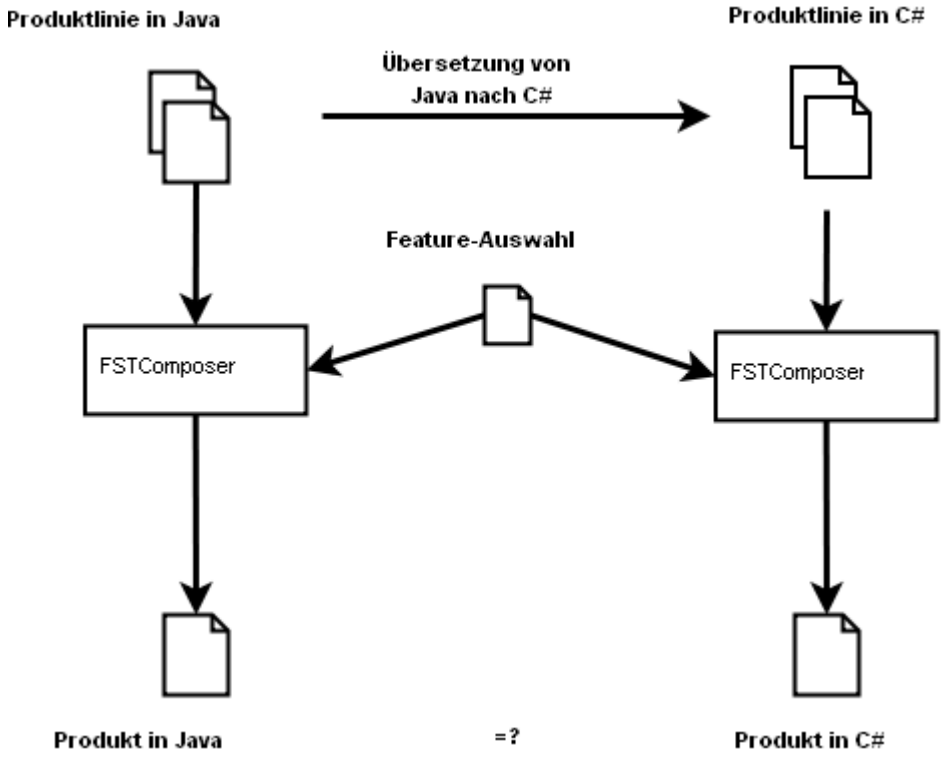


Abb. 28:Evaluierung mit übersetzten Java-Produktlinien

Weiterhin wird die Produktlinie mit Typparametern erweitert. Weil der FST-Composer keine Java-Typparameter behandeln kann, enthalten die Java-Produktlinien keine typparametrisierten Klassen. Bei Graphenalgorithmen ist die Verwendung von Typparametern aber besonders günstig. Weil in dieser Arbeit C#-Typparameter implementiert wurden, wurden die Produktlinien um Typparameter erweitert. So konnte die Implementierung der Typparameter getestet werden.

Portierung eines Feature-faktorisierten Programms von Java/ Jak nach C#

Für die Übersetzung der Java Produktlinie nach C# wurde zunächst das Tool Microsoft Java Language Conversion Assistant 3.0 genutzt. Es ermöglicht die automatische Übersetzung von Java Sourcecode nach C#. Da der resultierende C#-Code aber sehr umständlich ist, wurde die Übersetzung letztlich ohne Tool-Unterstützung vorgenommen.

Für die Übersetzung gibt es einige einfache Anweisungen, die man mit einer Suchen/ Ersetzen-Funktion ausführen kann. Dazu gehören zum Beispiel die Ände-

[28]Die Graph Produktline (GPL) wurde in [LB01] vorgestellt.

rung aller `import`-Schlüsselwörter in `using`, das Anfügen des Schlüsselwortes `virtual` vor jede Java-Methode[29]. Einige andere Probleme lassen sich nicht so einfach lösen. Zum Beispiel erlaubt C# keine anonymen Klassen. Sie müssen also durch innere Klassen simuliert werden, wobei auf Namenskonflikte geachtet werden muss. Ein weiteres Problem entsteht falls der Java Code den Wildcard-Operator `?` nutzt. Dieser Operator steht in C#-Typparametern nicht zur Verfügung. Es muss also eine andere softwaretechnische Lösung gefunden werden.

