Verfolgen von Kodekopien zur Defektvermeidung in Eclipse

Diplomarbeit (Informatik)

eingereicht durch:

Sofoklis Papadopoulos, Matrikelnr.: 2798882
am 15.01.2007

Institut für Informatik, Freie Universität Berlin,

Arbeitsgruppe Software-Engineering

Erstprüfer: Prof. Dr. Lutz Prechelt

Zweitprüfer: Prof. Dr. Elfriede Fehr

1. Eidesstattliche Erklärung:

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Diplomarbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben sind.

Berlin, den 15.01.2007 ……..
(Unterschrift)

3. Danksagungen

Ich möchte mich bei Sebastian Jekutsch für die gute Betreuung während meiner Arbeit bedanken.

Ein Dank geht an Herrn Prof.Dr. Lutz Prechelt den Leiter der Arbeitsgruppe Software-Engineering.

Ein Dankeschön geht an Frau Prof.Dr. Elfriede Fehr für die Zweitkorrektur.

Mein Dank gilt meinen Eltern, die mich immer unterstützt haben.

Bei meiner Lebenspartnerin Meltem Tunca möchte ich mich für ihre lange Geduld bedanken.

Des Weiteren möchte ich mich bei meinem Freund Stefan Ahl für seinen Zuspruch und seine offene Kritik bedanken.

Inhaltsverzeichnis

4. ZUSAMMENFASSUNG **6**

5. EINLEITUNG **6**

5.1 Allgemeine Einführung 6
5.2 Problemstellung 9
5.3 Motivation 11

6. THEORIE DER COPYPASTECHANGE-EPISODE **13**

6.1 Untersuchungen von CopyPasteChange-Praktiken 13
6.2 CopyPasteChange-Episoden 15
6.2.1 Aktivitäten die zur CopyPasteChange-Episode führen 15
6.2.2 Halbformale Beschreibung der CPC-Episode 18
6.2.3 Mehrmalige Einfügungen von kopierten oder ausgeschnittenen Kode 20
6.2.4 Kodefragmente in verschiedenen Dateien 20
6.2.5 Warum ist die bisherige Sichtweise von Original und Kopie aufzugeben? 21
6.2.6 Wann werden Klone in die Klonfamilie aufgenommen? 24
6.2.7 Wie lange müssen Klone beobachtet werden? 25
6.3 Formalisierung von CopyPasteChange-Episoden 26
6.3.1 Erläuterungen zu der Formalisierung 26
6.3.2 Welche C&P-Aktivitäten führen zur Verfolgung von Klonen? 28
6.3.3 Problem der Stellenidentität 30
6.3.4 Wie lange stehen Klone unter Beobachtung ? 33
6.3.5 Abschluss einer CopyPasteChange-Episode 37

7. IMPLEMENTIERUNG **38**

7.1 Arbeitsweise des ElectroCodeoGram (ECG) 39
7.2 Identifikation von CopyPasteChange-Episoden mittels ECG 39
7.2.1 Relevante Ereignisse des ECG 39
7.2.2 Vorgehensweise um das Problem der Klonverfolgung zu lösen 42
7.3 In die ECG-Software integrierte Episodenerkenner 42
7.3.1 Das EpisodeRecognizerIntermediateModule 42
7.3.2 Der EpisodeRecognizerManager 43
7.3.3 Das EpisodeRecognizer Interface 43
7.3.4 Beispiel für einen Episodenerkenner 44
7.4 Der CopyPasteChangeEpisoden-Erkenner als Zustandsautomat 46
7.5 Umsetzung des CPC-Episoden-Zustandsautomaten 58

8. EVALUATION **79**

8.1 Umgebungsbedingungen 79
8.2 Evaluationsbeschreibung 82
8.3 Evaluationssergebnis 83

9. FAZIT UND AUSBLICK **95**

10. REFERENZEN **96**

11. ANHANG **96**

4. Zusammenfassung

Fehler[1] entstehen in der Software auf unterschiedlichste Art und aus unterschiedlichen Gründen. Sie werden unbewusst von einem Programmierer produziert. Diese unbewusst produzierten Fehler führen zu Defekten[2] in der Software. Defekte können zu einem nicht erwarteten Verhalten der Software führen, die bezüglich der Spezifikation und der Anforderungen an diese Software nicht korrekt ist. Programmierer benutzen beim Kodieren der Software bestimmte Methoden, die ihnen ihre Tätigkeit, das Programmieren erleichtert. Eine häufig angewandte Methode ist das Selektieren von Kodeschnipseln[3] des Quelltextes der Software nach bestimmten zuvor persönlich durchdachten Kriterien. Diese selektierten Kodeschnipsel werden nach dem Selektieren entweder *ausgeschnitten* oder *kopiert*, um sie dann an einer anderen oder mehreren verschiedenen Stellen des Quelltextes *einzufügen*.
Nach dem Einfügen dieser Kodeschnipsel, werden sie zum jeweiligen Kontext hin angepasst. Diese Methodik nennt man im Allgemeinen *Copy And Paste* (also kopieren und einfügen). Wenn nun der ausgeschnittene oder kopierte Kodeschnipsel, also das O*riginal*, verändert wird, sollten auch alle eingefügten und angepassten Kodeschnipsel, die *Kopien* des Originals, ebenfalls verändert werden. Um die Veränderung der Kodekopien durchführen zu können müssen diese Kodekopien und das Original während der Evolution einer Software hinsichtlich etwaiger Veränderungen beobachtet werden.
Im Verlauf dieser Arbeit müssen wir uns von dieser Sichtweise, also von einem Original und von einer oder mehreren Kopien zu sprechen trennen und uns an eine differenziertere Betrachtung dieser *CopyAndPaste-Methode* annähern.
Der oben beschriebene Sachverhalt, das Verfolgen von Kodekopien in Eclipse[1] mit dem Fernziel der Defektvermeidung, soll mit Hilfe dieser Arbeit durchleuchtet, analysiert und schließlich ermöglicht werden.

5. Einleitung

5.1 Allgemeine Einführung

In der modernen Softwareentwicklung und insbesondere bei der Programmierung, also dem Implementieren, können Fehler auf verschiedenste Weise und aus unterschiedlichen Gründen durch den Programmierer produziert werden. Diese unentdeckten, eingefügten Fehler können dazu führen, dass die zu entwerfende und implementierende Software dann schließlich nicht mehr die gewünschten Ergebnisse liefert.
Eingeschlichene und nicht erkannte Fehler können also folglich zu Defekten in der Software führen. Den Softwareherstellern drohen dadurch möglicherweise wirtschaftliche Schäden durch Schadensersatzansprüche ihres Kunden. Ferner kommen zu den eben erwähnten wirtschaftlichen Schäden womöglich noch Risiken hinzu, die Menschenleben bedrohen

[1] Ein Fehler wird durch einen Menschen verursacht.
[2] Ein Defekt in der Software entsteht durch einen oder mehrere Fehler, die in den Programmkode produziert worden sind.
[3] Ein aus einer oder mehrerer Zeilen bestehendes Kodefragment des Quelltextes einer Software.

könnten (je nach dem um was für eine Art von Software es sich handelt, z.B. Reaktorkontrollsoftware eines Kernkraftwerks).
Diese produzierten Fehler können auf verschiedenste Arten klassifiziert werden. Zum Beispiel die Klassifizierung danach, ob Fehler während des Implementierens erkannt werden und damit auch unverzüglich und ohne größeren Aufwand behoben werden können. Im Gegensatz dazu die Klassifizierung von Fehlern, die nach Auslieferung und Inbetriebnahme der Software erkannt werden. Diese Fehler sind dann zu diesem Zeitpunkt meistens nicht mehr ohne größeren Aufwand und nur noch mit zusätzlich verbundenen hohen Kosten zu beheben.

Andererseits können Fehler semantischer oder syntaktischer Natur sein. Syntaktische Fehler beziehen sich auf die nicht eingehaltene Syntax, also die Struktur des Programms und auf die Regeln dieser Struktur. Diese Regeln nennt man auch Syntaxregeln. Die syntaktischen Fehler in einem Programm führen dazu, dass es nicht korrekt kompiliert, also übersetzt wird und dadurch folglich nicht ausführbar ist. Der Programmierer muss diesen syntaktischen Fehler zunächst beheben, damit er das Programm übersetzen und dann natürlich auch ausführen kann.

Entwicklungsumgebungen, so genannte IDE's (steht für Integrated Development Environment), können diese Art von Fehler bereits erkennen und für den Programmierer visuell erkennbar machen. Die Entwicklungsumgebung Eclipse signalisiert solche Syntaxfehler und gibt dem Programmierer Lösungsvorschlage beim Implementieren von Java-Programmen zum Auswählen an. Das Fehlen einer schließenden Klammer in einem komplexen Klammerausdruck ist z.B. ein typischer Syntaxfehler.

```
        .
        .
        .
if(((dr.no < bond )&&(bond > bourne)&&(magnum == poirot)){
      do something . . .
}
        .
        .
        .
```

Abb.1: Komplexer Klammerausdruck mit fehlender abschließender Klammer.

Die IDE'S erleichtern bei dieser Art von Fehlern die Arbeit des Programmierers. Sie unterbreiten dem Programmierer Lösungsvorschläge, wie die von ihm produzierten Syntaxfehler bereinigt werden sollen. Der Programmierer muss nur noch abwägen welcher Lösungsvorschlag zum gewünschten Ziel führt und wählt diesen aus. Die IDE führt dann die vom Programmierer gewählte Aktion zur Beseitigung des Fehlers aus. Bei der Benutzung von Klassen aus nicht importierten Klassenbibliotheken zum Beispiel, wird ein Fehler signalisiert und es wird seitens der IDE vorgeschlagen das benötigte Paket zu importieren. Will der Programmierer dieses tun, kann er den Lösungsvorschlag annehmen und von der IDE ausführen lassen. Die syntaktischen Fehler sind demnach leicht zu erkennen und ebenso leicht zu beheben, da die Syntax bestimmten Regeln zu gehorchen hat. Die Syntax ist korrekt, wenn sie mit den Regeln übereinstimmt, dagegen ist sie nicht korrekt, wenn sie nicht diesen Syntaxregeln entspricht.

Bei den semantischen Fehlern, oftmals auch logische Fehler genannt, sieht das weitaus schwieriger aus. Das Programm wird übersetzt, ist also ausführbar, aber es tut nicht das, wofür es eigentlich konzipiert wurde. Es führt tatsächlich genau das aus, was der Programmierer dem Programm, mit seinen in den Programmkode produzierten semantischen Fehlern, aufgetragen hat! Oder aber das Programm wird gestartet und führt eben durch diesen produzierten logischen Prorammierfehler während der Laufzeit zum Programmabbruch!

Ein semantischer Fehler tritt z.B. auf, wenn in einem Vector mittels einer *for-Schleife* nach einem bestimmten Element gesucht werden soll und der Programmierer die Laufvariable *i* falsch initialisiert und dadurch ein Laufzeitfehler provoziert wird.

```
.
.
.
Vector vec = new Vector();

.
.  //verschiedene Elemente in vec einfügen in einer anderen
.  //Methode

      for(int i = 0; i = vec.size(); i++){
        Element elem = vec.get(i);
         if(elem == searchedElement){
          System.out.println("Element:"+elem+"gefunden! ");
.
.
.
```

Abb. 2: Semantischer Fehler durch falsches Initialisieren der Laufvariable i.

Abbildung 2 macht deutlich, dass der Programmierer bei dieser Schleife eines nicht bedacht hat. Wenn der Vector Elemente des Typs Element enthält, wird diese for-Schleife immer mit einem Laufzeitfehler abbrechen. Kann die Methode die für das Einfügen der Elemente verantwortlich ist, aus irgendwelchen Gründen kein Element einfügen, so wird diese for-Schleife niemals mit einem Laufzeitfehler abbrechen. Es ist leicht nachzuvollziehen, dass noch weitaus kompliziertere, verschachtelte und unübersichtliche Programmlogik mit semantischen Fehlern existieren kann. Vor allem in der objektorientierten Programmierung kann man leicht den Überblick bei komplexen Softwaresystemen mit sehr vielen implementierten und benutzten Klassen und ihren auf sich arbeitenden Methoden verlieren.
Diese logischen Fehler sollten, so früh wie möglich, also noch im Entwicklungsprozess der Software erkannt werden. Zusätzlich wäre es eine immense Erleichterung, wenn dem Programmierer auch bei semantischen Fehlern Lösungsvorschläge durch die IDE angeboten werden würden, wie bei den weiter oben beschriebenen syntaktischen Fehlern. Diese Lösungsvorschläge sollten dann im idealen Fall dazu führen, dass die Veränderungen des Programmkodes hinsichtlich der Lösungsvorschläge nicht mehr zu dem vorher geschilderten Versagen[4] der Software führen. Es wird praktisch während der Fehlerentstehung eine entsprechende Meldung an den Programmierer geleitet. Beim Kopieren und Einfügen können solche Fehler auftreten, da der Programmierer eventuell in einem kopierten

[4] Ein Versagen ist ein falsches Verhalten des Programms bezüglich der Spezifikation.

Kodefragment bereits einen logischen Fehler produziert hat, der schließlich beim Einfügen an ein oder mehreren Stellen weiterverbreitet wird.

Um Defekte, die durch produzierte Fehler während der Implementierung entstehen, früh genug zu erkennen und gegebenenfalls abzuwenden, könnte es eine Lösung sein sich den Prozess der Softwareentwicklung und hier genauer den Programmierprozess zu betrachten und zu analysieren.

Das führt zu dem Begriff des *Mikroprozesses*[5] *der Softwareentwicklung* [1], der von Sebastian Jekutsch eingeführt wurde. Der Mikroprozess umfasst alle Aktivitäten eines Programmierers während des Implementierens von Software. Bei der Verwendung einer Entwicklungsumgebung während der Softwareerstellung fallen unter diese Aktivitäten z.B. das Kodieren des Programmkodes, das Öffnen eines Fensters in der Enwicklungsumgebung, das Debuggen des Kodes, das Ausführen des Programms, das Kopieren von Kode, das Einfügen von kopiertem Kode und vieles mehr. Bei der Betrachtung und der Analyse des Mikroprozesses der Softwareentwicklung ist man in der Lage die Klonverbreitung in einer Software, die durch das Kopieren und Einfügen von Kode ein Problem darstellt, explizit zu beobachten.
Alle diese Aktivitäten des Mikroprozesses sind Ereignisse, die in Beziehung zueinander stehen. In bestimmter Reihenfolge sequentiell aufgetreten können diese Ereignisse so genannte Episoden bilden.

Eine Episode ist z.B. die *CopyPasteChange-Episode*. Um eine *CopyPasteChange-Episode* handelt es sich, wenn der Programmierer ein Kodeschnipsel im Quelltext selektiert und kopiert, um schließlich diesen kopierten Kodeschnipsel dann an anderer Stelle im Quelltext ein oder mehrere Male einzufügen und nach eigenen Vorstellungen abzuändern.

Mit Hilfe geeigneter Werkzeuge und Techniken, kann man nun diese Ereignisse und die dadurch resultierenden Episoden erfassen, erkennen, analysieren und auswerten, um so letztendlich Methoden zur Defekterkennung und damit zur Defektvermeidung entwickeln zu können. Das *ElectroCodeoGram* (ECG)[2] ist eine Software, die es ermöglicht genau solche Ereignisse, die während des Programmierprozesses unter Benutzung der Entwicklungsumgebung Eclipse auftreten, aufzuzeichnen. Das ECG besitzt Sensoren, die diese Ereignisse, in diesem Zusammenhang nennt man sie Mikroprozessereignisse, aufzeichnet. Diese Mikroprozessereignisse und die in ihnen enthaltenen Informationen müssen dann nur noch aufbereitet und analysiert werden, um bezüglich der Episoden neue Erkenntnisse zu liefern. Diese Erkenntnisse können somit verwendet werden um Defekte in der Software bereits im Entstehungsprozess der Software zu ermitteln.

5.2 Problemstellung

Programmierer kopieren sehr oft Kodeschnipsel um sie an anderer Stelle des Quelltextes einzufügen. Sie kopieren diese Kodeschnipsel aus verschiedenen Quellen wie Softwaredokumentationen, Algorithmenbeschreibungen, nicht selbst produziertem Quelltext oder auch aus ihren eigenem produzierten Quelltext, um diese Kodeschnipsel schließlich im aktuellen Softwareprojekt zu verwenden. Im Verlauf des Softwareentwicklungsprozesses

[5] Mikroprozess der Softwareentwicklung (Sebastian Jekutsch) siehe: http://projects.mi.fu-berlin.de/w/bin/view/SE/MicroprocessHome

ändern Programmierer diese ehemals eingefügten Kodeschnipsel oft ab, da sich Bedingungen in Bezug auf diesen Kodeschnipsel verändert haben. Wenn die Idee aus der das Kopieren entstand als nicht ganz nützlich erwiesen hat, entfernen sie diesen eingefügten Kodeschnipsel sogar in voller Länge aus dem Quelltext der Software.

```
Event cutEvent = null;
//new cpcTree data-structure to store the cut-event in the vector cpcTree.vEvents
this.cpcTree = new CPCTree();
cutEvent = new Event(cuttedcode, userName, projectName, activity, cuttedfile,
                     cuttedcodeStartLine, cuttedcodeEndLine, cuttedDate);
if((cutEvent != null) && (cpcTree != null)){
    this.cpcTree.vEvents.add(cutEvent);
}

treeSize = this.cpcTree.vEvents.size();
```

Abb.3: Ausschnitt eines Quelltextes mit blau markiertem und kopiertem Kodeschnipsel.

Bei Betrachtung der Abbildung 3 kann man sich nun leicht vorstellen, dass dieser blau markierte Kodeschnipsel aus einer Java-Klasse A.java in derselben Java-Klasse an einer anderen Stelle eingefügt werden könnte. Natürlich kann man diesen Kodeschnipsel auch in einer beliebigen anderen Java-Klasse einfügen. Der Programmierer würde diesen Kodeschnipsel an eine für ihn aus bestimmten Gründen stimmige Stelle einfügen.
Beim Kopieren und Einfügen wird nun oft unnötig Kode dupliziert. Ein vorher kopierter und eingefügter Kodeschnipsel, der einen semantischen Fehler enthält, wird aber zudem genau diesen Fehler an die eingefügte Stelle im Kode vermutlich vererben. So hat man diesen Fehler unbeabsichtigt dupliziert. Man kann erahnen was für ein Ausmaß die Defekte solcher Kodeeinfügungen entwickeln können.

Andererseits wird wie bereits erwähnt Kode unnötigerweise dupliziert, was auch zu späteren Nebeneffekten führen kann. Dieser Effekt tritt auf, wenn man beim Warten der Software mehrere Stellen im Quelltext hat, die man Anpassen muss, da diese Kodestellen in einer Relation zueinander stehen und somit Abhängigkeiten zwischen ihnen bestehen.
Im Falle der Fehlerduplizierung ist die Wartung der Software ebenfalls problematisch, da hier zunächst die Kodekopien lokalisiert werden müssen, um dann die Fehler zu beseitigen. Aber wie kann sich ein Programmierer während der Evolution der Software an genau alle Kodekopien erinnern, die er jemals für diese Software erzeugt hat?

Forscher aus dem Bereich der Softwareentwicklung empfehlen das Programmierer diese Art von Programmiermethode vermeiden sollten [4] [5], da wie schon erwähnt diese durch Kopieren und Einfügen erzeugten Duplikate problematisch z.B. im Hinblick auf die Wartung der Software sind.

Trotz der Kenntnis dieser Problematik bleibt das Kopieren und das Einfügen weiterhin eine oft verwendete Praxis der Programmierer. Diesen Sachverhalt untersuchte Miryung Kim in einer Studie [6], die auch die Basis meiner Arbeit ist. Da die Programmiertechnik des Kopierens und Einfügens von Kode in der Praxis auch von Profiprogrammierern weiterhin stetig verwendet wird, sollte die Forschung wissenschaftlich entwickelte Methoden für die Lösung dieser Problematik der Kodekopien anbieten. Im Rahmen meiner Arbeit werde ich ein Werkzeug zur Verfügung stellen, dass kopierte und eingefügte Kodeschnipsel über die Zeit hinweg verfolgt und bei eventuellen Anpassungen, also textlichen Abänderungen dieser beobachteten Kodeschnipsel, Warnungen an den Programmierer leitet, um mögliche daraus

resultierende Defekte zu vermeiden. Diese Defekte können bei inkonsistenten Veränderungen dieser Kodeschnipsel entstehen, da zwischen diesen Kodeschnipsel Abhängigkeiten bestehen.

Unter Beobachtung stehen solche Kodekopien, die während einer Softwareentwicklung unter Verwendung der Java-Entwicklungsumgebung Eclipse produziert werden. Diese Kopien nennt man Klone. Dabei ist zu beachten, dass die zu beobachtenden Klone während des evolutionären Prozesses der Softwareerstellung zum einen ihre Positionen innerhalb des Quelltextes, in denen sie eingefügt wurden, zeilenweise laufend verändern können und zum anderen auch ihr textliches Aussehen ständig verändern können. Das Problem der Stellenidentität muss somit gelöst werden. Ferner kommt die Frage auf, wie lange diese Klone die nach und nach im Verlauf des Implementierens Veränderungen unterlaufen, beobachtet werden müssen? Die textliche Veränderung kann bis zu dem Extremfall des Löschens dieses zu beobachtenden Klons führen. Die Problematik des Löschens, also der Verlust der Identität des zu beobachtenden Klons, muss ebenso Berücksichtigung finden. Die Informationen der Mikroprozessereignisse müssen somit aufbereitet und analysiert werden, um Ergebnisse bezüglich einer CPC-Episoden Erkennung zu liefern.

5.3 Motivation

Im Gegensatz zum Rat der Forscher keine Copy&Paste (C&P) Programmierpraktik anzuwenden, werden von den Programmierern trotzdem weiterhin kopierte Kodeschnipsel oft als Schablonen wiederverwendet und nach dem Einfügen an den jeweiligen Kontext hin angepasst. Es ist deswegen von immenser Wichtigkeit sich mit diesem Problem der Kodekopien zu beschäftigen und letztendlich Methoden und Werkzeuge zu entwickeln, mit denen sich diese Klone während der Evolution der Software beobachten lassen können. Bei eventuellen Änderungen dieser unter Beobachtung stehenden Klone müssen Warnungen an den Programmierer gesendet werden, um die jeweiligen in Beziehung stehenden Klone ebenfalls konsistent abzuändern.

C&P-Aktionen und die daraus resultierenden Klone ersparen nicht nur das wiederholte Eintippen von Kode, sondern reflektieren unter Anderem auch wichtige Designentscheidungen des Programmierers wieder. Diese Designentscheidungen betreffen nicht unbedingt das Softwaredesign des gesamten Systems, sondern eben nur das gerade betreffende Problem, das es zu lösen gilt. Also ist mit Designentscheidung meist ein kleiner Kodeblock gemeint, der dann eben aber auch eine gewisse Kodestruktur enthält, so dass Miryung Kim hier von Design spricht. Es kann sich aber durchaus auch um eine gesamte Klassenstruktur handeln, die der Programmierer eben gerade verwenden will, um diese Struktur dann für sein zu lösendes Problem anzupassen.

```
.
.
. //Original kopiert

for(Iterator iter = messages.iterator();iter.hasNext();){
      Message current = (Message)iter.next();
      IFile iFile=
   WorkspaceUtils.getFile(currentgetFirstLocation().getClassName());
      .
      .
      .
}

-----------------------------------------------------------------------------------------------

.
.
.//Kopie eingefügt und innere Logik der for-Schleife abgeändert
for(Iterator iter = messages.iterator();iter.hasNext();){
      Message current= (Message)iter.next();
      MessageLocation messLoc= current.getLastLocation();
      IFilemarkFile=
            WorkspaceUtils.getFile(messLoc.getClasName());
.
.
.
```

Abb.4: Kopieren eines Schleifenkonstrukts (Designentscheidung) und modifizierte innere Logik dieser Kopie, die an eine andere Stelle im Quellkode eingefügt wurde.

Abbildung 4 zeigt, dass die Intention des Programmierers hier folgende war:
Er hatte ein bestimmtes Problem zu lösen, und nach Überlegungen kam er zu dem Schluss, dass die for-Schleife aus einem bestimmten Grund für die Lösung seines Problems verwendet werden könnte. Er kopierte diese Kontrollstruktur und fügte sie anderenorts ein, um sie anschließend dann, das Problem lösend, anzupassen.
Bei dieser Art von Designentscheidungen werden Abhängigkeiten zwischen den zu beobachtenden Kodeschnipseln erzeugt, da sie in gewisser Weise in Beziehung zueinander stehen.

Erzeugte Abhängigkeiten, zwischen den kopierten und eingefügten Kodeschnipseln, sind nützlich um die Software zu verstehen. Die Studie von Miryung Kim ergab, dass Programmierer ihr Erinnerungsvermögen über die Historie ihrer C&P-Aktionen benutzen, um die beobachtenden Kodeschnipsel zu verändern oder diese zu restrukturieren. Gleichwohl gibt sie an, dass die Erinnerung eines Programmierers an seine C&P-Aktionen aber eben auch kurzlebig und inakkurat sein kann. Das Beobachten der in Abhängigkeit zueinander stehenden Kodeschnipsel übernimmt dann das zu implementierende Beobachtungs-programm und entlastet das Gedächtnis des Programmierers. Somit wird keine Veränderung bei den in Abhängigkeit zueinander stehenden Klonen vernachlässigt und die Veränderungen können konsistent in allen beobachteten in Relation stehenden Klonen durchgeführt werden. Das Beobachtungsprogramm bzw. das Klonverfolgungsprogramm unterstützt während der Evolution der Software somit praktisch den Programmierer, als eine Art vorverlagerte Pflege und Wartung der Software. Bei dieser Klonbeobachtung können somit also Defekte bei der Entstehung von dupliziertem Kode vermieden werden.

