Grosser Beleg

Flexible Codemodifikation im Elucidative Programming

bearbeitet von

Martin Heinzerling
geboren am 11. April 1985 in Merseburg

Technische Universität Dresden
Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Betreuer: Dipl.-Inf. Andreas Bartho
Hochschullehrer: Prof. Dr. rer. nat. habil. Uwe Aßmann

Eingereicht am 19. Oktober 2009
D.2.7 Documentation/Enhancement, D.2.6 Programming Environments
Elucidative Programming, Dokumentation, Java, Codemodifikation

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Martin Heinzerling
Dresden, den 19. Oktober 2009

Zusammenfassung

Elucidative Programming, als neuerer Ansatz zur Entwicklung von Frameworkdokumentation und Tutorials, stellt neue Anforderungen an eine Entwicklungsumgebung. Die Prämisse des unveränderlichen Originalquelltextes erfordert einen flexiblen Umgang mit erforderlichen Änderungen und Modifikationen, hin zum für die Dokumentation benötigten Quellcode.

Diese Arbeit setzt sich zunächst mit den Arten der möglichen Bearbeitungen auseinander und zeigt anschließend eine exemplarische Umsetzung in der Entwicklungsumgebung DEFT.

Inhaltsverzeichnis

1 Einleitung **3**

1.1 Motivation 4

1.2 Aufgabenstellung 4

1.3 Aufbau dieser Arbeit 5

1.4 Elucidative Programming 6

1.5 DEFT 6

1.6 Verwandte Arbeiten 7

2 Analyse **8**

2.1 Codemodifikationen in aktueller Literatur 9

2.1.1 Hervorhebung 9

2.1.2 Inhaltliche Erweiterungen 9

2.1.3 Inhaltliche Ersetzungen 10

2.1.4 Weitere Modifikationen 10

2.2 DEFT-Architektur 11

2.3 Weitere Anforderungen 12

3 Implementierung **14**

3.1 Hervorhebung 15

3.2 Indikator für Schutz und Veränderung 15

3.3 Layoutoperation auf dem Quelltext 16

3.4 Veränderung am Syntaxbaum 16

3.5 Entfernen unnötiger Leerzeichen 17

3.6 Speicherung und Wiederanwendung der Veränderungen 18

3.7 Vorüberlegung zur Integration 19

4 Zusammenfassung und Ausblick **20**

A Verzeichnisse **21**
Abbildungsverzeichnis 22
Tabellenverzeichnis 22
Quellcodeverzeichnis 22
Literaturverzeichnis 22

B Dokumentation **26**
B.1 Einen StyleContainer implementieren 27
B.2 Umfang des ASTLayouters 27
B.3 Verwendung des ASTModifiers 28
B.3.1 CommentTypeVisitor 29
B.3.2 SyntaxCheckerVisitor 29
B.4 Anwendung von Werkzeugen und Dekorationen 29
B.5 Quellcode/Testfälle/Javadoc 30

Kapitel 1

Einleitung

Im folgenden Kapitel wird zunächst auf die Motivation und Aufgabenstellung die dieser Arbeit zugrunde liegt eingegangen. Weiterhin werden einige Grundlagen vorgestellt, die für das weitere Verständnis notwendig sind.

1.1 Motivation

Mit zunehmend größeren und komplexeren Anwendungen gewinnt die Dokumentation überproportional an Bedeutung. Auch in der wachsenden Gemeinde der offenen Projekte und Schnittstellen sind schon viele Projekte an einer mäßigen oder veralteten Dokumentation und damit der fehlenden Akzeptanz der Benutzer gescheitert.

Nach (Nørmark, 2000a) stehen dem Dokumentationswillen der Entwickler vier Grundprobleme[1] gegenüber, die durch das Konzept des Elucitative Programming (Abschnitt 1.4) angegangen werden können. Sobald die Grundprobleme aber durch eine entsprechende Entwicklungsumgebung (Abschnitt 1.5) gelöst sind, taucht ein weiteres Problem bzgl. der Akzeptanz durch den Anwender auf.

Jeder Entwickler hat nicht nur einen eigenen Stil bei der Programmierung, sondern auch bei der Dokumentation. Ein Tool muss auf diese Wünsche Rücksicht nehmen und einen daraus resultierenden großen Umfang an Bearbeitungsmöglichkeiten des Quellcodes innerhalb der Dokumentation zulassen. Ohne diese Flexibilität wird auch das beste Werkzeug nicht in der Lage sein, dem Motivationsproblem in der großen Menge der Entwickler flächendeckend zu begegnen.

1.2 Aufgabenstellung

Das Elucidative Programming (EP) ist eine neue, hypertextbasierte Art der Softwaredokumentation. Es werden so genannte Tutorials erzeugt, die neben Text auch Quellcodeausschnitte oder Hyperlinks auf Quellcode enthalten. Zur Erzeugung und Wartung solcher Tutorials ist Werkzeugunterstützung nützlich, beispielsweise der am Lehrstuhl entwickelte Tutorialeditor DEFT. In DEFT werden Codeausschnitte nicht physisch in das Tutorial eingefügt, stattdessen werden Referenzen zwischen Quellcode und der entsprechenden Stelle im Tutorial gehalten, welche erst beim Export in ein statisches Format, z.B. HTML oder PDF aufgelöst werden. Ändert sich der referenzierte Quellcode, werden die eingebetteten Codeausschnitte somit automatisch aktualisiert.

Oftmals möchte man in Tutorials aber den Code modifiziert darstellen, z.B. für Methoden nur die Signatur, oder aus einer Methode nur ausgewählte Statements. Ebenfalls wünschenswert ist, erläuternde Kommentare einzufügen, die im Originalcode nicht vorhanden sind. Derzeit funktionieren diese so genannten Formate derart, dass angegeben wird, welche Teile des Syntaxbaums des Codeausschnittes herausgefiltert und ggf. durch Alternativtext ersetzt werden sollen.

[1] Sauberer Quellcode ohne unnötige Kommentare, unübersichtliche Beziehung zw. Quellcode und externer Dokumentation, Überlastung bei parallelem Programmierieren und Dokumentieren, Motivationsproblem

Dieser Ansatz hat jedoch einige Unzulänglichkeiten:

- *Der Alternativtext ist eine reine Zeichenkette. Es ist nicht möglich, ihn beispielsweise als Kommentar auszuzeichnen und ins Syntaxhighlighting einzubeziehen.*
- *Es ist nicht möglich zu definieren, welche Auswirkungen ein Weglassen bestimmter Tokens auf das Layout des restlichen Codes hat. Sollen existierende Leerzeichen und Leerzeilen beibehalten werden oder nicht? Sollen neue Leerzeichen und Leerzeilen eingefügt werden?*
- *Manchmal möchte man gar keinen Code auslassen/ ersetzen, sondern durch Farbwahl bestimmte Stellen hervorheben. Auch eine Kombination von farblicher Hervorhebung und Auslassung von Code ist nützlich. Das ist derzeit noch nicht möglich.*

In dieser Arbeit ist zunächst zu untersuchen, welche Art von Codemodifikationen und -hervorhebungen in der Praxis verwendet werden. Es sollen sowohl Online-Tutorials als auch gedrucke Werke, z.B. Programmierlehrbücher, in die Betrachtung einbezogen werden. Ebenfalls zu untersuchen ist, ob es bereits Ansätze gibt, Codeausschnitte (semi)automatisch zu verändern. Auf den Rechercheergebnissen aufbauend ist der bisherige Mechanismus zur Codefragmentmodifikation in DEFT zu erweitern, um eine umfangreichere und flexiblere Spezifikation für Veränderungen und Hervorhebungen von Code zu ermöglichen.

