Proc. 2nd Workshop "Automatische Bewertung von Programmieraufgaben",
Wolfenbüttel, 2015. http://ceur-ws.org/Vol-1496/paper7.pdf

Zur Einschätzung von Programmierfähigkeiten — *Jedem* Programmieranfänger über die Schultern schauen

Bastian Schulten[1], Frank Höppner[2]

Abstract: Am besten kann man den Kenntnisstand und die Fähigkeiten eines Studierenden in einer Programmierveranstaltung einschätzen, wenn man ihm während des Programmierens über die Schulter schaut. Man erkennt, ob systematisch vorgegangen wird, welche Sprachkonstrukte sicher sitzen oder nicht, wieviel Trial & Error im Spiel ist, etc. Da die Anzahl der Teilnehmer die der Betreuer aber um ein Vielfaches übersteigt, ist eine solche direkte Beobachtung schwer möglich. Wir beschreiben vorbereitende Arbeiten und erste Teilergebnisse für die Erprobung eines Werkzeugs, das eine *nachträgliche* Analyse der Entstehung einer Lösung erlaubt, um über (manuelle oder semiautomatische) Auswertungen Rückschlüsse auf Wissenslücken und damit hilfreiche Aufgaben oder sinnvolle Stoff-Wiederholungen zu erlangen.

1 Einleitung

Die Bewertung von Programmierfähigkeiten ist mühsam und aufwendig, weil eine Vielzahl von Aspekten zu berücksichtigen ist. Als harte Fakten muss das Endprodukt natürlich die geforderte Funktionalität erfüllen, was durch Unit-Tests sehr gut bewerkstelligt werden kann. Die Funktionalität mag als alleiniges Kriterium einer finalen Abgabe oder eines Abschlussprojektes geeignet sein, aber im Laufe des Semesters – in der Zeit, in der ein Studierender noch lernt, noch Fehler machen darf – ist es möglicherweise nicht das einzige und informativste Kriterium.

Gerade bei großen Studierendengruppen wird oftmals eine Bearbeitung von Aufgaben in Zweiergruppen angeboten. Um die Relevanz der Bearbeitung von Aufgaben zu betonen, werden Programmieraufgaben oft als Teilleistungen ausgeschrieben. Das kann wiederum schwächere Studierende dazu verleiten, sich stärkeren anzuschließen, um sich die Teilleistungen auf jeden Fall zu sichern. Im schlimmsten Fall führt das dazu, dass die schwächeren Studierenden die Erstellung der Aufgaben weitestmöglich anderen überlassen, um kein Risiko einzugehen. Dabei berauben sie sich selbst der wichtigen Praxis.

Kriterien für die Abnahme einer Aufgabe sind oft Funktionalität und Form, wie es bspw. durch JUnit und Checkstyle automatisiert geprüft werden kann. So hilfreich diese Werkzeuge sind, sie tragen bewusst auch zu einer *Begradigung* oder *Normierung* der Lösungen bei, so dass sich die Abgaben mehr ähneln, als es der Kenntnisstand der Studierenden vermuten ließe: Ob die Lösung in einem Zug runterprogrammiert oder das Ende einer langen Trial & Error Odyssee darstellt, kann man dem Quelltext am Ende meist nicht mehr ansehen. Wenn ein Kommilitone zum Schluss geholfen hat, helfen danach weder Checkstyle

[1] Ostfalia Hochschule, Fakultät Informatik, Am Exer 2, Wolfenbüttel, b.schulten@ostfalia.de
[2] Ostfalia Hochschule, Fakultät Informatik, Am Exer 2, Wolfenbüttel, f.hoeppner@ostfalia.de

noch JUnit dabei, den Studierenden das nötige Feedback über den eigenen Leistungsstand zurückzuspiegeln.

Unsere Hypothese ist, dass man den Kenntnisstand eines Studierenden am besten beurteilen kann, indem man ihm zuschaut: Wie geht er die Lösung an? Hat er einen Plan oder hangelt er sich von Fehlermeldung zu Fehlermeldung? Folgt er blind den Vorschlägen der Entwicklungsumgebung? Wählt er eine sinnvolle Reihenfolge der Lösung von Teilproblemen? Diese Beobachtung verbietet sich im Regelfall aus Aufwandsgründen (abgesehen von Prüfungssituationen). Diese Arbeit ist ein erster, vorläufiger Schritt hin zu einem toolgestützten *über die Schulter schauen*. Die Grundidee ist die Aufzeichnung der Entstehungsgeschichte eines Programms, so dass eine nachgelagerte Beurteilung der Herangehensweise möglich ist.