6. Theorie der CopyPasteChange-Episode

In Kapitel 6 wird die Theorie zu der CopyPasteChange-Episode entwickelt und formal erfasst. Zunächst wird in Kapitel 6.1 auf die Praxis von CopyPasteChange-Praktiken eingegangen, die die Grundlage für den Theorieteil des 6. Kapitels bilden.

6.1 Untersuchungen von CopyPasteChange-Praktiken

Den theoretischen Hintergrund und damit die Grundlage dieser Arbeit erarbeitete ich mir aus der wissenschaftlichen Arbeit von Miryung Kim. Sie untersuchte in einer Studie die Copy - und Pastepraktiken (C&P-Praktiken) von Softwareprogrammierern. Das Hauptanliegen dieser Studie war, dass man C&P-Gewohnheiten von Programmierern untersuchte, um daraus Folgerungen zu schließen, die einen ersten Hinweis aufzeigen, wie man das Problem der Klone tiefer verstehen und lösen könnte. Das Problem der Klone äußert sich darin, dass Kode zum einen dupliziert wird und zum anderen das Abhängigkeiten zwischen diesen duplizierten Kodes erzeugt werden. Bei der Duplizierung von Kode werden eventuell enthaltene Fehler des kopierten und eingefügten Klons an die neue Programmkodestelle weitervererbt.

Kim und ihre wissenschaftlichen Kollegen beobachteten und untersuchten Profiprogrammierer während ihrer Programmiertätigkeit die mit Eclipse Software entwickelten. Sie verwendeten dazu einen eigens implementierten Logger, der die editierenden Aktivitäten der Programmierer aufnahm. Zusätzlich haben sie ein Werkzeug benutzt, das ihnen ermöglichte diese aufgenommenen Tätigkeiten visuell abspielen zu können, um das Aufgenommene Material schließlich zu untersuchen. Nach der Analyse der C&P-Praktiken der Programmierer erstellten sie eine Aufstellung dieser Praktiken, die eben auf dieser Analyse basierte.

Die Untersuchung begann damit fünf Programmierer bei der Programmierung in Java über eine Zeit von 50 Stunden zu beobachten. Nach dieser Beobachtung befragten sie die Programmierer jeweils zweimal ungefähr 30 Minuten bis zu einer Stunde lang. Sie identifizierten und dokumentierten verschiedene individuelle C&P-Operationen. Sie bezeichneten diese Operationen als *Instanzen*. Eine Instanz besteht aus einer Copy-Aktion bzw. Cut-Aktion von Kodeschnipseln auf die dann unmittelbar darauf eine bzw. mehrere Paste-Aktionen dieser Kodeschnipsel folgen. Zusätzlich zu diesen Aktionen enthält eine Instanz Informationen, die das Verändern dieser Kodeschnipsel betrifft.
Es folgte die Kategorisierung dieser beobachteten Instanzen. Wobei sie sich auf die vefahrenstechnischen Schritte, die zu diesen Kopien führten, konzentrierten. Wie eben auch auf die syntaktischen Einheiten dieser kopierten oder ausgeschnittenen Inhalte. Bei den syntaktischen Einheiten unterschieden sie zwischen Typen, Bezeichner, Blöcken, Methoden und Klassen.

Ferner dokumentierten sie für jede C&P-Instanz die Beziehung zwischen kopiertem oder ausgeschnittenem Kodeschnipsel und Kodestellen im Quellkode. Zusätzlich analysierten sie die evolutionär bedingten Veränderungen dieser Instanzen indem sie diese duplizierten Kodefragmente beobachteten wie sie sich veränderten. Nach all diesen Untersuchungen sprachen sie mit den Programmierern, um sich ihre Interpretationen der C&P-Aktionen der

Programmierer von Programmierern bestätigen zu lassen. Das Ergebnis dieser Untersuchung war, dass sie insgesamt 460 C&P-Instanzen ermittelten. Sie errechneten einen durchschnittlichen Wert von 16 C&P-Instanzen pro der Stunde. Um nun erkennen zu können wie oft C&P-Operationen mit verschieden langem Kode auftauchten, haben sie diese C&P-Instanzen in vier verschiedene Gruppen sortiert und zählten sie.

Die Einteilung bezüglich der Länge des kopierten und eingefügten Kodes wurde folgendermaßen vorgenommen:

1. Gruppe: Typen, Variablen, Methodennamen
2. Gruppe: Kodeblöcke
3. Gruppe: Methoden (also mit Methodenrumpf)
4. Gruppe: Klassen

Circa *74%* der C&P-Instanzen fielen unter die erste Gruppe. Das bedeutet, dass in dieser Kategorie Kode kopiert wurde, der eine Zeile im Quellkode ausmacht. Zu diesen Einzeilern gehören z.B. die Deklaration von Variablen, die Definition von Typenamen oder schlicht die Kopie eines Methodennamens. In diesen Fällen glauben Kim und ihre Kollegen, dass das Kopieren und das Einfügen nur ausgeübt wurde, um sich Zeit und das Tippen zu sparen. Aber ungefähr *25%* der ermittelten C&P-Instanzen beinhalteten das Kopieren und Einfügen eines gesamten Kodeblockes oder einer Methode mit Methodenrumpf also, um die Gruppe zwei und die Gruppe drei der obigen Auflistung.

Es wurde auch das Kopieren von Klassen samt Inhalt registriert, die in die Gruppe vier eingeordnet wurden. Bei den Kategorien zwei bis vier handelt es sich laut Kim um eine Klassifizierung, die die so genannten *Schablonen* beschreiben. Diese Schablonen reflektieren dabei eine bestimmte Designentscheidung des Programmierers wider. Designentscheidung meint hier nicht das Design des Systems im Gesamten, sondern vielmehr das Design bezogen auf eine bestimmte Struktur eines Kodeteils im Programm, die der Programmierer wieder verwenden möchte (*Beispiel siehe Abb. 4*). Diese Schablonen werden im Programmkode durch C&P Aktionen im gesamten Programmkode verbreitet.

Die Schablonen teilen sich wiederum in zwei Gruppen auf. Zu der einen Gruppe gehören die *syntaktischen* Schablonen, da sie hierbei wieder verwendbare syntaktische Elemente ausmachen, die der Programmierer nutzen möchte.

Folgendes Beispiel soll zeigen was eine syntaktische Schablone ausmacht:

```
//statement kopiert und mehrmals an anderen Stellen eingefügt

protectedClasses.add("java.lang.Object");

-----------------------------------------------------------------------

//Benutzen der syntaktischen Schablone für eine
//Methodeninitialisierung (Modifizierungen sind unterstrichen)

protectedClasses.add("java.lang.ref.Reference");
```

Abb.5: Beispiel für eine syntaktische Schablone.

Zu der anderen, also der zweiten Gruppe, gehören die so genannten *semantischen* Schablonen. Sie beschreiben wieder verwendbare Programmierlogik, die der Programmierer nutzen möchte (nochmals sei auf *Abb.4 verwiesen*). Im Rahmen dieser Arbeit sind vor allem diese semantischen Schablonen interessant. Für diese semantischen Schablonen soll der oben erwähnte Warnmechanismus greifen.

Wenn man nun diese 25% Aufkommen der Schablonen mit der Anzahl der C&P-Instanzen multipliziert, also 16 in der Stunde, dann ist das Ergebnis folgendes:

Ein Programmierer erzeugt *vier nicht triviale* C&P-*Abhängigkeiten* pro Stunde zwischen den kopierten und eingefügten Kodefragmenten.

Diese Abhängigkeiten resultieren aus der Tatsache, dass die kopierten Kodeschnipsel die statistisch gesehen unter das obige Ergebnis fallen, jeweils eine Schablone für eine andere Stelle im Programmkode darstellen. Die Abhängigkeit ergibt sich aus dem Kontext des jeweiligen Sachverhalts zwischen diesen Kodeschnipseln.

6.2 CopyPasteChange-Episoden

In diesem Kapitel gehe ich nochmals genauer und tiefer auf die CopyPasteChange-Episoden (CPC-Episoden) und die Theorie ein, die zu dieser Episode zu beachten ist. Zunächst besteht eine Episode allgemein gesehen aus einzelnen Aktivitäten. Diese Aktivitäten, die eine CPC-Episode ausmachen, werden nun im Folgenden beschrieben.

6.2.1 Aktivitäten die zur CopyPasteChange-Episode führen

Folgende Aktionen des Anwenders bilden in der Reihenfolge ihrer Beschreibung eine CPC-Episode:

- *copy*, das *Kopieren* eines Textteils beliebiger Länge innerhalb einer Datei mittels Tastatur oder Maus. Dieser Textteil bezeichnet das *Original*.

- *paste*, das *Einfügen* des zuvor kopierten Textteils an eine beliebige Stelle einer Datei ebenfalls mittels Tatstatur oder Maus. Dieser Textteil bezeichnet die *Kopie*.

- *change*, das *Ändern* mittels Tastatur oder Maus des zuvor in die Datei eingefügten Textteils. Bezeichnet die Modifikation der Kopie

Für copy verwende ich in dieser Arbeit das deutsche Verb *kopieren*. Das englische Wort paste ersetze ich durch das deutsche Verb *einfügen* und für change verwende ich das deutsche Verb *verändern*, *abändern* oder *modifizieren*. Abbildung 6 verdeutlicht nochmals die oben beschriebenen einzelnen Aktionen kopieren, einfügen und modifizieren.

```
//CPCEpisodeRecognizer is in state == START & activity == copy
if((this.state == CopyPasteChangeEpisodeState.START) && (activity.equals("copy"))){
    //set this recognizers-state to COPY since this event was a copy-event
    this.state = CopyPasteChangeEpisodeState.COPY;
    code = ECGParser.getSingleNodeValue("selection", document);
    copiedDate = timestamp;  // synonym for startDate of this episode
```

a) Der blau markierte Text bezeichnet hier die Aktion des Kopierens seitens des Anwenders.
Der kopierte Textteil ist das Original.

```
if((this.state == CopyPasteChangeEpisodeState.START) && (activity.equals("copy"))){
    //set this recognizers-state to COPY since this event was a copy-event
    this.state = CopyPasteChangeEpisodeState.COPY;
```

b) Der zuvor kopierte Textteil wurde nun an eine andere Stelle bzw. in einer anderen Datei eingefügt.
Diese Aktion bezeichnet das Einfügen seitens des Anwenders. Der eingefügte Textteil ist somit also die Kopie.

```
else if((this.state == CopyPasteChangeEpisodeState.START) && (activity.equals("cut"))){
    this.state = CopyPasteChangeEpisodeState.CUT;
    code = ECGParser.getSingleNodeValue("selection", document);
```

c) Der zuvor eingefügte Textteil wurde nun modifiziert und bezeichnet die Aktion des Veränderns.
Der Anwender modifizierte also mit dieser Aktion die Kopie.

Abb.6: Kopieren, einfügen und modifizieren eines Textes.

Bei den eben beschriebenen Aktionen des Programmierers werden die Begriffe *Original* und *Kopie* verwendet. Im Moment ist es noch nicht fehlerhaft diese Begriffe im Kontext von Kopieren und Einfügen von Textteilen zu verwenden. Um die allgemeine Theorie zu Kopieren und Einfügen zu erläutern, können wir mit diesen Begriffen zunächst weiterarbeiten. Im weiteren Verlauf dieser Arbeit, werde ich aber erklären warum man sich im Rahmen dieser Arbeit, also der Klonverfolgung, von der Begrifflichkeit Original und Kopie trennen muss. Für die Klonverfolgung ist eine präzisere Sicht und detailliertere Beschreibung notwendig.

Es wurde oben in Abbildung 6 das Original bzw. die Kopie erwähnt. Das ist der Textteil, der während einer Kopieraktion im Clipboard abgelegt und gespeichert wird bzw. an anderer Stelle eingefügt wird. Mit der Kopie bezeichnet man folglich den eingefügten Textteil, der im Clipboard gespeichert ist und somit textlich dem zuvor kopierten Textteil gleicht, also somit dem Original. Bei der allgemeinen Sicht des Kopieren und Einfügen, verwendet man die Fachbegriffe Original und Kopie für die Beziehung zwischen diesen beiden Textteilen.
Die einzelnen Aktionen des Kopierens, des Einfügens und des Abänderns müssen zeitlich sequentiell hintereinander ausgeführt werden, um inhaltlich einer gemeinsamen CPC-Episode anzugehören. Wobei das Kopieren und das Einfügen nicht nur sequentiell ablaufen müssen, vielmehr darf nach dem Kopieren keine weitere Kopieraktion folgen um inhaltlich der gleichen CPC-Episode anzugehören. Nach einem erneuten Kopieren würde der zuvor kopierte Textteil verloren gehen. Es würde tatsächlich dann das zuletzt kopierte eingefügt werden. Das wiederum könnte dann ein verschiedener Text verglichen zu dem zuvor kopiertem aber nicht mehr im Clipboard vorhandenen Text sein.

Folgende Abbildung soll das eben geschilderte wiedergeben:

```
        blockchanges++;
        linenumber = bc.getBlockStart();
        for (LineChange lc: bc.getLineChanges()) {
            if (lc.isChange()) {inside = event.changeLine(linenumber,lc.getContents());}
            if (lc.isDeletion()) {inside = event.deleteLine(linenumber);}
            if (lc.isInsertion()) {inside = event.insertLine(linenumber,lc.getContents());}
            if (inside) event_inside_change = true;
            linenumber++;
        }
    }
    /*****************************************************************
```

a) Kopierter Textteil ist damit im Clipboard gespeichert.

```
        boolean inside = false;
        for (BlockChange bc: blockChanges) {
            blockchanges++;
            linenumber = bc.getBlockStart();
            for (LineChange lc: bc.getLineChanges()) {
                if (lc.isChange()) {inside = event.changeLine(linenumber,lc.getContents());}
                if (lc.isDeletion()) {inside = event.deleteLine(linenumber);}
                if (lc.isInsertion()) {inside = event.insertLine(linenumber,lc.getContents());}
                if (inside) event_inside_change = true;
                linenumber++;
            }
        }
        /*****************************************************************
```

b) Erneutes Kopieren eines neuen erweiterten Textteils, der nun aktuell im Clipboard gespeichert ist.

```
 for (BlockChange bc: blockChanges) {
     blockchanges++;
     linenumber = bc.getBlockStart();
     for (LineChange lc: bc.getLineChanges()) {
         if (lc.isChange()) {inside = event.changeLine(linenumber,lc.getContents());}
         if (lc.isDeletion()) {inside = event.deleteLine(linenumber);}
         if (lc.isInsertion()) {inside = event.insertLine(linenumber,lc.getContents());}
         if (inside) event_inside_change = true;
         linenumber++;
     }
 }
```

c) Nach dem Einfügen steht an der neuen Stelle der zuletzt kopierte und im Clipboard gespeicherte Inhalt.

Abb.7: Beispiel des Inhalts des Clipboards bei einem zweimaligem Kopieren und anschließendem Einfügen.

Das bedeutet, dass der unmittelbar nach der Kopieraktion eingefügte Text wirklich dem zuletzt kopierten Text entspricht.
Schlussendlich können die Modifizierungen der zuvor kopierten und danach eingefügten Textteile dann zeitlich entweder unmittelbar nach dem Einfügen oder auch zeitlich gesehen später stattfinden. Bei jeder Modifizierungsaktion der kopierten und eingefügten Textteile, die inhaltlich zu einer CPC-Episode gehören, ist somit eine CPC-Episode abgeschlossen.

6.2.2 Halbformale Beschreibung der CPC-Episode

Halbformale Beschreibung dieser allgemeinen CopyPasteChange-Episode ein:

1) Zu einer CPC-Episode gehören folgende Aktionen:

   - **kopieren**
   - **einfügen**
   - **abändern** (**modifizieren**, **verändern**)

2) Damit die Aktionen aus 1 eine gemeinsame CPC-Episode bilden können, müssen sich die Aktionen auf einen gemeinsamen Textinhalt X beziehen.

3) Die Aktionen müssen in einer vorbestimmten Reihenfolge ausgeführt werden:

   Kopieren von Textteil X => Einfügen des Textteils X => Modifizieren des Textteils X

   = CopyPasteChange-Episode

Das ist zunächst eine einfache halbformale Beschreibung der CPC-Episode. Diese Beschreibung wird für ein korrektes Arbeiten des Kodekopieverfolgungsprogramms aber nicht ausreichen. Sie muss erweitert und noch expliziter in die Tiefe gehend definiert werden. Das werde ich im nächsten Kapitel (6.3 Formalisierung von CopyPasteChange-Episoden) meiner Ausarbeitung ausführlich beschreiben.

Die Aktionen, die oben in der halbformalen Beschreibung aufgeführt sind, werde ich ab jetzt *Aktivitäten*[6] nennen, da sie tatsächliche Aktivitäten des Programmierers sind und in Verbindung mit dem Mikroprozess der Softwareentwicklung ebenfalls als Aktivitäten bezeichnet werden.

Wenn man sich also nochmals die Aktivitäten betrachtet und dazu die Theorie aus dem vorherigen Kapitel gegenüberstellt fällt auf, das ein Punkt bezüglich der Aktivitäten bisher keine Beachtung gefunden hat. In Kapitel 6.1 wurde herausgearbeitet das zwischen den Kodefragmenten, die kopiert und eingefügt werden, eine Beziehung besteht. Zum einen da Kode dupliziert wird und zum anderen, da kopierte Kodefragmente Schablonen für einen anderen Programmteil darstellen, wenn sie eingefügt werden. Diese eingefügten Schablonen bestehen so lange aus dem gleichen Kode wie ihre Originale, bis sie abgeändert werden. Das ist zumindest bei den kopierten Kodefragmenten sicher so, die Schablonen für den Programmierer an anderer Stelle im Programmkode darstellen und nach dem Einfügen nach Bedarf angepasst werden. Man kann sich nun folgende Frage stellen. Ist es ebenso ein zu berücksichtigender Fall, wenn der Programmierer ein Kodefragment aus dem Programmkode ausschneidet und diesen Kodeschnipsel an einer anderen Stelle im Programmkode einfügt?

[6] Eine Aktivität ist in Verbindung mit dem Mikroprozess ein Ereignis, z.B. beim Programmieren sind das Ausführen eines Programms oder das Öffnen eines Editors etc. Aktivitäten.

Die Antwort liegt auf der Hand. Wenn ein Kodefragment ausgeschnitten wird und an eine andere Stelle im Programmkode eingefügt wird, existiert keine Beziehung zu einem anderen Kodefragment im Programmkode. Somit ist dieser Fall nicht zu berücksichtigen.
Anders ist es der Fall, wenn der zuvor ausgeschnittene Kodeschnipsel an mindestens zwei verschiedenen Stellen im Programmkode eingefügt wird. Dann ist diese Konstellation sehr wohl zu berücksichtigen, da nun tatsächlich eine Beziehung zwischen mindestens zwei Kodefragmenten besteht.

Bei der gesamten Problematik der Klone, also der zuvor kopierten und danach eingefügten Kodeschnipseln, darf man nicht vergessen, dass es sich dabei um Abhängigkeiten zwischen diesen Kodeschnipseln handelt. Wenn das Original ausgeschnitten wird und an eine andere Stelle im Programmkode eingefügt wird, existiert keine Kopie und folglich eben auch keine Beziehung zu einem anderen Kodefragment. Der Kodeschnipsel wurde gerade ausgeschnitten, also existiert er im Programmkode an dieser Stelle nicht mehr. Wird der ausgeschnittene Kode nun aber mindestens zweimal an verschiedenen Stellen im Programmkode eingefügt, hat der Programmierer zwei in Beziehung zueinander stehende Kodeschnipsel erzeugt. Der Programmierer hat eine Schablone sozusagen entfernt aber an zwei verschiedenen Stellen im Programmkode wieder aus bestimmten Gründen, die weiter oben schon beschrieben worden sind, eingefügt. Im Ergebnis hat der Programmierer hier wie beim Kopieren und Einfügen Kode dupliziert.

Bei dem Kopieren eines Kodefragments, das dann an einer anderen Stelle im Programmkode eingefügt wird, gilt das obige Problem des Ausschneidens und Einfügens nicht. Beim Kopieren eines Kodeschnipsels ist dieser Kodeschnipsel ja weiterhin im Programmkode vorhanden. Das ist das im Programmkode vorhandene Original. Nun wird nach der Kopieraktivität dieser Kodeschnipsel an einer anderen Stelle im Programmkode eingefügt. Bei diesem eingefügten Kodeschnipsel handelt es sich damit um die Kopie. Somit existiert im Programmkode zweimal derselbe Kode. Es wurde Kode dupliziert und diese zwei Kodestellen stehen nun in Beziehung zueinander.

6.2.3 Mehrmalige Einfügungen von kopierten oder ausgeschnittenen Kode

Zu beachten und zu verarbeiten sind zwei Fälle die auftreten können:

Fall 1: Kopieren und Einfügen

Wenn der Programmierer ein Kodefragment X kopiert und es dann mindestens einmal einfügt, daraus somit diese oben erwähnte Abhängigkeitsbeziehung zwischen diesen beiden Kodestellen entsteht, dann werden *n* Einfügungen dieses Kodefragments X diese Beziehung nicht aufheben. Vielmehr stehen nun sogar alle *n+1* Kodestellen, also das einmal kopierte Kodefragment X und die n-mal eingefügten Kodefragmente X in Beziehung zueinander. Zwischen den n Einfügungen darf kein Kopieren oder Ausschneiden eines Kodefragments erfolgen.

Fall 2: Ausschneiden und Einfügen

Bei Betrachtungen von ausgeschnittenen und eingefügten Kodefragmenten ist folgendes festzuhalten. Wenn der Programmierer ein Kodefragment X ausschneidet und *einmal* einfügt, existiert wie bereits vorher beschrieben kein weiteres Kodefragment von X im Programmkode und somit keine Beziehung zu einem weiteren gleichen Kodefragment X. Wenn aber der Programmierer das ausgeschnittene Kodefragment X mindestens *zweimal* an anderen Stellen im Programmkode einfügt, stehen diese zwei Kodefragmente mit X als Inhalt zueinander in Beziehung. Bei *n-maligen* Einfügungen von X gilt dasselbe wie im zuvor beschriebenen Fall 1 und zwar stehen dann alle *n-mal* eingefügten Kodefragmente mit X als Textinhalt zueinander in Beziehung. Auch hier gilt, dass zwischen den n Einfügungen kein Kopieren oder Ausschneiden eines anderen Kodefragments erfolgen darf.

6.2.4 Kodefragmente in verschiedenen Dateien

Kodefragmente, die durch Copy-Aktivitäten und Paste-Aktivitäten aus verschiedenen Dateien entstanden sind, müssen nun ebenfalls beobachtet werden, damit konsistente Abänderungen des Kodes möglich werden.

Datei: a.java

```
boolean inside = false;
for (BlockChange bc: blockChanges) {
    blockchanges++;
    linenumber = bc.getBlockStart();
    for (LineChange lc: bc.getLineChanges()) {
        if (lc.isChange()) {inside = event.changeLine(linenumber,lc.getContents());}
        if (lc.isDeletion()) {inside = event.deleteLine(linenumber);}
        if (lc.isInsertion()) {inside = event.insertLine(linenumber,lc.getContents());}
        if (inside) event_inside_change = true;
        linenumber++;
    }
}
/*****************************************************************
```

Die Schleife wurde in Datei a.java kopiert und soll in Datei m.java eingefügt weden, da die Logik dieser Schleife zu der Lösung des Problems in m.java beiträgt.

Datei: m.java

```
for (BlockChange bc: blockChanges) {
    blockchanges++;
    linenumber = bc.getBlockStart();
    for (LineChange lc: bc.getLineChanges()) {
        if (lc.isChange()) {inside = event.changeLine(linenumber,lc.getContents());}
        if (lc.isDeletion()) {inside = event.deleteLine(linenumber);}
        if (lc.isInsertion()) {inside = event.insertLine(linenumber,lc.getContents());}
        if (inside) event_inside_change = true;
        linenumber++;
    }
}
```

Die Schleife wurde in der Datei m.java eingefügt. Es müssen nun beide Kodefragmente dateiübergreifend beobachtet werden, um eventuelle Modifizierungen konsistent abzuändern.

Abb.8: Beispiel für die Notwendigkeit des dateiübergreifenden Erfassen und Bearbeiten von kopierten und eingefügten Kodefragmenten.

6.2.5 Warum ist die bisherige Sichtweise von Original und Kopie aufzugeben?

Am Anfang dieses Kapitels habe ich darauf hingewiesen, dass die allgemeine Sichtweise von Kopieren und Einfügen und das damit einhergehende Verständnis von Original und Kopie im Verlauf dieser Arbeit aufgegeben werden muss. Ich werde nun darauf eingehen und erklären warum dieses so ist.