Ein Problem, das bei der Portierung aufgetreten ist, wird im Folgenden detaillierter behandelt. Bei der Verwendung eines Java-Iterators ist es verboten während der Iteration die Struktur der zugrunde liegenden Liste zu verändern. Es ist aber nicht verboten Listenelemente auszutauschen. Genau in diesem Punkt unterscheiden sich die Implementierungen des Java-Iterators und des C#-Enumerators. Der C#-Enumerator erlaubt kein Austauschen von Listenelementen.

In dem Feature MSTPrim der Produktlinie GPL wurde genau dieser Umstand in Java ausgenutzt. Es wurden während der Iteration über eine Liste die Listenelemente umsortiert. Diese Implementierung konnte aufgrund der oben beschriebenen Unterschiede zwischen Java und C# nicht übernommen werden. Es wurde eine sehr einfache Lösung gewählt, indem eine Kopie der Original-Liste erstellt wurde und alle Änderungen auf dieser ausgeführt wurden während weiter auf der Original-Liste iteriert wurde. Diese Lösung ist nicht weiter kompliziert, aber das Problem zeigt mit welch subtilen Unterschieden man während einer Portierung zwischen zwei auf den ersten Blick so ähnlichen Programmiersprachen rechnen muss.

Es wird festgestellt, dass alle Probleme, die während der Übersetzung aufgetreten sind, auf Unterschiede oder Beschränkungen der Programmiersprachen bzw. APIs zurückzuführen sind. Die oft restriktiv formulierten Kompositionsregeln [Kapitel 3] haben in keinem einzigen Fall zu Problemen geführt.

Refaktorisierung der portierten Produktlinie mit Typparametern

Bei der Portierung sind viele Klassen entstanden, die nur die Methoden `next()` und `hasNext()` enthalten. Diese waren nötig um den Java-Iterator mittels des C#-Enumerators zu simulieren. Dieses Konstrukt ist sehr unschön weil es viel Code-Replikation erfordert. Es wurde darum im Zuge der Refaktorisierung durch eine generische Klasse `Iterator < RT, ET>` ersetzt. Diese Klasse simuliert die Funktionalität des Java-Iterators und ist durch Verwendung des Enumerator-Konstrukts in C# realisiert.

Die Typparameter geben den Enumerator-Typ (ET) und den Rückgabetyp (RT) an. Es wird über eine Liste vom Typ ET iteriert. Der Typ RT gibt an welcher Typ von der `next()` Funktion zurückgegeben wird. Meist stimmen diese Typen überein. Es kann jedoch eine Konversionsfunktion angegeben werden, die den aktuellen Enumerator-Rückgabewert in den Rückgabetyp umwandelt. Diese Funktion ist mittels des Delegates `delegate RT Transformer(ET val)` definiert.

Diese Komplikation war nötig um die gesamte Funktionalität der Java-Iteratoren in Kombination mit anonymen Klassen zu simulieren. Aufgrund dieser Erweiterung konnte der restliche Teil der Implementierung weitgehend ohne weitere Anpassung übernommen werden.

[29] Andernfalls können Methoden nicht überschrieben werden.

Vergleich der komponierten Produktlinie in Java mit den komponierten Produktlinien in C#

In diesem Abschnitt wird beschrieben wie die portierten Produktlinien letztendlich verglichen werden. Durch diesen Vergleich soll ein Indiz für die Korrektheit der C#-Erweiterung des FSTComposer gewonnen werden.

Aus der Java-Produktlinie standen bereits neun Feature-Auswahl-Dateien der Produktlinie zur Verfügung. Diese Feature-Auswahlen funktionieren natürlich auch mit der portierten Produktlinie weil die Namen der Features beibehalten wurden. Zum Vergleich werden nun alle Produkte die durch diese Feature-Auswahlen definiert werden sowohl in Java als auch in C# erstellt. Weiterhin werden die entsprechenden Produkte ausgeführt, und ihre (Kommandozeilen-) Ausgaben verglichen. Dieser Vergleich ergab das alle Produkte kompilieren und alle Ausgaben der C#-Produkte im Wesentlichen ihren Java-Äquivalenten entsprechen.

