Automatisiertes Feedback für block-basierte Programmiersprachen

Florian Obermüller[1], Luisa Greifenstein[2], Ute Heuer[3], Gordon Fraser[4]

Abstract: Block-basierte Programmiersprachen wie Scratch oder mBlock ermöglichen motivierende und einfache erste Programmierversuche, aber ohne Feedback sowohl zu fehlerhaften oder umständlichen als auch zu gut gelungenen Programmabschnitten werden viele Lernchancen ungenutzt bleiben. Um diesem Problem zu begegnen, stellen wir in diesem Beitrag das von uns entwickelte automatisierte Feedback-Werkzeug LitterBox vor. LitterBox unterstützt Lernende, indem es Programme auf bekannte Codemuster überprüft und für jede gefundene Instanz eines Musters Erläuterungen generiert, die sich direkt auf das vorliegende Programm beziehen und diese am zugehörigen Codeabschnitt visualisiert. Darüber hinaus kann LitterBox Lehrenden helfen, schnell einen Überblick über prinzipielle Lernrückstände und -fortschritte ihrer Schüler*innen zu erhalten. In einer ersten Evaluation mit Grundschullehramtsstudierenden ohne universitäre Informatikausbildung (n = 142) hat sich LitterBox generell als sehr hilfreich beim Erstellen funktionaler und lesbarer Programme erwiesen. Aus den Rückmeldungen der Studierenden konnten außerdem Kriterien für gute Hinweise abgeleitet werden. LitterBox wird von uns als Web-Frontend bereitgestellt oder kann Open Source bezogen und um neue Codemuster erweitert werden.

Keywords: Scratch; mBlock; Codequalität; Hinweisgenerierung; Codemuster; Evaluation

1 Einleitung

Für den Erwerb von ersten Programmierfähigkeiten sind block-basierte Sprachen wie Scratch [Ma10] und mBlock äußerst beliebt [MD20]. Obwohl es relativ einfach ist mit solchen Sprachen erste Programme zu erstellen, da sie versuchen syntaktische Probleme zu vermeiden, können Fehler vorhanden sein. Dies kann unter anderem daran liegen, dass gerade junge Lernende, die wenig Programmierkonzepte kennen oder lernhinderliche Vorstellungen haben, diese Sprachen verwenden. Aber auch wenn die Projekte sich wie erwartet verhalten, kann die Codequalität niedrig sein: Scratch Projekte enthalten eine nicht vernachlässigbare Anzahl an *Code Smells* (i.e., unschöner Code minderer Qualität) [AH16, HSH16]. Dies kann sich negativ auf die Verständlichkeit der Programme auswirken [HA16].

Um diese Qualitätsprobleme und Fehler zu beseitigen, benötigen die Lernenden Unterstützung. Diese wird in den meisten Fällen von Lehrkräften bereitgestellt, jedoch können

[1] Universität Passau, Lst. Software Engineering II, florian.obermueller@uni-passau.de
[2] Universität Passau, Didaktik der Informatik, luisa.greifenstein@uni-passau.de
[3] Universität Passau, Didaktik der Informatik, ute.heuer@uni-passau.de
[4] Universität Passau, Lst. Software Engineering II, gordon.fraser@uni-passau.de

doi:10.18420/infos2023-030

1.

Möglicher Fehler in der Figur **Katze**: **Nutzloser Vergleich**

Problem:
Bei deinem Vergleich erhältst du immer dasselbe Ergebnis: FALSCH. Der Code nach dem Warte-Baustein wird nie ausgeführt.

Verbesserungsidee:
Vermutlich wolltest du eine Variable Leben im Vergleich verwenden. Lege diese in der Kategorie Variable an und ziehe den neuen Baustein in das Feld.