2 Verwandte Arbeiten

2.1 Evolution eines (Quell-) Textes

Angeregt zu diesem Ansatz haben Arbeiten über die Beurteilung der Fähigkeiten von angehenden Schriftstellern: Gab es einen Entwurf? Wieviele Überarbeitungen gab es? Wurde der Text *in einem Stück* geschrieben oder musste ständig vor und zurückgesprungen werden? (Vgl. [Ma87, PR96]) Als Unterstützung für eine solche Beurteilung wurden in diesen Disziplinen diverse Editoren geschaffen, die jeden Tastendruck bei der Entstehung eines Aufsatzes dokumentieren (zum Beispiel[3] ScriptLog), um diese Fragen im Nachhinein zur Beurteilung heranziehen zu können. In den letzten Jahren gab es aber auch im Software-Engineering verstärkt Bestrebungen, die Zwischenstände aus Versionsverwaltungen (wie SVN, CVS) automatisch zu analysieren, um den wahren Umfang von Änderungen in Code-Repositories abzuschätzen [CCP07], Bug-Tracking Einträge und Change-Log Einträge miteinander zu verknüpfen [BB09] oder die Entwicklung einer Codebasis über der Zeit zu visualisieren [Ha13]. Eine noch feiner granulierte Analyse der Entstehung eines Programms erscheint für industrielle Anwendungen weniger sinnvoll, könnte im Bereich der Programmierausbildung – analog zu der Beurteilung von Autoren – aber ein bisher unausgeschöpftes Potential bergen.

2.2 Programmverständnis

Ebenfalls relevant ist die Disziplin des *Program Understanding* [WY96], bei der es u.a. darum geht, die Anweisungen eines Programms den verschiedenen Teilzielen des Autors zuzuordnen[4], um so ihre Bedeutung für die Gesamtlösung zu verstehen. Die Vorstellung ist, dass der Entwickler einen Plan zur Umsetzung verfolgt, der aus verschiedenen Teilplänen besteht. Die Rekonstruktion des ursprünglichen Plans anhand des Quelltexts kann

[3] Siehe bspw. http://www.writingpro.eu/logging_programs.php

[4] z.B. besteht das Teilziel, einen Zähler zu implementieren, aus den Schritten Deklaration, Initialisierung und Inkrementierung, die jeweils an der richtigen Stelle (vor/innerhalb einer Schleife) stehen müssen.

auf Grundlage einer manuell gepflegten Hierarchie von Plänen/Aktionen inkl. zahlreicher Nebenbedingungen gelingen [QYW98]. Diese Aktionen repräsentieren das Wissen von Programmierern (Erfahrung, gute Programmierpraxis), das aber bei Programmier-Anfängern meist erst noch entwickelt werden muss. Insofern ist der Erfolg dieser Techniken bei Programmierlaboren fraglich. Andere Arbeiten verzichten auf ein *tieferes* Verständnis und prognostizieren auf Basis von einfachen Kennzahlen der statischen Analyse (Anzahl Blöcke, Schleifen, ...) z.B. die Art des implementierten Sortier-Algorithmus [TMK08].

2.3 Verständnisprobleme von Programmieranfängern

In der Literatur sind viele Fehlkonzepte von Programmieranfängern dokumentiert [BA01, KP10]. Wir listen einige typische Probleme von Programmieranfängern auf, die vor allem aber auch aus der eigenen Erfahrung im Bereich der Java-Programmierausbildung bekannt sind:

1. Richtung der Zuweisung: Wird die Anweisung `a=b;` als Zuweisung von links nach rechts, rechts nach links, oder als Vergleich aufgefasst?
2. Ausführungsreihenfolge von Anweisungen der `for`-Schleife: Falsche Vorstellungen über den Zeitpunkt, zu dem die Bestandteile der Anweisung ausgeführt werden, provozieren ein Trial & Error Vorgehen.
3. Statische Elemente: Wurde zuerst dem prozeduralen Paradigma gefolgt (Java: statische Methoden und Attribute), fällt es den Studierenden beim Wechsel zum objektorientierten Paradigma oft schwer, diesen statischen Elementen zu entsagen. Eine ungebrochene Verwendung des Schlüsselwortes `static` ist ein Indiz, dass das nötige Verständnis noch nicht erworben wurde.
4. Copy & Paste Bann: Meistens wissen die Studierenden, dass sie Copy & Paste nicht verwenden sollen, machen es aber dennoch.
5. Test-Handling: Wird das Potential ggf. bereitgestellter Tests genutzt? Stellen die Studierenden sicher, dass sie bei der fortwährenden Erweiterung ihrer Lösung keine grünen Testfälle mehr *verlieren*?
6. Weitsicht: Eine Aufgabe mittlerer Größe, für die JUnit-Tests vorliegen, ist typischerweise in Form von Interfaces vorspezifiziert. Die Studierenden kennen also ein Spektrum an Klassen und Methoden, die sie implementieren müssen. Wie gehen Studierende diese Aufgabe an? Erkennen sie, was aufeinander aufbaut, und fangen am Ende der Abhängigkeitskette an? u.v.a.m.

3 Eine Beispielsituation

Zur Illustration einer typischen Labor-Situation betrachten wir die stark vereinfachte Mini-Aufgabe aus Abb. 1, die einem Studenten zu einem Zeitpunkt gestellt wird, zu dem Objekte eingeführt wurden. Davor wurde Java nur prozedural behandelt (statische Attribute,

statische Methoden). Daher bereitet es einigen Studierenden erfahrungsgemäß Schwierigkeiten, sich vom Schlüsselwort `static` loszusagen, wenn sie die Konzepte dahinter noch nicht verstanden haben. Ist das Konzept angekommen, ist die Aufgabe trivial und ein entsprechender Test (analog zum Inhalt von `main()`) bestätigt das richtige Ergebnis 5.

Aufgabe: Ein X soll eine ganze Zahl speichern, die über get() abgefragt und über set(int) neu gesetzt werden kann. In der main()-Methode legen Sie zwei X-Objekte an, setzen ihren Wert auf 2 bzw. 3, addieren den einen zum anderen Wert und geben die Summe auf dem Bildschirm aus.

```
1  public class X {
2    public static void main(String[] args) {
3      X x1 = new X();
4      X x2 = new X();
5      x1.set(2);
6      x2.set(3);
7      x1.add(x2);
8      System.out.println(x1.get());
9    }
10 }
```

Abb. 1: Ein sehr einfaches Beispielproblem.

```
1  public class X {
2    static int i;
3    static int get() { return i; }
4    static void set(int ai) { i=ai; }
5    static void add(X x) { i+=x.i; }
6
7    public static void main(String[] args) {
8      X x1 = new X();
9      X x2 = new X();
10     x1.set(2);
11     x2.set(3);
12     x1.add(x2);
13     System.out.println(x1.get());
14   }
15 }
```

Abb. 2: Ein Zwischenstand.

```
1  public class X {
2    static int i;
3    static int get() { return i; }
4    static void set(int ai) { i=ai; }
5    static void add(X x) { i+=X.i; }
6
7    public static void main(String[] args) {
8      X x1 = new X();
9      X x2 = new X();
10     X.set(2);
11     X.set(3);
12     X.add(x2);
13     System.out.println(X.get());
14   }
15 }
```

Abb. 3: Eine Sackgasse.

Beginnt der Student aber wie gewohnt mit statischen Attributen, zeigt Abb. 2 einen möglichen Zwischenstand, der aber natürlich das falsche Ergebnis (nämlich 6) liefert. Durch die Ausgabe oder den Unit-Test weiß der Student, dass etwas nicht stimmt, aber was unternimmt er? Ein (leider) typisches Verhalten von Studierenden ist es, sich hier (unreflektiert) Ratschläge von der Entwicklungsumgebung zu holen. Eclipse macht mehrere Vorschläge. Einer ist *change access to static* für die Zeilen 10-13 aus Abb. 1. Folgt der Student konsistent diesem Vorschlag (er könnte potentiell auch inkonsistent verschiedenen Vorschlägen folgen), führt ihn der Weg zu Abb. 3. Eine letzte verbleibende Warnung gibt es dort für Zeile 8: *the value of local variable x1 is not used.* Diese Meldung sollte ihm zu denken geben (Wie kann ich zwei Zahlen addieren, wenn eine davon gar nicht benutzt wird?), resultiert aber möglicherweise nur in einem *Rückbau* aller bisher vorgenommenen Änderungen. Der zweite Ratschlag, den Eclipse beim Stand von Abb. 2 macht, ist *remove*

static modifier of set(). Folgt er wieder konsistent demselben Vorschlag bei allen Warnungen, führt ihn dies zu der richtigen Lösung, ohne sich seinem eigenen Verständnisproblem je gestellt zu haben.