Der einzige Unterschied liegt in der Auswahl des repräsentativen Elements eines Zyklus im Graphen. Dieser Unterschied tritt bei mehreren Feature-Auswahlen auf. Der Zyklus wird in der Ausgabe durch dieses repräsentative Element identifiziert. Das repräsentative Element ist in den Java-Ausgaben stets `1` in den C# Ausgaben ist es `7`.

Bei der Suche nach einem Zyklus werden die Knoten des Graphen in einer Liste gespeichert und sortiert. Dabei wird für die Sortierung eine Vergleichsfunktion verwendet bezüglich derer alle Knoten dieses speziellen Graphen gleich sind. In Java wird für die Sortierung die Funktion `Collections.sort` verwendet. In C# sind es die Funktionen `ArrayList.Sort` (in der Implementierung ohne Typparameter) und `List<T>.Sort`. Die Sortierfunktion in Java ist stabil, das bedeutet, dass die Reihenfolge von Elemente die bezüglich der Vergleichsfunktion gleich sind nicht verändert wird. Die Sortierfunktionen in C# nutzen offensichtlich einen nicht-stabilen Sortieralgorithmus. In C# steht vor der Sortierung der Knoten 1 am Anfang der Liste. Nach der Sortierung steht der Knoten 7 am Anfang. Darum wird Knoten 7 als Repräsentant gewählt. In Java wird die Reihenfolge nicht geändert, darum wird Knoten 1 gewählt. Dieser Unterschied in der Implementierung der Sortierfunktionen zeigt wiederum wie subtil die Unterschiede von APIs sein können.

Mit diesem Test konnte gezeigt werden, dass der FSTComposer für diese spezielle Produktlinie ein vermutlich korrektes Ergebnis erzeugt. Es konnte nicht gezeigt werden ob fortgeschrittene Anwendungsmöglichkeiten wie zum Beispiel die Änderung eines Zugriffsmodifiers richtig implementiert wurden, weil diese in der Produktlinie nicht genutzt werden. Außerdem konnte gezeigt werden, dass das Programm auch für größere Produktlinien mit einer guten Laufzeit ausgeführt wird.

4.3 Test der Webservice-Produktlinie

Diese Produktlinie enthält die Features eines Webservices, der mit Visual Studio 2005 in C# geschrieben wurde. Der Webservice stellt einen einfachen Rechner zur Verfügung der addieren, subtrahieren, multiplizieren und dividieren (auf Integern) kann. Die Produktline enthält ein Feature für jede Grundrechenart. Fast der gesamte Code zu diesem Webservice wird vom Visual Studio generiert. Nur die

konkrete Implementierung der Methoden muss manuell vorgenommen werden. Dieser Code wird in der folgenden Abbildung dargestellt.

```
12  [WebMethod] public int add (int arg1,int arg2){
13      return arg1 + arg2;
14  }
15  [WebMethod] public int sub (int arg1,int arg2){
16      return arg1 - arg2;
17  }
18  [WebMethod] public int mult (int arg1,int arg2){
19      return arg1 * arg2;
20  }
21  [WebMethod] public int div (int arg1,int arg2){
22      return arg1 / arg2;
23  }
```

Abb. 29:CalcWebservice Implementierung

Jede der Methoden ist in einem eigenen Feature gekapselt. Diese Darstellung entsteht, wenn man alle vier Features komponiert.