Weitere Informationen

Abb. 1: Beispielfehlermuster: Anstatt die Variable Leben mit dem Wert 2 zu vergleichen, wird das Wort „Leben" mit dem Wert 2 verglichen. Dieser Vergleich resultiert natürlich immer im gleichen Ergebnis, ein Verbesserungsvorschlag wird unterbreitet.

diese durch eine Vielzahl an Handmeldungen im Unterricht überfordert sein oder sind in Selbstlernszenarien gar nicht verfügbar. Um diesem Problem entgegenzuwirken, haben wir LitterBox entwickelt, ein automatisiertes Feedback-Tool für Scratch und mBlock Programme. Dabei sind wir von der Beobachtung ausgegangen, dass sich viele Fehler auf ähnliche lernhinderliche Vorstellungen zurückführen lassen und dadurch bestimmte *Fehlermuster* erkennbar sind. LitterBox identifiziert Fehler und unschönen Code in Scratch Projekten, indem es den Quellcode analysiert und auf Überdeckungen mit definierten Fehlermustern und Code Smells prüft. Für jedes gefundene Problem kombiniert LitterBox eine visuelle Darstellung des betroffenen Codeabschnitts, in dem die Position des Problems hervorgehoben ist, mit einem erklärenden Hinweistext (siehe Abb. 1). Dabei sollen die Hinweise keine direkten Lösungen darstellen, sondern die Lernenden bei der Verbesserung des Programms unterstützen. Während dieses korrigierende Feedback gut für die Entwicklung kognitiver Fähigkeiten ist [WZH20], haben sich positive Rückmeldungen als vorteilhafter für motivationale Aspekte gezeigt [Ha09]. Weiterhin kann rein negatives Feedback schädlich für die Selbstwirksamkeit und intrinsische Motivation sein [RD00, WZH20]. Intrinsische Motivation wiederum kann den Umgang mit Feedback beeinflussen: Motivierte Lernende verarbeiten Feedback besser und positives Feedback motiviert die Lernenden stärker als negatives Feedback [DT15]. Um effektives Lernen zu ermöglichen, sollten Informationen zu Fehlern und richtigem Verhalten aufgezeigt werden, um kognitive und motivationale Aspekte anzusprechen. Hierzu gibt es in LitterBox Code Perfumes. Diese können als Gegenteil von Smells angesehen werden und markieren passende Praktiken.

LitterBox ist als Webanwendung verfügbar, um Lernenden die Verwendung zu erleichtern. In einem ersten Praxistest mit 142 Grundschullehramtsstudierenden wurde das Werkzeug als hilfreich bewertet und das Hervorheben der positiven Aspekte als bestärkend empfunden. Durch die Rückmeldungen konnten Kriterien für gute Hinweise gefunden werden.

2 Qualitätsprobleme und Fehler in block-basierten Programmen

Scratch [Ma10] und mBlock sind bei Lehrkräften und Programmieranfänger*innen beliebt: Durch die block-basierte Natur lassen sich einfach erste Programme entwickeln und durch die verschiedenen Blockformen wird versucht Syntaxfehler zu vermeiden. Dadurch wird der Erwerb von ersten Programmierkonzepten erleichtert, ohne Syntaxbefehle lernen zu müssen [WW15]. Obwohl dieser Ansatz sehr ermutigend für Lernende ist, kann beobachtet werden, dass Scratch-Nutzende negative Angewohnheiten entwickeln [MSABA11]. So wurde gezeigt, dass es weit verbreitete Vorkommen von Qualitätsproblemen in Scratch-Projekten gibt [AH16, HSH16], welche sich als sogenannte Code Smells manifestieren.

Code Smells Selbst ein fehlerfreies Scratch-Projekt kann Code mit mangelnder Qualität enthalten. Code Smells beschreiben dabei konkrete Qualitätsprobleme, die nicht falsch sind, da sie den korrekten Programmablauf nicht beeinflussen, sich aber negativ auf den Lernfortschritt auswirken können, da sie das Code Verständnis und die Veränderbarkeit hemmen [HA16]. Dadurch wird auch die Gefahr von Fehlern in zukünftigen Änderungen erhöht [Fo99]. Code Smells aus anderen Programmiersprachen sind dabei auch in ähnlicher Form in Scratch auffindbar, beispielsweise Code Duplizierung und lange Skripte. Erste Analysewerkzeuge wie Hairball [Bo13] und Quality Hound [TT17] konnten einige Code Smells finden, sind aber für die Verwendung mit Scratch 3 zu alt.