1.3 Aufbau dieser Arbeit

Dieser Arbeit liegt eine strikte Trennung der Abstraktionsgrade zugunde. Zunächst wird im restlichen Verlauf dieses Kapitels die Theorie hinter dem Konzept des Elucidative Programming dargestellt und die bisherige Umsetzung in einem Entwicklungswerkzeug kurz angedeutet.

Das zweite Kapitel fasst anschließend die Ergebnisse einer umfassenden Literaturstudie bezüglich möglicher Codemodifikationen zusammen und leitet parallel die daraus resultierenden Anforderungen für ein Entwicklungswerkzeug ab.

Die allgemeinen Anforderungen finden im dritten Kapitel auf dem konkreten Fall DEFT ihre Anwendung. Dabei wurden die konkreten Implementierungsdetails in den Anhang B ausgelagert, so dass man auch ohne die detaillierten Kenntnisse zu DEFT einen schnellen Überblick erhält. Diese Trennung soll im Weiteren auch die Anwendbarkeit auf andere Entwicklungswerkzeuge vereinfachen.

Kapitel vier fasst diese Arbeit nochmals kurz zusammen.

1.4 Elucidative Programming

Das Konzept des *Elucidative Programming*s(EP) aus (Nørmark, 2000a) leitet sich aus dem einige Jahre zuvor definierten Konzept des *Literate Programming*s(LP) (Knuth, 1984) ab. Beim LP wird auf eine räumliche Trennung des Quellcodes und der Dokumentation verzichtet. Der Entwickler ist dazu angehalten, fortlaufenden Text im Sinne der Dokumentation zu schreiben und in diesen die besprochenen Codeabschnitte einzubetten. Im Anschluss kann ein komplettes Programm und eine gewohnte Dokumentation generiert werden. (*tangle*/*weave*) Trotz einer zunehmenden Werkzeugunterstützung[2] konnte sich das Konzept nie wirklich durchsetzen.

Elucidative Programming
Literate Programming
tangle
weave

EP hingegen ist trotz des Namens keine Form des Programmierens. Der Schwerpunkt liegt hier auf einer strikten Trennung des Quellcodes und der dazugehörigen Dokumentation. Nachdem der Quellcode auf herkömmliche Weise erstellt wurde, wird dieser bzw. Ausschnitte davon als Referenz in die Dokumentation eingefügt. Wichtig ist hierbei, dass der Originalquellcode stets unverändert bleibt. Abschließend kann wieder eine interaktive Dokumentation generiert werden.

Die Verbreitung dieser Art der Dokumentation ist zur Zeit noch sehr überschaubar. Die wohl bekanntest Umsetzung ist unter [3] zu finden. Im weiteren Verlauf dieser Arbeit wird der Schwerpunkt der Betrachtung auf dem an der Technischen Universität Dresden entwickelte DEFT (Abschnitt 1.5) liegen.

1.5 DEFT

DEFT[4] ist das Akronym für *Development Environment For Tutorials*. Dabei entsprechen Tutorials den in den vorherigen Abschnitten erläuterten Dokumentationen mit Quellcodeabschnitten. In der aktuellen Version können beliebige Quellcodedateien[5] in ein Repository geladen und so für die Verwendung in Tutorials vorbereitet werden. Mit Hilfe einer graphischen Benutzeroberfläche können Codeabschnitte anschließend aus dem Syntaxbaum direkt in den Text eingebettet werden.

Diese Grundfunktionalität, die durch das EP definiert wird, wird durch einen WYSIWYG-Editor[6] für die Textpassagen und automatisches Syntaxhighlighting der Quellcodeabschnitte erweitert. Im Vergleich zu anderen Werkzeugen dieser Art sind keine Annotationen im Quelltext mehr nötig (vgl. Abschnitt

[2](Williams, 1992), (Briggs, 1993), (Knuth und Levy, 1994), (Johnson und Johnson, 1997)
[3]http://www.cs.aau.dk/~normark/scheme/styles/xml-in-laml/elucidator-2/man/elucidator.html
[4]http://deftproject.org
[5]derzeit in Java und C#, aber auch jede andere Sprache ist möglich
[6]What You See Is What You Get

2.2) und der Benutzer kann über ein einfaches Interface die Auswahl der sichtbaren Teile des Quellcodes steuern. Bei Änderungen und Weiterentwicklungen des ursprünglichen Quelltextes wird der Benutzer darauf aufmerksam gemacht, die betreffenden Abschnitte in den Tutorials zu überprüfen.

Weiterführende Details sind in (Bartho, 2009) oder unter [4] zu finden.

1.6 Verwandte Arbeiten

Der hier untersuchten Aufgabenstellung wurde bisher in der Literatur nur wenig Aufmerksamkeit geschenkt. In (Vestdam, 2002) werden sogenannte „komplexe Variationen“ angedeutet um die Modifikation von Codeabschnitten zu benennen, ohne diese Überlegungen jedoch zu Ende zu führen. Auch (Aguiar u. a., 2004) zeigt in Auszügen einige Änderungsmöglichkeiten bezogen auf Java.

Kapitel 2

Analyse

Das folgende Kapitel liefert eine Übersicht über in der Literatur häufig verwendete Modifikationen in Quellcodeabschnitten. Aufgrund des großen Umfanges an Lehrbüchern, Tutorials und Dokumentationen sind zu jeder Modifikation nur einige wenige Beispiele genannt. Auf die explizite Angabe von Beispielen wird in den trivialen Fällen verzichtet. Weiterhin spiegelt die Gliederung dieses Kapitels eine grobe Klassifikation der Modifikationen wieder, die auch im weiteren Verlauf dieser Arbeit beibehalten wird. Parallel dazu werden die Anforderungen an die Erweiterung von DEFT aufgezeigt, die daraus resultieren.

2.1 Codemodifikationen in aktueller Literatur

2.1.1 Hervorhebung

Die wohl einfachste aber zugleich häufigste Modifikation ist das Hervorheben bestimmter Abschnitte. Dabei werden neue Konzepte oder relevante Zeichenketten durch Veränderung des Schriftbildes gekennzeichnet. In der klassischen, gedruckten Literatur ist die verbreitetste Form das fett setzen der Zeichenkette.[1] Alternativ dazu werden im monochromen Druck auch das Unter- bzw. Durchstreichen[2], Kursivschrift[3] oder Versalien eingesetzt[3].