Zusätzlich zu der C#-Implementierung existiert eine WSDL-Beschreibung des Webservices. In dieser XML-kodierten Beschreibung werden die zur Verfügung gestellten Services definiert. Diese Datei wird je nach ausgewählten Features komponiert. Es werden also in diesem Test tatsächlich verschiedene Artefakttypen gemeinsam komponiert („.cs“ und „..fxml“). Die Definition eines speziellen Dateiformates für Feature-XML-Dateien („.fxml“) war nötig, weil, anders als in normalem XML, für die Umsetzung in einen FST jeder Knoten einen eindeutigen Namen haben muss. In XML können mehrere identische Knoten auf derselben Ebene existieren. Dies ist nicht mit dem Prinzip des FSTComposer kompatibel. Darum wurde für Feature-XML jedem Knoten ein zusätzliches Attribut „fstname“ hinzugefügt. Das wird genutzt um den Knoten zu identifizieren. Falls der Knoten über seinen Typen auf der Hierarchieebene eindeutig identifiziert wird, ist das Attribut nicht nötig.

Mit diesem Test wurde gezeigt, dass auch andere Programmtypen als einfache (Konsolen-) Applikationen für die Feature-Komposition geeignet sind. Außerdem wurde die Anwendung des FSTComposer auf verschiedene Artefakttypen demonstriert. Ein weiterer möglicher Artefakttyp wären Dokumentationsdateien wie z. B. Readme-Dateien oder Nutzeranleitungen in LaTex.

5. Fazit

In dieser Arbeit wurden die Möglichkeiten der Komposition von in der Programmiersprache C# verfassten Features untersucht. Die aus dieser Untersuchung gewonnenen Ergebnisse wurden als Erweiterung des Tools „FSTComposer“ implementiert. Diese Erweiterung ermöglicht es mit dem FSTComposer Features, die (teilweise) in C# geschrieben sind, zu komponieren.

Besonderes Ziel bei der Untersuchung und Implementierung war es den Benutzer zu unterstützen, indem die Erstellung von fehlerhaften Ergebnisprogrammen möglichst verhindert wird. Dieses Ziel konnte bis auf wenige Ausnahmen erreicht werden. Die Ausnahmen wurden diskutiert und es muss dem Entwickler überlassen werden Situationen die diese Ausnahmen hervorrufen zu vermeiden.

Ein weiteres Ziel was es, möglichst große Teile des Kompositionsalgorithmus kommutativ zu gestalten. Das bedeutet, dass die Reihenfolge in der Features komponiert werden nicht beachtet werden muss. Es war von vornherein klar, dass dieses Ziel für die Komposition von Methoden nur schwer zu erreichen ist. Es war aber möglich die Kompositionen aller nicht-methoden-ähnlichen Konstrukte kommutativ zu gestalten. Dadurch wird eine vereinfachte Benutzung des Tools und ein geringerer Einarbeitungsaufwand für den Entwickler erreicht.

Schwierigkeiten bei der Untersuchung sind durch die syntaktische und semantische Komplexität der Strukturen in C# entstanden. In einigen Fällen wurden nur die einfachsten der diskutierten Kompositionsalgorithmen tatsächlich implementiert. In der Evaluierung hat sich jedoch gezeigt, dass auch mit diesen eine ausreichend flexible Programmierung möglich ist. Die Praxis muss an dieser Stelle zeigen welche weiteren, speziellen Kompositionsmöglichkeiten sinnvoll und lohnend sind. Hier wäre dann der Ansatzpunkt für weitergehende Arbeiten gegeben.

Die Evaluierung wurde mit drei sehr verschiedenen Produktlinien durchgeführt. Eine simple Produktlinie wurde für die Fehlersuche während der Implementierungsphase entwickelt. Eine größere Produktlinie wurde genutzt um den praxisnahen Einsatz zu simulieren und mit der dritten Produktlinie wurde gezeigt, dass sowohl Code, als auch XML- Dateien komponiert werden können. Insgesamt hat sich diese Struktur zur Evaluierung bewährt. Es konnten alle wesentlichen Anwendungsbereiche automatisiert getestet werden und viele Fehler wurden schon frühzeitig erkannt und korrigiert.

6. Literaturverzeichnis

AA07 Albahari, J., & Albahari, B. (2007). C# 3.0 In a Nutshell: A Desktop Quick Reference; covers LINQ and .NET 3.5 CLR and Core Classes (3. ed.). Beijing: O'Reilly.

ALM+08 Apel, S., Lengauer, C., Möller, B., & Kästner, C. (2008). An Algebra for Features and Feature Composition. In Lecture notes in computer science: . Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST) (pp. 36-50). Springer-Verlag.