Im Gegensatz zu Code Smells, die für den Programmablauf harmlos sind, kann es natürlich auch Programmabschnitte geben, die bei der Ausführung des Programms zu Fehlern führen. Viele dieser Fehler lassen sich auf die gleichen Ursprünge im Code zurückführen und zu Fehlermustern abstrahieren. Vorherige Untersuchungen haben für diese eine weite Verbreitung nachgewiesen [Fr20]. Das Finden und Verbessern von Fehlern in eigenen Programmen ist als Bestandteil des Lernprozess ein wichtiger Faktor und kann mit passenden Techniken unterstützt werden [MR19]. Gerade Lehrkräfte könnten jedoch von automatisierter Fehleridentifikation profitieren, um Lernende effektiver und effizienter zu unterstützen [Gr21]. Die Fehlermuster lassen sich dabei in mehrere Kategorien einordnen:

Scratch-spezifische Fehler Fehler werden durch den Umgang mit der Bühne und den Figuren erzeugt. Beispielsweise kann das Programm Ereignisverarbeiter verwenden, die auf einen Bühnenbildwechsel warten, ohne dass das Ereignis jemals eintreten kann.

Syntaktische Fehler Obwohl Blöcke an sich nur gültig kombiniert werden können, gibt es Ausnahmen. So sind alle Ausdrücke runde oder spitze Blöcke, sodass sich beispielsweise auch der Block für den Kostümnamen dort verwenden lässt, wo Scratch eine Farbe erwartet, was keinen Sinn ergibt. Auch eigene Blöcke, die in Scratch definiert werden können, gehören zu dieser Kategorie, da deren Signatur nicht validiert wird. Dadurch können mehrere Parameter den gleichen Namen haben, oder Parameter außerhalb des eigenen Blocks verwendet werden.

Allgemeine Fehler Natürlich ist es auch möglich in SCRATCH Fehler zu erzeugen, die auch in anderen Programmiersprachen vorkommen können. So ist es möglich mit eigenen Blöcken oder Nachrichten unendliche Rekursionen zu erzeugen, Variablen oder Attribute können nicht initialisiert sein und damit den Datenfluss stören, und es kann auch zu Anomalien im Kontrollfluss durch fehlende Wiederholungen kommen.

Probleme aus diesen Kategorien treten auch in MBLOCK auf, da es auf SCRATCH basiert. Jedoch stellt es zusätzliche Blöcke bereit, um Lernroboter zu programmieren. Dabei ist es möglich Werte der in den Robotern verbauten Sensoren auszulesen und die Aktoren wie Motoren und LED-Matrizen anzusteuern. Deshalb werden zusätzliche eigene Muster benötigt, um die spezifischen Probleme im Umgang mit der realen Welt aufzuzeigen [Ob22]. Hier liegt der Fokus auf den Messbereichen der Sensoren, da nur Vergleiche innerhalb der gültigen Bereiche Sinn ergeben, und dem richtigen Ein- und Ausschalten der Aktoren.

3 Gute Programmierpraktiken

Analog dazu, dass schlechter Code „smelly" ist, also stinkt, kann guter Code als „wohlduftend" bezeichnet werden. Hier wird von Code Perfumes gesprochen [Ob21]. Folglich sind Code Perfumes Idiome, die auf die korrekte Verwendung von Programmierkonzepten oder gute Praktiken hinweisen. Ob das Programm fehlerfrei funktioniert, wird jedoch nicht gewährleistet, da auch die unpassende Kombination von guten Codeteilen zu unerwarteten Ergebnissen führen kann. Es gibt mehrere Herangehensweisen um Code als gut einzustufen:

Lösungsmuster Wenn Fehlermuster von korrektem Code abweichen, repräsentieren die zugrundeliegenden korrekten Codeteile passende Wege bestimmtes Verhalten zu implementieren. Diese können einem Code Perfume als Grundlage dienen. Jedoch implizieren nicht alle Fehlermuster ein Lösungsmuster. So ist das Verwenden einer Wiederholung ohne einen Fehler zu verursachen nicht wert, als positiv kommentiert zu werden, da hier nur die Abwesenheit von Fehlermustern und nicht das Vorhandensein von Lösungsmustern gelobt werden würde. Lösungsmuster bieten zusätzliche Chancen für Feedback, wenn das passende Fehlermuster vorhanden ist: Was an einer Stelle nicht richtig gemacht wurde, ist in einem anderen Codestück schön gelöst und die Codestellen können verglichen werden.