Die erhaltene Version liefert das gewünschte Ergebnis, ein etwaiger Unit-Test ebenso. Vom Endergebnis her ist es nicht von der Lösung eines Studenten, der die Problematik durchdrungen hat, zu unterscheiden. Die wichtige Information über die Programmierfähigkeit trägt *der Weg zum Ziel*, der üblicherweise nicht bekannt ist.

4 Lösungsvorschlag

4.1 Ansatz

Als Lösungsansatz für das skizzierte Problem wird hier eine Beobachtung der Studierenden während der Programmentwicklung vorgeschlagen. Die Erprobung im Rahmen einer größeren Studierendengruppe steht noch bevor, bisher wurden nur im kleineren Rahmen Entstehungsprotokolle erhoben. Die (geplanten) Elemente sind im Einzelnen:

• Automatische Protokollierung der Entstehungsgeschichte der Lösung durch ein Plug-In in der Entwicklungsumgebung. Dies ermöglicht eine Playback-Funktion, so dass die Entstehungsgeschichte im Nachhinein (zusammen mit dem Student oder nur durch den Betreuer) betrachtet werden kann. Das Replay muss nicht in Echtzeit erfolgen und ist damit zeitsparend, aber das eigentliche Ziel ist eine (semi-) automatische Analyse der Aufzeichnungen.

• Eine klare Kommunikation an die Studierenden, dass die aufgezeichneten Informationen in keiner Weise für die Bewertung herangezogen werden, dafür gibt es Klausuren und/oder gesonderte Aufgaben/Projekte. Das Ziel ist es, Verständnisprobleme oder Schwächen direkt und zielgerichtet im Labor oder der begleitenden Veranstaltung ansprechen zu können, *bevor* es zu einer Klausur oder einem finalen Abschlussprojekt kommt. Dieses Ziel sollte im Sinne der Studierenden sein, so dass sie sich freiwillig für eine Teilnahme entscheiden. Die Protokollierung wird nicht zur Pflicht erhoben (ist abschaltbar).

• Im Idealfall liefern Indikatoren, die aus den Protokollen abgeleitet wurden, dem Betreuer ein Indiz über problematische Abschnitte der Lösung, so dass eine gemeinsame Reflektion gezielt angestoßen werden kann (und nicht erst nach Verständnisproblemen *gesucht* werden muss). Häufig auftretende Probleme können dem Dozenten helfen, in der begleitenden Vorlesung problematischen Stoff zu wiederholen und Hinweise auf weitere Aufgaben (in der gleichen Richtung) zu liefern.

4.2 Verarbeitung

Die einfachste Form der Verarbeitung der Protokolle ist der Einsatz einer Playback-Funktion. Sie ermöglicht die schrittweise Rekonstruktion, ist aber weiterhin zeitintensiv.

Für die automatische Auswertung wird nach jeder Änderung (mit Hilfe der Java Compiler API) der aktuelle Quelltext und/oder der Syntaxbaum erzeugt, um Unterschiede in Quelltextzeilen und/oder Knoten des Syntaxbaums zu identifizieren. Der Vorteil der lückenlosen Protokollierung ist, dass sich die Quelltexte bzw. Syntaxbäume von Schritt zu Schritt nur minimal unterscheiden und somit eine Beurteilung der (Art der) Änderung erleichtert wird (vgl. ChangeDistiller [Fl07]). Quelltextzeilen und/oder Syntaxbaum-Knoten verschiedener Evolutionsschritte werden einander zugeordnet und mit Änderungszählern ausgestattet. Dabei liefert die Analyse auf Basis des Syntaxbaums klare Vorteile: (1) Die Änderung kann präziser lokalisiert werden (z.B. Abbruchbedingung einer `for`-Schleife), (2) Formatierungen oder Kommentare beeinflussen zwar die Position im Quelltext, aber nicht die Lage des Baumknotens.

Außerdem können die einzelnen Änderungsereignisse zu abstrakteren Aktionen gebündelt werden, etwa *Bearbeitung der Methode f()*, die in der Zeit verortet werden (Beginn / Ende der Bearbeitung). Entsprechende Zeiträume lassen sich für die Erfüllung einzelner Tests oder die Compilierfähigkeit von Klassen generieren.