MPP Apel, S. (Wintersemester 07/08). Vorlesung Moderne Programmierparadigmen an der Universität Passau.

AL08 Apel, S., & Lengauer, C. (2008). Superimposition: A Language-Independent Approach to Software Composition. In Lecture notes in computer science: . Proceedings of the 7th International Symposium, SC 2008, Budapest, Hungary, March 29-30, 2008 (pp. 20-35). Springer-Verlag.

BSR03 Batory, D., Sarvela, J. Neal, & Rauschmayer, A. (2003). Scaling stepwise refinement. In . ICSE '03: Proceedings of the 25th International Conference on Software Engineering (pp. 187-197). Washington, DC, USA: IEEE Computer Society.

CW85 Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. ACM Comput. Surv., 17(4), 471-523.

CN07 Clements, P., & Northrop, L. (2007). Software product lines: Practices and patterns (6. printing). SEI series in software engineering. Boston: Addison-Wesley.

CE05 Czarnecki, K., & Eisenecker, U. W. (2005). Generative programming: Methods, tools, and applications (6. print.). Boston: Addison Wesley.

ECMA Spezifkation der Programmiersprache C#: ECMA-334 (4th Edition). (2006). Online verfügbar unter http://www.ecma-international.org/publications/standards/Ecma-334.htm

GHJ+95 Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

OOP Gregor Snelting (Wintersemester 07/08). Vorlesung Objektorientierte Programmierung an der Universität Passau.

GUN00 Gunnerson, E. (2000): C#. Die neue Sprache für Microsofts .NET-Plattform. Herausgegeben von Galileo Press GmbH. Online verfügbar unter http://www.galileocomputing.de/openbook/csharp/, zuletzt geprüft am 20.06.2008.

KCH+90 Kang, K. C. , Cohen, S. G., Hess, J. A., Novak, W. E., & Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Carnegie-Mellon

KAE07 Kästner, C. (2007). CIDE: Decomposing Legacy Applications into Features. In . Proceedings of the 11th International Software Product Line Conference (SPLC), second volume (Demonstration) (pp. 149-150).

KLM+97 Kiczales, G., Lamping, J., Mendhekar, A., Meada, C., Lopes, C., & Loingtier, J.-M., et al. (1997). Aspect- Oriented Programming. In M. Ak¸sit (Ed.), Lecture notes in computer science: Vol. 1241. Object-oriented programming. 11th European conference, Jyväskylä, Finland, June 9 - 13, 1997 ; proceedings (pp. 220–242). Berlin: Springer.

LB01 Lopez-Herrejon, R. E., & Batory, D. (2001). A Standard Problem for Evaluating Product-Line Methodologies. Lecture Notes in Computer Science: . Proceedings of the Third International Conference, GCSE 2001 Erfurt, Germany, September 10–13, 2001 (pp. 10-24). Springer-Verlag.

VB9 Programmiersprachenspezifikation, Version 9.0: The Microsoft Visual Basic Language Specification.

PRE97 Prehofer, C. (1997). Feature-Oriented Programming: A Fresh Look at Objects. Lecture Notes in Computer Science: . Proceedings of the 11th European Conference Jyväskylä, Finland, June 9–13, 1997 (p. 419). Springer-Verlag.

TOH+99 Tarr, P. L., Ossher, H., Harrison, W. H., & Stanley M. Sutton Jr. (1999). N Degrees of Separation: Multi-Dimensional Separation of Concerns. In . International Conference on Software Engineering .

IEEE94 The Institute of Electrical and Electronics Engineers (Ed.) (1994). IEEE Software Engineering Standards Collection.