Elementarmuster & Evidenzvariablen Die Nutzung bestimmter Blöcke kann ein Hinweis auf Problemlösekompetenzen und logisches Denken sein [AB18]. Elementarmuster fokussieren sich auf die korrekte Verwendung solcher Codestrukturen und können für positives Feedback verwendet werden. Ein Beispiel hierfür ist die Verwendung von geschachtelten *Falls dann* oder *Falls dann sonst* Blöcken, da so der Kontrollfluss für mehrere voneinander abhängige Entscheidungen modelliert werden kann. Evidenzvariablen sind eng mit Elementarmustern verwandt und wollen durch Messung algorithmischer Aspekte Evidenz für Computational Thinking Kompetenzen aufzeigen [SF13]. Hier werden Perspektiven wie Koordination zwischen Figuren, Benutzerinteraktion und Initialisierung betrachtet.

Spielprogrammierung Interaktive Spiele sind ein zentraler Bestandteil von SCRATCH. Spiele enthalten häufig spezifische Muster, deren Auftreten nach dem GCS 2.0 Modell wiederum Aufschluss über die Entwicklung von Computational Thinking geben kann [We20]. Beispielsweise kann eine Steuerung mit den Pfeiltasten als Code Perfume angezeigt werden.

4 Das LITTERBOX Werkzeug

4.1 Konsolenanwendung

LITTERBOX kann als Konsolenanwendung die Anzahl an gefundenen Mustern ausgeben. Daneben ist auch eine Ausgabe der Analyse als CSV Datei möglich, in der auch die Ergebnisse eines ganzen Ordners mit Projekten festgehalten werden kann, was es ermöglicht die Dateien einer ganzen Klasse in einem Aufruf zu analysieren. LITTERBOX kann außerdem eine detaillierte Ausgabe mit Orten der Muster und Hinweisen erzeugen, welche für die Webansicht verwendet wird.

Codemuster detektieren Das Detektieren von Fehlermustern, Code Smells und Perfumes in SCRATCH und MBLOCK Programmen ist das Hauptaugenmerk von LITTERBOX. Für diese Analyse benötigt LITTERBOX das Projekt als .sb3 oder .mblock Datei. LITTERBOX gliedert das Programm in einem abstrakten Syntaxbaum, in welchem jeder Block als eigene Klasse dargestellt wird. Sogenannte „Finder" überprüfen welche der Codemuster vorhanden sind, indem sie als Besucher über den Baum navigieren und die spezifischen Blockkombinationen identifizieren. Diese Modellierung ermöglicht das effiziente Hinzufügen neuer Muster zu LITTERBOX. Aus dem Syntaxbaum wird außerdem ein Kontrollflussgraph erzeugt, der beispielsweise zur Feststellung fehlender Initialisierung verwendet wird.

Codemetriken errechnen Neben dem Finden von Codemustern kann LITTERBOX auch Statistiken für den Code errechnen. Diese können dazu dienen, sich einen Eindruck von der Größe, Komplexität und Struktur eines Programms zu machen. Bei diesen Metriken sind sowohl rein deskriptive Werte wie Blockanzahl, zyklomatische Komplexität oder Anzahl an Figuren als auch der wertende Computational Thinking Score [MLRRG15] vorhanden.

4.2 Webansicht

Um Lernenden die Verwendung von LITTERBOX zu erleichtern, stellen wir eine Webanwendung zur Verfügung. Diese bietet Deutsch, Englisch und Spanisch als Sprachen an. Benutzer*innen können eine .sb3/.mblock Datei hochladen oder die Projekt-ID eines öffentlich geteilten SCRATCH-Projekts angeben. Danach werden alle gefundenen Code Perfumes, Code Smells und Fehlermuster nach Kategorie sortiert angezeigt. Dabei wird für alle Instanzen die Figur, das enthaltende Skript und eine textuelle Erklärung verwendet (siehe Abb. 1). Bei Code Perfumes wird erklärt, wieso der Code gut ist, bei Fehlermustern und Code Smells wird das Problem erläutert und eine mögliche Lösung vorgeschlagen.