Das allgemeine Verständnis von Original und Kopie genügte den bisherigen Ansprüchen das Phänomen Kopieren und Einfügen zu verstehen und erklären zu können. Nun nähere wich mich aber der Formalisierung dieses Phänomens, die im folgenden Kapitel 6.3 Formalisierung der CopyPasteChange-Episode folgen soll. Jetzt ist ein tieferes Verständnis nötig, um dieses Problem in dem späteren Kodekopieverfolgungsprogramm korrekt abbilden zu können.
Geht man weiter von der allgemeinen Sichtweise aus, dann würde implizit annehmen, dass immer nur die Kopie angepasst werden müsste, falls das Original abgeändert wird. Das ist so, da man ja mit der Verwendung des Begriffes Original deutlich macht, dass dieses

Kodefragment das Original und somit die Schablone für die Kopie ist. Es ist sozusagen eine hierarchische Beziehung, wobei das Original immer über der Kopie steht, denn ändert sich das Original dann könnte auch die Kopie zu ändern sein. Es ist aber von größter Notwendigkeit diese Beziehung nicht nur in dieser einen Richtung zu betrachten, sondern vielmehr auch zu hinterfragen, ob in entgegen gesetzter Richtung nicht auch eine Beziehung zwischen Kopie und Original existiert. Diese Richtung ist existent und das möchte ich an einem Beispiel auf der folgenden Abbildung verdeutlichen.

```
      .
      .                    KOPIE
      .
        // Hervorgegangene und abgeändert Kopie aus unten kopiertem Original
       // und nun weitere Modifizierung (unterstrichen).

1     for(Iterator iter = messages.iterator();iter.hasNext();){
2           Message current= (Message)iter.next();
3           MessageLocation messLoc= current.getLastLocation();
4           IFile markFile =
5                 WorkspaceUtils.getFile(messLoc.getClasName());
6
7                  // global definierter Schreiber
8                  prWriter.println(markFile.toString());
      .
      .
      .
```

```
      .
      .                    ORIGINAL
      .
      // Original, müsste vielleicht auch mit dieser Modifizierung
      // bedacht werden, damit die Konsistenz der Programmlogik
      // erhalten bleibt!

1     for(Iterator iter = messages.iterator();iter.hasNext();){
2           Message current = (Message)iter.next();
3           IFile iFile =
4     WorkspaceUtils.getFile(current.getFirstLocation().getClassName())
5
6                  // global definierter Schreiber
7                  prWriter.println(iFile.toString());
      .
      .
      .
```

Abb.9: Beispiel zweier for-Kontrollstrukturen, die die Sichtweise von Original und Kopie verschwimmen lässt (for-Schleifen aus Abb. 4).

In Abbildung 9 sieht man, das nach dem Einfügen und erstmaligem modifizieren der Kopie, also dem Anpassen des Kodes zu dem Programmkontext an dieser Stelle, zu einem späteren Zeitpunkt der Implementierung nochmals eine Modifizierung der inneren Logik vorgenommen wurde. Der Programmierer fügte in Zeile 3 der Kopie eine neue Variable `messLoc` (`MessageLocation messLoc= current.getLastLocation()` in die innere Logik der Kopie ein. Und bei der Variablendeklaration in Zeile 3 der Kopie ruft er im Methodenaufruf die neue Variable `messLoc` auf. Das kann nun eine neue Anforderung an das Programm sein, die der Programmierer mit dieser zusätzlichen Anweisung umsetzen

möchte. Damit die Konsistenz der Software erhalten bleibt könnte es aber nun der Fall sein, dass diese neue Variable laut neuen Anforderungen an das Programm, ebenfalls in dem Kodefragment das das Original ausmacht eingefügt werden müsste. Also müsste der Programmierer in der inneren Logik des Originals ebenfalls diese neue Variable erzeugen und die Variable `iFile` wiederum durch den Methodenaufruf mit dieser neu erzeugten Variable erzeugen.
Hier verschwimmt die Sichtweise von Original in Kopie, denn so wie zunächst beschrieben wurde, dass eine Kopie angepasst werden müsste, wenn sich das Original verändert, müsste eventuell genauso das Original angepasst werden, falls die Kopien modifiziert wurden. Bei der bisherigen allgemeinen Sichtweise würde der Programmierer aber niemals gewarnt werden, wenn sich die Kopie verändert. Damit ist klar, dass der Programmierer auch gewarnt werden muss, falls sich die Kopie eines Originals verändert. Er muss zumindest darauf zumindest hingewiesen werden, dass er gerade eine Kopie modifiziert hat und gegebenenfalls auch das Original abändern muss. Das bedeutet, dass man nach mehrmaligem gegenseitigem Anpassen nicht mehr feststellen kann, was das Original und was die Kopie ausmacht, um sie tatsächlich voneinander unterscheiden zu können. Aus diesem Grund werde ich den Begriff des Klons definieren.

Definition eines Klons:

Ein Klon ist ein Kodefragment,

- das durch eine Copy-Aktivität und mindestens einer Paste-Aktivität des Programmierers *dupliziert* wurde.

- das entweder durch eine Cut-Aktivität und mindestens zwei aufeinander folgenden Paste-Aktivitäten dupliziert wurde.

Der Begriff des Klons weist somit auf die Ähnlichkeit zwischen Kodefragmenten hin. Bei der neuen exakteren Betrachtung dieser Kodefragmente spricht man somit nicht mehr von Original und Kopie, sondern von dem *Ursprung* und den *Klonen*.

Warum verwende ich die Terminologie des Klons?

Einem Ursprung geht sozusagen eine Geburt voraus. Das ist der Fall, wenn der Programmierer das Kodefragment kopiert und es im Clipboard gespeichert ist. Ein Ursprung hat irgendeinen Anfangsstatus, der sich im Verlauf der Programmevolution sicher verändern kann. Es folgt nun das Einfügen dieses im Clipboard befindlichen Kodefragments an eine andere Stelle im Programmkode. Nun ist Kode dupliziert worden was genau den Klon ausmacht. Denn ein Klon entsteht dadurch, dass man die Gene eines Lebewesens nutzt um den Klon zu erzeugen. Das ist der Fall bei dem Einfügen des im Clipboard befindlichen Kodefragments. Wenn nun der Programmierer ein Kodefragment kopiert und einfügt, hat er eine *Klonfamilie* erzeugt. Diese Klonfamilie beschreibt eine Menge, die es nun im Klonverfolgungsprogramm zu *beobachten* gilt. Bei jedem erneuten kopierten Klon einer Klonfamilie und darauf folgendem Einfügen dieses Klons in den Programmkode, muss dieser neu erzeugte Klon ebenfalls in diese Klonfamilie aufgenommen werden. Der neu aufgenommene Klon wird nun ebenfalls *beobachtet*, eben genau wie seine übrigen Klonfamilienmitglieder. Das verändern der Klone wird verfolgt und der Programmierer kann dahingehend informiert werden, dass er eine Änderung eines in der Klonfamilie befindlichen Klons vorgenommen hat. Daraus folgt, dass der Programmierer wenn erforderlich alle übrigen Familienmitglieder konsistent abändern kann.

Analog ist der eben beschriebene Sachverhalt auch auf ausgeschnittene und eingefügte Kodefragmente anzuwenden, wobei das Ausschneiden eines Kodefragments ein Spezialfall darstellt. Dieser Spezialfall wird in Kapitel 6.3 erklärt, wo ich auf die Formalisierung eingehe.

6.2.6 Wann werden Klone in die Klonfamilie aufgenommen?

Diese Beschreibung ist analog zu 6.2.3 Fall 1 und Fall 2.Es gilt zwei Fälle zu betrachten um diese Frage zu beantworten.

1. Fall: Kopieren und Einfügen

Erzeugen der Klonfamilie:

Der Programmierer kopiert ein Kodefragment. Das ist der Ursprung. Nun fügt der Programmierer den Ursprung anderenorts ein. Das bedeutet, es wurde eine Beziehung zwischen dem Ursprung und dem Eingefügten erzeugt. Das Eingefügte Kodefragment entspricht dem Klon. Es werden Ursprung und Klon in die Klonfamilie aufgenommen. Damit ist klar, dass nach dem Kopieren mindestens ein einmaliges Einfügen des Kodeteils zu erfolgen hat.

Weiteres Einfügen von Klonen in die Klonfamilie:

Wird ein Klonfamilienmitglied kopiert und in den Programmkode eingefügt, entsteht ein neuer Klon und er wird in die Klonfamilie aufgenommen.

2 .Fall: Ausschneiden und Einfügen

Erzeugen der Klonfamilie:

In dem Fall, dass ein Kodefragment ausgeschnitten und dann eingefügt wird, besteht noch keine Beobachtungspflicht. Das resultiert daraus, dass wenn ein Kodefragment ausgeschnitten wird, es nicht mehr im Programmkode existent ist und somit nach dem Einfügen eben nur dieses eine ausgeschnittene Kodefragment im Programmkode vorhanden ist. Es besteht damit keine Beziehung zu dem ausgeschnittenen Kodeschnipsel im Programmkode. Folglich werden erst nach einem zweimaligen Einfügen eines zuvor ausgeschnittenen Kodefragments diese Kodeschnipsel in eine neue Klonfamilie aufgenommen.

Weiteres Einfügen von Klonen in die Klonfamilie:

Wenn ein Klonfamilienmitglied durch ein Ausschneiden aus einer Klonfamilie entfernt wird und anschließend wieder in den Programmkode eingefügt wird, muss dieser Klon in die Klonfamilie aufgenommen werden.

6.2.7 Wie lange müssen Klone beobachtet werden?

Die Objekte die in einer Klonfamilie aufgenommen wurden, werden nun von dem Klonverfolgungsprogramm beobachtet. Änderungen an einem Klon dieser Klonfamilie müssen dokumentiert werden. Eine Warnung wird generiert, um den Programmierer auf die konsistenten Änderungen der übrigen Klonfamilienmitglieder hinzuweisen. Das Beobachten des Klonvefolgungsprogramms wird so lange durchgeführt, wie folgende Punkte gelten:

1) Klone sind noch im Programmkode existent sind, d.h. das der Klon nicht insgesamt textlich gelöscht wurde.

2) Klone sich nicht um ein Maß so voneinander textlich entfernt haben, dass sie sich nicht mehr ähnlich sind.

3) Die Anzahl der Klone in der Klonfamilie muss größer als eins sein, da sonst keine Beziehung zu einem anderen Klon in der Familie existiert.

Zu Punkt 1 eins ist folgendes zu sagen. Ist ein Klon nicht mehr im Programmkode vorhanden, dann wird er auch aus der Klonfamilie gelöscht. Er ist nicht mehr existent, also muss er auch nicht mehr verfolgt werden.

Wenn nach dem Löschen in dieser Klonfamilie nur noch ein zu beobachtender Klon vorhanden ist, tritt Punkt 3 in Kraft und die gesamte Klonfamilie wird gelöscht.

Punkt zwei macht deutlich, dass wenn sich Klone textlich zu sehr voneinander entfernen, sie nicht mehr beobachtet werden. Er verbleibt in der Klonfamilie aber ist nicht mehr ähnlich genug zu den anderen Klonfamilienmitgliedern in seiner Klonfamilie. Der Warnmechanismus greift aber nur bei unter Beobachtung stehenden Klonen. Wenn ein Klon einen zu großen Unterschied zu seinen übrigen Klonfamilienmitgliedern aufweist, steht dieser Klon nicht mehr unter Beobachtung und wird bei eventuellen Änderungen seiner übrigen unter Beobachtung stehenden Klonfamilienmitgliedern bei einer Warnung nicht berücksichtigt. Das heißt, dass wenn dieser zuvor aus der Beobachtung genommene Klon nach einer erneuten Veränderung sich wieder den übrigen Klonen in der Klonfamilie textlich annähert, kann er wieder in die Verfolgung mit aufgenommen werden. Somit wird er auch wieder bei eventuellen Warnmeldungen mitberücksichtigt.
Das wiederum bedeutet, dass ein Klon wirklich erst gänzlich aus der Verfolgung herausgenommen wird, wenn er textlich aus dem Programmkode gelöscht wird, wenn also Punkt 1 in Kraft tritt. Die Anzahl der Klone in einer Klonfamilie muss somit mindestens zwei betragen, ansonsten wird diese Klonfamilie nicht mehr berücksichtigt.

Zu Punkt zwei ist noch hinzuzufügen, dass die textliche Verschiedenheit der Klone mittels eines String-Vergleichsalgorithmus verglichen wird. Nach einigem Recherchieren bin ich auf die Editierdistanz nach Levenshtein[7] gestoßen. Die Levenshtein-Distanz wird im Klonverfolgungsprogramm verwendet, um die Unterschiedlichkeit zweier Strings zu berechnen.

Definition Levenshtein-Distanz:

Die Editierdistanz ermittelt aus zwei gegebenen Strings s und t die minimale Anzahl der folgenden Editierschritte:

- Löschen eines Zeichens

- Verändern eines Zeichens

- Einfügen eines Zeichens,

die benötigt werden, um s in t zu verwandeln.

Daraus folgt:

Je größer die Levenshtein-Distanz d, desto unterschiedlicher sind die zwei Strings s und t.

Beispiel:

1) Wenn s = “Hello World“ und t = “Hallo Welt“ .

2) Dann ermittelt sich die Levenshtein-Distanz aus s und t zu d =4.

Auf den Levenshtein-Algorithmus gehe ich konkreter in Kapitel 7 ein. Zunächst reicht diese kurze Beschreibung aus, um grundsätzlich zu verstehen, wie die Unterschiedlichkeit von Klonen in einer Klonfamilie ermittelt wird.

6.3 Formalisierung von CopyPasteChange-Episoden

6.3.1 Erläuterungen zu der Formalisierung

i, **j**, **k**, **l**, **m**, **n**, **z1**, **z2** := ganze Zahlen für die Zeilenangaben der Klone**,**
wobei i <= j, k <= l, m <= n, z1 <= z2

A.java := Eine Quelltextdatei aus Eclipse

B.java := Eine Quelltextdatei aus Eclipse

Cut := Ausschneide-Aktivität

Copy := Kopier-Aktivität

Paste := Einfüge-Aktivität

Copy (A/x,z1,z2) := Kopieren von Kodeschnipsel x aus Datei A.java von Zeile z1 bis Zeile z2

Paste (A/x,z1,z2) := Einfügen von Kodeschnipsel x in Datei A von Zeile z1 bis Zeile z2

Cut (B/y,z1,z2) := Ausschneiden von Kodeschnipsel y aus Datei B.java von Zeile z1 bis Zeile z2

Change (X/x,z1,z2) := Das textliche Verändern des Programmkodes einer Datei X.java mit Zeilenanfang z1, Zeilenende z2 und dem veränderten Kode x.

Clone(X/x,i,j) := Ein Klon nach Zeilenanpassungen und Textabänderungen.

Erzeugung von Klonfamilien:

Klonfamilie A = {(Copy (Y/x,z1,z2)), (Paste (Y/x,i,j)), (Paste (Y/x,k,l))*}
:= Klonfamilie X, eine Menge der in Relation stehenden Copy- und Pasteaktivitäten.

Klonfamilie B = (Cut (Y/x,z1,z2) {(Paste (Y/x,i,j)), (Paste (Y/x,k,l))+}
:= Klonfamilie X, eine Menge der in Relation stehenden Cut- und Pasteaktivitäten.

Man beachte: Der Kode aus der Cut-Aktivität wird nicht in die Klonfamilie als zu beobachtender Kode eingefügt, da er im Programmkode, aus dem er herausgeschnitten wurde nicht mehr existent ist. Es besteht keine Beziehung zu einem anderen Kode im Programmkode. Somit ist dieser Kode aus der Cut-Aktivität nicht als Element dieser Klonfamilie einzufügen.

Weiteres Einfügen in Klonfamilien im Falle einer Copy-Aktivität:

Die unten aufgeführte Klonfamilie X soll eine Klonfamilie repräsentieren, bei deren Klonelementen bereits Change-Aktivitäten durchgeführt worden sind. Das soll heißen das Zeilenanpassungen und Textveränderungen vorgenommen wurden.

Klonfamilie X = {(Clone(Y/x,z1,z2)),…, (Clone(Z/x,i,j))}

Mit den Aktivitäten **(Copy (Y/x,z1,z2)** und **(Paste (Y/x,z1,z2)** wird ein weiterer Klon in die Klonfamilie X aufgenommen.

Klonfamilie X = {(Clone(Y/x,z1,z2)),…, (Clone(Z/x,z1,z2)), (Paste (Y/x,z1,z2)}

Man beachte: Der Klon, der mittels Copy-Aktivität kopiert wird, wird nicht in die Klonfamilie eingefügt. Er ist bereits in der Klonfamilie als Element enthalten.

Weiteres Einfügen in Klonfamilien im Falle einer Cut-Aktivität:

Klonfamilie X = {(Clone(Y/x,z1,z2)),…, (Clone(Z/x,i,k))}

Mit den Aktivitäten **(Cut (Z/x,i,k)** und **(Paste (Z/x,m,n)** wird der Klon in die Klonfamilie X wieder eingefügt.

Klonfamilie X = {(Clone(Y/x,z1,z2)),…, (Clone(Z/x,m,n))}

Bem*erkung*:

- A.java und B.java kann auch durch eine beliebige Datei X.java ersetzt werden.

Es gibt drei Problembereiche die berücksichtigt werden müssen, wenn man sich das Verfolgen von Kodekopien über die Zeit hinweg in einem Editor zur Aufgabe macht. Der erste Problembereich ist mit der folgenden Frage verbunden. Welche C&P-Aktivitäten führen dazu, dass Klone überhaupt verfolgt werden? Diese Frage wird im ersten Punkt geklärt werden. Der zweite Problembereich umfasst das Problem der Stellenidentität und das wird im zweiten Punkt gelöst. Der dritte und letzte Problembereich umfasst die textliche Ähnlichkeit von Klonen gegenüber den anderen Klonen in ihrer Klonfamilie und die damit verbundene Frage: Wann wird ein beobachteter Klon aus der Beobachtung seiner Klonfamilie entlassen? Diese Frage wird in Punkt 3 geklärt.

1. Punkt:

6.3.2 Welche C&P-Aktivitäten führen zur Verfolgung von Klonen?

Zunächst muss bestimmt werden, welche Copy- und Paste-Aktivitäten dazu führen, dass kopierte oder ausgeschnittene und schließlich eingefügte Textteile, also die Klone, Berücksichtigung bei der Verfolgung finden.

Für alle aufgeführten Aktivitäten gilt, dass sie keiner Einschränkung bei der Verwendung von gleichnamigen Dateien oder unterschiedlichen Dateien unterliegen.
Für Copy(B/y,856,1200) Paste(D/y,410,754) kann man auch Copy(V/y,856,1200) Paste (V/y,410,754) angeben, falls genau diese Dateien gerade an dieser Aktion beteiligt sind.

a) **Cut (A/x,i,j) Paste (A/x,k,l) Paste (A/x,m,n)**

Ursprung ist hier der Kodeschnipsel aus Paste (A/x,i,j) und der Klon aus der Aktivität Paste (A/x,k,l). Die daraus resultierende zu beobachtende Klonfamilie {(Paste (A/x,k,l)),(Paste (A/x,m,n))}.

b) **Cut (A/x,i,j) Paste (B/x,k,l) Paste (C/x,m,n)**

Siehe Erklärung a) mit dem Unterschied das hier unterschiedliche Dateien verwendet werden. Aus den Aktivitäten resultierende Klonfamilie {{(Paste (B/x,k,l)), (Paste (C/x,m,n))}.

c) **Cut (A/x,i,j) Paste1 (B/x,k,l)…PasteN (X/x,m,n), mit N > 1 und i,k,m<=j,l,n**

Ursprung ist aus Paste1 (B/x,k,l) und die Klone sind aus den restlichen N -1 Paste-Aktivitäten bis PasteN (X/x,m,n). Die aus den Copy- und Pasteaktivitäten resultierende Klonfamilie {(Paste1 (B/x,k,l)),…, (PasteN (X/x,m,nl))}.

d) **Copy (B/y,i,j) Paste (B/y,k,l)**

Ursprung aus Copy B/y(i,j) und Klon aus Paste B/y(k,l). Die zu beobachtende Klonfamilie {(Copy B/y(i,j)),(Paste B/y(k,l))}.

e) **Copy (B/y,i,j) Paste (D/y,k,l)**

Siehe Erklärung d) mit dem Unterschied das hier unterschiedliche Dateien verwendet werden. Die aus den Copy- und Pasteaktivitäten resultierende Klonfamilie {(Copy (B/y,i,j)),(Paste (D/y,k,l))}.

f) **Copy (B/x,i,j) Paste1 (A/x,k,l)…PasteN (Y/x,m,n), mit N > 1 und l,k,m<=j,l,n**

Ursprung aus der Copy (B/x,i,j) und die Klone aus Paste1 (A/x,k,l) bis PasteN (Y/x,m,n). Die resultierende Klonfamilie {(Copy (B/x,i,j)), (Paste1 (A/x,k,l)),…,(PasteN (Y/x,m,n))} die es zu beobachten gilt.

g) **Klonfamilie A = {(Clone(Y/x,i,j)),…, (Clone(Z/x,k,l))}**

(Copy(Y/x,i,j) Paste(Y/x,m,n)

=> Klonfamilie A = {(Clone(Y/x,i,j)),…, (Clone(Z/x,k,l)), Paste(Y/x,m,n)}

Ein Klon einer Klonfamilie wird mittels Copy-Aktivität und Paste-Aktivität in die Klonfamilie aufgenommen.

h) **Klonfamilie B = {(Clone(Y/x,i,j)),…, (Clone(Z/x,k,l))}**

(Cut(Z/x,k,l) Paste(Z/x,m,n)

=> Klonfamilie B = {(Clone(Y/x,i,j)),…,Paste(Z/x,m,n)}

Ein Klon einer Klonfamilie wird mittels Cut-Aktivität und Paste-Aktivität in die Klonfamilie wieder aufgenommen.

Die resultierenden Klonfamilien und die in Ihnen enthaltenen Familienmitglieder werden nun durch das Klonverfolgungsprogramm nun verfolgt.

2. Punkt:

6.3.3 Problem der Stellenidentität

Die Stellenidentität ist der nächste Punkt, der berücksichtigt werden muss. Ein Quelltext beliebiger Länge, also ein aus einigen Zeilen bestehender Kodeschnipsel aus einer Datei, wird sicherlich im Laufe der Zeit seine Position im Programmkode verändern. Daraus folgt, dass jede zeilenweise Verschiebung von Programmkode zu berücksichtigen ist. Also müssen auch Änderungen von Programmkode, der nicht zu beobachten ist, erkannt werden. Das ist auch genau so wünschenswert, da selbst zeilenweise Verschiebungen von nicht beobachtetem Kode Auswirkungen auf die Zeilennummern der beobachteten Klone haben können.
Es gibt dabei im Wesentlichen zwei Fälle die beobachtet werden müssen. Es können Abänderungen des Programmkodes zeilenweise vor den beobachteten Klonen der Klonfamilien vollzogen werden. Es können Abänderungen des Programmkodes zeilenweise hinter den beobachteten Klonen der Klonfamilien erfolgen. Die Abänderungen von Programmkode der in die zu beobachteten Zeilen eines Klons fällt und was dabei zu beachten ist wird in Kapitel 6.3.3 beschrieben.

i, **j**, **k**, **l**, **m**, **p, n** **:=** Ganze Zahlen für den Zeilenbereich eines Klons und einer Änderung des Programmkodes,
wobei n die Zeilenanzahl markiert und es gilt:
(m < i,,j und k,l) und (p > i,,j und k,l)

Klonfamilie A = {(Clone(Y/x,i,j)),…,(Clone(Z/x,k,l))}

Löschen von n Zeilen aus Datei Y ab Zeile m:

Es gilt: (m < i,,j und k,l) und (p > i,,j und k,l)

- m-n Zeilen wird der darunter liegende Programmkode nach oben gezogen.
  => **{(Clone(Y/x,i-n,j-n)}**.

Löschen von n Zeilen aus Datei Y ab Zeile p:

- falls p-n > i,,j gilt, müssen keine Zeilen angepasst werden.
- falls p-n <= ,j und p-n > i gilt, müssen Zeilen angepasst werden.
- falls p-n <= i und p-n < =j gilt, werden die Klone gelöscht.

Einfügen von n Zeilen in Datei Z nach Zeile m:

- m+n Zeilen wird der Programmkode erweitert und Zeilen müssen angepasst werden.
  => **{(Clone(Z/x,k+n,l+n)**

Einfügen von n Zeilen in Datei Z nach Zeile p:

- da p > k,l ist, wird dieser Fall ignoriert.

Beispiel zu Fall 1: Zeilenanpassung von Klonen, bei Abänderung von Programmkode, der zeilenweise vor den beobachteten Klonen in den Klonfamilien steht.

```
class A {
   .
   .
   .
45 if(ypsilon = = delta){              | Zeile 45
46    while(zeta != ypsilon)
47         supernatural += theForce;       Ursprung:
48         zeta = supernatural;            Copy(A/x,45,50)
49         zeta ++;
50 }                                   | Zeile 50
   .
   .
}//END class A
```

```
class B {
  .
  .
   if(ypsilon = = delta){              | Zeile 245
      while(zeta != ypsilon)
      supernatural += theForce;          Klon:
           zeta = supernatural;          Paste(B/x,245,250)
           zeta ++;
   }                                   | Zeile 250
  .
  .
 }//END class B
```

```
class A {
   .
   // drei neue Zeilen vor dem beobachteten Klon wurden eingefügt
   // Veränderte Zeilen entsprechen Kode c.
   // Die Zeilenangaben müssen aktualisiert werden !
   StringBuffer sB = new StringBuffer(anyString);
   StringBuffer revSB = sB.reverse();
   StringBuffer addedRevSB = revSB.append(adddedString;
   .
48 if(ypsilon = = delta){              | Zeile 48
49    while(zeta != ypsilon)              Change(A/c,12,14)
50         supernatural += theForce;       Ursprung Zeilen anpassen:
51         zeta = supernatural;            Copy(A/x,48,53)
52         zeta ++;
53 }                                    | Zeile 5
   .
   .
}//END class A
```

Abb.10: Einfügen eines Klons aus Datei A.java in Datei B.java und Einfügen von drei Kodezeilen in A.java und erforderliche Zeilenanpassung des Klons aus Copy A/x(45,51).