7. Anhang

Reduktion in C#

```
1  using System.Collections.Generic;
2
3  static class Redux {
4      public delegate T operation<T>(T x, T y);
5      public static T reduce<T>(operation<T> op,
6          T init, List<T> remainingElements) {
7          if (remainingElements.Count > 0) {
8              // remove first element from the list
9              T current = remainingElements[0];
10             remainingElements.RemoveAt(0);
11             // call the recursive reduction
12             return reduction<T>(op, op(init, current),
13                 remainingElements);
14         }
15         else {
16             return init;
17         }
18     }
19
20     // Verwendung
21     static void Main() {
22         int red = reduce<int>(add, 0,
23             new List<int>(new[] { 1, 2, 3 }));
24         System.Console.WriteLine(red);
25         System.Console.ReadLine();
26
27         // Implementierung mit Lambda-Funktion
28         //(nur in C# 3.5)
29         // der Typ der Lambda-Funktion
30         // wird automatisch von C# inferiert
31         /*
32         int red2 = reduce<int>(
33             ((x,y)=>x+y), 0,
34             new List<int>(new[] { 1, 2, 3 }));
35         System.Console.WriteLine(red2);
36         System.Console.ReadLine();
37          */
38     }
39     private static int add(int x, int y) {
40         return x + y;
41     }
42 }
```

Abb. 30:Reduktion in C#

Tabelle der FST-Knotentypen

Knotentyp	Verwendung
`AttributeListVertex`	Repräsentiert eine Liste von Attributen
`AttributeVertex`	Ein Element einer Attribut-Liste
`ClassVertex`	Repräsentiert eine C#-Klassendeklaration
`CompilationUnitVertex`	Repräsentiert eine „.cs“-Datei
`ConstructorBodyVertex`	Der Rumpf eines Konstruktors (analog zu MethodBodyVertex)
`ConstructorInitialisationVertex`	Initialisierung des Konstruktors (`base` oder `this`)
`ConstructorVertex`	Ein Konstruktor einer C#-Klasse
`DelegateVertex`	Repräsentiert eine Delegate-deklaration
`EnumMemberVertex`	Ein Element einer Enum-Deklaration
`EnumVertex`	Die Deklaration eines Enums
`FieldInitialisationVertex`	Initialisierung eines Feldes mit einem Default-Wert
`FieldTypeVertex`	Typ eines Feldes
`FieldVertex`	Repräsentiert ein Feld
`InputParameterArrayVertex`	Die InputParameter-FSTKnotentypen repräsentieren die Struktur der Parameter einer Methode oder eines Konstruktors.
`InputParameterListVertex`	
`InputParameterModifierVertex`	
`InputParameterNameVertex`	
`InputParameterTypeVertex`	
`InterfaceVertex`	Repräsentiert ein C#-Interface
`MemberDeclarationVertex`	Abstrakter Obertyp für `MethodVertex`, `FieldVertex` und `PropertyVertex` (siehe [3.2.15])
`MethodBodyVertex`	Rumpf einer Methode
`MethodVertex`	Repräsentiert eine Methode
`ModifierListVertex`	Liste der Modifier von z. B. Einer Methode, eines Feldes oder einer Klasse
`NamespaceDeclarationVertex`	Repräsentiert die Deklaration eines `Namespace`
`PropertyAccessorBodyVertex`	Rumpf eines Property Accessors (vgl. `MethodBodyVertex`)

Knotentyp	Verwendung
`PropertyAccessorVertex`	Accessor einer Property (vgl. `MethodVertex`)
`PropertyTypeVertex`	Typ einer Property
`PropertyVertex`	Repräsentiert eine Property
`ReturnTypeVertex`	Rückgabewert von z. B. einer Methode
`StructVertex`	Repräsentiert ein C#-Struct
`SupertypeListVertex`	Supertypliste einer Typdeklaration
`SupertypeVertex`	Element einer Supertypliste
`TypeDeclarationVertex`	Abstrakter Obertyp von `ClassVertex`, `StructVertex`, `InterfaceVertex`, `DelegateVertex` und `EnumVertex` (siehe [3.2.15])
`TypeParameterConstrainstListVertex`	Die Typparameter-Knotentypen repräsentieren die Struktur der Typparametrisierung einer Methode, einer Klasse oder eines Interfaces.
`TypeParameterConstraintsVertex`	
`TypeParameterListVertex`	
`TypeParameterVertex`	
`UsingAliasVertex`	Alias eines `UsingVertex` (optional)
`UsingListVertex`	Speichert alle `using`-Deklarationen am Anfang einer Struktur (z. B. `CompilationUnitVertex`)
`UsingVertex`	Element einer `using`-Liste

Mitgelieferte Software

Mit dieser Arbeit wird eine CD geliefert, auf der sich die entwickelte Software in der endgültigen Version befindet. Die Software ist als Eclipse[30]- Projekt auf der CD zu finden. Das Projekt ist bereits fertig konfiguriert. Es ist also keine weitere Installation (abgesehen von der Eclipse- Installation) nötig.