5 Praxisbeispiel

5.1 Rahmenbedingungen

Um Kriterien für gute Hinweise zu finden, verwendeten 142 Grundschullehramtsstudierende LitterBox in der Webansicht. Zuvor hatten die Studierenden in zwei Einheiten à 90 Minuten Scratch kennengelernt, verschiedene Aufgabentypen bearbeitet und mit der Programmierung des Spiels „Schifffahrt”[5] begonnen. In der dritten Einheit programmierten die Studierenden das Spiel mit der zusätzlichen Unterstützung durch LitterBox weiter. Alle Studierenden haben einen Hinweis bewertet, was zu 142 Rückmeldungen führte. Jede Rückmeldung besteht aus einer numerischen Bewertung zwischen 0 und 2 (Daumen runter, Daumen gerade, Daumen hoch wie in Abb. 1) und einer textlichen Bewertung des Hinweises. Die Studierenden konnten frei wählen, welchen der Hinweise, die sie zu ihrem Programm erhalten hatten, sie bewerten wollten und wie ausführlich ihre textliche Bewertung war.

5.2 Ergebnisse

Abb. 2 zeigt, wie die Studierenden einen ihrer erhaltenen Hinweise numerisch bewerteten und Abb. 3 veranschaulicht die genannten Gründe für die jeweilige Bewertung. Die Studierenden scheinen den Hinweisen positiv gegenüberzustehen (Abb. 2): 50.4 % gaben dem ausgewählten Hinweis einen ‚Daumen hoch’ und 16.2 % einen ‚Daumen runter’. Bei den Code Perfumes ist diese positive Tendenz noch deutlicher (Abb. 2), wenn auch nicht signifikant höher als bei den Hinweisen zu schlechten Programmteilen ($p = .119$, $\hat{A}_{12} = .34$). Die Studierenden nannten diverse Gründe für ihre numerische Bewertung in den textlichen Bewertungen, durch welche Kriterien für (weniger) hilfreiche Hinweise in Bezug auf die Funktion und die Darstellung extrahiert werden konnten (Abb. 3).

5.2.1 Funktionen

In positiver Hinsicht erwähnten die Studierenden LitterBox als hilfreich zur generellen Unterstützung, Anzeige, Verbesserung, Erklärung und zum Erkenntnisgewinn (Abb. 3a). Die häufige Nennung der *Anzeige*-Funktion lässt sich insbesondere auf die Code Perfumes und weniger auf die Fehlermuster zurückführen: Die Anzeige wurde bei den Code Perfumes signifikant häufiger erwähnt als bei Hinweisen zu schlechten Programmteilen ($p < .001$, $\hat{A}_{12} = .21$) und bei den Code Smells wiederum signifikant häufiger als bei den Fehlermustern ($p = .037$, $\hat{A}_{12} = .44$). Die Anzeige der Code Perfumes beschreiben die Studierenden auch als affektiv bestärkend: „Freut mich, dass ich das richtig gemacht habe!” (S83) und „Toll, dass man auch gelobt wird!” (S98). Hinsichtlich des *Erkenntnisgewinns* fanden die Studierenden insbesondere Code Smells hilfreich und erwähnten sie diesbezüglich signifikant häufiger als die Fehlermuster ($p = .029$, $\hat{A}_{12} = .45$). Auch innerhalb der Code Smells wie beispielsweise ‚Langes Skript’ gibt es unterschiedliche Meinungen: So erklärt S7 „Durch die Erklärung

[5] angelehnt an https://projects.raspberrypi.org/en/projects/boat-race, zuletzt abgerufen am 31.01.2023

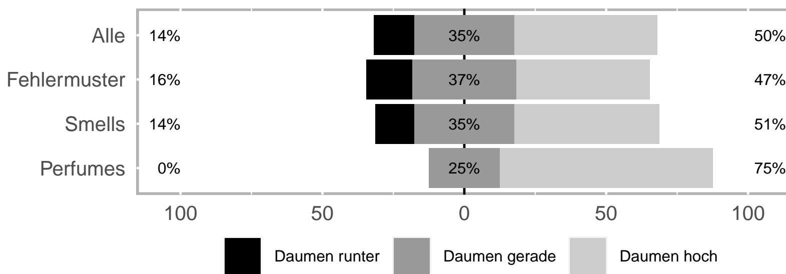

Abb. 2: Verteilung der Bewertungen eines erhaltenen Hinweises.