Da Elucidative Programming vorwiegend in einer interaktiven oder sogar hypertextbasierten Umgebung eingesetzt wird, gibt es keine formalen Gründe, sich bloß auf diese Formen zu beschränken. Vorstellbar sind auch farbige Hervorhebungen oder das Ändern der Schriftart. Bei Blöcken ist auch das Einrahmen oder das Ändern der Hintergrundfarbe denkbar und sinnvoll. Aus diesen Überlegungen resultieren folgende neue Anforderungen an die Erweiterung von DEFT.

Anforderung R1: DER QUELLCODE MUSS UM BELIEBIGE FORMATIERUNGEN ERWEITERBAR SEIN. **R1**

Anforderung R2: DIE ART DER FORMATIERUNG MUSS AUCH FÜR ZUKÜNFTIGE EXPORTFORMATE ANPASSBAR SEIN. **R2**

2.1.2 Inhaltliche Erweiterungen

Modifikationen, die das Layout betreffen sind relativ einfach in gedruckten oder kompilierten Werken zu erkennen. Weniger offensichtlich sind Erweiterungen im Quellcode der Dokumentation, die im Produktionscode nicht vorhanden waren.

Das wohl häufigste Szenario in dieser Kategorie ist das Anfügen von Kommentaren.[4] Besonders in Tutorials für Anfänger neigen Autoren dazu jede Programmzeile einzeln zu kommentieren, was innerhalb des Produktionscodes denkbar ungeeignet ist. Aber selbst im „normalen“ Quellcode sind zu umfangreiche Kommentare nicht förderlich für die Sauberkeit des Codes (vgl. Nørmark, 2000a).

Für das nötige Verständnis des Quellcodes sorgen aber neben Kommentaren auch zusätzliche Ausgaben oder sogar das Einbinden einer Funktion, die

[1] (Fitzgerald, 2007, S. 91), (Steiner, 1991, S. 251), (Ullenboom, 2007, S. 524, 718, 754), (Meyer, 1998, S. 249), (Ullenboom, 2007, S. 306), (Hewitt, 2009, S. 396), (Born und Born, 2006, S. 626), (Pomberger und Dobler, 2008)

[2] (Sturm, 2009, S. 255), (Martin, 2009, S. 234), (Ganor, 2007)

[3] (Weiss, 1989, S. 138)

[4] (Hajji, 1998, S. 604), (Ullenboom, 2007, S. 524), (Riggs, 2003, S. 341), (Jackson und McClellan, 1997, S. 329), (Stroustrup, 1990, S. 481), (Wirfs-Brock und McKean, 2003, S. 212), (Jan, 2007), (MSDN, 2009)

gewöhnlich an einer andere Stelle zu finden ist.[5]

Die Abstraktion dieser Gedanken führt zu den folgenden neuen Anforderungen.

Anforderung R3: In den Quellcode müssen neue Programmelemente (Token) oder Freitext eingefügt werden können. **R3**

Anforderung R4: Das Einfügen soll nicht nur in Textform, sondern unter voller Semantik für spätere Bearbeitungsschritte erfolgen. **R4**

Anforderung R5: Das Einfügen muss auch für zukünftige Programmiersprachen erweiterbar sein. **R5**

Anforderung R6: *Optional.* Das Einfügen muss an der eingefügten Stelle syntaktisch erlaubt sein. **R6**

2.1.3 Inhaltliche Ersetzungen

Wenn aus dem Originalquellcode einzelne oder mehre Anweisungen nicht benötigt werden, werden diese, sofern sie nicht komplett ausgeblendet werden, häufig durch „...“ ersetzt. Diese Zeichen treten dann sowohl horizontal als auch vertikal auf.[6] Abstrahiert man dieses Verhalten, ist auch das Ersetzen mehrerer Anweisungen durch eine Pseudofunktion möglich.[7]

Anforderung R7: Ersetzen beliebiger Token oder Tokenfolgen durch andere Token oder Freitext, unter Berücksichtigung der Anforderungen R4, R5 und R6. **R7**

Anforderung R8: Syntaktisch oder semantisch inkorrekte Einfügungen dürfen weitere Verarbeitungsschritte nicht stören oder unmöglich machen. **R8**

2.1.4 Weitere Modifikationen

In kleinen Testumgebungen oder wenn nur Teile eines Systems weitergegeben und dokumentiert werden, kommt es durchaus vor, dass der Quellcode nicht vollständig ausgeblendet oder ersetzt werden soll, sondern nur auskommentiert.[8]

[5] (ChilkatSoftware, 2007)

[6] (Bulterman und Rutledge, 2004, S. 287), (Lamprecht, 1988, S. 127), (Kamp und Pudlatz, 1974, S. 232), (Sturm, 2009, S. 255), (Riggs, 2003, S. 340), (Hirte, 2004)

[7] z.B. mehre Ausgabeanweisungen zu einer nicht real existieren Funktion `some_output()`

[8] (Appu, 2002, S. 182), (Paypal, 2008)

```
/*vorher*/
int somevalue;
...
somevalue=someCalculation(foo, bar);
...
/*nachher*/
int somevalue;
...
//somevalue=someCalculation(foo, bar);
somevalue=3;
...
```

Listing 2.1: Auskommentieren von Quellcode

Anforderung R9: TOKENFOLGEN MÜSSEN SEMANTISCH AUSKOMMENTIERBAR SEIN. **R9**

Weiterhin werden sowohl in gedruckter als auch in Hypertextform Querverweise zwischen Quellcode und Fließtext durch diverse Symbole, Marker bzw. Fußnoten visualisiert.[9]

Anforderung R10: EINFÜGEN VON MARKERN IN DEN QUELLTEXT. **R10**

In gedruckter Literatur steht dem Autor meist weniger Raum zur Verfügung und im Tutorialquellcode kann nicht so großzügig mit Leerraum umgegangen werden wie im Originalquelltext. Exemplarisch sei hier das Zusammenziehen von Zeilen bei kurzen get()- und set()-Anweisungen oder bei einzeiligen Blockanweisungen[10] genannt.[11] Abstrahiert folgt daraus folgende Anforderung:

Anforderung R11: EINFÜGEN UND ENTFERNEN VON LEERZEICHEN, LEERZEILEN UND ZEILENUMBRÜCHEN IN/AUS DEM QUELLTEXT. **R11**

2.2 DEFT-Architektur

Der folgende Abschnitt soll einen kurzen Überblick über die für die Erweiterungen relevante Architektur von DEFT geben. Im Vorgriff auf die Implementierung soll weiterhin dargelegt werden, dass diese Architektur ohne signifikante strukturelle Änderungen die gegebenen Anforderungen erfüllen kann.