In Abbildung 10 ist dargestellt, dass aus der Datei A.java ein Kodefragment (A/x,45,50) kopiert und in Datei B.java als Klon (B/x,245,250) eingefügt wird. Mit dieser Aktion ist eine neue Klonfamilie erzeugt worden. In dieser Klonfamilie befinden sich diese beiden Kodeteile und stehen ab sofort unter Beobachtung.
Wenn nun im Laufe der Softwareprogrammierung Veränderungen an den Klassen A.java und B.java vorgenommen werden, z.B. werden drei neue Kodezeilen in der Datei A.java eingefügt, muss das auch Berücksichtigung bei den entsprechenden Elementen der Klonfamilien finden, die diese beobachten. Die drei Kodezeilen werden bei diesem Beispiel

ab Zeile 10 eingefügt, somit verschiebt sich der unter Zeile 10 stehende Kode um drei Zeilen. Diese zeilenweise Veränderung des gesamten Programmkodes von A.java muss nun auch in dem betreffenden Element der Klonfamilie um diese Zeilen angepasst werden, falls der unter Beobachtung stehende Klon in der Klonfamilie zeilenweise unter der veränderten Kodestelle im Programmkode steht. Damit muss nur eine Zeilenanpassung für den Ursprung berechnet werden. Der Klon aus der Datei B.java wird keiner Zeilenberechnung unterzogen, da hier keine Abänderung des zeilenweise vor ihm stehenden Programmkodes vollzogen wurde.

Sowohl bei dem Ursprung aus (A/x,45,50) und dem Klon aus (B/x,245,251), ist es jeweils nur von Interesse wenn vor diesen beiden Elementen aus der Klonfamilie etwas textlich im Programmkode hinzugefügt oder gelöscht wird.
Das bedeutet im Hinblick auf das Klonverfolgungsprogramm, dass Zeilenaktualisierungen nur vorgenommen werden müssen, falls beobachtete Klone aus den Klonfamilien in der Datei in der die Änderung zu verzeichnen ist, zeilenweise unter dieser abgeänderten Kodestelle stehen. Der nächste abzuhandelnde Fall wird auf der folgenden Seite in Abbildung 11 gezeigt.

Beispiel zu Fall 2: Keine Zeilenanpassung von Klonen, bei Abänderung von Programmkode, der zeilenweise nach den beobachteten Klonen in den Klonfamilien steht.

```
class A {
   .
   .
   .
   if(ypsilon == delta){              | Zeile 45
      while(zeta != ypsilon)
         supernatural += theForce;       Ursprung:
         zeta = supernatural;            Copy(A/x,45,50)
         zeta ++;
   }                                  | Zeile 50
   .
   .
}//END class A
```

```
class B {
   .
   .
   if(ypsilon == delta){              | Zeile 245
      while(zeta != ypsilon)
      supernatural += theForce;        Klon:
            zeta = supernatural;       Paste(B/x,245,250)
            zeta ++;
   }                                  | Zeile 250
   .
   .
}//END class B
```

```
class B {
   .
   .
   if(ypsilon == delta){              | Zeile 245
      while(zeta != ypsilon)             Change(A/c,377,379)
         supernatural += theForce;       Klon:
         zeta = supernatural;            Paste(B/x,245,250)
         zeta ++;
   }                                  | Zeile 250
```

```
    .
    .
    .
// drei neue Zeilen nach dem beobachteten Klon wurden eingefügt
// Veränderte Zeilen entsprechen Kode c.
// Die Zeilenangaben müssen nicht aktualisiert werden !
StringTokenizer sT = new StringTokenizer(anyString);
StringBuffer revSB = sB.reverse();
StringBuffer addedRevSB = revSB.append(adddedString;}//END class B
```

Abb.11: Programmkode wird in Datei B.java abgeändert, erfordert aber keine Zeilenanpassungen seitens des Klons aus Paste(B/x,245,251), da Änderung zeilenweise nach den Zeilen des Klons erfolgte .

Die Abbildung 11 zeigt, dass zeilenweise Veränderungen von Programmkode der zeilenweise nach beobachteten Klonen aus den Klonfamilien steht, keine Berücksichtigung finden muss. Da es sich hierbei um Programmkodeveränderungen handelt, die die Klone zeilenweise nicht direkt betreffen. Da die Programmkodeänderung aus Change(A/c,377,379) zeilenweise nach dem Klon aus Paste (B/x,245,250) liegt, wird hier keine Aktualisierung des Klons durchgeführt. Die Anfangszeile und Endzeile der Kodeänderung (377 und 379) sind größer der Anfangszeile und Endzeile des Klons (245 und 250). Das bedeutet für das Kodekopieverfolgungsprogramm, dass keine Aktualisierungen an Klonen in den Klonfamilien vorgenommen werden müssen, wenn sie zeilenweise nicht davon betroffen sind.

Was ist aber, falls zeilenweise und textliche Änderungen des Programmkodes Klone direkt betreffen?

Mit direkt betreffen ist hier die zeilenweise und die textliche Abänderung innerhalb der Zeilen des Klons gemeint. Das ist dann ein Stellen- und Textaktualisierungsproblem. Ich habe das zu einem neuen Punkt zusammengefasst, da hierbei nicht nur die Stellenaktualisierungen zu erfassen sondern auch die textliche Ähnlichkeit gegenüber den anderen Klonfamilienmitgliedern zu ermitteln ist.
Diese textliche Ähnlichkeitsüberprüfung wird mit einem Stringvergleichsalgorithmus durchgeführt. Das Ergebnis dieses Vergleichs ermittelt, ob dieser Klon weiter beobachtet oder aus der Verfolgung herausgenommen wird. Das wird nun im folgenden dritten Punkt erörtert werden.
Es sei hier erwähnt, dass eine Veränderung des Programmkodes die oben beschriebenen zwei Fälle des Stellenidentitätsproblems (Punkt 2) und den Fall der Veränderung eines Klons innerhalb seiner Zeilen aus Punkt 3 umfassen kann, wenn sich die zu beobachteten Klone innerhalb einer Datei befinden.

3. Punkt:

6.3.4 Wie lange stehen Klone unter Beobachtung ?

Unter Punkt 3, der Berücksichtigung finden muss, fallen die textliche Aktualisierung der unter Beobachtung stehenden Klone und die damit einhergehende Überprüfung der Ähnlichkeit gegenüber den anderen Klonfamilienmitgliedern. Schließlich können sich im Laufe der Implementierung einer Software auch Kodestellen der beobachteten Klone textlich verändern. Diese textlichen Veränderungen von beobachteten Klonen müssen natürlich verfolgt, erkannt und aktualisiert werden. Damit zusammenhängend sind bei diesen

Änderungen auch Zeilenverschiebungen der Zeilen von beobachteten Klonen möglich. Hierbei ist es erforderlich nicht nur die textlichen Veränderungen der unter Beobachtung stehenden Klone zu erfassen, sondern auch eventuelle Zeilenverschiebungen zu erfassen. Textliche Veränderungen des Programmkodes, die nicht unter Beobachtung stehen, werden unter Punkt 2 erfasst, da es zeilenweise Verschiebungen geben kann. Falls Kodezeilen, die nicht unter Beobachtung stehen, textlich verändert werden, so wird dieser Fall nicht berücksichtigt. Diese Änderung hat keine zeilenweise Verschiebung zur Folge, sondern nur eine textliche Abänderung dieser Zeilen. Wenn doch Zeilenverschiebungen erfolgen, fällt das unter Punkt 2.

Unter Beobachtung stehende Klonfamilie:

{(Clone(A/x,i,j)),(Clone(B/x,k,l)),Clone(X/x,m,n)}.

Falls

a) **Change(A/y,c,d)** mit **c >= i & d <= j | c < i & d <= j | c <= j & d >j**

oder

b) **Change(B/y,c,d)** mit **c >= k & d <= l | c < k & d <= l | c <= k & d >l**

oder

c) **Change(X/y,c,d)** mit **c >= m & d <= n | c < m & d <= n | c <= m & d >n**,

mit **i**,**j**,**k**,**l**,**m**,**n**,**c**,**d** aus **1..N** und **N >=1** mit **N** aus der Menge der ganzen Zahlen.

=> Veränderung vornehmen.

Im Fall a) muss der **Clone(A/x,i,j)** aktualisiert werden.
Im Fall b) wird der **Clone(B/x,k,l)** abgeändert.
Oder im Fall c) wird der **Clone(X/x,m,n)** aktualisiert.

Nach erfolgreicher Aktualisierung des betreffenden Klons muss folgendes ermittelt werden:

Wenn der Klon nach der Aktualisierung mit einem Klonfamilienmitglied seiner Familie einen textlichen Unterschied von weniger als 50% aufweist, steht er weiterhin unter Beobachtung.
Wenn der Klon aber mit keinem seiner Familienmitglieder einen textlichen Unterschied von weniger als 50% aufweisen kann, dann wird er aus der Beobachtung genommen. Er verbleibt aber weiterhin in der Klonfamilie. So lange er nicht unter Beobachtung steht, wird er bei möglichen Warnungen die diese Klonfamilienmitglieder betreffen, nicht mitberücksichtigt. Er hat zu diesem Zeitpunkt keine Ähnlichkeit mit seinen Klonfamilienmitgliedern. Der Klon wird bei eventuellen späteren Aktualisierungen weiterhin berücksichtigt, da er sich nach diesen Aktualisierungen seinen Klonfamilienmitgliedern textlich wieder annähern kann. Folglich wird dieser Klon bei Warnmeldungen mit den übrigen beobachteten Klonen wieder aufgeführt.

Der textliche Unterschied zweier Klone wird durch die Levenshtein-Distanz ermittelt, Beispiel:

```
class A {
   .
   .
   .
   if(ypsilon == delta){              | Zeile 45
      while(zeta != ypsilon)
         supernatural += theForce;       Ursprung:
         zeta = supernatural;            Copy(A/x,45,50)
         zeta ++;
   }                                  | Zeile 50
   .
   .
}//END class A
```

```
class B {
  .
  .
    // Verändern des Kodes des beobachteten Klons, Kode c wird eingefügt.
    // Es muss eine Zeilen- und Textanpassung erfolgen!
    // Nach bestandener Überprüfung der Ähnlichkeit zu seinen in Beziehung
    // stehenden  Klonmitgliedern bleibt der Klon in der Klonfamilie
    // unter Beobachtung. Ansonsten wird dieser Klon nicht mehr beobachtet.
   if(ypsilon == delta){              |  Zeile 245
      if(delta != zeta){
            delta++;                    Klon textlich und zeilenweise anpassen:
      }
      while(zeta != ypsilon)              Paste(B/x+ (Change(B/c,246,248)),245,250)
      supernatural += theForce;
           zeta = supernatural;          => Paste(B/x+c,245,253)
           zeta ++;
   }                                  | Zeile 253
  .
  .
}//END class B
```

Abb.12: Zeilenweise und textliche Abänderungen des Programmkodes, die ein Klon einer Klonfamilie direkt betreffen.

Abbildung 12 zeigt, dass bei der Veränderung eines Klons, dieser den Änderungen entsprechend aktualisiert werden muss. Es werden die aktuellen Zeilennummern und der aktuelle Kode nach dieser Änderung ermittelt. Nach den erforderlichen Aktualisierungen wird mittels Levenshtein-Distanz ermittelt, ob dieser eben aktualisierte Klon keine Ähnlichkeit mehr gegenüber seinen Klonfamilienmitgliedern aufweist.
Falls ein Klon textlich insgesamt aus dem Programmkode durch eine Aktivität des Programmierers gelöscht wird, so wird er auch aus der Klonfamilie gelöscht. Das bedeutet, dass die Klonfamilie mit n Elementen vor der Löschaktivität, nach dem Löschen aus n-1 Elementen besteht. Das Klonverfolgungsprogramm muss also die textliche Veränderung ermitteln, die Levenshtein-Editierdistanz zu den anderen Familienmitgliedern berechnen und entscheiden, ob dieser Klon aus der Verfolgung genommen wird oder weiterverfolgt wird.

Falls eine Klonfamilie nur noch ein Element enthält, dann wird diese Klonfamilie ebenfalls gelöscht, da keine Beziehung zu einem weiteren Klon in der Klonfamilie besteht. Das bedeutet, dass das Klonverfolgungsprogramm erkennen muss, wann eine Klonfamilie nur

noch ein Element enthält, um diese Klonfamilie dann zu löschen. Die folgenden Abbildungen 13 und 14 zeigen diesen Sachverhalt.

```
class B {
  .
  .
   if(ypsilon == delta){              | Zeile 245
      if(delta != zeta){
            delta++;                    Klon: Paste(B/x+c,245,2530)
      }
      while(zeta != ypsilon)            Klonfamilie:
      supernatural += theForce;         {(Copy(A/x,45,50)), (Paste(B/xc,245,2530))}
}
           zeta = supernatural;
           zeta ++;
   }                                  | Zeile 253
  .
  .
 }//END class B
```

```
class B {
   .
   .
   .
}//END class B
```

| Zeile 245 bis 253 aus Programmkode gelöscht.
Daraus folgt das das Element aus Paste(B/xc,245,2530) aus der Klonfamilie zu löschen ist.
Daraus folgt, dass wenn die Anzahl der Elemente der Klonfamilie = 1 ist, dann ist die Klonfamilie aus der Verfolgung nehmen.
Klonfamilie: {(Copy(A/x,45,50))} # Elemente = 1 => Klonfamilie aus der Beobachtung nehmen.

Abb.13: Löschen des Klons hat zur Folge, dass ein Klon aus der Klonfamilie gelöscht wird. Klonfamilie wird aus der Beobachtung genommen, da sie nur noch ein Element enthält.

```
class B {
  .
  .
   if(ypsilon == delta){              | Zeile 245
      if(delta != zeta){
            delta++;                    Klon: Paste(B/x+c,245,2530)
      }
      while(zeta != ypsilon)
      supernatural += theForce;
}
           zeta = supernatural;
           zeta ++;
   }                                  | Zeile 253
  .
  .
 }//END class B
```

Klonfamilie: {(Copy(A/x,45,50)), (Paste(B/xc,245,253)),(Paste (C/x+c,300,305))}

```
class B {
  .
```

```
 .
   // Verändern des Kodes des beobachteten Klons, Kode c wird gelöscht.
   // Es muss eine Zeilen- und Textanpassung erfolgen!
   // Nach bestandener Überprüfung der Ähnlichkeit zu seinen in Beziehung
   // stehenden  Klonmitgliedern bleibt der Klon in der Klonfamilie
   // unter Beobachtung. Ansonsten wird dieser Klon nicht mehr beobachtet.
  if(ypsilon == delta){              | Zeile 245
     if(delta != zeta){
           delta++;
     }
     StringTokenizer sT = new StringTokenizer(anyString);
     StringBuffer revSB = sB.reverse();
  }                                  | Zeile 251
 .
 .
}//END class B
```

Klon textlich und zeilenweise anpassen und Editierdistanz zu allen Familienmitgliedern berechnen:

Paste(B/x+ (Change(B/c,246,248)),249,251) => Paste(B/x-c,245,251) aber liegt nach Vergleich der Ähnlichkeit über dem Maß und muss somit aus der Verfolgung gestrichen werden.

Klonfamilie: {(Copy(A/x,45,50)), Paste (C/x+c,300,305))
 # Elemente > 1 => Klonfamilie weiter beobachten.

Abb.14: Abänderung des Programmkodes hat zur Folge, dass ein Klon aus der Klonfamilie gelöscht wird. Klonfamilie wird weiter beobachtet.

6.3.5 Abschluss einer CopyPasteChange-Episode

Es wird nun formal beschrieben, welche Aktivitäten zum Abschluss einer CopyPasteChange-Episode und damit im Ergebnis zu Warnmeldungen führen.
Warnmeldungen werden generiert, wenn eine Klonfamilie unter Beobachtung steht und ein unter Beobachtung stehendes Klonfamilienmitglied textlich verändert wird.

Klonfamilien und Textveränderungsaktionen die zu Warnmeldungen führen:

Bemerkung: Eine Textveränderung, also die Change-Aktivität kann auch eine Zeilenveränderung umfassen. Zeilenverschiebungen sind das Ergebnis des Löschens von einer oder mehrerer Zeilen oder das Einfügen von einer oder mehrerer Zeilen.

Klonfamilie Y = {(Clone(A/x,i,j)),…,(Clone(X/x,k,l))}.

- Textliche Veränderung aus Change-Aktivität löst Warnmeldungen aus,

Change((A/y,c,d) |…| (X/y,a ,b)),

mit **c >= i**, **d <= j**, **a >= k**, **b <= l**,

wobei **a**,**b**,**c**,**d** ,**i**,**j**,**k**,**l** aus **1..N** und **N >=1** mit **N** aus der Menge der ganzen Zahlen.

Diese eben formal beschriebene Veränderung eines Klons mittels Change, führt zum Abschluss einer CopyPasteChange-Episode der Klonfamilie. Daraus folgt, dass eine Warnmeldung generiert werden muss.

Folgende Information soll mit dieser Warnung für alle Klone in der Klonfamilie, außer für den gerade veränderten Klon, ausgeben werden:

- Der Zeitpunkt des Abschlusses der CopyPasteChange-Episode dieser Klonfamilie.

- Der Name des Anwenders, der durch seine Programmieraktivitäten diese CopyPasteChange-Episode ausgelöst hat.

- Die Klone, die in dieser Klonfamilie enthalten sind und unter Beobachtung stehen. Die Klone werden repräsentiert durch den Dateinamen in denen sie als Text enthalten sind zusammen mit ihrer Anfangs und Endzeile.

Beispiel:

Folgende Klonfamilie steht unter Beobachtung:

{(Clone(A/x,45,50)), (Clone(B/x,245,253)),(Clone (C/x,300,305))}

Nun folgt eine Änderung eines Klonfamilienmitglieds mit **Change (C/c,302,303)**.
Klon **(Clone (C/x,300,305)** unterliegt also einer textlichen Veränderung von **Zeile 302** bis **Zeile 303** in der beobachteten Datei **C**, wobei **c** die textliche Veränderung ausmacht.
Es werden Warnungen generiert, die alle unter Beobachtung stehenden Klone aufführt, außer **Clone(C/x,300,305)**. Nun muss die Textaktualisierung bei **Clone(C/x,300,305)** durchgeführt werden und der Stringvergleichsalgorithmus überprüft, ob der Klon noch eine Ähnlichkeit gegenüber einem Klonfamilienmitglied seiner Familie aufweist
Weist dieser gerade veränderte Klon keine Ähnlichkeit gegenüber einem seiner übrigen Familienmitgliedern auf, so wird er aus der Beobachtung genommen. Der Klon wird aber bei folgenden Änderungen mitberücksichtigt. Er unterliegt dann wieder der Ähnlickeits-überprüfung, um feststellen zu können, ob er sich textlich an die übrigen Klone angenähert hat. Falls das der Fall ist, wird er erneut weiterbeobachtet und kann auch wieder mit einer Warnmeldung aufgeführt werden.

7. Implementierung

In diesem Kapitel wird die softwaretechnische Umsetzung des Klonverfolgungsprogramms erläutert. Das Klonverfolgungsprogramm, also der CopyPasteChange-Episodenerkenner (CPC-Episodenerkenner) wird in die ElectroCodeoGram-Software (ECG) integriert. Das ECG ist ein Protokollierungswerkzeug. Es zeichnet Aktivitäten eines Programmierers während der Softwareentwicklung mit der Entwicklungsumgebung Eclipse auf.

Die Arbeitsweise des ECG und die aus dieser Software für den CPC-Episodenerkenner benötigten Grundlagen werden im folgenden Kapitel 7.1 kurz und knapp beschrieben.

In Kapitel 7.2 wird erläutert, wie die Identifikation von CPC-Episoden im Hinblick auf das ECG umgesetzt wird.

In Kapitel 7.3 wird auf schon vorhandene Episodenerkenner in der ECG-Software eingegangen. Kapitel 7.4 beschreibt den CPC-Episodenerkenner aus softwaretechnischer Sicht. Kapitel 7.5 knüpft an Kapitel 7.4 an und geht auf die konkrete Implementierung als Zustandsautomat ein.

Für ein detaillierteres Befassen mit der ECG-Software verweise ich erneut auf die Diplomarbeit von Frank Schlesinger[2].

7.1 Arbeitsweise des ElectroCodeoGram (ECG)

Das ECG_Lab ist eine Art Sammelstelle für die aufgezeichneten Ereignisse im ECG. Mit dem ECG_Lab ist es möglich unter Verwendung verschiedener Module ad hoc Analysen durchzuführen. Der ECG_Sensor zeichnet Mikroprozessereignisse aus Eclipse heraus auf und sendet diese an das ECG_Lab. Der ECG_Sensor wertet die Ereignisse aus und kodiert sie in ein XML Dokument. Dieses XML Dokument repräsentiert das jeweilige ausgewertete Ereignis. So lange wie das ECG_Lab läuft und der Programmierer programmiert, werden die vom ECG_Sensor ermittelten und gesendeten Mikroprozessereignisse aufgezeichnet.

7.2 Identifikation von CopyPasteChange-Episoden mittels ECG

7.2.1 Relevante Ereignisse des ECG

Bei einer CPC-Episode handelt es sich um eine zeitlich sequentielle Abfolge von bestimmten einzelnen Aktivitäten während des Mikroprozesses der Softwareentwicklung.

Die Aktivitäten sind:

- **Ausschneiden eines Textes X oder Kopieren eines Textes X**
- **Einfügen eines Textes X**
- **Verändern des Programmkodes**

Das Klonverfolgungsprogramm wird in das ECG integriert und reagiert auf so genannte Ereignisse, die mittels ECG_Sensor erfasst werden. Diese Ereignisse entsprechen genau den oben angegebenen Aktivitäten.

Grundsätzlich ist das Klonverfolgungsprogramm somit abhängig von der Korrektheit der gesendeten Ereignisse und der in ihnen enthaltenen Informationen. Korrektheit der Ereignisse bedeutet, dass diese Ereignisse auch tatsächlich exakt die Aktivitäten des Programmierers abbilden, die der Programmierer gerade ausführt.

Das Klonverfolgungsprogramm muss entsprechend der beschriebenen Theorie ab Kapitel 6 auf diese gesendeten Ereignisse reagieren.

Folgende Aktivitäten des Programmierers lösen Ereignisse aus, die den Input für das Klonverfolgungsprogramm beschreiben:

- ***Copy***, das Kopieren eines Kodeteils beliebiger Länge aus einem Quelltext heraus mittels Tastatur oder Maus.

- ***Cut***, das Ausschneiden einer beliebigen Stelle eines Quelltextes mittels Tastatur oder Maus.

- ***Paste***, das Einfügen des zuvor kopierten oder ausgeschnittenen Kodeteils an eine beliebige Stelle eines Quelltextes mittels Tatstatur oder Maus.

- ***Change***, wird durch das Abändern des Programmkodes ausgelöst. Hierbei wird ein ***Codechange-Ereignis*** generiert, das aber nicht direkt von dem Klonverfolgungsprogramm verarbeitet wird.

Die konkreten Ereignisse, die gesendet und von dem Klonverfolgungsprogramm verarbeitet werden, sind:

- ***Textoperation***-Ereignisse

- ***Linediff***-Ereignisse

Unter den Textoperation-Ereignissen werden die Aktivitäten Cut, Copy und Paste zusammengefasst. Das Linediff-Ereignis umfasst die Veränderung des Programmkodes. Diese Ereignisse werden durch die so genannten Mikrosensordatentypen (msdt) umgesetzt. In der ECG-Software sind sie als ***msdt.textoperation.xsd*** und ***msdt.linediff.xsd***, XML-Dateien realisiert. Der msdt.linediff.xsd wird aus einem ***msdt.codechange.xsd*** ermittelt. Das Codechange-Ereignis drückt nichts Weiteres als das Ändern des Kodes in einer Datei aus. Das Codechange-Ereignis enthält das gesamte Dokument nach einer Änderung des Kodes. Differenzen des Programmkodes können demnach nach zwei Codechange-Ereignissen, durch das Linediff-Ereignis generiert werden.

Das msdt.textoperation.xsd enthält folgende Informationen für die Cut, -Copy, und Paste-Aktivität:

Cut)

- ***version***: Gibt die Version des Mikrosensordatentyps an.
- ***creator***: Ein String der den Erzeuger dieses Ereignisses beschreibt. Meist ein Sensor oder ein Module des ECG.
- ***username***: Name des Programmierers, der gerade in Eclipse programmiert.

- ***projectname***: Name des Projekts, dass der Programmierer bearbeitet.
- ***id***: Ein Identifikationsnummer der auf der Resource basiert dem dieses Ereignis Gemeldet wurde.
- ***activity***: Cut
- ***editorname***: Der aktive Editor während des Sendens dieses Ereignisses.
- ***selection***: Inhalt der Selektion eines Textinhalts eines Editors.
- ***startline***: Startzeile der Selektion.
- ***endline***: Endzeile der Selektion.
- ***offset***: Offset der Selektion.

Copy)

Alle Elemente wie bei Cut.

.

Ausser:

- ***activity***: Copy.

Paste)

Alle Elemente wie bei Cut und Copy.

Ausser:

- ***activity***: Paste.
- ***clipboard***: Inhalt des Clipboards.