(Unter-)Kategorie	% Bewertungen
Funktion	71.6%
generelle Unterstützung	27%
Anzeige von Mustern	24.3%
Programm-Verbesserung	18.9%
Erklärung von Konzepten	16.2%
Erkenntnisgewinn	13.5%
Darstellung	39.2%
Verständlichkeit	33.8%
Anschaulichkeit	9.5%

(a) Positive Aspekte und deren Anteil an gesamten positiven Kommentaren.

(Unter-)Kategorie	% Bewertungen
Funktion	40.7%
Überschneidung	16.9%
False Positives	10.2%
keine Analyse der Semantik	6.8%
kein autom. Verbessern	6.8%
Darstellung	62.7%
unklare Vorgehensweise	32.2%
Erklärung	11.9%
Auffinden von Blöcken	8.5%
Unverständlichkeit	8.5%
ungenaue Angaben	3.4%

(b) Negative Aspekte und deren Anteil an gesamten negativen Kommentaren.

Abb. 3: Positive bzw. negative Aspekte eines erhaltenen Hinweises.

habe ich verstanden, dass mein Skript logischerweise viel übersichtlicher wird, wenn ich in kürzeren Blöcken programmiere", wobei S73 findet „Persönlich empfinde ich den Abschnitt meines Skriptes so übersichtlicher als wenn er in tausende Schnipsel zerteilt ist."

In negativer Hinsicht erwähnten die Studierenden die Überschneidung mit anderen Codeabschnitten des Programms, das Vorkommen von False Positives und dass die Semantik nicht analysiert und das Programm nicht automatisch verbessert wurde (Abb. 3b). Die *Überschneidung* wurde signifikant häufiger bei den Code Perfumes als bei den Fehlermustern erwähnt ($p = .009$, $\hat{A}_{12} = .61$). Das kann damit erklärt werden, dass sich Kontrollstrukturen und Programmierkonzepte der Code Perfumes teilweise nur geringfügig wenn auch relevant von anderen Mustern unterscheiden. Am Beispiel des Code Perfumes ‚Bedingungsüberprüfung in Wiederholung' schreibt S47 „In einem anderem Code wurde das als [Smell] ‚aktives Warten' bemängelt, das verwirrt mich nun ein wenig". Der einzige Unterschied, der nicht beachtet wurde, liegt darin, dass beim aktiven Warten das Skript direkt nach

Eintritt der Bedingung beendet wird, wodurch (im Gegensatz zur ‚Bedingungsüberprüfung in Wiederholung') ein *warte bis* ausreichend wäre. Als weitere Funktionen wünschten sich einzelne Studierende eine *automatische Verbesserung* und eine *Analyse der Semantik*. Ersteres könnte mithilfe automatischen Refactorings, das bereits für Scratch-Programme untersucht wurde [Ad21], umgesetzt werden. Dies ist nach der Untersuchung für einige der Muster in LitterBox integriert worden. Zweiteres wird von LitterBox nicht abgedeckt, da es Scratch-Programme nicht ausführt. Hierfür könnten Code-Testwerkzeuge wie Whisker [SKF19] verwendet werden. Diese unpassende Vorstellung zeigt, dass neben Funktionsweise auch Grenzen automatisierter Feedback-Werkzeuge kurz erklärt werden sollten, um Nutzenden bei der Interpretation des Feedbacks zu helfen [Ma21].

5.2.2 Darstellung

Als positiv empfanden viele Studierende die Verständlichkeit und einzelne Studierende die Anschaulichkeit (Abb. 3a). Dabei wurde die *Verständlichkeit* signifikant häufiger bezüglich der Code Smells als den Fehlermustern erwähnt ($p = .033$, $\hat{A}_{12} = .58$). Das mag daran liegen, dass Fehlermustern komplexere Sachverhalte (z.B. Kontrollstrukturen) zugrunde liegen, während Code Smells eher grundlegende Kriterien wie Codelesbarkeit fokussieren. Bei der *Anschaulichkeit* schätzten die Studierenden die Hervorhebung der relevanten Stelle: „mit der Einfärbung und dem Pfeil kann man gut sehen wo der Fehler sich befindet" (S92).