[9] (Steppan, 2007, S. 358), (Horstmann und Budd, 2009, S. 598), (Nørmark, 2000b)
[10] if, for, while, ...
[11] (Hunt und Thomas, 2000, S. 31), (Martin, 2009, S. 89)

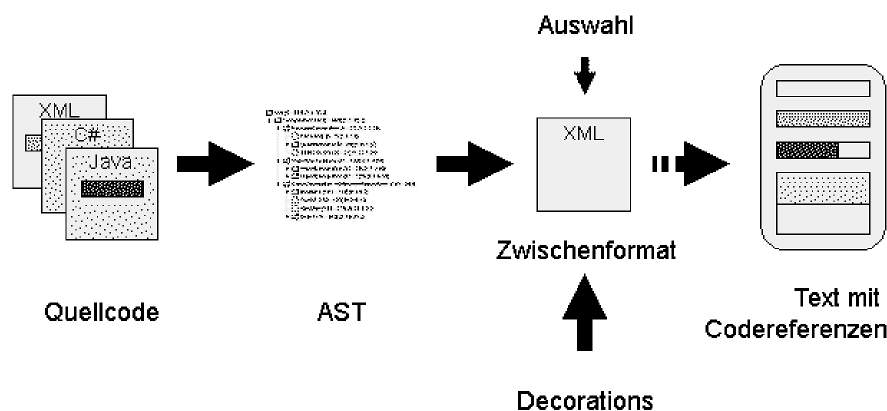

Abbildung 2.1: DEFT-Architektur

Abbildung 2.1 zeigt den schematischen Aufbau der aktuellen DEFT-Architektur. Es werden komplette Quelldateien eingelesen und als *abstrakter Syntaxbaum* (AST) in einer internen Repräsentation abgelegt. Um das Layout des Quellcodes zu erhalten, wird zu jedem Token die Position aus der Quelldatei gespeichert. Jeder Knoten kann mit sogenannten *Decorator*s[12] um weitere Informationen[13] für die Weiterverarbeitung und Ausgabe angereichert werden.

abstrakter Syntaxbaum

Decorator

Dieser angereicherte Syntaxbaum wird dann unter Einbeziehung der Informationen, welche Abschnitte sichtbar bzw. ausgeblendet sind, in einem XML-Zwischenformat an die weiteren Aufbereitungsschritte übergeben. Abschließend kann der Quellcode durch verschiedene Exporter für diverse Ausgabeformate[14] vorbereitet werden.

Durch prototypische Implementierungen konnte nachgewiesen werden, dass jede optische Änderung über die Decorators und jede strukturelle Änderung als Vorverarbeitung auf dem Syntaxbaum erfolgen kann.

2.3 Weitere Anforderungen

Aus den allgemeinen Anforderungen und den Überlegungen zur aktuellen Architektur folgt eine weitere entscheidende Anforderung:

Anforderung R12: Einfache und erweiterbare Möglichkeit der persistenten Speicherung der Modifikationen. **R12**

Weiterhin führt das Speichern der Position derzeit zu einem unerwünschten

[12]kein direkter Bezug zum Decorator-Pattern
[13]z.B. Zeilennummer, oder ob des Token ein Schlüsselwort ist
[14]derzeit HTML; später auch PDF und weitere

Nebeneffekt beim Ausblenden einzelner Token. Dabei bleiben diverse Leerzeichen sichtbar stehen, da diese nur implizit über die gespeicherte Position gegeben sind.[15]

Anforderung R13: Entfernen unnötiger Leerzeichen beim Ausblenden einzelner Token. **R13**

[15] z.B. Ausblenden von Variablenbezeichnern führt zu `public void test(int␣)`

Kapitel 3

Implementierung

Das folgende Kapitel stellt die aus der Anforderungsanalyse resultierende Umsetzung als Erweiterung von DEFT vor. Hierbei liegt der Schwerpunkt auf der abstrakten Einteilung der Werkzeuge. Bezug zur konkreten Implementierung wird im Anhang B oder über die Hinweise zur Javadoc in den Fußnoten hergestellt.

Sämtliche hier vorgestellten Module wurden testgetrieben entwickelt, somit können praktische Anwendungsfälle und weitere Details ohne Schwierigkeiten den Testfällen (Abschnitt B.5) entnommen werden.

3.1 Hervorhebung

Das Modul „Hervorhebung“ zur Begegnung der Anforderung R1 baut auf dem bereits vorgestellten Konzept des Decorators[1] auf. Durch explizite Angabe eines XPath-Ausdruckes kann jeder Knoten im abstrakten Syntaxbaum adressiert und so diese zusätzliche Information angehängt werden.

Um der Anforderung R2 gerecht zu werden, werden die konkreten typografischen Änderungen in einem Container[2] gekapselt. Im Rahmen dieser Arbeit wurden zwei exemplarische Container implementiert. Da die aktuelle Version von DEFT nur den Export nach HTML vorsieht, unterstützt ein Containertyp[3] Cascading Style Sheet(CSS) in klassischer Deklarationensschreibweise[4], und ein weiterer Container CSS in Form von Klassen[5], wie sie bei dem Import einer externen CSS-Datei in einem Hypertextdokument benötigt werden. Beide und alle weiteren Containertypen können innerhalb des Syntaxbaumes beliebig kombiniert werden.

Mit weiterem Fortschreiten des Projektes kann sich dann mit der Frage auseinander gesetzt werden, ob und in welcher Form eine für alle Exportformate gemeinsame typografische Notation definiert werden muss. Auch eine Einschränkung der Mächtigkeit einer Stylesheet-Sprache könnte dem Benutzer zur Verfügung gestellt werden. Für eine Anleitung zur Implementierung weiterer Container sei auf Abschnitt B.1 verwiesen.

3.2 Indikator für Schutz und Veränderung

In der Analyse wurde festgestellt, dass weitere Verarbeitungsschritte mit möglichen Änderungen an der Struktur des Quelltextes bzw. Syntaxbaumes Schwierigkeiten haben könnten. Daher wurde nach Anforderung R8 ein weiterer Decorator zur Indikation eines geschützten Abschnittes[6] eingeführt. Dieser beinhaltet neben der reinen An- oder Abwesenheit keine weitere Funktionalität.

Analog dazu wurde ebenfalls ein Decorator zur Markierung geänderter Abschnitte[7] definiert, der derzeit zwar für den Programmablauf nicht relevant ist, aber dem Benutzer vorgenommene Änderungen anzeigen kann.