Das msdt.linediff.xsd enthält folgende Informationen für die Change-Aktivität:

- ***documentname***: Der Dateiname in der die Kodeänderung vorgenommen wurde.
- ***lines***: Eine Menge von aufeinander folgenden Kodezeilen die folgende Information enthalten:

  - ***type***: Art der Änderung:

    - ***changed***: Zeile wurde mit neuem Inhalt ersetzt.
    - ***inserted***: Zeile wurde neu eingefügt.
    - ***deleted***: Zeile wurde gelöscht.

  - ***linenumber***: Die Nummer der Zeile.
  - ***from***: Der Inhalt der Zeile vor der Änderung.
  - ***to***: Der Inhalt der Zeile nach der Änderung.

Der Mikrosensordatentyp msdt.textoperation.xsd enthält alle benötigten Informationen um die Cut, -Copy und Paste-Aktivitäten zu verarbeiten. Der Mikrosensordatentyp msdt.linediff.xsd enthält die benötigten Informationen zum Verarbeiten der Abänderungen eines Programmkodes.

7.2.2 Vorgehensweise um das Problem der Klonverfolgung zu lösen

Die Ereignisse und ihre Daten werden in eine Ausgabedatei geschrieben. Diese Ausgabedatei ist die Log-Datei. Das Klonverfolgungsprogramm liest zeilenweise die Log-Datei, erkennt die relevanten Ereignisse und liest die in den Ereignissen enthaltenen Daten. Die Daten werden vom Klonverfolgungsprogramm verarbeitet. Wird eine der CopyPasteChange-Episode entsprechende Reihenfolge der Ereignisse erkannt, dann generiert das Klonverfolgungsprogramm Warnungsereignisse. Diese Warnungen, eine Menge von ***cpcwarnings*** wird dann ebenfalls in eine Log-Datei geschrieben.

Für das Klonverfolgungsprogramm sind folgende Schritte zu beachten. Die Schritte geben eine erste algorithmische Beschreibung der Arbeitsweise des Programms an.

- Die Log-Datei, die die Ereignisse enthält, muss eingelesen werden.
- Die Ereignisse Textoperation und Linediff müssen identifiziert und verarbeitet werden.
- Sobald in einer Datei, die unter Beobachtung steht, textlich und oder zeilenweise Veränderungen vorgenommen werden, müssen bezüglich aller Klonfamilien die diese Dateien beobachten, alle relevanten Aktualisierungen ermittelt und evaluiert werden. Das ist notwendig, da die von dieser Datei betroffenen Kodestellen (Klone in der Klonfamilie) im Hinblick auf die gerade vollzogene Änderung ebenfalls konsistent abgeändert werden müssen.
- Falls ein unter Beobachtung stehender Klon (d.h. etwaige Copy -, Cut- oder Paste-Aktivitäten wurden bereits verarbeitet) textlich verändert wird, werden die Warnmeldungen generiert. Das ist im Ergebnis das Abschließen einer CPC-Episode.
- Wenn ein Klon, der unter Beobachtung steht textlich gelöscht wird, so wird dieser Klon aus der Klonfamilie ebenfalls gelöscht. Besteht diese Klonfamilie nur noch aus einem beobachteten Klon, wird diese Klonfamilie aus der Verfolgung genommen.
- Falls ein unter Beobachtung stehender Klon so stark verändert wird, dass er zu den anderen Klonfamilienmitgliedern keine Ähnlichkeit mehr aufweist, wird er zunächst aus der Beobachtung genommen. Falls er bei darauf folgenden Abänderungen wieder Ähnlichkeit gegenüber den anderen Klonen in der Klonfamilie aufweist, wird er erneut unter Beobachtung gestellt.

7.3 In die ECG-Software integrierte Episodenerkenner

7.3.1 Das EpisodeRecognizerIntermediateModule

Das EpisodeRecognizerIntermediateModule ist ein Episodenerkenner-Container für eine konkrete Episodenerkenner Implementierung. Der Container kann genau einen konkreten Episodenerkenner enthalten. Falls verschiedenartige Episodenerkenner benötigt werden,

d.h. für jede Art einer Episode ein Episodenerkenner, benötigt jeder Erkenner eine neue Instanz eines EpisodeRecognizerIntermediateModule. Diese Instanz wird mit dem neuen konkreten Episodenerkenner konfiguriert.

7.3.2 Der EpisodeRecognizerManager

Ein EpisodeRecognizerManager verwaltet Instanzen eines konkreten Episodenerkenners.
Es steht immer mindestens eine erzeugte Instanz eines Episodenerkenners im Initialzustand zur Verfügung. Das ist so realisiert worden, da es möglich sein kann, dass potentielle Episoden parallel existieren und somit auch parallel abgearbeitet werden müssen. Bei der CPC-Episodenerkennung ist dies der Fall. Für jedes EpisodeRecognizerIntermediateModule existiert genau ein EpisodeRecognizerManager, der genau eine Art von Episodenerkenner verwaltet. Der EpisodeRecognizerManager ruft mit seiner Analyse-Methode alle bei ihm registrierten Erkenner auf, die dann wiederum ihre implementierte Analyse-Methode ausführen

7.3.3 Das EpisodeRecognizer Interface

Alle zu realisierenden Episodenerkenner müssen das EpisodeRecognizer Interface implementieren. Das Interface definiert eine minimale Implementierungschnittstelle für Episodenerkenner.

Der in 7.3.2 vorgestellte EpisodeRecognizerManager erzeugt dynamisch Instanzen der registrierten Erkenner, daher muss ein Default-Konstruktor seitens der Erkenner angeboten werden. Abbildung 15 zeigt die Definition des Interface.

```
/*
 * (c) Freie Universität Berlin - AG SoftwareEngineering - 2006
 */

package org.electrocodeogram.module.intermediate.implementation;

import org.electrocodeogram.event.ValidEventPacket;

public interface EpisodeRecognizer {

      boolean isInInitialState();

      boolean isInFinalState();

    ValidEventPacket analyse(ValidEventPacket packet,
                                     Long minDuration);

}
```

Abb.15: EpisodeRecognizer Interface aus der ECG-Software

7.3.4 Beispiel für einen Episodenerkenner

Im Modul ECG_EpisodeRecognizerIntermediateModule sind alle bisherigen implementierten Episodenerkenner umgesetzt worden. Um für die Problemstellung meiner Arbeit einen Ansatzpunkt zu finden, welches Softwaredesign für das Klonverfolgungsprogramm in Frage käme, habe ich mir die Umsetzung bisheriger implementierter Episodenerkenner betrachtet und analysiert. Ich möchte nun ein Beispiel eines einfachen Episodenerkenners angeben, um zu zeigen woran ich mich bei meiner Designentscheidung für das Klonverfolgungsprogramms orientierte.

Die Aufgabe des WindowActiveEpisodeRecognizer ist folgende:

- Die Dauer der Aktivität des Programmierers in einem Hauptfenster der Anwendung Eclipse zu ermitteln und zu erkennen.

- Die relevanten Ereignisse, die der Automat verarbeitet (msdt-Typen) sind:

    - msdt.window.xsd
    - msdt.part.xsd
    - msdt.dialog.xsd
    - msdt.editor.xsd

- Als Ergebnis eine Episode generieren, die die Dauer dieses Ereignisses als Information enthält und an den Aufrufer, den EpisodeRecognizerManager, zurückgibt.

Der WindowActiveEpisodeRecognizer ist, wie alle übrigen Episodenerkenner, softwaretechnisch als Zustandsautomat realisiert worden. Der Automat hat definierte Zustände, wie START, WINDOWACTIVE und STOP. Wenn der WindowActiveEpisodeRecognizer sich im START-Zustand befindet, das ist der so genannte Initialzustand, ist noch kein Fenster aktiviert worden. Im WINDOWACTIVE-Zustand wurde ein Fenster aktiviert. Wenn das Fenster deaktiviert wurde, wechselt der Automat in den STOP-Zustand. Befindet sich der WindowActiveEpisodeRecognizer-Automat Im STOP-Zustand, so ist das der Endzustand des Automaten. Der Manager überprüft, ob ein bei ihm registrierter Erkenner in seiner Liste vorhanden ist, der sich im STOP-Zustand befindet. Falls er einen findet, wird er aus seiner Liste gelöscht. Denn dieser Automat, der sich im STOP-Zustand befindet, würde auf kein einziges folgendes Ereignis mehr reagieren. Falls sich in der Liste des Managers kein WindowActiveEpisodeRecognizer-Automat mehr im Initialzustand vorfindet, dann erzeugt der Manager eine neue Automateninstanz, der sich im Initialzustand START befindet.

Im Rahmen meiner Arbeit ist somit der Weg bezüglich der Realisierung des zu implementierenden Klonverfolgungsprogramms implizit vorgegeben gewesen.
Auf der folgenden Abbildung sind Auszüge aus der Implementierung des Episodenerkenners WindowActiveEpisodeRecognizer aufgeführt. An diese Implementierung lehnte ich den Entwurf des Klonverfolgungsprogramms an. Das Klonverfolgungsprogramm würde somit als Zustandsautomat realisiert werden.

```
       .
       .
       .
public class WindowActiveEpisodeRecognizer  extends AbstractSingleEpisodeRecognizer
{

    /**
     * State types for this recognizer
     */
    public enum WindowActiveEpisodeState {
        /**
         * Ínitial state, no window recognized yet
         */
        START,

        /**
         * Window has been activated
         */
        WINDOWACTIVE,

        /**
         * Final state after window deactivate
         */
        STOP
    }
       .
       .
       .

public ValidEventPacket analyseSingle(ValidEventPacket packet, long
                                                        minDuration)
{

       ValidEventPacket event = null;
       String msdt = packet.getMicroSensorDataType().getName();

       try {
        if (msdt.equals("msdt.window.xsd") ||
            msdt.equals("msdt.part.xsd")   ||
            msdt.equals("msdt.dialog.xsd") ||
            msdt.equals("msdt.editor.xsd"))
        {

             Document document = packet.getDocument();
             Date timestamp = packet.getTimeStamp();

             String activity = ECGParser.getSingleNodeValue("activity",
                                                            document);
             String windowname = "";

             if (msdt.equals("msdt.window.xsd"))
                  windowname = ECGParser.getSingleNodeValue("windowname",
                                                              document);
            if (msdt.equals("msdt.dialog.xsd"))
             windowname = ECGParser.getSingleNodeValue("dialogname",
                                                              document);

             if (state == WindowActiveEpisodeState.START &&
                     (activity.equals("activated") ||
                        activity.equals("opened"))) {

                     state = WindowActiveEpisodeState.WINDOWACTIVE;
                     activeWindow = windowname;
                     startDate = timestamp;
             }

             else if (msdt.equals("msdt.window.xsd") &&
                       state == WindowActiveEpisodeState.WINDOWACTIVE &&
                         (activity.equals("deactivated") ||
                        activity.equals("closed")))
             {
               if (activeWindow == null) {
                    activeWindow = windowname;
               }
```

```
                    event = generateEpisode("msdt.windowactive.xsd",
                            minDuration,
                            ECGParser.getSingleNodeValue("username",
                                                document),
                            ECGParser.getSingleNodeValueIfAvailable
                          ("projectname",     document),
                            timestamp,startDate,activeWindow);
                            activeWindow = null;
                            startDate = null;
                            state = WindowActiveEpisodeState.STOP;
                }

            }

        } catch (NodeException e) {
              logger.log(Level.SEVERE, "Could not read XML string in
              WindowActiveEpisodeRecognizer.");
              logger.log(Level.FINE, e.getMessage());
        }

// Bemerkung: Hier wird nur ein einziges event oder null an den Aufrufer,
//            (EpisodeManager) für WindowActiveEpisodeRecognizer-Objekte
//            zurückgegeben!
      return event;

 }
```

Abb.16: EpisodeRecognizer Interface aus der ECG-Software.

Weitere Beispiele für Episodenerkenner aus dem EpisodeRecognizerIntermediateModule sind z.B. der FileActiveEpisodeRecognizer, der PartActiveEpisodeRecognizer und der RunActiveEpisodeRecognizer.

7.4 Der CopyPasteChangeEpisoden-Erkenner als Zustandsautomat

Im Hinblick auf die Realisierung des CPC-Episodenerkenners musste ich mir Gedanken bezüglich des softwaretechnischen Umsetzens des Automaten machen. Der CPC-Episodenerkenner würde mit Sicherheit mehr Zustände berücksichtigen müssen, als der oben vorgestellte WindowActiveEpisodeRecognizer, der mit drei Zuständen realisiert wurde.
Damit würde auch der Kode komplexer werden als bei einem Zustandautomaten, der drei Zustände abarbeiten muss.
Der CPC-Episodenerkenner muss zwei Ereignis-Signale verarbeiten, die er aus einer Log-Datei liest. Zum einen das Textoperation-Ereignis und zum anderen das Linediff-Ereignis, dass aus einem Codechange generiert wird. Wobei ein Textoperation-Ereignis entweder einer Copy-, Cut-oder Paste-Aktivität entspricht. Der Automat muss auf diese Ereignisse entsprechend reagieren und gegebenenfalls seinen Zustand ändern.

Das Linediff-Ereignis, dass die Veränderung des Programmkodes ermittelt, stand zunächst noch nicht zur Verfügung. Dieser Ereignistyp gekoppelt mit der Software, die die Veränderungen bezüglich des Programmkodes berechnet, musste noch entwickelt werden. Es stand aber das Textoperation-Ereignis zur Verfügung.

Zunächst konzentrierte ich mich somit auf das Textoperation-Ereignis. Das Textoperation-Ereignis trägt zur Erzeugung der Klonfamilien bei. Die Aktivitäten, die mit diesem Ereignistyp abgebildet sind, betreffen ja genau das Kopieren oder Ausschneiden und das Einfügen von Programmkode.

Das Textoperation-Ereignis umfasst wie oben kurz erwähnt:

- Die Copy-Aktivität (copy)
- Die Cut-Aktivität (cut)
- Die Paste-Aktivität (paste)
- Die in Klammern aufgeführten Aktivitäten sind für den Zustandsübergang wichtig.

Zustände des CPC-Episodenerkenner, die am Erzeugen der Klonfamilien beteiligt sind:

- START
- COPY
- CUT
- CUTPASTE
- PASTE
- STOP

Ein Episodenerkenner befindet sich, wenn er vom EpisodenManager erzeugt wird, Initial im START-Zustand. Nun ist der CPC-Episodenerkenner bereit Ereignisse zu verarbeiten. Wenn der Automat sich im COPY-Zustand befindet, hat er gerade ein Copy-Ereignis aus der Log-Datei gelesen. Ist er im CUT-Zustand, wurde ein Cut-Ereignis aus der Log-Datei gelesen.
In den CUTPASTE-Zustand wechselt der CPC-Episodenerkenner, wenn nach einem Cut-Ereignis unmittelbar ein Paste-Ereignis auftritt. Das impliziert, dass nach dem Cut-Ereignis zunächst ein Codechange-Ereignis erfolgt, da der Programmkode einer Veränderung unterlag. Im PASTE-Zustand befindet er sich, wenn ein Paste-Ereignis gelesen wurde. Im STOP-Zustand befindet sich der Automat, wenn er auf kein folgendes Ereignis entsprechend reagieren kann.

Das Abarbeiten von Textoperation-Ereignissen führt zu folgendem Zustandsdiagramm:

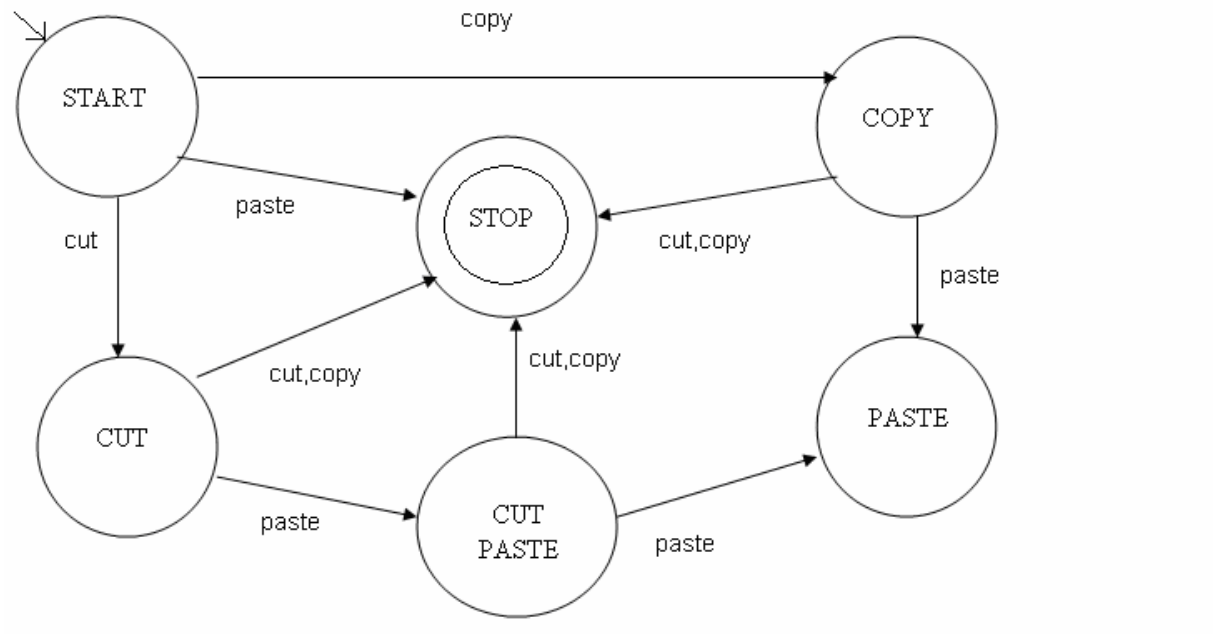


Abb.17: Zustandsdiagramm für das Verarbeiten von Textoperation-Ereignissen des CPC-Automaten.

Mit den in Abbildung 17 aufgeführten Zuständen und Zustandsübergängen kann der Automat die Textoperation-Ereignisse verarbeiten.

Beispiele für einige Zustandswechsel einer Klonfamilienerzeugung:

- Der CPC-Episodenerkenner befindet sich im Initialzustand START.

- Der Programmierer kopiert Programmkode aus Datei Test2.java. Der CPC-Episodenerkenner reagiert auf das Copy-Ereignis und wechselt in den COPY-Zustand.

```
public class Test2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Nasenbär21");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär23");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");

        System.out.println("Nasenbär221");
        System.out.println("Nasenbär222");
        System.out.println("Nasenbär223");
        System.out.println("Nasenbär224");
        System.out.println("Nasenbär225");
    }
}
```

CPC- Episodenerkenner mit folgenden Ausgabe-Daten (Diese Ausgabe diente nur zu Testzwecken und zur Evaluierung des CopyPasteChangeEpisodeRecognizer):

Ausgabe:

- Nummer des CPC-Episodenerkenner.

- Aktueller Zustand des CPC-Episodenerkenner.

- Elemente der Klonfamilie *vor* Bearbeitung des Ereignisses, mit Dateiname, Kode, Startzeile und Endzeile.

- Zu verarbeitendes Ereignis.

- Eventuelle CPCWarning-Meldung.

- Elemente der Klonfamilie *nach* Bearbeitung des Ereignisses, mit Dateiname, Kode, Startzeile und Endzeile.

- Nummer des CPC-Episodenerkenner.

- Aktueller Zustand des CPC-Episodenerkenner.

Bemerkung: Die Zeilenangaben in der Ausgabe müssen um 1 inkrementiert werden, damit sie den Zeilen im Editor von Eclipse entsprechen.
Bei der Ausgabe für die CPCWarning-Ereignisse allerdings stimmen die Zeilenangaben mit den Zeilenangaben des Editors überein.

```
nr: 1
zustand: START
msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: copy
datei:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
```

```
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");
startzeile: 5
endzeile: 9
zeitstempel: Fri Dec 29 21:10:44 CET 2006
nr: 1
zustand: COPY
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5                        //Zeilen um 1 inkrementieren !
endline: 9                          //Zeilen um 1 inkrementieren !
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");
```

- Der kopierte Kode wird in der Datei Test1.java an eine Stelle eingefügt und der CPC-Episodenerkenner verarbeitet das Paste-Ereignis und wechselt in den PASTE-Zustand.

```
public class Test1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub

        System.out.println("Nasenbär21");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär23");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");
    }
}
```

CPC-Episodenerkenner Nummer 1 Ausgabedaten:

```
nr: 1
zustand: COPY

filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");

msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: paste
datei:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");
```

```
startzeile: 11
endzeile: 11
zeitstempel: Fri Dec 29 21:10:49 CET 2006

nr: 1
zustand: PASTE
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");
```

- Ist ein CPC-Episodenerkenner im START-Zustand und er erhält z.B. als Signal ein Paste-Ereignis, dann wechselt er in den STOP-Zustand.

CPC-Episodenerkenner Nummer 5 Ausgabedaten:

```
nr: 5
zustand: START


msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: paste
datei:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
code: System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
startzeile: 9
endzeile: 13
zeitstempel: Fri Dec 29 21:11:15 CET 2006

nr: 5
zustand: STOP
```

Die restlichen Zustände werden entsprechend dem Zustandsdiagramm verarbeitet. Die drei Beispiele sollen zunächst genügen, um die grundsätzliche Arbeitsweise des Automaten nachvollziehen zu können. Der Automat muss noch um das Linediff-Ereignis und den CHANGE-Zustand erweitert werden. Das Linediff-Ereignis ist zusammen mit den Copy-, Cut- und Paste-Ereignissen für die Zustandsübergänge relevant. Der CHANGE-Zustand bildet ab, dass ein Klon des CPC-Episodenerkenners verändert wurde.

Mit dem Linediff-Ereignis ist dann die Abänderung des Programmkodes und somit die Abänderung von beobachteten Klonen abgedeckt. Das erweiterte Zustandsdiagramm auf der folgenden Seite zeigt wie der Zustandsautomat nach der Erweiterung Ereignisse verarbeitet.

Wenn das Linediff-Ereignis mit in die Betrachtung genommen wird, dann führt das zu folgendem Zustandsdiagramm:

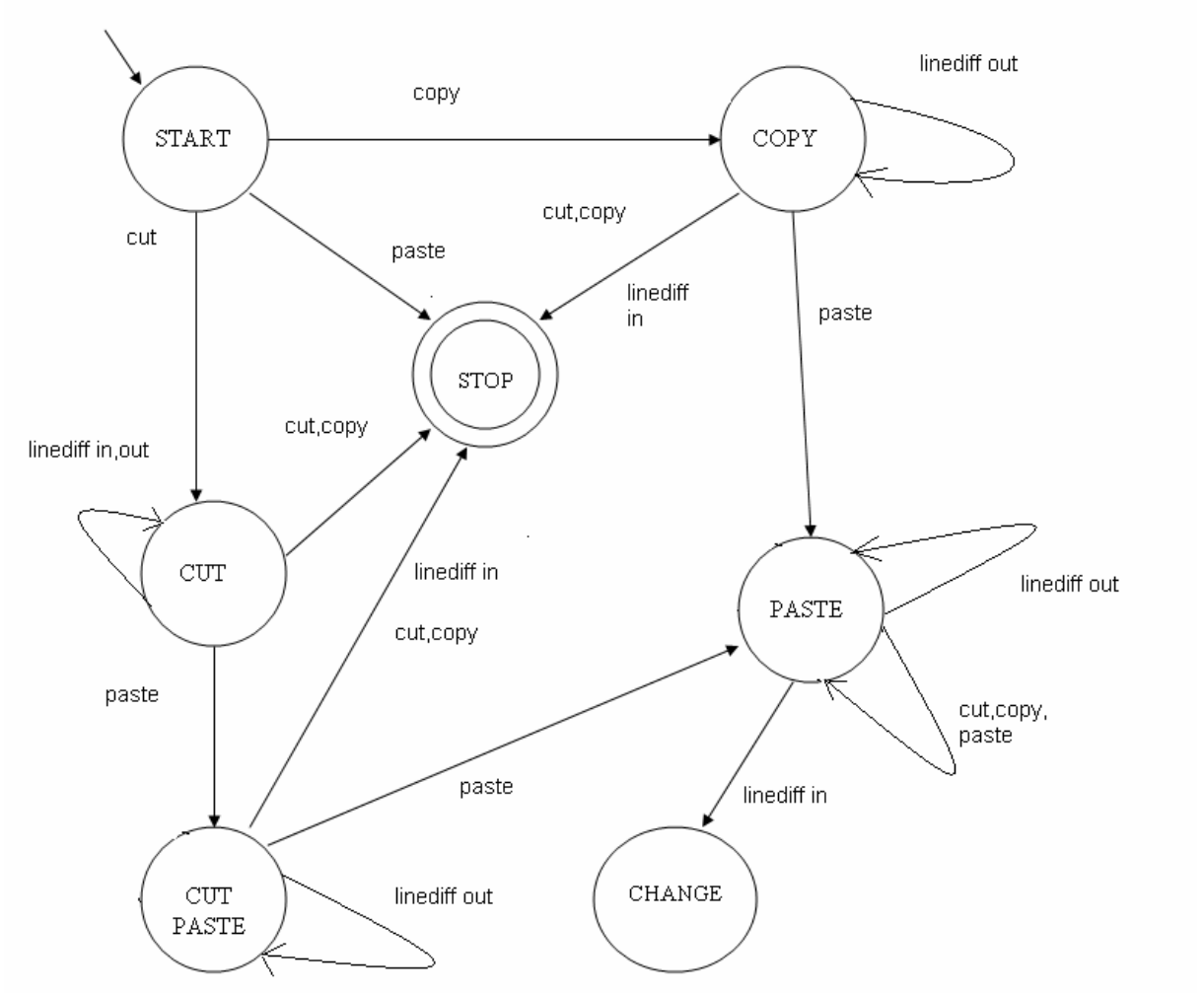


Abb.18: Erweitertes Zustandsdiagramm für das Verarbeiten von Textoperation-Ereignissen und Linediff-Ereignissen des CPC-Automaten.