Negativ erwähnt wurden die teilweise unklare Vorgehensweise, die Erklärung, das Auffinden von Blöcken, Unverständlichkeit und ungenaue Angaben (Abb. 3b). Die *unklare Vorgehensweise* beim Verbessern von Mustern stellte für einige Studierende einen negativen Aspekt der Hinweise dar. Gleichzeitig gibt LitterBox bewusst nicht zu konkrete Lösungsvorschläge um die Nutzenden kognitiv zu aktivieren, was relevant für den Transfer von Kenntnissen ist, die über Hinweise erworben wurden [MJWP19]. Hier muss ein Mittelmaß oder eine geeignete Möglichkeit der Differenzierung gefunden werden, die über den Button ‚weitere Infos' bereits angedacht wurde. Außerdem könnten *genauere Angaben* gemacht und das *Auffinden von Blöcken* vereinfacht werden. Ersteres zielt auf genaue Angaben zur Definition des Code Smells ‚Langes Skript' ab: „Allerdings stellt sich jetzt die Frage, wie klein die einzelnen Teile wirklich sein sollen" (S22). Zweiteres könnte wie folgt umgesetzt werden „Man hätte noch sagen können wo man das Attribut findet, z.B. im Skript Bewegung" (S35) oder „das Attribut ‚Richtung' könnte blau [...] sein, damit man es leichter findet" (S54).

6 Fazit

Block-basierte Sprachen sind ein beliebter Einstiegspunkt zum Programmieren. Lernende und Lehrkräfte profitieren von zusätzlichen Hinweisen zu guten und schlechten Programmteilen. Diese stellt LitterBox durch ein innovatives Webinterface bereit. Eine erste Erprobung hat gezeigt, dass die generierten Hinweise hilfreich und motivierend sind. Um LitterBox auszuprobieren sind Webinterface `https://scratch-litterbox.org` und Quellcode `https://github.com/se2p/LitterBox` verfügbar.

Literatur

[AB18] Amanullah, Kashif; Bell, Tim: Analysing students' scratch programs and addressing issues using elementary patterns. In: 2018 IEEE Frontiers in Education Conference (FIE). IEEE, S. 1–5, 2018.

[Ad21] Adler, Felix; Fraser, Gordon; Gründinger, Eva; Körber, Nina; Labrenz, Simon; Lerchenberger, Jonas; Lukasczyk, Stephan; Schweikl, Sebastian: Improving Readability of Scratch Programs with Search-based Refactoring. In: 21st International Working Conference on Source Code Analysis and Manipulation. IEEE, S. 120–130, 2021.

[AH16] Aivaloglou, Efthimia; Hermans, Felienne: How kids code and how we know: An exploratory study on the Scratch repository. In: Proceedings of the ACM Conference on International Computing Education Research. S. 53–61, 2016.

[Bo13] Boe, Bryce; Hill, Charlotte; Len, Michelle; Dreschler, Greg; Conrad, Phillip; Franklin, Diana: Hairball: Lint-inspired static analysis of scratch projects. S. 215–220, 03 2013.

[DT15] DePasque, Samantha; Tricomi, Elizabeth: Effects of intrinsic motivation on feedback processing during learning. NeuroImage, 119:175–186, 2015.

[Fo99] Fowler, Martin: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA, 1999.

[Fr20] Frädrich, Christoph; Obermüller, Florian; Körber, Nina; Heuer, Ute; Fraser, Gordon: Common Bugs in Scratch Programs. In: Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education. S. 89–95, 2020.

[Gr21] Greifenstein, Luisa; Obermüller, Florian; Wasmeier, Ewald; Heuer, Ute; Fraser, Gordon: Effects of Hints on Debugging Scratch Programs: An Empirical Study with Primary School Teachers in Training. In: The 16th Workshop in Primary and Secondary Computing Education. S. 1–10, 2021.

[Ha09] Hattie, John: Visible Learning: A Synthesis of Over 800 Meta-Analyses Relating to Achievement. 01 2009.

[HA16] Hermans, Felienne; Aivaloglou, Efthimia: Do code smells hamper novice programming? A controlled experiment on Scratch programs. In: 2016 IEEE 24th International Conference on Program Comprehension (ICPC). S. 1–10, May 2016.