[1] s. Javadoc StyleDecorator
[2] s. Javadoc StyleContainer
[3] s. Javadoc CssDeclarationStyleContainer
[4] [attribute:value;]
[5] s. Javadoc CssClassStyleContainer
[6] s. Javadoc ProtectedDecorator
[7] s. Javadoc ModifiedDecorator

3.3 Layoutoperation auf dem Quelltext

Die weiteren Anforderungen wurden auf drei Werkzeuge zur Modifikation des Quellcodes aufgeteilt. Die erste Gruppe, und damit das erste Werkzeug[8], umfasst Operationen, die einfache Veränderungen am Satz des Quelltextes zulassen. (Anforderung R11)

- Einfügen einer Anzahl von Leerzeichen vor/nach einem Token
- Entfernen aller Leerzeichen vor/nach einem Token
- Einfügen einer Anzahl von Leerzeilen vor/nach einem Token
- Entfernen aller Leerzeilen vor/nach einem Token
- Einfügen eines Zeilenumbruchs innerhalb einer Zeile vor/nach einem Token
- Entfernen eines Zeilenumbruchs vor/nach einem Token
- Einrückung einer Menge von Token

Diese Operationen können nach Integration in die GUI dem Benutzer vollständig zur Verfügung gestellt werden, legen aber andererseits auch den Grundstein für die komplexeren Modifikationen in den nächsten beiden Abschnitten.

Die konkrete Verwendung ist im Abschnitt B.2 beschrieben.

3.4 Veränderung am Syntaxbaum

Änderungen, die über das Layout des Quelltextes hinaus reichen, werden in einem weiteren Werkzeug[9] zusammengefasst.

- Einfügen von Knoten oder Teilbäumen vor/nach einem Token
- Ersetzen eines einzelnen oder einer Liste von Knoten
- Umwandeln von Knoten in Kommentartoken

Die ersten beiden Funktionen decken die Anforderungen R3 und R7 ab. Durch das direkte Einfügen in den Syntaxbaum ist in der weiteren Verarbeitung nicht mehr zu unterscheiden, welche Elemente aus dem Originalquelltext stammen oder welche hinzugefügt wurden (Anforderung R4). An dieser Stelle kann bei Bedarf der Indikator für veränderten Code in der Ausgabe sichtbar gemacht werden. Durch die Arbeit auf den abstrakten Token sind zunächst alle

[8]s. Javadoc ASTLayouter
[9]s. Javadoc ASTModifier

Operationen von einer konkreten Programmiersprache der Quelldatei unabhängig (Anforderung R5). Auch die Anforderung R10, das Einfügen von Markern, bedarf zunächst keiner gesonderten Behandlung, sondern kann entweder durch ein Texttoken mit einen spezifischen Syntax oder ein neues Token erfolgen. In beiden Fällen kann die Definition einer solchen Struktur durch die nachfolgenden Arbeitsschritte nach eigenen Anforderungen erfolgen.

Für das syntaktisch korrekte Einfügen und Ersetzen (Anforderung R6) stellt das Werkzeug eine Schnittstelle für einen Visitor[10,11] zur Verfügung, der die Korrektheit der Operation prüft. Auf eine vollständige Beispielimplementierung wurde verzichtet und lediglich ein Visitor[12] vorbereitet, der das Einfügen an jeder Stelle erlaubt. Es wäre auch vorstellbar, dass eine zukünftige Distribution neben den beiden genannten noch einen Visitor beinhaltet, der nur Kommentare zulässt, oder andere spezifische Einschränkungen vorgibt.

Ebenfalls sprachabhängig ist die Umwandlung einer Menge von Token in einen (oder mehrere) Kommentare (Anforderung R9). Da es selbst innerhalb einer Programmiersprache diverse Kommentarformen gibt, wurde hier ebenfalls eine Lösung über einen spezifischen Visitor[13] implementiert. Die aktuelle Version bietet somit die Umwandlung in einen Java-Blockkommentar[14] oder in mehrere Java-Zeilenkommentare[15] an. Auch hier wäre eine spätere Erweiterung auf z.B. Javadoc-Kommentare oder Kommentare anderer Sprachen vorstellbar.

Hinweise zur Verwendung und der Definition eigener Visitor sind Abschnitt B.3 zu entnehmen.

3.5 Entfernen unnötiger Leerzeichen

Um dem Problem der überzähligen Leerzeichen (Anforderung R13) zu begegnen, wurde ein weiteres Werkzeug[16] implementiert. Dieses setzt beim Ausblenden der Token bzw. Knoten an und entfernt Leerraum nach einer zuvor definierten Spezifikation. In einer sprachabhängigen Datei kann dazu angegeben werden, in welcher Richtung Leerzeichen entfernt werden müssen, wenn ein Token ausgeblendet wird. Der Aufbau ist Listing 3.1 zu entnehmen.

```
<?xml version="1.0" encoding="utf-8"?>
<autotrim language="Java" version="1.6">
 <!-- h=none|left|right|both
```

[10](Gamma u. a., 1993)
[11]s. Javadoc SyntaxCheckerVisitor
[12]s. Javadoc NoSyntaxCheckerVisitor
[13]s. Javadoc CommentTypeVisitor
[14]s. Javadoc JavaBlockCommentVisitor
[15]s. Javadoc JavaLineCommentVisitor
[16]s. Javadoc TokenAutoTrimmer

```
      v=none|before|after|both -->
 <trim h="left" v="none">
  ABSTRACT, FINAL, STATIC, SYNCHRONIZED,
 ...
 </trim>
 ...
</autotrim>
```

Listing 3.1: Automatisches Entfernen unnötiger Leerzeichen

Der autotrim-Tag enthält neben einer Angabe zur beschriebenen Sprache eine Menge von trim-Tags. Diese Tags gruppieren diverse Kombinationen aus horizontalen und vertikalen Trimmungen und enthalten eine kommagetrennte Auflistung[17] von sämtlichen auf diese Weise zu behandelnden Token und Knoten. Ein nicht gelistetes Element führt zu der Annahme, dass keine Trimmung nötig ist.

3.6 Speicherung und Wiederanwendung der Veränderungen

Die dem EP zugrunde liegende Veränderlichkeit des Originalquellcodes macht es erforderlich, dass nicht in erster Linie das Ergebnis der angewendeten Modifikationen, sondern die Abfolge Selbiger gespeichert werden muss. Um dies der Architektur und der Anforderung R12 entsprechend flexibel und offen für zukünftige Erweiterungen zu halten, wurde das in Listing 3.2 dargestellte Format zur Speicherung definiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<codesnippet>
  <modifications>
    <modification class="FACTORY">
      <param name="NAME">WERT</param>
      ...
    </modification>
   ...
  </modifications>
  <decorations>
    <decoration class="FACTORY">
      <param name="NAME">WERT</param>
      ...
```

[17] Leerzeichen und Zeilenumbrüche werden ignoriert