4.3 Präsentation und Nutzung

Sind die Protokolle verarbeitet, können sie auf verschiedene Weise aufbereitet werden, um Einblicke in den Kenntnisstand Einzelner oder der Gruppe zu geben:

```
1:public class X {
1:   private int value;
1:   public int get() {
1:      return value;
1:   }
2:   public void set(int value) {
1:      this.value = value;
1:   }
1:   public void add(X x) {
1:      this.value += x.get();
1:   }
1:   public static void main(String[] args) {
1:      X x1 = new X();
1:      X x2 = new X();
1:      x1.set(2);
1:      x2.set(3);
1:      x1.add(x2);
1:      System.out.println(x1.get());
1:   }
1:}
```

Abb. 4: Lösung eines Studenten mit den nötigen Kenntnissen.

```
1:public class X {
3:   int i;
4:   void set(int ai) {
1:      i = ai;
1:   }
3:   int get() {
1:      return i;
1:   }
3:   void add(X x) {
3:      i += x.i;
1:   }
1:   public static void main(String[] args) {
1:      X x1 = new X();
1:      X x2 = new X();
3:      x1.set(2);
3:      x2.set(3);
3:      x1.add(x2);
3:      System.out.println(x1.get());
1:   }
1:}
```

Abb. 5: Lösung bei static-Schwierigkeiten. (gelb: mind. 2x, orange: mind. 3x modifiziert)

• Indikatoren – In ersten Tests hat sich der Eindruck ergeben, dass die Anzahl der compilierbaren Zwischenschritte ein Indikator für den Reifegrad der Programmierfähigkeit sein könnte. Wer noch unsicher ist, wird sich von compilierbarem Zwischenstand zu compilierbarem Zwischenstand bewegen – sei es, weil sich nur kleine Schritte zugetraut werden, oder weil den Fehlermarkierungen am Editor sehr viel Aufmerksamkeit gezollt wird. Je größer die eigene Sicherheit und je weiter die Gedanken schon vorauseilen, desto eher werden spontan weitere Aspekte berücksichtigt, noch bevor die letzte Anweisungen syntaktisch korrekt aufgeschrieben wurde. Viele weitere Indikatoren sind denkbar, wie etwa die Lokalität von Änderungen (Fokus bei der Bearbeitung).

• HeatMap – Verständnisprobleme (wie die Probleme 1-3 des Abschnitts 2.3) führen dazu, dass die Studierenden für eine funktionierende Lösung (Unit-Tests erfüllt) bestimmte Stellen wieder und wieder anfassen, bis *es richtig läuft*. Die Änderungshäufigkeit kann im Code farbcodiert dargestellt werden, so dass Regionen hoher Änderungen sofort als *Hot Spot* erkannt werden können. Damit sind die Stellen, die schwer gefallen sind, leicht zu erkennen. Für die vom finalen Quelltext her nicht unterscheidbaren Beispiellösungen aus Kap. 3 ergibt sich (bei der aktuell zeilenbasierten Änderungszählung) ein klarer Unterschied in Abb. 4-5. (Erfolgt die Änderungszählung und Farbcodierung nicht zeilenweise sondern pro Syntaxbaum-Element wird noch deutlicher markiert, dass es eine große Unsicherheit bei den Modifikatoren gab.)

• Bearbeitungsmuster – Die Änderungshistorie kann verkürzt durch die Darstellung der Zeitintervalle, in denen bestimmte Methoden bearbeitet oder Tests erfüllt wurden, visualisiert werden (vgl. Abb. 6). Diese kompakte Darstellung erlaubt die Beurteilung von Problemen 5 und 6 aus Abs. 2.3. Mit Methoden der Sequenzanalyse (z.B. [PHB12]) können die Entstehungshistorien aller Studierenden nach Gruppen unterschiedlicher Vorgehensweisen untersucht werden. Sind historische Protokolle verfügbar, wäre auch eine Art Ratgeber-Funktion denkbar, die Anfängern Tipps gibt, um welche Dinge sie sich als Nächstes kümmern könnten.