Abbildung 18 zeigt, dass das Zustandsdiagramm um einen Zustand erweitert wurde. Der CHANGE-Zustand wurde dazugefügt. Wenn der Automat sich im CHANGE-Zustand befindet, wurde einer seiner in der Klonfamilie befindlichen Klone abgeändert. Die Aktivitäten die zu Zustandsübergängen führen sind um das *linediff in* und das *linediff out* erweitert worden. Wobei das *linediff in* die Bedeutung einer Zeilen-und Textveränderung innerhalb der Zeilen des beobachteten Klons darstellt. Hierbei müssen CPCWarning-Ereignisse gesendet werden. Das *linediff out* bedeutet, dass außerhalb der Zeilen des beobachteten Klons Programmkodeänderungen vorgenommen wurden.

Beispiel eines Durchlaufs durch den Zustandsgraphen in Abbildung 18:

Ich greife auf das vorige Beispiel für einige Zustandswechsel einer Klonfamilie zurück, gehe dieses Beispiel bis zum Schluss durch, erspare mir aber die bereits bekannten aufgeführten Datenausgaben nochmals anzugeben.

- Der CPC-Episodenerkenner befindet sich im Initialzustand START.

- Der Programmierer kopiert Programmkode. Und der Erkenner wechselt in den COPY-Zustand, da er das Textoperation-Ereignis Copy verarbeiten muss.

```
public class Test2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Nasenbär21");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär23");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");

        System.out.println("Nasenbär221");
        System.out.println("Nasenbär222");
        System.out.println("Nasenbär223");
        System.out.println("Nasenbär224");
        System.out.println("Nasenbär225");
    }
}
```

- Der kopierte Kode wird in einer anderen Datei an eine Stelle im Programmkode eingefügt und der CPC-Automat wechselt nachdem er das Textoperation-Ereignis Paste verarbeitet hat in den PASTE-Zustand.

```
public class Test1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub


        System.out.println("Nasenbär21");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär23");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");
    }
}
```

- Nun kopiert der Programmierer erneut ein Kodefragment. Es wird eine neue Klonfamilie erzeugt und der neu erzeugte CPC-Episodenerkenner, der sich zuvor im Initialzustand befand wechselt in den COPY-Zustand.

```java
public class Test2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Nasenbär21");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär23");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");

        System.out.println("Nasenbär221");
        System.out.println("Nasenbär222");
        System.out.println("Nasenbär223");
        System.out.println("Nasenbär224");
        System.out.println("Nasenbär225");
    }
}
```

Der neu erzeugte CPC-Episodenerkenner mit folgenden Ausgabe-Daten:

```
nr: 4
zustand: START


msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: copy
datei:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
code: System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
startzeile: 11
endzeile: 15
zeitstempel: Fri Dec 29 21:10:57 CET 2006

nr: 4
zustand: COPY
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
```

- Der Programmierer markiert die ersten drei Zeilen des Klons aus dem CPC-Episodenerkenner mit der Nummer 1 in der Datei Test1.java. Hierbei wird kein Ereignis erfasst und es wird auch kein Ereignis verarbeitet.

```
public class Test1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub


        System.out.println("Nasenbär21");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär23");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");
    }
}
```

- Nun fügt der Programmierer das kopierte Kodefragment in die markierte Stelle in Datei Test1.java ein. Er verändert damit zum einen den beobachteten Klon aus dem ersten Paste-Ereignis, somit muss der CPC-Episodenerkenner mit der Nummer 1 auf das daraus resultierende Linediff-Ereignis reagieren. Der Erkenner muss seinen beobachteten Klon aktualisieren. Zum anderen muss der CPC-Episodenerkenner mit der Nummer 4 auf das Textoperation-Ereignis Paste reagieren und einen neuen Klon in die Klonfamilie aufnehmen.

```
public class Test1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Nasenbär221");
        System.out.println("Nasenbär222");
        System.out.println("Nasenbär223");
        System.out.println("Nasenbär224");
        System.out.println("Nasenbär225");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");
    }
}
```

Ausgabe des CPC-Episodenerkenner mit der Nummer 1:

```
nr: 1
zustand: PASTE

filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");

msdt-typ: msdt.linediff.xsd
pakettyp: linediff
linedifftyp: changed
datei:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
zeitstempel: Fri Dec 29 21:11:15 CET 2006

aktueller event wird beobachtet: true
cpcwarning time: Fri Dec 29 21:11:15 CET 2006
event not deleted and valid
file:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
startline: 6 endline: 10     //Zeilenangaben entsprechen den Zeilen
                             // des Eclipse-Editors !
nr: 1
zustand: CHANGE
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär223");
       System.out.println("Nasenbär224");
            System.out.println("Nasenbär225");
            System.out.println("Nasenbär24");
            System.out.println("Nasenbär25");
```

Ausgabe des CPC-Automaten mit der Nummer 4:

```
nr: 4
zustand: COPY

filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
```

```
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");

msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: paste
datei:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
code: System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
startzeile: 9
endzeile: 13
zeitstempel: Fri Dec 29 21:11:15 CET 2006

nr: 4
zustand: PASTE
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 9
endline: 13
code: System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
```

Als Ergebnis kann notiert werden, dass die beiden CPC-Episodenerkenner die Ereignisse korrekt abgearbeitet haben. Die beiden Automaten befinden sich im für sie jeweiligen korrekten Zustand nach Verarbeitung der Ereignisse. Der CPC-Episodenerkenner Nummer 1 ist im CHANGE-Zustand, da ein Klon verändert wurde.

Der Programmkode des Klons aus CPC-Automat Nummer 1 hat vor Veränderung:

```
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
startline: 11
endline: 15
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");
```

Der Programmkode des Klons aus CPC-Episodenerkenner 1 nach Veränderung:

```
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär223");
       System.out.println("Nasenbär224");
            System.out.println("Nasenbär225");
            System.out.println("Nasenbär24");
            System.out.println("Nasenbär25");
```

Er hat die Aktualisierung ebenfalls korrekt abgearbeitet. Der CPC-Episodenerkenner mit der Nummer 4 befindet sich im PASTE-Zustand, da er auf das Copy-und Paste-Ereignis reagieren musste und dieses abarbeiten musste.

7.5 Umsetzung des CPC-Episoden-Zustandsautomaten

In diesem Kapitel soll ein Überblick und das Verständnis über die Softwarearchitektur des CopyPasteChangeEpisodeRecognizer, (kurz CPC-EpisodeRecognizer) gegeben werden. Im Folgenden werden die implementierten Klassen des CPC-EpisodeRecognizer vorgestellt.

Nach dem Festlegen der benötigten Zustände des CPC-EpisodeRecognizer, in Kapitel 7.4 erläutert, konnte mit der Implementierung begonnen werden. In Kapitel 7.3.4 habe ich bereits beschrieben, dass das Software-Design bezüglich des zu implementierenden Klonverfolgungsprogramms bereits implizit feststand.
Ich orientierte mich an den bereits implementierten Zustandsautomaten im Module ECG_EpisodeRecognizerIntermediateModule.

Bei dem CPC-EpisodeRecognizer mussten die sieben Zustände umgesetzt werden, die in Kapitel 7.4 beschrieben wurden. Zunächst soll das Vorgehensmodell vorgestellt werden, dass kurz beschreiben soll, wie der CPC-EpisodeRecognizer konkret arbeitet.

Generelles Modell:

1. Alle Aktionen beim Programmieren werden mit zwei Konzepten beschrieben:

   a) Generelle Textoperationen, die Textoperation-Ereignisse:

      - CUT
      - COPY
      - PASTE

   b) Tatsächliche Zeilenoperationen, die Linediff-Ereignisse. Sie repräsentieren eine Veränderung des Programmkodes

2. Ein CPC-EpisodeRecognizer muss prüfen, ob eine generelle Textoperation ausgewertet werden muss.
3. Ein CPC-EpisodeRecognizer muss prüfen, ob die Änderung eines beliebigen Programm-abschnitts ihn selbst, also einen seiner Klone betrifft.

4. Falls die Änderung eines Linediff-Ereignisses einen seiner Klone betrifft, werden die CPCWarning-Ereignisse für die betroffenen Klone gesendet.

Um dieses generelle Vorgehensmodell zu realisieren, wurden zwei Klassen implementiert. Zum einen die Klasse `CopyPasteChangeEpisodeRecognizer.java`, die das EpisodeRecognizer-Interface implementiert und den Zustandsautomaten realisiert. Zum anderen die Klasse `Clone.java`, die die beobachteten Programmstellen repräsentieren. In der Klasse Clone.java werden die Aktualisierungen bezüglich der Linediff-Ereignisse abgearbeitet.

Beide Klassen sind in das folgende Paket integriert:

`package org.electrocodeogram.module.intermediate.implementation.recognizers`

Clone.java

Die Klasse Clone.java beinhaltet alle benötigten Methoden, um Aktualisierungen auf den Clone-Objekten auszuführen. Ein Clone-Objekt repräsentiert eine konkrete Programmstelle. Um dieses abbilden zu können, benötigen sie die Informationen die eine Programmstelle ausmacht. Ein Clone-Objekt speichert somit den Kode, den Dateinamen in der sich dieser Kode befindet und die Start- und Endzeile.

Mit einer Boolean-Variable deleted steht die Information zur Verfügung, ob das Clone-Objekt gelöscht wurde. Die Boolean-Variable valid beschreibt, ob das Clone-Objekt beobachtet wird.

Alle Linediff-Ereignisse beziehen sich auf Veränderungen im Programmkode. Falls Zeilenveränderungen den Kode eines Clone-Objekts betreffen, müssen diese Aktualisierungen durchgeführt werden. Die Veränderungen werden zeilenweise registriert und durch das Linediff-Ereignis geliefert. Wegen der zeilenweisen Orientierung ist in der Klasse Clone.java eine Variable vLines vom Typ Vector vereinbart, in der die Kodezeilen des Clone-Objekts gespeichert sind. Mit der Methode buildCloneCodeLines(), aufgerufen im Konstruktor, wird der Kode des Clone-Objekts in Zeilen gespalten und in vLines eingefügt. Diese können nun, falls erforderlich, bequem manipuliert werden. Mit der Methode void calcCloneEndLineNumber()), die auch im Konstruktor aufgerufen wird, wird die Endzeile des Clone-Objekts berechnet, da bei Textoperation-Ereignissen die Endzeile nicht mitgeliefert wird.

Die Methode deleteLine(int linenumber) löscht eine Zeile mit der Zeilenummer linenumber. Wird über der Stelle des Klons eine Zeile gelöscht, dann werden Start- und Endzeile dekrementiert. Wenn innerhalb der Zeilen des Klons eine Zeile gelöscht wird, so wird die betreffende Zeile in vLines des Clone-Objekts ebenfalls gelöscht. Die Zeilennummern werden um eins dekrementiert.

Mit der Methode insertLine(int linenumber, String codeLine) wird eine Zeile in vLines des Clone-Objekts eingefügt, falls die eingefügte Zeile innerhalb der Zeilen des Klons eingefügt wurde. Falls die eingefügte Zeile über der Stelle des Klons eingefügt wurde, werden die Start- und Endzeilen inkrementiert.
Mit der Methode changeLine(int linenumber, String codeLine) wird eine Zeile des Clone-Objekts verändert. Veränderungen einer Kodezeile die nicht das Clone-Objekt betreffen, werden nicht verarbeitet, da aus dieser Veränderung keine Zeilenveränderungen

hervorgehen. Programmkodeveränderungen die keinen Klon betreffen sind nicht relevant und müssen somit nicht verarbeitet werden.

CopyPasteChangeEpisodeRecognizer.java (CPCEpisodeRecognizer)

Der CPCEpisodeRecognizer hat die Aufgabe die Ereignisse zu verarbeiten und die Klone in der Klonfamilie zu verwalten. Weiterhin wird in dieser Klasse die Berechnung des Unterschiedlichkeitsmaßes bezüglich der Klone in der Klonfamilie ausgeführt.

Die CPCEpisodeRecognizer-Klasse implementiert das EpisodeRecognizer-Interface, das in Kapitel 7.3.3 bereits vorgestellt wurde. Es muss daher Implementierungen für die Methoden

- isInInitialState()
- isInFinalState()
- analyse(ValidEventPacketpacketlongminDuration)

zur Verfügung stellen. Ferner muss ein Default-Konstruktor für die Erzeugung von CPCEpisodeRecognizer-Objekten bereitgestellt werden.

Die Methode isInInitialState() gibt ein True zurück, falls sich der CPCEpisodeRecognizer im START-Zustand befindet. Die Methode isInFinalState() gibt als Rückgabewert ein True zurück, falls er sich im STOP-Zustand befindet. Befindet er sich im STOP-Zustand, wird er von dem EpisodenManager, der alle erzeugten CPCEpisodeRecognizer-Objekte verwaltet, aus seiner Verwaltungsliste gestrichen.

Mit der Konstante private static final double STRING_MEASURE kann das Unterschiedlichkeitsmaß für den Stringvergleich justiert werden. Es drückt die erlaubte Unterschiedlichkeit zwischen einem String x und einem String y aus. Der Default-Wert ist bei 50 eingestellt. Will man die Unterschiedlichkeit zwischen den Klonen gering halten, so nimmt man kleinere Double-Werte. Das heißt, dass die Klone auch schneller aus der Beobachtung genommen werden. Wählt man einen größeren Wert, so stehen die Klone, da das Unterschiedlichkeitsmaß größer wird, länger unter Beobachtung.

Mit der Konstante private static final int STRING_LENGTH kann man die Länge der Klone bestimmen, die beobachtet werden sollen. Die Verfolgung der Klone wird somit mit dieser Konstante gesteuert. Wenn man bei der Verfolgung aus Analysegründen auch kleinere Klone zulassen möchte, dann weist man dieser Konstanten einen kleineren Integer-Wert zu. Falls die Granularität gröber bezüglich der Länge der Klone sein soll, dann weist man ihr einen höheren Integer-Wert zu. Der Default-Wert liegt hier bei 100.

Die Variable ArrayList<ValidEventPacket> validEventPacketList wird zur Speicherung der CPCWarning-Ereignisse genutzt. Die ArrayList ist der Rückgabewert an den Episoden-Manager.

In der Variable private Vector<Clone> CloneFamilyVector werden die zu beobachteten Klone verwaltet. Das ist die Struktur, die der Klonfamilie entspricht. Die Klone entsprechen Objekten der Klasse Clone.java. Für die Klasse Vector<E> habe ich mich entschieden, weil sie die Funktionalität anbietet, die ich für die Verwaltung der Klonfamilie benötige. Mit der Methode boolean add(E o) wird z.B. ein Element an das Ende der Liste eingefügt. Mit der Methode public E firstElement() wird das erste Element im Vector geliefert. Diese Methode

wird eingesetzt, wenn im CUT-Zustand oder im COPY-Zustand auf das erste Element im CloneFamilyVector zugegriffen wird. Die Methode public setElementAt(E obj, int index) wird im CUT-Zustand benötigt, wenn der ausgeschnittene Klon, der bereits im CloneFamilyVector enthalten ist, durch den eingefügten Klon ersetzt wird (siehe dazu Kapitel 6.3 S.27). Zu dem kann man den Vector bequem mit einem Iterator durchlaufen.

Mit dem Konstrukt

```
private enum CopyPasteChangeEpisodeState{START,COPY,CUT,CUTPASTE
                                                    PASTE,CHANGE,STOP}
```

werden die Zustände des CPCEpisodeRecognizer repräsentiert.

Die Variable private CopyPasteChangeEpisodeState STATE enthält den aktuellen Zustand des CPCEpisodeRecognizers.

In einer Switch-Anweisung sind die Zustandsübergänge und die Abarbeitung der Zustände umgesetzt. In den einzelnen Case-Einträgen erfolgt die Abarbeitung der Zustände.

Eine Besonderheit ist bei den Zuständen CUTPASTE und PASTE zu beachten. Wenn nach einer Paste-Aktivität, mit einem bestimmten Zeitstempel t, sich der CPCEpisodeRecognizer im CUTPASTE- oder PASTE-Zustand befindet, darf er auf darauf folgende Linediff-Ereignisse, die denselben Zeitstempel t besitzen, nicht reagieren. Das resultiert daraus, dass Programmkodeveränderungen nicht immer als ein einziges Linediff-Ereignis ausgedrückt werden. Vielmehr können die Abänderungen durchaus durch mehrere Linediff-Ereignisse abgebildet werden. Mehrere hintereinander folgende Linediff-Ereignisse, die genau der Paste-Aktivität entsprechen werden somit nicht berücksichtigt. Es würde sonst Kode doppelt in einem Clone-Objekt gespeichert werden.

Im Konstruktor public CopyPasteChangeEpisodeRecognizer() wird der CPCEpisodeRecognizer auf den Initialzustand START gesetzt. Nun ist der CPCEpisodeRecognizer bereit auf die gelesenen Ereignisse entsprechend zu reagieren. Im Konstruktor werden zusätzlich zu dem Setzen des Initialzustands Elemente für das XML-Dokument desCPCWarning-Ereignisses initialisiert.

Die Abarbeitung der gelesenen Signale wird in der Methode Collection<ValidEventPacket> analyse(ValidEventPacket packet,long minDuration) vollzogen. Der Rückgabewert dieser Methode ist entweder null oder eine Menge von CPCWarning-Ereignissen. Diese Ereignisse beziehen sich auf die beobachteten Clone-Objekte eines CPCEpisodeRecognizer-Objekts bezüglich eines Linediff-Ereignisses. In der analyse-Methode werden die Zustände abhängig von den Signalen und dem aktuellen Zustand entsprechend gesetzt.

Ist ein Linediff-Ereignis erkannt, werden aus dem Linediff-Ereignis BlockChange-Objekte mittels einer Parse-Methode erzeugt. Diese BlockChange-Objekte enthalten LineChange-Objekte. Die LineChange-Objekte enthalten wiederum die erforderlichen Zeilen, die die Programmkodeveränderungen ausmachen. Falls solch ein LineChange-Objekt ein beobachtetes Clone-Objekt betrifft, werden die Zeilenveränderungen durchgeführt. Drückt sich die Veränderung in Form von Zeilenverschiebungen aus, werden nur die Zeilennummern aktualisiert. Drückt sich die Veränderung in Form einer Textveränderung innerhalb des Clone-Objekts aus, so wird der Kode aktualisiert. Falls zusätzlich zur Kodeveränderung auch Zeilennummernanpassungen nötig sind, wird dieses ebenfalls durchgeführt.

Ist das LineChange-Objekt vom Typ Changed, wird die Methode boolean changeLine(int linenumber, String codeLine) aufgerufen, die die Aktualisierung der Zeile durchführt. Wenn das LineChange-Objekt vom Typ Inserted ist, wird die boolean insertLine(int linenumber, String codeLine) aufgerufen, die für das Einfügen der neuen Zeile verantwortlich ist. Falls das LineChange-Objekt vom Typ Deleted ist, so wird mittels deleteLine(int linenumber), die betreffende Zeile gelöscht. Alle drei Zeilenmanipulationsmethoden liefern ein True, falls die Änderung innerhalb der Zeilen eines Clone-Objekts stattfinden. Sie liefern False, wenn die Programmkodeänderung außerhalb der Zeilen eines Clone-Objekts stattfinden.

Mit der Methode void validateCloneFamily() wird nach einer Aktualisierung eines Clone-Objekts A geprüft, ob mindestens ein Clone-Objekt B in dem CloneFamilyVector existiert, so dass A verglichen mit B unter dem Vergleichsmaß liegt. Es wird in einer For-Schleife einmal durch alle Clone-Objekte des CloneFamilyVector iteriert. Alle Clone-Objekte werden außer mit sich selbst validiert.

```
boolean valid = false;
int size = CloneFamilyVector.size();
int j = 0;
for (int i = 0; i<size; i++) {
      valid = false;
      j = 0;
      while (j<size & !valid) {
        if (j!=i) {
          if (!isOverStringMeasure(i,j)) {
            CloneFamilyVector.get(i).setValid(valid=true);
          }
        }
        j++;
      }
      if(valid==false){
        CloneFamilyVector.get(i).setValid(false);
}
```

Mit der Methode isOverStringMeasure(int i, int j) wird in der If-Abfrage ermittelt, ob das Clone-Objekt mit dem Index i im CloneFamilyVector im Vergleich zu dem Clone-Objekt mit dem Index j unter dem Vergleichsmaß liegt. Falls das der Fall ist, wird das Boolean-Flag isValid des Clone-Objekts i auf True gesetzt, und es wird mit dem nächsten Clone-Objekt fortgefahren. Wenn das nicht der Fall ist, wird das Boolean-Flag auf False gesetzt.

Innerhalb der Methode isOverStringMeasure(int i, int j) wird ein Objekt der inneren Klasse private StringDifferMeasurement erzeugt, dass für die eigentliche Vergleichsberechnung zuständig ist. Das Objekt wird mit den Kodes der Clone-Objekten initialisiert, die mit Index i und Index j aus dem CloneFamilyVector geholt werden.

```
StringDifferMeasurement stringdiff =
        new StringDifferMeasurement(cloneCode, diffEvalCloneCode)
```

Die Methode int countNotDeletedClones() zählt die Anzahl der Clone-Objekte im CloneFamilyVector, die nicht gelöscht wurden. Diese Methode wird in der analyse-Methode aufgerufen, um zu überprüfen, ob die Anzahl der nicht gelöschten Clone-Objekte größer 1 ist. Das ist erforderlich, denn wenn die Anzahl der nicht gelöschten Clone-Objekte gleich 1 ist, existieren keine Clone-Objekte in dem CloneFamiliyVector, die in Beziehung zueinander stehen. Der Zustand des CPCEpisodeRecognizer wird auf STOP gesetzt und der EpisodeManager löscht dieses CPCEpisodeRecognizer-Objekt aus seiner Verwaltung.

Die Methode int countValidAndNotDeletedClones() wird in der analyse-Methode aufgerufen und ermittelt die Anzahl der noch beobachteten und nicht gelöschten Klone. Diese Methode wird genutzt, um vor dem Senden eines CPCWarning-Ereignisses festzustellen, ob die Anzahl der beobachteten und nicht gelöschten Clone-Objekte größer 1 ist. Wenn das der Fall ist, kann ein CPCWarning-Ereignis in die validEventPacketList eingefügt werden. Wenn das nicht der Fall ist, wird kein CPCWarning-Ereignis in die Liste eingefügt.

In der privaten Klasse StringDifferMeasurement wird der eigentliche Stringvergleichs-algorithmus durchgeführt. Der Stringvergleich wird mittels der Levensthein-Distanz berechnet. Die Levenshtein-Distanz ist ein nach dynamischer Programmierung verfahrender Algorithmus.

Algorithmus Levensthein-Distanz:

Gegeben zwei Strings s und t.

1 Schritt:

- Setze n auf die Länge des Strings s.
- Setze m auf die Länge des Strings t.
- Wenn n = 0 dann return m und exit.
- Wenn m = 0 dann return n und exit.
- Erzeuge eine Matrix d mit 0..n Zeilen und 0..m Spalten, mit i Laufvariable für die Zeilen und j als Laufvariable für die Spalten.

2 Schritt:

- Initialisiere die erste Zeile mit 0..n.
- Initialisiere die erste Spalte mit 0..m.

3.Schritt:

- Durchlaufe jedes Zeichen von s von i = 1 bis n.

4.Schritt:

- Durchlaufe jedes Zeichen von s von i = 1 bis n.

5.Schritt:

- Falls s[i] = t[j] , dann Kosten = 0.
- Falls s[i] != t[j], dann Kosten = 1.

6.Schritt:

- Setze die Zelle der Matrix d[i,j] auf das Minimum(d[i-1,,j] +1, d[I,j-1] + 1, d[i-1,j-1] + Kosten)

7.Schritt:

- Wenn die Schritte 3,4,5 und 6 komplett durch sind, kann die Levenshtein-Distanz in d[n,m] abgelesen werden.

Das StringDifferMeasurement-Objekt erhält im Konstruktor als Parameter die zwei zu vergleichenden Strings s und t und führt den Algorithmus auf diese Strings aus. Die Distanz, die am Ende der Berechnung in d[n,m] steht, wird nun verwendet um die prozentuale Unterschiedlichkeit des Strings s zu t zu ermitteln. In der Methode stringDifferMeasurement(int lengthFirst, int percentVal) wird dieser Wert berechnet.

Es folgen nun ein paar ausgewählte Testfälle, um einen Einblick in die Arbeitsweise des CPCEpisodeRecognizer zu gewinnen. Die restlichen dokumentierten Testfälle befinden sich auf der beiliegenden CD-ROM.

Ausgewählte Testfälle:

Achtung:

- Die Ausgabe, die nach jedem Bild den aktuellen Zustand des CPCEpisodeRecognizer wiedergibt, ist nur zu Testzwecken vorgesehen.

- Wichtige Informationen in der Ausgabe, z.B. Zustand des CPCEpisodeRecognizer oder ein Generieren eines CPCWarning-Ereignisses, werden in fetten Lettern hervorgehoben.

1) Kopieren und Einfügen eines Kodefragments.