```
    </decoration>
    ..
  </decorations>
</codesnippet>
```

Listing 3.2: Speicherung der Veränderungen

Unter dem gemeinsamen Dokumentknoten können Modifikationen, im Sinne der Veränderung des AST, und Dekorationen, als rein visuelle Erweiterungen wie zuvor definiert, angegeben werden. Diese Trennung ist eine bewusst gewählte Unterstützung für den Benutzer, so dass alle Modifikationen auch sicher von den Dekorationen behandelt werden und keine Konflikte bezüglich der Abarbeitungsfolge auftreten. Der innere Aufbau ist dann hingegen bei beiden Tag-Arten identisch. Das class-Attribut gibt eine Javaklasse an, die als Factory[18] fungiert und ein dafür nötiges Interface [19] implementiert. Dieser Klasse können beliebige weitere Parameter übergeben werden. Eine Auflistung der aktuellen Factories und benötigter Parameter kann Abschnitt B.4 entnommen werden. Dort sind auch weitere Informationen zur Implementation einer eigenen Factory für zukünftige Werkzeuge oder Decorator zu finden.

3.7 Vorüberlegung zur Integration

Auch wenn der Schwerpunkt dieser Arbeit im Backend lag, soll dieser Abschnitt noch einen Vorschlag zur Integration in die graphische Benutzeroberfläche anbieten.

Der im vorherigen Abschnitt vorgestellten textuellen Repräsentation der auszuführenden Änderungen steht eine gleichwertige interne Repräsentation[20] gegenüber. Neben dem Laden und Speichern dieser Daten kann dem Benutzer so ein listenartiges Interface angeboten werden, welches die Konfiguration der hier vorgestellten (und weiterer) Werkzeuge ermöglicht. Das einfache Hinzufügen und Entfernen von Dekorationen und Werkzeuganwendungen sollte ein essentieller Bestandteil einer erhöhten Benutzerfreundlichkeit sein.

Im Gespräch mit potentiellen Nutzern hat sich auch der Wunsch nach einer Anwendung von DEFT in Form eines Kommandozeilenprogramms heraus kristallisiert. Auch hier könnte das vorgestellte Format ein erster Schritt in diese Richtung sein.

[18](Gamma u. a., 1993)
[19]s. Javadoc Invoker
[20]s. Javadoc Persistence

Kapitel 4

Zusammenfassung und Ausblick

Im Verlauf dieser Arbeit konnte aufgezeigt werden, dass sämtliche von Entwicklern und Autoren benutzten Modifikationen in Tutorials und in allgemeinen Dokumentationen durch eine Entwicklungsumgebung im Sinne des Elucidative Programming umgesetzt werden können. Weiterhin wurde eine Lösung aufgezeigt, die Änderungen sinnvoll zur erneuten Anwendung zu speichern.

Der nächste Schritt muss eine vollständige Integration der erarbeiteten Werkzeuge in die nächsten Versionen von DEFT sein. Der Schwerpunkt liegt auf der Integration in die GUI und der Benutzerfreundlichkeit für den Endanwender.

Weiterführend gilt es die Frage zu beantworten, welche Modifikationen auf anderen Artefakten in Tutorials (z.B. UML-Diagrammen usw.) möglich und nötig sind. Dabei wäre es auch interessant zu klären, wie diese Modifikationen abstrahiert werden können, um ein möglichst breites Spektrum an Artefakten abzudecken.

Anhang A

Verzeichnisse

Abbildungsverzeichnis

2.1 DEFT-Architektur 12

Tabellenverzeichnis

B.1 Mögliche Werkzeug- und Dekorationsanwendungen 31

Listings

2.1 Auskommentieren von Quellcode 11
3.1 Automatisches Entfernen unnötiger Leerzeichen 17
3.2 Speicherung der Veränderungen 18

Literaturverzeichnis

[Aguiar u. a. 2004] AGUIAR, Ademar ; DAVID, Gabriel ; BADROS, Greg: *JavaML 2.0: Enriching the Markup Language for Java Source Code.* 2004

[Appu 2002] APPU, Ashok: *Making use of PHP.* Wiley, 2002

[Bartho 2009] BARTHO, Andreas: Creating and maintaining tutorials with DEFT. In: *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference* (2009), May, S. 309–310. – URL `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5090072&isnumber=5090011`. – ISSN 1063-6897

[Born und Born 2006] BORN, Günter ; BORN, Benjamin: *Visual Basic 2005.* Entwickler-Press, 2006

[Briggs 1993] BRIGGS, Preston: Nuweb, A Simple Literate Programming Tool / Rice University. Houston, TX, USA, 1993. – Forschungsbericht

[Bulterman und Rutledge 2004] BULTERMAN, Dick C. A. ; RUTLEDGE, Lloyd: *SMIL 2.0.* Springer, 2004

[ChilkatSoftware 2007] CHILKATSOFTWARE: Drop Only the BR Tags. (2007). – URL `http://www.example-code.com/csharp/convert-html-to-xml-5.asp`

[Fitzgerald 2007] FITZGERALD, Michael: *Ruby - kurz & gut.* O'Reilly, 2007

[Gamma u. a. 1993] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Abstraction and Reuse of Object-Oriented Design.* 1993

[Ganor 2007] GANOR, Roy: Refactoring PHP Code. In: *Zend Developer Zone* (2007). – URL `http://devzone.zend.com/article/2514`

[Hajji 1998] HAJJI, Farid: *Perl.* Addison-Wesley, 1998

[Hewitt 2009] HEWITT, Eben: *Java SOA cookbook.* O'Reilly, 2009

[Hirte 2004] Hirte, Rolf: Programmierung mit MS Visual Basic. (2004). – URL http://rhirte.de/vb/grafgdi.htm

[Horstmann und Budd 2009] Horstmann, Cay S. ; Budd, Timothy: *Big C++*. Wiley, 2009

[Hunt und Thomas 2000] Hunt, Andrew ; Thomas, David: *The Pragmatic Programmer*. Addison-Wesley, 2000

[Jackson und McClellan 1997] Jackson, Jerry R. ; McClellan, Alan L.: *JAVA by example*. SunSoft Press, 1997

[Jan 2007] Jan: Java-Anwendung erstellen für Mac OS X. In: *Java User Group Berlin Brandenburg* (2007). – URL http://www.jug-bb.de/2007/08/java-anwendung-erstellen-fur-mac-os-x/

[Johnson und Johnson 1997] Johnson, Andrew L. ; Johnson, Brad C.: Literate Programming Using Noweb. In: *Linux Journal* 42 (1997), Oct, S. 64–69. – URL ftp://ftp.ssc.com/pub/lj/listings/issue42/2188.tgz. – ISSN 1075-3583

[Kamp und Pudlatz 1974] Kamp, Hermann ; Pudlatz, Hilmar: *Einführung in die Programmiersprache PL/I*. Vieweg, 1974

[Knuth 1984] Knuth, Donald E.: Literate Programming. In: *Comput. J.* 27 (1984), Nr. 2, S. 97–111

[Knuth und Levy 1994] Knuth, Donald E. ; Levy, Silvio: *The CWEB System of Structured Documentation: Version 3.0*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1994. – ISBN 0201575698

[Lamprecht 1988] Lamprecht, Günther: *Einführung in die Programmiersprache SIMULA*. Vieweg, 1988

[Martin 2009] Martin, Robert C.: *Clean Code*. Pearson Education, 2009

[McLaughlin 2000] McLaughlin, Brett: *Java and XML*. O'Reilly, 2000

[Meyer 1998] Meyer, André: *Java Foundation Classes 1.1 mit Swing 1.0*. Addison-Wesley-Longman, 1998

[MSDN 2009] MSDN: Lernprogramm für Indexer. (2009). – URL http://msdn.microsoft.com/de-de/library/aa288465(VS.71).aspx

[Nørmark 2000a] Nørmark, Kurt: Requirements for an Elucidative Programming Environment. In: *IWPC '00: Proceedings of the 8th International Workshop on Program Comprehension*. Washington, DC, USA : IEEE

Computer Society, 2000, S. 119. – URL http://www.cs.aau.dk/~normark/elucidative-programming/papers/req-paper.pdf. – ISBN 0-7695-0656-9

[Nørmark 2000b] Nørmark, Kurt: Time Conversion Tutorial. (2000). – URL http://www.cs.aau.dk/~normark/elucidative-programming/time-conversion/time/time.html

[Nørmark u. a. 2000] Nørmark, Kurt ; Andersen, Max ; Christensen, Claus ; Kumar, Vathanan ; Staun-Pedersen, Søren ; Sørensen, Kristian: Elucidative programming in Java. In: *IPCC/SIGDOC '00: Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation.* Piscataway, NJ, USA : IEEE Educational Activities Department, 2000, S. 483–495. – ISBN 0-7803-6431-7

[Paypal 2008] Paypal: Instant Payment Notification - Code Samples. (2008). – URL https://www.paypal.com/us/cgi-bin/webscr?cmd=p/pdn/ipn-codesamples-pop-outside

[Pomberger und Dobler 2008] Pomberger, Gustav ; Dobler, Heinz: *Algorithmen und Datenstrukturen.* Pearson Studium, 2008

[Riggs 2003] Riggs, Roger: *Programming wireless devices with the Java 2 Platform.* Addison-Wesley, 2003

[Steiner 1991] Steiner, Josef: *Turbo Pascal 6.0.* Markt-und-Technik-Verlag, 1991

[Steppan 2007] Steppan, Bernhard: *Einstieg in Java 6.* Galileo-Press, 2007

[Stroustrup 1990] Stroustrup, Bjarne: *Die C++ Programmiersprache.* Addison-Wesley, 1990

[Sturm 2009] Sturm, Eberhard: *The new PL/I.* Vieweg + Teubner, 2009

[Ullenboom 2007] Ullenboom, Christian: *Java ist auch eine Insel.* Galileo-Press, 2007

[Vestdam 2002] Vestdam, Thomas: *Generating Consistent Program Tutorials.* 2002

[Weiss 1989] Weiss, Dieter: *Occam 2.* Hanser, 1989

[Williams 1992] Williams, Ross: FunnelWeb User's Manual / University of Adelaide. Adelaide, South Australia, Australia, 1992. – Forschungsbericht

[Wirfs-Brock und McKean 2003] Wirfs-Brock, Rebecca ; McKean, Alan: *Object design.* Addison-Wesley, 2003

Anhang B

Dokumentation

Im Anhang B werden Fragestellungen im Zusammenhang mit der konkreten Implementierung behandelt. Weiterhin wird ein Überblick über die mögliche Verwendung der in der Arbeit vorgestellten Werkzeuge gegeben.

B.1 Einen StyleContainer implementieren

Ein konkreter `StyleContainer` erbt drei Methoden.