Die in dieser Arbeit entwickelten Implementierungen sind in den folgenden `packages` zu finden:

FST-Knoten-Klassen	`de.apel.fa.fst.csharp`
Aufruf des C#-Parsers und Erstellung des FST	`de.apel.fa.builder.csharp`
Erweiterung des FSTComposer [Kapi-	`de.apel.fa.composer` und

[30]Software-Entwicklungsumgebung, http://www.eclipse.org/

tel 4.2]	`de.apel.fa.fst`

Der eigentliche C#-Parser wurde mit dem Parsergenerator JavaCC aus einer Grammatik erzeugt. Diese Grammatik (`cs.jj`) ist ebenfalls in dem Eclipse-Workspace enthalten. Der erzeugte Parser ist in dem Eclipse-Projekt `CS_Parser`. Weil der FSTComposer dieses Projekt nutzt um C#-Dateien in Syntaxbäume (und dann in FSTs) umzuwandeln, existiert eine Abhängigkeit des FSTComposer-Projekts von dem Projekt `CS_Parser`.

Die Klasse `de.apel.fa.composer.composer` enthält die `main`-Methode des FSTComposer. Über diese Klasse kann das Programm gestartet werden. Folgende Kommandozeilenparameter sind dabei wichtig:

`--expression <file>`	Gibt an welche Features komponiert werden sollen (vgl. Feature-Auswahl in [Kapitel 2.3.1]).
`--base-directory <dir>`	Verzeichnis in dem die Features liegen.
`--write --output-directory <dir>`	Verzeichnis in das die komponierten Dateien gespeichert werden sollen.

Weitere Kommandozeilenparameter sind auf der Webseite zum FSTComposer[31] zu finden.

Die in dieser Arbeit beschriebenen Testproduktlinien sind ebenfalls in dem Projekt enthalten. Sie können über die folgenden Aufrufe ausgeführt werden.

SimpleTests (Test an kleinen Beispielklassen [Kap. 5.1])	`--expression examples/CSharpTests/Test.expression --base-directory examples/CSharpTests/ --write --output-directory examples/CSharpTests`
GraphProduktLine [Kap. 5.2]	`--expression examples/GPLcsharpTest/Test<i>.expression --base-directory examples/GPLcsharpTest/ --write --output-directory examples/GPLcsharpTest/`
GraphProduktLine [Kap. 5.2] mit Typparameter-Refaktorisierung	`--expression examples/GPLcsharpGenerics/Test<i>.expression --base-directory examples/GPLcsharpGenerics/ --write --output-directory examples/GPLcsharpGenerics/`
Webservice-Produktlinie [Kap. 5.3]	`--expression examples/CalcWebServiceCS/TestAll.-expression --base-directory examples/CalcWebServiceCS/ --write --output-directory examples/CalcWebServiceCS/`

Die Platzhalter `<i>` in den Aufrufen der GraphProduktLine stehen für eine Ziffer (1-9) mit der man die gewünschte Feature-Auswahl angeben muss. Es stehen bereits 9 vorkonfigurierte Auswahlen zur Verfügung.

Als Testplattform für diese Arbeit wurde ein Notebook mit folgenden Spezifikationen verwendet:

Typ	Sony Vaio PCG-8W1M
Betriebs-	Windows XP 5.1 (Build 2600) Service Pack 2

[31] http://www.infosun.fim.uni-passau.de/cl/staff/apel/FSTComposer/

system	
RAM	2048 MB
CPU	Intel(R) Core(TM)2 CPU T7200@2.00GHz
Eclipse-Version	Eclipse 3.3.1.1

Eidestattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Bneutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Alexander von Rhein