

Innovative Lehrkräftebildung, digitally  
enhanced.



# Innovative Lehrkräftebildung, digitally enhanced.

*Multimodale Impulse aus dem Projekt  
SKILL.de*

*INES BRACHMANN; MIRJAM DICK;  
BENJAMIN HEURICH; BENCE  
LUKÁCS; UND ELIŠKA WÖLFL*

*ANDREA SIEBER; ANDREAS  
MICHLER; THOMAS STELZL; URS  
HACKSTEIN; TOBIAS KAISER;  
ROMINA SEEFRIED; REGINA HOLZE;  
PETRA MAYRHOFER; MATTHIAS  
BRANDL; KARSTEN FITZ; KARLA  
MÜLLER; JULIA SIWEK; JOHANNES*

*PRZYBILLA; FLORIAN  
ZITZELSBERGER; DOROTHE KNAPP;  
UND DANIELA WAWRA*

PASSAU





*Innovative Lehrkräftebildung, digitally enhanced. Copyright © by Ines Brachmann; Mirjam Dick; Benjamin Heurich; Bence Lukács; und Eliška Wölfel is licensed under a [Creative Commons Nammensnennung 4.0 International](https://creativecommons.org/licenses/by/4.0/), except where otherwise noted.*

Zitationshinweis: I. Brachmann, M. Dick, B. Heurich, B. Lukács & E. Wölfel (Hrsg.), Innovative Lehrkräftebildung, digitally enhanced. Multimodale Impulse aus dem Projekt SKILL.de.

# 30. Reporting Good Code to Encourage Learners

FLORIAN OBERMÜLLER; UTE HEUER; UND GORDON FRASER

## Abstract

Block-based programming languages like Scratch or mBlock motivate children to be creative while learning to program. Even though the creation of programs is simplified in block-based languages, learning to program can nevertheless be challenging. Automated tools therefore support learners by providing feedback. Linters check the program for potential bugs or code smells in their programs. Even when this feedback is elaborate and constructive, it still is purely negative and ignores what learners have done correctly in their programs. In this paper we present the concept of code perfumes as the counterpart to code smells, indicating the correct application of programming practices considered to be good. By analysing what learners did right we hope to encourage learners, and to provide teachers and students another view of learners' progress. Using a catalogue of 25 code

perfumes for Scratch and 18 for mBlock, we empirically demonstrate that these represent frequent practices.

## Introduction

Learning to program can be challenging and frustrating (Hansen & Eddy, 2007), therefore block-based programming languages like Scratch (Maloney et al., 2010) or mBlock aim to remove some common obstacles, such as the need to memorize programming commands and to produce complex, but syntactically valid textual structures (Bau et al., 2017) by using visual blocks, a single window user interface layout and the minimal command set. While these features and the community helped Scratch and mBlock to become widely used in the computer science education context (McGill & Decker, 2020), the reduced complexity can neither ensure correctness nor good code quality (Frädrich et al., 2020; Hermans et al., 2016; Obermüller et al., 2022; Techapalokul & Tilevich, 2017b). So, block-based code can still inhibit bug patterns and code smells, which negatively influence the young learners coding habits and computational thinking skills (Hermans & Aivaloglou, 2016).

Program analysis Tools like LitterBox (Fraser et al., 2021), Hairball (Boe et al., 2013) and Quality Hound (Techapalokul & Tilevich, 2017a) can detect these problematic patterns to address the problem. Such tools can provide help for the




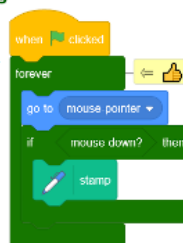
learners by pointing out bugs and code smells along with feedback to help avoid the same problems in the future. However, this way only negative aspects of the projects are highlighted.

For acquiring further cognitive skills this corrective feedback is very useful (Wisniewski et al., 2020). Contrasting this, negative feedback might harm autonomy and self-efficacy resulting in decreased intrinsic motivation (Ryan & Deci, 2000; Wisniewski et al., 2020). Furthermore, this may lead to negative reactions to negative feedback as learners handle feedback better when they are motivated (DePasque & Tricomi, 2015). Positive feedback is considered to have better effects on the motivational aspects especially on task level (Hattie, 2009). Concluding effective feedback should include information about errors as well as good, correct behaviour so that both cognitive and motivational aspects are addressed.