```
public abstract void parse(String text)
public abstract String renderValueAttribute()
public abstract String renderIdAttribute()
```

Die Methode `parse` liest das angegeben Format ein und speichert es in einer internen Repräsentation ab. Diese Methode wird ggf. auch vom Konstruktor verwendet.

Die anderen beiden Methoden liefern eine für das XML-Zwischenformat aufbereitete textuelle Form des Containerinhalts und einen Bezeichner, der den Containertyp für weitere Verarbeitungen angibt. Zur Orientierung ist ein Blick in die Klassen `CssDeclarationStyleContainer` und `CssClassStyleContainer` angeraten.

Im Java-Quellcode finden die Container anschließend als Argument für die `StyleInformation`s Verwendung. Alternativ dazu kann der Containertyp auch in der in Abschnitt 3.6 vorgestellten XML-Datei angegeben werden. Weitere Details dazu in Abschnitt B.4.

B.2 Umfang des ASTLayouters

Für den Entwickler besteht die Möglichkeit, den ASTLayouter direkt aufzurufen und so in eigenen Tools zu verwenden. Hierbei wir dem Konstruktor eine serialisierte Version des Syntaxbaumes übergeben.

Anschließen stehen Funktionen zum Einfügen von Leerzeichen

```
public void insertSpacesBefore(Token targetToken, int offset)
public void insertSpacesAfter(Token targetToken, int offset)
```

Einfügen von Leerzeilen

```
public void insertLinesBefore(Token targetToken, int offset)
public void insertLinesBefore(int offset)
public void insertLinesAfter(Token targetToken, int offset)
```

Einfügen und Entfernen von Zeilenumbrüchen

```
public void insertLineBreakBefore(Token targetToken)
public void insertLineBreakAfter(Token targetToken)
public void removeLineBreakAfter(Token targetToken)
public void removeLineBreakBefore(Token targetToken)
```

Trimmen von Leerzeichen und Leerzeilen

```
public void trim(Token targetToken)
public void trimLeft(Token targetToken)
public void trimRight(Token targetToken)
```

```
public void trimLinesBefore(Token targetToken)
public void trimLinesAfter(Token targetToken)
```

Einrücken von Tokenfolgen

```
public void indentRelative(Token startToken, Token endToken,
        int offset)
public void indentRelative(int offset)
public void reset()
```

zur Verfügung.

Nach Abschluss aller Operationen **muss** einmalig die Methode `repairOffset` aufgerufen werden.

Der Aufruf durch den Benutzer erfolgt über die in Abschnitt 3.6 vorgestellte XML-Datei bzw. zukünftig über die GUI. Weitere Details dazu in Abschnitt B.4.

B.3 Verwendung des ASTModifiers

Auch der `ASTModifier` kann innerhalb neuer Werkzeuge direkt verwendet werden. Sämtliche Methoden erhalten den aktuellen Syntaxbaum sowie einen XPath-Ausdruck des Zielknotens, an dem die Veränderung ansetzt.

Im Weiteren benötigen die Methoden zum Einfügen den einzufügenden Teilbaum und können noch Angaben zu Leerzeichen und -zeilen um die betroffene Stelle enthalten.