Textoperation-Ereignis: Copy-Aktivität

```
public class Test2 {
    public static void main(String[] args) {

        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");

        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
    }
}
```

nr: 1
zustand: START

msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: copy
datei: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");

System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
startzeile: 5
endzeile: 9
zeitstempel: Thu Jan 11 07:02:11 CET 2007

nr: 1
zustand: COPY
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");

Textoperation-Ereignis: Paste-Aktivität

```
public class Test1 {

    public static void main(String[] args) {
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");

        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
    }
}
```

Der CPCEpisodeRecognizer wechselt in den PASTE-Zustand.

---

nr: 1
zustand: COPY

filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");

```
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
```

msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: paste
datei: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
code: System.out.println("Nasenbär2");

```
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
```

startzeile: 11
endzeile: 11
zeitstempel: Thu Jan 11 07:02:16 CET 2007

nr: 1
zustand: PASTE
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär2");

```
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
```

filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär2");

```
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
                    System.out.println("Nasenbär2");
```

Auf die Paste-Aktivität folgt das Linediff-Ereignis, dass genau die Veränderung des Programmkodes nach der Paste-Aktivität in Datei Test1.java ausmacht. Darauf darf der CPCEpisodeRecognizer nicht reagieren, da der Klon in die Klonfamilie eingefügt wurde. Falls der CPCEpisodeRecognizer in seiner Klonfamilie weitere Klone enthält, die Programmkode beobachten der nach dem eingefügten Kodefragment liegt, so müssen diese Klone und deren Zeilen aktualisiert werden.

Linediff-Ereignis nach Paste-Aktivität:

Der CPCEpisodeRecognizer verbleibt im PASTE-Zustand und er führt keine Aktualisierungen durch.

---

nr: 1
zustand: PASTE

filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false

isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");

msdt-typ: msdt.linediff.xsd
pakettyp: linediff
linedifftyp: changed
datei: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
zeitstempel: Thu Jan 11 07:02:16 CET 2007

eventCode:System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
diffevaleventCode: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
The length of the first string is : 172
--------=-------- second string is: 172
editieroperationen: 0
STRING_DIFFER_MEASUREMENT < 50.0%: 0.0
verschiedenheit: 0.0
eventCode:System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
diffevaleventCode: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
The length of the first string is : 172
--------=-------- second string is: 172
editieroperationen: 0
STRING_DIFFER_MEASUREMENT < 50.0%: 0.0
verschiedenheit: 0.0
nr: 1
zustand: PASTE
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");

System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");

Der CPCEpisodeRecognizer ist im PASTE-Zustand verblieben, ein korrektes Ergebnis.

2) Ausschneiden, Einfügen eines Kodefragments und Verändern des eingefügten Klons:

Textoperation-Ereignis: Cut-Aktivität

```
public class Test2 {
    public static void main(String[] args) {

        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");

        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");

    }
}
```

```
public class Test2 {
    public static void main(String[] args) {


        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär22");

    }
}
```

---

nr: 1
zustand: START

msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: cut
datei: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
startzeile: 5
endzeile: 9
zeitstempel: Thu Jan 11 06:32:18 CET 2007

nr: 1
zustand: CUT
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true

startline: 5
endline: 9
code:
```
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
```

Auf das anschließende Linediff-Ereignis, darf der CPCEpisodeRecognizer nicht reagieren, da das Linediff-Ereignis diese Cut-Aktivität ausdrückt. Falls der CPCEpisodeRecognizer in seiner Klonfamilie weitere Klone enthält, die Programmkode beobachten, der nach dem ausgeschnittenen Kodefragment liegt, so müssen diese Klone und deren Zeilen aktualisiert werden.

Linediff-Ereignis:

---

nr: 1
zustand: CUT

filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code:
```
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
```

msdt-typ: msdt.linediff.xsd
pakettyp: linediff
linedifftyp: inserted
datei: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
zeitstempel: Thu Jan 11 06:32:25 CET 2007

nr: 1
zustand: CUT
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code:
```
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
```

Textoperation-Ereignis: Paste-Aktivität

```
public class Test1 {

    public static void main(String[] args) {
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");

        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
    }
}
```

nr: 1
zustand: CUT

filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 9
code:
```
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
```

msdt-typ: msdt.textoperation.xsd
pakettyp: textoperation
aktivität: paste
datei: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
code:
```
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
```
startzeile: 11
endzeile: 11
zeitstempel: Thu Jan 11 06:32:26 CET 2007

nr: 1
zustand: CUTPASTE
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15

code: System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");
System.out.println("Nasenbär2");

Linediff-Ereignis:

```
public class Test1 {

    public static void main(String[] args) {
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär1");

        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär2");
        System.out.println("Nasenbär");
    }
}
```

Die letzte Zeile des ausgeschnittenen und eingefügten Kodefragments wird verändert. Der CPCEpisodeRecognizer muss in den STOP-Zustand wechseln, da er sich im CUTPASTE-Zustand befand und eine Veränderung innerhalb der Zeilen des beobachteten Klons vorgenommen wurde.

```
nr: 1
zustand: CUTPASTE

filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
code:        System.out.println("Nasenbär2");
             System.out.println("Nasenbär2");
             System.out.println("Nasenbär2");
             System.out.println("Nasenbär2");
             System.out.println("Nasenbär2");

msdt-typ: msdt.linediff.xsd
pakettyp: linediff
linedifftyp: changed
datei:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
zeitstempel: Thu Jan 11 06:32:35 CET 2007

nr: 1
zustand: STOP
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 11
endline: 15
```

```
code:        System.out.println("Nasenbär2");
       System.out.println("Nasenbär2");        System.out.println("Nasenbär2");
       System.out.println("Nasenbär2");        System.out.println("Nasenbär");
```

Auf die Ausgaben bei Copy-, Cut- und Paste-Aktivitäten werde ich nun verzichten, da in den Testfällen 1 und 2 gezeigt wurde, dass diese Fälle korrekt abgearbeitet werden.

3) Kopieren und Einfügen eines Klons. Löschen der ersten 5 Zeilen des Klons.

Textoperation-Ereignis: Copy-Aktivität:

```
public class Test2 {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Nasenbär21");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär23");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");

        System.out.println("Nasenbär221");
        System.out.println("Nasenbär222");
        System.out.println("Nasenbär223");
        System.out.println("Nasenbär224");
        System.out.println("Nasenbär225");

        System.out.println("Nasenbär2221");
        System.out.println("Nasenbär2222");
        System.out.println("Nasenbär2223");
        System.out.println("Nasenbär2224");
        System.out.println("Nasenbär2225");
    }
```

Textoperation-Ereignis: Paste-Aktivität:

```
public class Test1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Nasenbär21");
        System.out.println("Nasenbär22");
        System.out.println("Nasenbär23");
        System.out.println("Nasenbär24");
        System.out.println("Nasenbär25");

        System.out.println("Nasenbär221");
        System.out.println("Nasenbär222");
        System.out.println("Nasenbär223");
        System.out.println("Nasenbär224");
        System.out.println("Nasenbär225");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär12");
        System.out.println("Nasenbär13");
```

Das folgende Linediff-Ereignis wird ignoriert, da kein weiterer unter Beobachtung stehender Klon existiert, der nach dem eingefügten Programmkode steht.

Linediff-Ereignis:

---

nr: 1
zustand: PASTE

filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 15
code: System.out.println("Nasenbär21");
System.out.println("Nasenbär22");
System.out.println("Nasenbär23");
System.out.println("Nasenbär24");
System.out.println("Nasenbär25");

System.out.println("Nasenbär221");
System.out.println("Nasenbär222");
System.out.println("Nasenbär223");
System.out.println("Nasenbär224");
System.out.println("Nasenbär225");
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 9
endline: 19

```
code: System.out.println("Nasenbär21");
                System.out.println("Nasenbär22");
                System.out.println("Nasenbär23");
                System.out.println("Nasenbär24");
                System.out.println("Nasenbär25");

                System.out.println("Nasenbär221");
                System.out.println("Nasenbär222");
                System.out.println("Nasenbär223");
                System.out.println("Nasenbär224");
                System.out.println("Nasenbär225");

msdt-typ: msdt.linediff.xsd
pakettyp: linediff
linedifftyp: inserted
datei: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
zeitstempel: Fri Dec 29 08:11:08 CET 2006

eventCode:System.out.println("Nasenbär21");
                System.out.println("Nasenbär22");
                System.out.println("Nasenbär23");
                System.out.println("Nasenbär24");
                System.out.println("Nasenbär25");

                System.out.println("Nasenbär221");
                System.out.println("Nasenbär222");
                System.out.println("Nasenbär223");
                System.out.println("Nasenbär224");
                System.out.println("Nasenbär225");
diffevaleventCode: System.out.println("Nasenbär21");
                System.out.println("Nasenbär22");
                System.out.println("Nasenbär23");
                System.out.println("Nasenbär24");
                System.out.println("Nasenbär25");

                System.out.println("Nasenbär221");
                System.out.println("Nasenbär222");
                System.out.println("Nasenbär223");
                System.out.println("Nasenbär224");
                System.out.println("Nasenbär225");
The length of the first string is : 367
--------=-------- second string is: 367
editieroperationen: 0
STRING_DIFFER_MEASUREMENT < 50.0%: 0.0
verschiedenheit: 0.0
eventCode:System.out.println("Nasenbär21");
                System.out.println("Nasenbär22");
                System.out.println("Nasenbär23");
                System.out.println("Nasenbär24");
                System.out.println("Nasenbär25");

                System.out.println("Nasenbär221");
                System.out.println("Nasenbär222");
                System.out.println("Nasenbär223");
                System.out.println("Nasenbär224");
                System.out.println("Nasenbär225");
diffevaleventCode: System.out.println("Nasenbär21");
                System.out.println("Nasenbär22");
                System.out.println("Nasenbär23");
                System.out.println("Nasenbär24");
                System.out.println("Nasenbär25");

                System.out.println("Nasenbär221");
                System.out.println("Nasenbär222");
                System.out.println("Nasenbär223");
```

```
                    System.out.println("Nasenbär224");
                    System.out.println("Nasenbär225");
```
The length of the first string is : 367
--------=-------- second string is: 367
editieroperationen: 0
STRING_DIFFER_MEASUREMENT < 50.0%: 0.0
verschiedenheit: 0.0
nr: 1
zustand: PASTE
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 15
code:
```
System.out.println("Nasenbär21");
                    System.out.println("Nasenbär22");
                    System.out.println("Nasenbär23");
                    System.out.println("Nasenbär24");
                    System.out.println("Nasenbär25");

                    System.out.println("Nasenbär221");
                    System.out.println("Nasenbär222");
                    System.out.println("Nasenbär223");
                    System.out.println("Nasenbär224");
                    System.out.println("Nasenbär225");
```
filename: src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 9
endline: 19
code:
```
System.out.println("Nasenbär21");
                    System.out.println("Nasenbär22");
                    System.out.println("Nasenbär23");
                    System.out.println("Nasenbär24");
                    System.out.println("Nasenbär25");

                    System.out.println("Nasenbär221");
                    System.out.println("Nasenbär222");
                    System.out.println("Nasenbär223");
                    System.out.println("Nasenbär224");
                    System.out.println("Nasenbär225");
```

Linediff-Ereignis: Die ersten fünf Zeilen des beobachteten Klons werden gelöscht.

```
public class Test1 {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub


        System.out.println("Nasenbär221");
        System.out.println("Nasenbär222");
        System.out.println("Nasenbär223");
        System.out.println("Nasenbär224");
        System.out.println("Nasenbär225");
        System.out.println("Nasenbär1");
        System.out.println("Nasenbär12");
        System.out.println("Nasenbär13");
        System.out.println("Nasenbär14");
        System.out.println("Nasenbär15");
    }
}
```

```
nr: 1
zustand: PASTE

filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: true
startline: 5
endline: 15
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");

             System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: true
startline: 9
endline: 19
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");

             System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
```

```
msdt-typ: msdt.linediff.xsd
pakettyp: linediff
linedifftyp: deleted
datei:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
zeitstempel: Fri Dec 29 08:11:33 CET 2006

aktueller event wird beobachtet: true
cpcwarning time: Fri Dec 29 08:11:33 CET 2006
event not deleted and valid
file:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
startline: 6 endline: 16

eventCode:System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");

             System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
diffevaleventCode:                     System.out.println("Nasenbär221");
       System.out.println("Nasenbär222");
       System.out.println("Nasenbär223");
       System.out.println("Nasenbär224");
       System.out.println("Nasenbär225");
The length of the first string is : 367
--------=-------- second string is: 182
editieroperationen: 185
STRING_DIFFER_MEASUREMENT > 50.0%: 50.40871934604905
verschiedenheit: 50.40871934604905
eventCode:                      System.out.println("Nasenbär221");
       System.out.println("Nasenbär222");
       System.out.println("Nasenbär223");
       System.out.println("Nasenbär224");
       System.out.println("Nasenbär225");
diffevaleventCode: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");

             System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
The length of the first string is : 182
--------=-------- second string is: 367
editieroperationen: 185
STRING_DIFFER_MEASUREMENT > 50.0%: 101.64835164835165
verschiedenheit: 101.64835164835165
nr: 1
zustand: CHANGE
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test2.java
isDeleted: false
isValid: false
startline: 5
endline: 15
code: System.out.println("Nasenbär21");
             System.out.println("Nasenbär22");
             System.out.println("Nasenbär23");
             System.out.println("Nasenbär24");
             System.out.println("Nasenbär25");

             System.out.println("Nasenbär221");
             System.out.println("Nasenbär222");
             System.out.println("Nasenbär223");
             System.out.println("Nasenbär224");
             System.out.println("Nasenbär225");
```

```
filename:
src/org/electrocodeogram/module/intermediate/implementation/recognizers/Test1.java
isDeleted: false
isValid: false
startline: 9
endline: 14
code:                      System.out.println("Nasenbär221");
       System.out.println("Nasenbär222");
       System.out.println("Nasenbär223");
       System.out.println("Nasenbär224");
       System.out.println("Nasenbär225");
```

Nach dem Löschen der ersten fünf Zeilen, also das Abändern des Klons, wird ein CPCWarning-Ereignis generiert. Der Klon wird allerdings so sehr verändert, dass er keine Ähnlichkeit mit seinen Familienmitgliedern aufweist. Nach der Validierung der Klonfamilienmitglieder stehen beide Klone nicht mehr unter Beobachtung. Das Boolean-Flag valid wird bei beiden Klonen auf False gesetzt. Somit können sie erst wieder in einem CPCWarning-Ereignis berücksichtigt werden, wenn sie sich beide durch etwaige Änderungen wieder ähnlich werden.

8. Evaluation

In Kapitel 8 wird auf die Evaluation des CPCEpisodeRecognizer eingegangen. In Kapitel 8.1 werden die Umgebungsbedingungen beschrieben, die zum Erhalt der Testdaten führten. Die Testdaten wurden als Input vom CPCEpisodeRecognizer verarbeitet. In Kapitel 8.2 wird eine Beschreibung der Evaluation gegeben. Kapitel 8.3 präsentiert die Ergebnisse der Evaluation.

8.1 Umgebungsbedingungen

Um eine Evaluierung des CPCEpisodeRecognizer durchführen zu können, war es erforderlich Datenmaterial zu erzeugen, das der CPCEpisodeRecognizer verarbeiten soll. Um geeignetes Datenmaterial zu erzeugen, wurde einem Programmierer eine Programmieraufgabe vorgelegt, die er versuchen sollte zu lösen. Seine Programmier-aktivitäten wurden mittels ECG-Software protokolliert. Als Entwicklungsumgebung wurde Eclipse verwendet. Als Programmieraufgabe wurde die unten stehende Aufgabe Rubik's Cube Solver[8] ausgewählt. Dem Programmierer wurden keine besonderen Hinweise bezüglich der Programmieraufgabe gegeben. Bezüglich der Aufgabe ist zu sagen, dass sie ausgewählt wurde, um möglichst viele Copy-, Cut- und Paste-Ereignisse als protokollierte Daten zu erhalten.

Die Aktivitäten des Programmierers wurden per Videokamera und mit dem Aufzeichnungsprogramm mediaCamAV verfolgt. Das von mediaCamAV aufgezeichnete Video wurde zusammen mit den Daten, die während der Programmierung erzeugt wurden, zur Evaluation herangezogen. Die während der Programmierung protokollierten Daten, sind durch den ECGEclipseSensor1.1.4 erzeugt worden. Der ECGEclipseSensor registriert die Copy-, Cut- oder Paste-Aktivitäten und sie werden als Mikrosensordatentypen in eine Log-Datei geschrieben. Die Log-Datei wird mittels in die ECG-Software integriertem CPCEpisodeRecognizer gelesen und verarbeitet. Der CPCEpisodeRecognizer erzeugt beim

analysieren der Log-Datei CPCWarning-Ereignisse, die gefiltert werden und in eine Datei *out_cpcwarning.log* geschrieben werden.

Folgende Daten wurden während der Programmierung der Aufgabe erzeugt:

- Die protokollierte Datenmenge der Cut-, Copy- und Paste-Aktivitäten umfasst eine Größe von 16,2 MB.

- Das aufgezeichnete Video hat eine Größe von 374 MB.

- Die Programmierdauer betrug 5 Stunden, startete um 11:18 Uhr und endete um 16.07 Uhr.

- Die Datei out_cpcwarning.log ist 299 KB groß.

Programmieraufgabe:

ACM South Central Region
2002 Programming Contest

Problem #2: Rubik's Cube Solver

Introduction

The Rubik's Cube was invented by Enro Rubik in 1974. It is a 3-dimensional puzzle made up of 26 smaller cubes. Each smaller cube has from one to three sides exposed for a total of 54 exposed sides. Each of these sides is assigned one of six colors, and each color is assigned to precisely nine exposed sides. The cube is manipulated by rotating any side of the cube by 90 degrees. It is considered solved when each side of the Rubik's Cube is entirely covered by one of the six colors.

You are a researcher at the Rubik's University and are working on an algorithm to solve a Rubik's Cube in the least possible number of moves from any starting position. To aid in your research, you need a program that will read in various Rubik's Cube configurations, perform operations on those configurations, and determine if the resulting cube is solved.

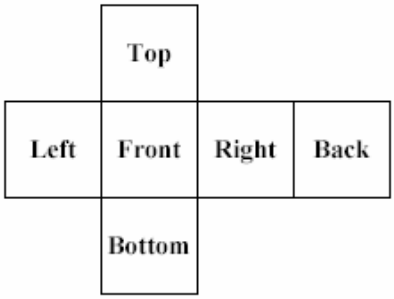
Input

Input to this problem will consist of a starting configuration for the Rubik's Cube and then one or more operations to perform on the cube.

The input configuration will look like:

```
      G W O
      G R R
      G B R
B R B R G Y W W W Y G O
G W B O G B Y B O W Y O
W R Y O Y B R Y R G O O
      B R Y
      B O W
      G Y W
```

Which follows this layout:



Each character in this grid represents the color of the piece of the cube. There is one space between each character in the grid and possibly many spaces before the first character on a line. The grid represents the cube as is if it were unfolded and flattened out. Each group of 9 characters (3 x 3 array) represents one side of the grid. The top of the cube is represented by the first 3 lines of input. The next 3 lines of input represent the left, front, right, and back sides in that order. The last 3 lines represent the bottom of the cube.

Following the initial configuration will be 1 more operations to perform on the cube. There are 12 possible operations that can be performed, each being a 90 degree rotation of one of the cube's 'faces' of 9 smaller cubes. Note that this results in the movement of 20 colored squares (8 on the face being rotated and 12 on the sides of the smaller cubes that make up that face). All 12 possible operations are listed in the table below with a description of how to perform that operation.

Output

For each data set, there will be exactly one line of output. If the cube is solved, then the word "Yes" is output on a line. If the cube remains unsolved, then the word "No" is output on a line.

Sample Input

```
START
      O O O
      O O O
      O G W
W W W Y B G R B B G G G
B Y W Y B G R G G W W R
B Y Y B O O B W W Y Y R
      R Y Y
      B R R
      G R R
front right
top right
left left
bottom right
END
START
      Y B B
      O G B
      G R B
W W B R Y Y R G G O O O
R O O G B R G R O Y W W
Y G Y O R G W B G O Y W
      B B R
      W Y Y
      R W W
back left
top right
left right
right right
front left
bottom right
right left
END
ENDOFINPUT
```

Sample Output

```
Yes
No
```

8.2 Evaluationsbeschreibung

Die Datenmenge der registrierten Aktivitäten des Programmierers ist sehr groß. Ich habe mich deswegen entschieden einen bestimmten Zeitabschnitt zu analysieren, da die erzeugte Log-Datei zu groß ist, um sie insgesamt auszuwerten.
Es wird *nicht* untersucht, ob trotz einer Änderung eines in einer Klonfamilie beobachteten Klons ein CPCWarning-Ereignis nicht generiert wurde.

- Es wird der Zeitabschnitt von 15:12 Uhr bis 16:07 Uhr analysiert.

- Für aufeinander folgende CPCWarning-Ereignisse eines CPCEpisodeRecognizer, wird untersucht, weshalb diese Ereignisse generiert wurden.

8.3 Evaluationssergebnis

Während der Programmierung der Aufgabe Rubik's Cube Solver wurden folgende Ereignisse und ihre Anzahl protokolliert:

- Textoperation-Ereignisse: 326

- Linediff-Ereignisse: 1139

Die Schalter STRING_MEASURE bzw. STRING_LENGTH im CPCEpisodeRecognizer wurden auf die Default-Werte 50 bzw.100 gesetzt. Zunächst ist zu sagen, dass der Programmierer während er die Aufgabe bearbeitete in der gesamten Zeit nicht eine semantische Schablone (Seite 14) geklont hat. Vielmehr hat er die gesamte Zeit über zwar strukturelle Klone erzeugt, aber diese waren allesamt syntaktische Klone (Seite 14), die wieder verwendbare syntaktische Elemente beinhalteten. Der Programmierer hatte die gesamte Zeit über an seiner Lösung für das Problem gefeilt, kam aber nicht dazu seinen Kode auszuführen.

Der Programmierer kopierte z.B. folgenden Kode:

```
private void front_left() {
    // face moves
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    // side moves
}
private void front_right() {

}
```

Die markierte Zeile ist ein typisches Beispiel für eine syntaktische Schablone.

Nun wird diese Schablone zweimal nach der kopierten Textstelle eingefügt:

```
private void front_left() {
    // face moves
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    // side moves
}
private void front_right() {

}
```

Nach dem Einfügen der syntaktischen Schablonen, benutzt er wiederum diese, um neue syntaktische Klone zu erzeugen:

```
private void front_left() {
    // Three front face lines have to be moved.
    // For each line three positions have to be changed.
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    // Four side lines have to be moved: top, left, bottom, right
}
private void front_right() {

}
```

```
private void front_left() {
    // Three front face lines have to be moved.
    // For each line three positions have to be changed.
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    // Four side lines have to be moved: top, left, bottom, right
}
private void front_right() {

}
```

Nach dem Einfügen, ändert er die syntaktischen Schablonen nach seinen Wünschen ab:

```
private void front_left() {
    // Three front face lines have to be moved.
    // For each line three positions have to be changed.
    Front.top_face_line.setElementAt(Front.top_face_line.get(2),0);
    Front.top_face_line.setElementAt(Front.mid_face_line.get(2),1);
    Front.top_face_line.setElementAt(Front.bottom_face_line.get(2),2);
    Front.mid_face_line.setElementAt(Front.top_face_line.get(1),0);
    Front.mid_face_line.setElementAt(Front.bottom_face_line.get(1),2);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),0);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),1);
    Front.bottom_face_line.setElementAt(Front.bottom_face_line.get(2),2);
    // Four side lines have to be moved: top, left, bottom, right
}
private void front_right() {

}
```

Das Kopieren dieser syntaktischen Schablonen zieht sich fort und der Programmierer fügt nach diesen Änderungen den gesamten Inhalt, der sich im Methodenrumpf der Methode private void front_left() befindet, in die übrigen Methoden dieser Klasse ein.

Um die Evaluation durchführen zu können mussten die mittels ECGEclipseSensor protokollierten Daten durch den CPCEpisodeRecognizer verarbeitet werden.

Der Input der protokollierten Daten und das Auswerten des CPCEpisodeRecognizer lieferte bei STRING_LENGTH = 100 und STRING_MEASURE = 50 folgendes Ergebnis:

	04.01.2007
Date	04.01.2007
Begin	11:12:54
End	16:07:07
msdt.user.xsd	
msdt.focus.xsd	
msdt.window.xsd	88
msdt.partactive.xsd	
msdt.fileactive.xsd	
msdt.run.xsd	
msdt.dialog.xsd	78
msdt.runactive.xsd	
msdt.system.xsd	
msdt.resource.xsd	281
msdt.antactive.xsd	
msdt.codestatus.xsd	2
msdt.rundebug.xsd	
msdt.linediff.xsd	1139
msdt.exactcodechange.xsd	1745
msdt.testrun.xsd	
msdt.linediffbase.xsd	2
msdt.windowactive.xsd	
msdt.cpcwarning.xsd	1094
msdt.codechange.xsd	1056
msdt.test.xsd	
msdt.editor.xsd	77
msdt.antrun.xsd	
msdt.part.xsd	15
msdt.textoperation.xsd	326
Events total	5903

Bei 1139 Linediff-Ereignissen, wurden 1094 CPCWarning-Ereignisse generiert. Die CPCWarning-Ereignisse werden bei einer Änderung innerhalb der Zeilen eines Klons in einer Klonfamilie erzeugt. Jedes einzelne CPCWarning-Ereignis repräsentiert dann jeden übrigen beobachteten Klon in seiner Klonfamilie. Für den Klon, der gerade verändert wurde, wird kein CPCWarning-Ereignis erzeugt. Die große Anzahl der CPCWarning-Ereignisse kommt dadurch zustande, dass in einer Klonfamilie mindestens zwei oder auch mehrere Klone enthalten sind. Bei jedem Linediff-Ereignis, dass den Klon betrifft, wird ein CPCWarning-Ereignis für die übrigen Klone seiner Klonfamilie erzeugt. Wenn eine Klonfamilie fünf beobachtete Klone enthält, dann werden bei einer Änderung eines Klons vier CPCWarning-Ereignisse generiert.