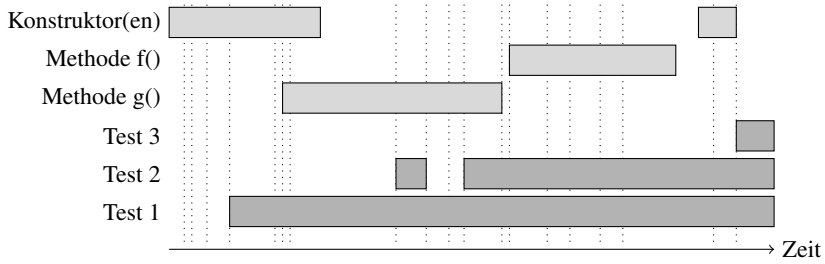

Abb. 6: Zeitleiste mit Zeitintervallen der Bearbeitung (hell) und Testerfüllung (dunkel). Vertikale Linien zeigen compilierbare Zustände an.

5 Fazit und Ausblick

Die Entstehungsgeschichte der Lösung zu einer Programmieraufgabe ist für den Dozenten eine wertvolle Hilfe zur Beurteilung des Kenntnisstands des Autors. Die Erfassung

der Historie in Form eines (persistenten) Protokolls erlaubt verschiedene Auswertungen (wie HeatMap, Bearbeitungssequenzen, ...), die rasch wertvolle Hinweise auf Defizite des Autors liefern können, aber auch eine Beurteilung des Fortschritts der Gruppe insgesamt ermöglichen. Bisher handelt es sich um vorläufige Untersuchungen im kleinen Rahmen; ob und wie weit sich der Ansatz zu einem nützlichen Werkzeug im Alltag der Programmierausbildung ausbauen lässt, soll im bevorstehenden Semester erprobt werden. Möglicherweise sind dazu andere als die bisher üblichen Aufgabentypen angezeigt, bspw. vorgegebene, fehlerhafte Quelltexte, die korrigiert werden sollen. Eine solche Korrektur ist i.a. einfacher als die Konstruktion einer komplett eigenständigen Lösung und hatte vielleicht deswegen bisher wenig Wert, aber die Möglichkeit zur Beobachtung bei der Fehlersuche kann aufzeigen, wo im Quelltext ein Student eine mögliche Ursache gesehen hat und damit wertvolle Einblicke in das Verständnis liefern.

Literaturverzeichnis

[BA01] Ben-Ari, M: Constructivism in computer science education. Journal of Computers in Mathematics and Science, 20:45–73, 2001.

[BB09] Bachmann, Adrian; Bernstein, Abraham: Data retrieval, processing and linking for software process data analysis. University of Zurich, Technical Report, (December), 2009.

[CCP07] Canfora, Gerardo; Cerulo, Luigi; Penta, Massimiliano Di: Identifying Changed Source Code Lines from Version Repositories. International Workshop on Mining Software Repositories, S. 14, Mai 2007.

[Fl07] Fluri, Beat; Würsch, Michael; Pinzger, Martin; Gall, Harald C.: Change Distilling : Tree Differencing for Fine- Grained Source Code Change Extraction. IEEE Trans. on Software Engineering, 33(11):725–743, 2007.

[Ha13] Hanjalic, A: ClonEvol: Visualizing software evolution with code clones. Software Visualization (VISSOFT), S. 1–4, 2013.

[KP10] Kaczmarczyk, LC; Petrick, ER: Identifying student misconceptions of programming. Proceedings Computer Science Education, S. 107–111, 2010.

[Ma87] Matsuhashi, Ann: Revising the plan and altering the text. In: Writing in real time: Modelling production processes. S. 197–223, 1987.

[PHB12] Peter, Sebastian; Höppner, Frank; Berthold, Michael R: Learning Pattern Graphs for Multivariate Temporal Pattern Retrieval. In: Proc. Int. Symp. Intelligent Data Analysis. Jgg. 7619 in LNCS. Springer, S. 264–275, 2012.

[PR96] Piolat, Annie; Roussey, Jean-Yves: Students' drafting strategies and text quality. Learning and Instruction, 6(2):111–129, Juni 1996.

[QYW98] Quilici, Alex; Yang, Qiang; Woods, Steven: Applying Plan Recognition Algorithms To Program Understanding. Automated Software Engineering, 5:347–372, 1998.

[TMK08] Taherkhani, Ahmad; Malmi, Lauri; Korhonen, Ari: Algorithm recognition by static analysis and its application in students' submissions assessment. Int. Conf. on Computing Education Research, S. 88–91, 2008.

[WY96] Woods, Steven; Yang, Q: The program understanding problem: analysis and a heuristic approach. In: 18th Int. Conf. Software Engineering. 1996.