```
public void appendAfter(TreeNode ast, TreeNode insert,
        XPath location,int emptyLinesBefore, int emptyLinesAfter,
        int spacesBefore, int spacesAfter)
public void appendBefore(TreeNode ast, TreeNode insert,
        XPath location, int emptyLinesBefore, int emptyLinesAfter,
        int spacesBefore,int spacesAfter)
public void replace(TreeNode ast, TreeNode insert,
        XPath location, int emptyLinesBefore, int emptyLinesAfter,
        int spacesBefore,int spacesAfter)
```

Bei der Ersetzung von mehreren Token definiert der Parameter `width` die Anzahl der nachfolgenden Geschwisterknoten, die neben dem Ziel auch betroffen sind. Die Angabe `-1` umfasst dabei alle möglichen Knoten.

```
public void replaceList(TreeNode ast, TreeNode insert, XPath
        location, int width, int emptyLinesBefore, int
        emptyLinesAfter, int spacesBefore, int spacesAfter)
public void changeToComment(TreeNode ast, XPath location,
        int width, CommentTypeVisitor v)
```

B.3.1 CommentTypeVisitor

Ein Visitor für die Umwandlung von Knoten in Kommentarknoten implementiert das Interface `CommentTypeVisitor` und stellt damit eine Methode

```
public abstract List<TreeNode> execute(List<TreeNode> nodes)
```

zur Verfügung. Innerhalb dieser werden die Knoten in Kommentare umgewandelt. Zur Unterstützung bietet die Klasse `CommentTypeVisitor` eine Methode

```
protected String treenodesToString(List<TreeNode> nodes)
```

an, welche eine Liste von TreeNodes in eine flache, die Positionen berücksichtigende, textuelle Repräsentation umwandelt. Beispielimplementierungen bieten die Klassen `JavaBlockCommentVisitor` und `JavaLineCommentVisitor`.

B.3.2 SyntaxCheckerVisitor

Ein für den Konstruktoraufruf des ASTModifiers notwendiger SyntaxChecker-Visitor implementiert eine Methode

```
public abstract boolean check(TreeNode parent, TreeNode insert,
        int insertAt)
```

die den Elternknoten, den einzufügenden Knoten und dessen Position innerhalb der Liste der Kindknoten benötigt. Als Rückgabe liefert dieser Aufruf `true`, wenn die Einfügung zulässig ist.

Der Aufruf sämtlicher Methoden durch den Benutzer erfolgt über die in Abschnitt 3.6 vorgestellte XML-Datei bzw. zukünftig über die GUI. Weitere Details dazu in Abschnitt B.4.

B.4 Anwendung von Werkzeugen und Dekorationen

Neben dem bereits in Abschnitt 3.6 vorgestellten XML-Format können die Werkzeugfolgen auch direkt zur Laufzeit aufgebaut werden. Nach der Instantiierung stellt ein `Persistence`-Objekt neben dem Laden aus bzw. dem Speichern in eine Datei Methoden zur Verfügung, um `Modifications` und `Decorations` aufzunehmen. Diese bieten beide einen Konstruktor, der einen `Invoker` und eine Parameter-Map benötigt.

```
public Item(Invoker invoker, Map<String, String> params)
```

Die Angabe der Parameter erfolgt analog zum XML-Format.

Nachdem ein `Persistence`-Objekt geladen oder erzeugt wurde, kann die Methode

```
public void apply(TreeNode ast)
```

aufgerufen werden und sämtliche Änderungen werden auf den Syntaxbaum angewendet.

Um bei Bedarf weitere Factories zu schreiben, implementieren diese das Interface `Invoker` mit der Methode

```
public void invoke(TreeNode ast, Map<String, String> params)
```

die die Anwendung des Werkzeuges oder der Dekoration realisiert. Gute Beispiele liefern hier die Klassen `StyleDecoratorInvoker` und `ASTModifierInvoker`.

Tabelle B.1 zeigt einen Überblick über alle aktuellen Invoker und deren Parameter.

B.5 Quellcode/Testfälle/Javadoc

Der Quellcode einschließlich Testfällen und Javadoc kann über [1] bezogen werden.

[1] http://www.mheinzerling.de/tud.php

MODIFIKATIONEN	
ASTLayouterInvoker (org.deft...tools.astlayouter.ASTLayouterInvoker)	
method	Mögliche Methoden: *insertSpacesBefore*, *insertSpacesAfter*, *insertLinesBefore*, *insertLinesAfter*, *insertLineBreakBefore*, *insertLineBreakAfter*, *removeLineBreakAfter*, *removeLineBreakBefore*, *trim*, *trimLeft*, *trimRight*, *trimLinesBefore*, *trimLinesAfter* und *indentRelative*
offset	Ganzzahl; nötig bei *insertSpacesBefore*, *insertSpacesAfter*, *insertLinesBefore*, *insertLinesAfter* und *indentRelative*
location	XPath des Zielknoten; notwendig bei allen außer *indentRelative*
startlocation	XPath des Startknoten; notwendig bei *indentRelative*
endlocation	XPath des Zielknoten; notwendig bei *indentRelative*
ASTModifierInvoker (org.deft...tools.astmodifier.ASTModifierInvoker.modifier)	
method	Mögliche Methoden: *appendAfter*, *appendBefore*, *replace*, *replaceList* und *changeToComment*
location	XPath des Zielknoten; notwendig
syntaxchecker	Klasse des `SyntaxCheckerVisitor`; optional (NoSyntaxCheckerVisitor)[a]
insert	Zeichenkette; DATEI+"||"+XPATH schneidet einen (Teil-)Baum aus einer Datei TOKENNAME+"||"+TOKENINHALT erzeugt einen Token (K_CLASS_BODY_DECLARATIONS | K_COMPILATION_UNIT | K_EXPRESSION | K_STATEMENTS)[b]+ „||"+QUELLTEXT|DATEI+"||"+XPATH; z.B. K_EXPRESSION||x=3||/Expression[c]; notwendig außer bei *changeToComment*
emptyLinesBefore	Ganzzahl; optional (0) außer bei *changeToComment*
emptyLinesAfter	Ganzzahl; optional (0) außer bei *changeToComment*
spacesBefore	Ganzzahl; optional (0) außer bei *changeToComment*
spacesAfter	Ganzzahl; optional (0) außer bei *changeToComment*
width	Ganzzahl; optional (0) bei *changeToComment* und *replaceList*
commentType	Klasse des `CommentTypeVisitor`; notwendig bei *changeToComment*

DEKORATIONEN	
StyleDecoratorInvoker (org.deft...decoration.style.StyleDecoratorInvoker)	
location	XPath der Zielknoten; notwendig
container	Klasse des `StyleContainer`; notwendig
value	zu parsender Inhalt des Containers
ModifiedDecoratorInvoker (...decoration.modify.ModifiedDecoratorInvoker)	
location	XPath der Zielknoten; notwendig
ProtectedDecoratorInvoker (...decoration.protect.ProtectedDecoratorInvoker)	
location	XPath der Zielknoten; notwendig

[a]Standardwert
[b]s.a. http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/ASTParser.html
[c]Im Rahmen dieser Arbeit nur Java

Tabelle B.1: Mögliche Werkzeug- und Dekorationsanwendungen