Es wird nun im Folgenden der Ausschnitt der CPCWarning-Datei analysiert. Da der Programmierer keine typischen semantischen Schablonen erzeugt hat, werde ich bei der Analyse einzig auf die Änderungen von Klonen und die generierten CPCWarning-Ereignisse eingehen.

1) CPCWarning-Ereignis von 15:12:30 bis 15:12:45:

Klonfamilie: {Clone(Cube.java,31,36), Clone (Cube.java,352,357)}

15:12:30 bis 15:12:45 Clone(Cube.java,351,356) wird verändert
Warnung: Cube.java 31,36 = Der Klon in der Datei Cube.java von Zeile 31 bis 36.

Bis 15:12:45 lag die Unterschiedlichkeit des Clone(Cube.java,351,356) zu seinem Klonfamilienmitglied unter 50%. Nach diesem Zeitpunkt wurden die 50% nicht unterschritten.

Der Clone(Cube.java,351,356) wird verändert von:

```
Top_Buffer.clone();
Left = Left_Buffer.clone();
Front = Front_Buffer.clone();
Right = Right_Buffer.clone();
Back = Back_Buffer.clone();
Bottom = Bottom_Buffer.clone();
```

Zu:

```
// Top
// Left
// Front
// Right
// Back = Back_Buffer.clone();
// Bottom = Bottom_Buffer.clone();
```

-> Veränderung liegt über dem STRING_MEASURE und somit wird der Clone(Cube.java,351,356) nicht mehr beobachtet.

Es werden alle Änderungen des Klons bis zum Zeitpunkt des Überschreitens des Unterschiedlichkeitsmaßes aktualisiert.

2) CPCWarning-Ereignisse von 15:18:56 bis 15:19:38

Klonfamilie: {Clone(Face.java,18,19),Clone(Face.java,20,21),Clone(Face.java,22,23)}

15:18:56 Clone(Face.java,20,21) wird verändert
Warnung: Face.java18,19 und Face.java 22,23

15:19:09 Clone(Face.java,22,23) wird verändert.
Warnung: Face.java 18, 19 und Face.java 20, 21

15:19:35 Clone(Face.java,18,19) wird verändert.
Warnung an: Face.java 20,21 und Face.java 22,23

15:19:38 Clone(Face.java,18,19) wird verändert.
Warnung: Face.java 20,21 und Face.java 22,23

Clone(Face.java,18,19) wird verändert von:

```
if ( !top_face_line.get(0).equals(top_face_line.get(1)) ||
     !top_face_line.get(0).equals(top_face_line.get(2)) ||
     !top_face_line.get(1).equals(top_face_line.get(2)) )
return false;
```

zu:

```
if (!top_face_line.get(0).equalsIgnoreCase(top_face_line.get(1)) ||
    !top_face_line.get(0).equals(top_face_line.get(2)) ||
    !top_face_line.get(1).equals(top_face_line.get(2)) )
return false;
```

Clone(Face.java,20,21) wird verändert von:

```
if ( !top_face_line.get(0).equals(top_face_line.get(1)) ||
     !top_face_line.get(0).equals(top_face_line.get(2)) ||
     !top_face_line.get(1).equals(top_face_line.get(2)) )
return false;
```

zu:

```
if ( !mid_face_line.get(0).equals(mid_face_line.get(1)) ||
     !mid_face_line.get(0).equals(mid_face_line.get(2)) ||
     !mid_face_line.get(1).equals(mid_face_line.get(2)) )
return false;
```

Clone(Face.java,22,23) wird verändert von:

```
if ( !top_face_line.get(0).equals(top_face_line.get(1)) ||
     !top_face_line.get(0).equals(top_face_line.get(2)) ||
     !top_face_line.get(1).equals(top_face_line.get(2)) )
return false;
```

zu:

```
if ( !bottom_face_line.get(0).equals(bottom_face_line.get(1)) ||
     !bottom_face_line.get(0).equals(bottom_face_line.get(2)) ||
     !bottom_face_line.get(1).equals(bottom_face_line.get(2)) )
return false;
```

3) CPCWarning-Ereignisse von 15:31:13 bis 15:31:41

Klonfamilie: {Clone(Cube.java,372,372),Clone(Cube.java,376,376)}

=> Diese Klonfamilie wird im Verlauf der Veränderungsaktivitäten ungültig.

Kode des Clone(Cube.java,372,372) vor Änderung:

```
input_line.trim().startsWith("front") && !input_line.trim().startsWith("left") &&
!input_line.trim().startsWith("right") && !input_line.trim().startsWith("back") &&
!input_line.trim().startsWith("top") && !input_line.trim().startsWith("bottom")
```

Kode des Clone (Cube.java,376,376)} vor Änderung:

```
input_line.trim().startsWith("front") && !input_line.trim().startsWith("left") &&
!input_line.trim().startsWith("right") && !input_line.trim().startsWith("back") &&
!input_line.trim().startsWith("top") && !input_line.trim().startsWith("bottom")
```

```
String input_line = null;
try {
    while ((input_line = br.readLine()) != null && !input_line.equalsIgnore
        if (input_line.equalsIgnoreCase("START")) {
            while ((input_line = br.readLine()) != null && !input_line.trim
                // Top

            }
            while (input_line.trim())
        }
        if (input_line.equalsIgnoreCase("END")) {
            System.out.println(solved());
        }
    }
```

In der rot markierten Zeile der While-Schleife wird der Methodenaufruf input_line.trim() durch ein kopierten Kodeschnipsel ersetzt.

```
try {
    while ((input_line = br.readLine()) != null && !input_line.equalsIgnoreCase("endofinput")) {
        if (input_line.equalsIgnoreCase("START")) {
            while ((input_line = br.readLine()) != null && !input_line.trim().startsWith("front") && !
                // Top

            }
            while (input_line.trim().startsWith("front") && !input_line.trim().startsWith("left") && !
        }
        if (input_line.equalsIgnoreCase("END")) {
            System.out.println(solved());
        }
    }
} catch (Exception e) {
    e.printStackTrace();
```

Die rot unterstrichenen Kodezeilen, sind die beiden Elemente der Klonfamilie. Nun ändert der Programmierer den Klammerausdruck ab.

```
try {
    while ((input_line = br.readLine()) != null && !input_line.equalsIgnoreCase("endofinput")) {
        if (input_line.equalsIgnoreCase("START")) {
            while ((input_line = br.readLine()) != null && !input_line.trim().startsWith("front") && !
                // Top

            }
            while (input_line.trim().startsWith("front") && !input_line.trim().startsWith("left") && !
        }
        if (input_line.equalsIgnoreCase("END")) {
            System.out.println(solved());
        }
    }
} catch (Exception e) {
    e.printStackTrace();
```

Er ersetzt den mit grünem Kreis markierten logischen UND-Ausdruck durch einen logischen ODER-Ausdruck.

```
String input_line = null;
try {
    while ((input_line = br.readLine()) != null && !input_line.equalsIgnoreCase("endofinput"))
        if (input_line.equalsIgnoreCase("START")) {
            while ((input_line = br.readLine()) != null && !input_line.trim().startsWith("fron
                // Top

            }
            while (input_line.trim().startsWith("front") || input_line.trim().startsWith("left
        }
        if (input_line.equalsIgnoreCase("END")) {
            System.out.println(solved());
        }
    }
} catch (Exception e) {
    e.printStackTrace();
```

Der grün markierte Bereich weist auf die Änderungen nach dem Einfügen in die Klammer der Bedingungsabfrage der While-Schleife hin. Der logische ODER-Ausdruck und vereinzelte Zeichen in der Zeile wurden abgeändet.

Kode des Clone (Cube.java,372,372) nach Änderung:

```
input_line.trim().startsWith("front") && !input_line.trim().startsWith("left") &&
!input_line.trim().startsWith("right") && !input_line.trim().startsWith("back") &&
!input_line.trim().startsWith("top") && !input_line.trim().startsWith("bottom")
```

Kode des Clone (Cube.java,376,376)} nach Änderung:

```
while (input_line.trim().startsWith("front") ||
input_line.trim().startsWith("left") && !input_line.trim().startsWith("right") &&
!input_line.trim().startsWith("back") && !input_line.trim().startsWith("top") &&
!input_line.trim().startsWith("bottom"))
```

Das zusätzliche Schlüsselwort **while** vor der Bedingungsklammer, wurde nicht durch den Programmierer beabsichtigt eingefügt, um den Klon zu Ändern. Vielmehr resultiert diese zusätzliche Einfügung des Schlüsselworts while daraus, dass die Klonbeobachtung, die Aktualisierung der Klone und das Linediff-Ereignis zeilenweise ausgerichtet sind. Der Index in der Zeile wurde bisher nicht berücksichtigt. Aus diesem Grund wurde nach einem Linediff-Ereignis die Veränderung aktualisiert. Dabei wurde das vor dem Klon stehende while mit der Klammer zu dem Kode hinzugefügt. Somit ist der Kode des Klons ungültig. Daraus folgt, dass die Klonfamilie ungültig und somit nicht mehr zu verfolgen ist, da die Klonfamilie dann aus nur einem gültigen Element besteht.

4) CPCWarning-Ereignisse von 15:50:14 bis 15:59:05

Klonfamilie: {Clone(Face.java,24,26),Clone(Face.java,30,32),Clone(Face.java,36,38)}

15:50:14 Clone(Face.java,30,32) wird verändert
Warnung: Face.java 24,26 und Face.java 36,38

15:50:15 Clone(Face.java, 30,32) wird verändert
Warnung: Face.java 24,26 und Face.java 36,38

15:50:17 Clone(Face.java, 30,32) wird verändert
Warnung: Face.java 24,26 und Face.java 36,38

15:50:21 Clone(Face.java, 36,38) wird verändert
Warnung: Face.java 24,26 und Face.java 30,32

15:50:23 Clone(Face.java, 36,38) wird verändert
Warnung: Face.java 24,26 und Face.java 30,32

15:50:24 Clone(Face.java, 36,38) wird verändert
Warnung: Face.java 24,26 und Face.java 30,32

15:56:35 Clone(Face.java, 24,26) wird verändert
Warnung: Face.java 30,32 und Face.java 36,38

15:57:25 Clone(Face.java, 24,26) wird verändert
Warnung: Face.java 30,32 und Face.java 36,38

15:57:28 Clone(Face.java, 24,26) wird verändert
Warnung: Face.java 30,32 und Face.java 36,38
.
. kontinuierliche Abänderungen, die alle aktualisiert werden
.
15:59:05 Clone(Face.java, 30,32) wird verändert
Warnung: Face.java 24,26 und Face.java 36,38

15:59:05 Clone(Face.java, 36,38) wird verändert
Warnung: Face.java 24,26 und Face.java 30,32

Sobald eine Änderung den Kode eines Klons betrifft, werden diese Aktualisierungen korrekt durchgeführt. Der Klon besteht nunmehr am Ende der Änderungsaktivitäten aus dem aktualisierten Kode.

Kode des Clone (Cube.java,24,26) vor Änderung:

```
top_face_line.setElementAt(firstColor,0);
top_face_line.setElementAt(secondColor,1);
top_face_line.setElementAt(thirdColor,2);
```

Kode des Clone (Cube.java,24,26) nach Änderung:

```
top_face_line.setElementAt(new Character(firstColor.trim().charAt(0)),0);
top_face_line.setElementAt(new Character(secondColor.trim().charAt(0)),1);
top_face_line.setElementAt(new Character(thirdColor.trim().charAt(0)),2);
```

Kode des Clone (Cube.java,30,32) vor Änderung:

```
top_face_line.setElementAt(firstColor,0);
top_face_line.setElementAt(secondColor,1);
top_face_line.setElementAt(thirdColor,2);
```

Kode des Clone (Cube.java,30,32) nach Änderung:

```
mid_face_line.setElementAt(new Character(firstColor.trim().charAt(0)),0);
mid_face_line.setElementAt(new Character(secondColor.trim().charAt(0)),1);
mid_face_line.setElementAt(new Character(thirdColor.trim().charAt(0)),2);
```

Kode des Clone (Cube.java,36,38) vor Änderung:

```
top_face_line.setElementAt(firstColor,0);
top_face_line.setElementAt(secondColor,1);
top_face_line.setElementAt(thirdColor,2);
```

Kode des Clone (Cube.java,36,38) nach Änderung:

```
bottom_face_line.setElementAt(new Character(firstColor.trim().charAt(0)),0);
bottom_face_line.setElementAt(new Character(secondColor.trim().charAt(0)),1);
bottom_face_line.setElementAt(new Character(thirdColor.trim().charAt(0)),2);
```

5) CPCWarning-Ereignisse von 16:03:55 bis 16:03:59

Die CPCWarning-Ereignisse beziehen sich auf die Klonfamilie:

{Clone(Cube.java,372,372),Clone(Cube.java,376,376)}

Diese Klonfamilie aus 3) ist wie bereits aus den oben erläuterten Gründen ungültig. Somit ist es nicht korrekt diese Klone weiter zu beobachten.

Durchlauf des Log-Files mit unterschiedlichem STRING_MEASURE und STRING_LENGTH:

Einstellungen STRING_MEASURE = 50 und STRING_LENGTH = 200

	04.01.2007
Date	04.01.2007
Begin	11:12:54
End	16:07:07
msdt.user.xsd	
msdt.focus.xsd	
msdt.window.xsd	88
msdt.partactive.xsd	
msdt.fileactive.xsd	
msdt.run.xsd	
msdt.dialog.xsd	78
msdt.runactive.xsd	
msdt.system.xsd	
msdt.resource.xsd	281
msdt.antactive.xsd	
msdt.codestatus.xsd	2
msdt.rundebug.xsd	
msdt.linediff.xsd	1139
msdt.exactcodechange.xsd	1745
msdt.testrun.xsd	
msdt.linediffbase.xsd	2
msdt.windowactive.xsd	
msdt.cpcwarning.xsd	786
msdt.codechange.xsd	1056
msdt.test.xsd	
msdt.editor.xsd	77
msdt.antrun.xsd	
msdt.part.xsd	15
msdt.textoperation.xsd	326
Events total	5595

Die Statistik-Tabelle zeigt, dass je größer die String-Länge gewählt wird, desto mehr verringert sich die Anzahl der beobachteten Klone. Die Anzahl der CPCWarning-Ereignisse gibt einen Rückschluss darauf. Das ist auch so zu erwarten, denn die Länge der erzeugten Klone erreicht irgendwann eine Grenze. Kürzere Klone werden häufiger erzeugt als längere Klone. Das zeigte auch die Studie von Kim und ihren Kollegen. Somit wird eine kleinere Menge der CPCWarning-Ereignisse generiert.

Einstellungen STRING_MEASURE = 50 und STRING_LENGTH = 500

	04.01.2007
Date	04.01.2007
Begin	11:12:54
End	16:07:07
msdt.user.xsd	
msdt.focus.xsd	
msdt.window.xsd	88
msdt.partactive.xsd	
msdt.fileactive.xsd	
msdt.run.xsd	
msdt.dialog.xsd	78
msdt.runactive.xsd	
msdt.system.xsd	
msdt.resource.xsd	281
msdt.antactive.xsd	
msdt.codestatus.xsd	2
msdt.rundebug.xsd	
msdt.linediff.xsd	1139
msdt.exactcodechange.xsd	1745
msdt.testrun.xsd	
msdt.linediffbase.xsd	2
msdt.windowactive.xsd	
msdt.cpcwarning.xsd	768
msdt.codechange.xsd	1056
msdt.test.xsd	
msdt.editor.xsd	77
msdt.antrun.xsd	
msdt.part.xsd	15
msdt.textoperation.xsd	326
Events total	5577

Das Resultat dieses Durchlaufes ist ebenfalls deutlich. Bei einer String-Länge von 500 verringert sich die Menge der CPCWarning-Ereignisse weiterhin von 786 auf 768. Das war auch so zu erwarten. Die Anzahl der Klone die den Wert STRING_LENGTH = 500 übersteigen, um in die Beobachtung aufgenommen zu werden, verringert sich.

Einstellungen STRING_MEASURE = 75 und STRING_LENGTH = 100

	04.01.2007
Date	04.01.2007
Begin	11:12:54
End	16:07:07
msdt.user.xsd	
msdt.focus.xsd	
msdt.window.xsd	88
msdt.partactive.xsd	
msdt.fileactive.xsd	
msdt.run.xsd	
msdt.dialog.xsd	78
msdt.runactive.xsd	
msdt.system.xsd	
msdt.resource.xsd	281
msdt.antactive.xsd	
msdt.codestatus.xsd	2
msdt.rundebug.xsd	
msdt.linediff.xsd	1139
msdt.exactcodechange.xsd	1745
msdt.testrun.xsd	
msdt.linediffbase.xsd	2
msdt.windowactive.xsd	
msdt.cpcwarning.xsd	1179
msdt.codechange.xsd	1056
msdt.test.xsd	
msdt.editor.xsd	77
msdt.antrun.xsd	
msdt.part.xsd	15
msdt.textoperation.xsd	326
Events total	5988

Die CPCWarning-Ereignisse steigen von 1094 bei STRING_MEASURE = 50 und STRING_LENGTH = 100 auf 1179 bei STRING_MEASURE = 75 und STRING_LENGTH = 100. Dieses Ergebnis lässt sich folgendermaßen interpretieren. Die in die Beobachtung aufgenommenen Klone werden länger beobachtet, da das Maß STRING_MEASURE von 50 auf den Wert 75 angehoben wurde. Es wird damit eine größere Anzahl der Veränderungen an den Klonen zugelassen. So kommt es, dass hierbei auch eine größere Anzahl der generierten CPCWarning-Ereignisse erzeugt wurden.

Einstellungen STRING_MEASURE = 30 und STRING_LENGTH = 100

	04.01.2007
Date	04.01.2007
Begin	11:12:54
End	16:07:07
msdt.user.xsd	
msdt.focus.xsd	
msdt.window.xsd	88
msdt.partactive.xsd	
msdt.fileactive.xsd	
msdt.run.xsd	
msdt.dialog.xsd	78
msdt.runactive.xsd	
msdt.system.xsd	
msdt.resource.xsd	281
msdt.antactive.xsd	
msdt.codestatus.xsd	2
msdt.rundebug.xsd	
msdt.linediff.xsd	1139
msdt.exactcodechange.xsd	1745
msdt.testrun.xsd	
msdt.linediffbase.xsd	2
msdt.windowactive.xsd	
msdt.cpcwarning.xsd	1078
msdt.codechange.xsd	1056
msdt.test.xsd	
msdt.editor.xsd	77
msdt.antrun.xsd	
msdt.part.xsd	15
msdt.textoperation.xsd	326
Events total	5887

Bei den Einstellungen STRING_MEASURE = 30 und STRING_LENGTH = 100 ist die Anzahl der CPCWarning-Ereignisse von STRING_MEASURE = 50 und STRING_LENGTH = 100 von 1094 leicht auf 1078 geschrumpft. Die Klone werden kürzer beobachtet und fallen somit schneller aus der Beobachtung.

9. Fazit und Ausblick

Die Arbeit hat gezeigt das der CPCEpisodeRecognizer, also das Klonverfolgungsprogramm, Klonfamilien und ihre Klone beobachten kann. Die Änderungen die im Programmkode während der Programmierung vollzogen werden, werden ebenfalls in den Klonen entsprechend aktualisiert, falls es den Kode der Klone betrifft. Die CPCWarning-Ereignisse werden generiert, sobald ein beobachteter Klon verändert wird. Der CPCEpisodeRecognizer ist eine robuste Software, die die Funktionalität bietet um Klone zu beobachten.
Es werden allerdings beide Arten der Schablonen beobachtet. Bei den syntaktischen Schablonen hat das die Evaluation deutlich gezeigt. Zu den semantischen Schablonen ist zu sagen, dass sie ebenfalls beobachtet werden. Das ist so, da Klone, die über den Längenkonstantenwert STRING_LENGTH hinausgehen unter Beobachtung stehen. Möchte man eine Gruppe der Schablonen von dem CPCEpisodeRecognizer konzentriert beobachten lassen, reicht die Länge eines Strings aber als alleiniges Auswahlkriterium nicht mehr aus, um diese Klone in die Beobachtung zu nehmen. Es müsste eine Logik dazugefügt werden, die erkennt, um welche Art von Schablone es sich handelt. So wird die Anzahl der beobachteten Klone vermutlich nicht so groß werden. Damit wird vermieden, dass alle möglichen Arten von kopierten und eingefügten Kodeschnipsel, die über das Längenmaß gehen, unter Beobachtung genommen werden. Daraus folgt, dass dann auch eine kleinere Menge von CPCWarning-Ereignissen generiert werden würden.

Die CPCWarning-Ereignisse werden nach jeder Veränderung eines Klons einer Klonfamilie generiert. Es hat sich gezeigt, dass im Verlauf einer Programmiersession eine große Menge dieser CPCWarning-Ereignisse erzeugt werden. Für eine spätere visuelle Warnung an den Programmierer ist das eine zu große Anzahl, um sich gezielt mit diesen Warnungen auseinanderzusetzen. Für eventuelle statistische Auswertungen ist diese Menge der CPCWarning-Ereignisse trotzdem interessant. Man könnte anhand der Anzahl der CPCWarning-Ereignisse erkennen, ob während der Entwicklung einer Software eine große Anzahl an Klonen erzeugt worden sind. Man kann mit diesem Wert Vergleiche zwischen Softwareprodukten ziehen, vorausgesetzt die Softwareprodukte wurden mit dem um die ECG-Software erweiterten Eclipse programmiert.

Die zeilenweise Orientierung bei den Programmkodeveränderungen ist zu grob. Es muss der Index mitberücksichtigt werden, damit die Klone eine exaktere Stellenidentität besitzen. So werden keine Klone zu ungültigen Klonfamilienelementen, die nicht mehr beobachtet werden dürfen.

Bis jetzt ist der CPCEpisodeRecognizer so implementiert, dass bei jeder neuen Cut-Aktivität und Copy-Aktivität neue Klonfamilien erzeugt werden. Das heißt es gibt eine Redundanz, falls aus einer Klonfamilie ein Klon kopiert und dann eingefügt wird. Hier wird eine neue Klonfamilie erzeugt. Der neu im Programmkode eingefügte Klon, der durch eine Copy-Aktivität eines Klons aus einer bereits bestehenden Klonfamilie erzeugt wurde, müsste in diese Klonfamilie als neues Klonelement eingefügt werden. Eine denkbare Erweiterung des CPCEpisodeRecognizer wäre, dass ein kopierter Klon einer Klonfamilie beim Einfügen in den Programmkode in eben genau diese Klonfamilie als neuer Klon eingefügt wird.

Das Feld der Episodenerkennung des Mikroprozesses ist noch wenig erforscht. Der CPCEpisodeRecognizer ist eine gute Basis, um auf kommende Forschungen bezüglich der Episodenerkennung aufbauen zu können. Es ist eine stabile und robuste Software, die es im Allgemeinen möglich macht Klone zu verfolgen. Mit den vorgeschlagenen Erweiterungen werden Klone eingegrenzt zudem detaillierter beobachtet und detaillierter erfasst.

10. Referenzen

[1] Eclipse, *Entwicklungsumgebung* (IDE): http://www.eclipse.org

[2] ECG,Diplomarbeit von Frank Schlesinger:
http://projects.mi.fu-berlin.de/w/bin/view/SE/ThesisEclipsemonitor

[3] Jekutsch, Sebastian: *Mikroprozess der Softwareentwicklung* siehe:
http://projects.mi.fu-berlin.de/w/bin/view/SE/MicroprocessHome .

[4] Fowler, Martin Refactoring: *Improving the Design of Existing Code*, Addison-Wesley, 2000.

[5] Hunt, A. und D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*, Addison-Wesley , 2000.

[6] M. Kim, L. Bergman, T.A. Lau und D. Notkin: *An Ethongraphic Study of Copy and Paste Programming Practices in OOPL.* In *Proc. of the Int'l Symposium on Empirical Software Engineering*, pages 83-92, 2004.

[7] Dan Gusfield: *Algorithms on Strings, Trees and Sequences*, Computer Science and Computational Biology, Camebridge University Press 1997.

[8]Programmieraufgabe: *Problem #2: Rubik's Cube Solver*, ACM South Central Region Programming Contest 2002.

11. Anhang

Inhalt CD-ROM:

- Im Ordner Software befinden sich die Klassen:

    - CopyPasteChangeEpisodeRecognizer.java

    - Clone.java

    - msdt.cpcwarning.xsd

- Im Ordner Testfälle befinden sich:

    - Die Testfälle, jeweils mit dem Video, den erzeugten CPCEpisodeRecognizer-Textdateien als Ausgabe und das out_cpcwarning.log.

- Im Ordner Evaluierung befinden sich:

    - Die Log-Datei mit den Textoperation-Ereignissen aus der Programmier-Session.

- Die CPCEpisodeRecognizer, also die Klonfamilien, die beim Abarbeiten der Log-Datei erzeugt wurden.

- Das out_cpcwarning.log, mit den gemeldeten CPCWarning-Ereignissen.