[HSH16] Hermans, Felienne; Stolee, Kathryn T.; Hoepelman, David: Smells in Block-Based Programming Languages. In: 2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). IEEE, S. 68–72, 2016.

[Ma10] Maloney, John; Resnick, Mitchel; Rusk, Natalie; Silverman, Brian; Eastmond, Evelyn: The Scratch Programming Language and Environment. ACM Transactions on Computing Education (TOCE), 10:16, 11 2010.

[Ma21] Marwan, Samiha; Shabrina, Preya; Milliken, Alex; Menezes, Ian; Catete, Veronica; Price, Thomas W; Barnes, Tiffany: Promoting students' progress-monitoring behavior during block-based programming. In: Proceedings of the 21st Koli Calling International Conference on Computing Education Research. S. 1–10, 2021.

[MD20] McGill, Monica M.; Decker, Adrienne: Tools, Languages, and Environments Used in Primary and Secondary Computing Education. In: Proceedings of the Conference on Innovation and Technology in Computer Science Education. ACM, S. 103–109, 2020.

[MJWP19] Marwan, Samiha; Jay Williams, Joseph; Price, Thomas: An Evaluation of the Impact of Automated Programming Hints on Performance and Learning. In: Proceedings of the ACM Conference on International Computing Education Research. S. 61–70, 2019.

[MLRRG15] Moreno-León, Jesús; Robles, Gregorio; Román-González, Marcos: Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. RED-Revista de Educación a Distancia, 09 2015.

[MR19] Michaeli, Tilman; Romeike, Ralf: Improving Debugging Skills in the Classroom: The Effects of Teaching a Systematic Debugging Process. In: Proceedings of the Workshop in Primary and Secondary Computing Education. WiPSCE'19. ACM, 2019.

[MSABA11] Meerbaum-Salant, Orni; Armoni, Michal; Ben-Ari, Mordechai: Habits of Programming in Scratch. In: Proceedings of the Conference on Innovation and Technology in Computer Science Education. ITiCSE '11. ACM, S. 168–172, 2011.

[Ob21] Obermüller, Florian; Bloch, Lena; Greifenstein, Luisa; Heuer, Ute; Fraser, Gordon: Code Perfumes: Reporting Good Code to Encourage Learners. In: The 16th Workshop in Primary and Secondary Computing Education. Association for Computing Machinery, New York, NY, USA, 2021.

[Ob22] Obermüller, Florian; Pernerstorfer, Robert; Bailey, Lisa; Heuer, Ute; Fraser, Gordon: Common Patterns in Block-Based Robot Programs. In: Proceedings of the 17th Workshop in Primary and Secondary Computing Education. WiPSCE '22, Association for Computing Machinery, New York, NY, USA, 2022.

[RD00] Ryan, Richard M; Deci, Edward L: Intrinsic and extrinsic motivations: Classic definitions and new directions. Contemporary educational psychology, 25(1):54–67, 2000.

[SF13] Seiter, Linda; Foreman, Brendan: Modeling the learning progressions of computational thinking of primary grade students. In: Proceedings of the ninth annual international ACM conference on International computing education research. S. 59–66, 2013.

[SKF19] Stahlbauer, Andreas; Kreis, Marvin; Fraser, Gordon: Testing scratch programs automatically. In: Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, S. 165–175, 2019.

[TT17] Techapalokul, Peeratham; Tilevich, Eli: Quality Hound — An online code smell analyzer for scratch programs. In: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). S. 337–338, Oct 2017.

[We20] Werner, Linda; Denner, Jill; Campe, Shannon; Torres, David M: Computational sophistication of games programmed by children: a model for its measurement. ACM Transactions on Computing Education (TOCE), 20(2):1–23, 2020.

[WW15] Weintrop, David; Wilensky, Uri: To Block or Not to Block, That is the Question: Students' Perceptions of Blocks-Based Programming. In: Proceedings of the International Conference on Interaction Design and Children. IDC '15. ACM, S. 199–208, 2015.

[WZH20] Wisniewski, Benedikt; Zierer, Klaus; Hattie, John: The power of feedback revisited: A meta-analysis of educational feedback research. Frontiers in Psychology, 2020.