Code perfumes are introduced as a counterpart to code smells, so automated analysis tools can also give feedback to the good parts in student projects. Figure 2 shows the positive feedback for continuously checking an event. Our previous research on code perfumes in both Scratch and mBlock is summarized in this paper (Obermüller et al., 2021; Obermüller et al., 2022). In detail this summary contains 25 code perfumes for Scratch, 18 for mBlock focusing on the CodeyRocky and mBot robot and an evaluation addressing the frequency of these perfumes.

#### Code perfume in sprite **Unicorn: Loop Sensing**

Very good! To ensure that you don't miss a  event you continuously check for it in a forever loop.



*Figure 1. Positive feedback given for correctly implementing a continuous check for an event*

# Background

## Analysing Block-Based Programs

Scratch and mBlock are block-based programming languages that aim to make programming more accessible for novices (Maloney et al., 2010). They favour exploration over recall and visually distinguish different categories of blocks (Bau et al., 2017). Blocks have different shapes that determine how statements can be combined, thus trying to prevent syntactical errors. However, the resulting code can nevertheless contain problems. An important means to provide feedback to learners is to identify common patterns of blocks in the learners' programs:

- Code smells are idioms that decrease the understandability of a program and increase the likelihood of introducing bugs occurring when modifying the code (Fowler, 1999). Multiple code smells for Scratch have either been inspired by other programming languages (Hermans et al., 2016) or have been defined specially for it (Moreno-León et al., 2015). There is evidence that code smells hamper the ability of learners to understand and modify existing code (Hermans & Aivaloglou, 2016) and that code smells can decrease the likelihood of projects being reused by other users (Techapalokul & Tilevich, 2017b). In order to detect code smells in Scratch projects there are automated tools to analyse the programs like Hairball (Boe et al., 2013), Quality Hound (Techapalokul & Tilevich, 2017a), and LitterBox (Fraser et al., 2021).
- Bug patterns refer to code idioms that are likely to be defects (Hovemeyer & Pugh, 2004). These can be detected automatically on source code (Louridas, 2006; Novak et al.,

2010). Bug patterns have been shown to appear frequently in Scratch projects (Frädrich et al., 2020) and there are automated code analysis tools that can be used to find them, such as LitterBox (Fraser et al., 2021). Note that instances of bug patterns in code are likely candidates of bugs, but are not guaranteed to be incorrect.

Another way of helping students with the task of finding and fixing bugs is through automated tests and program verification. These have been introduced to Scratch with the tools Whisker (Stahlbauer et al., 2019), Itch (Johnson, 2016) and Bastet (Stahlbauer et al., 2020). Both testing and verification require either task specific tests or a formal specification that describe the expected behaviour. Both can be tedious to create and are only fit for one task. In contrast, the patterns described above are task-independent and only need to be defined once in order to find them in any project. While for the usage in the Scratch context only virtual user input and program reactions have to be considered, mBlock projects need special handling.

## **Educational Robots**

Besides Scratch a popular way to introduce kids to programming is using educational robots (McGill & Decker, 2020). This has multiple reasons: First, robots offer an easy starting point as they usually can be controlled without a computer, such as the Ozobot robots (Greifenstein et al., 2022; Körber et al., 2021). Second, interacting with the environment rather than just using graphics on the computer can lead to higher learning motivation (Peng et al., 2020). Students have to tackle real world problems, like getting values from a sensor and letting the robot react based on the measurements. Third, programming educational robots leads to an acquisition of programming skills combined with abilities like spatial

thinking (Jung & Won, 2018) and the use of robots may also lead to further discussion about execution of the programs and what consequences can stem from it, for example as motors could be overstrained or other parts of the robot may be damaged when the robot is used wrong. Finally, robots are well suited for cross-curricular activities (e. g., physics, art, physical education) due to their sensors and actuators (Sullivan & Bers, 2016). As an example, the physics of a motion sensor can be discussed and measured data can be used in classic tasks of physics lessons like calculating the average speed of the robot from time needed and travelled way.

Educational robot programming environments are usually intended to help transition to solely computer-based programming without the use of other physical devices. For example, the MakeBlock line of robots and their mBlock<sup>1</sup> programming environment achieve this by using an extended version of Scratch (Maloney et al., 2010). mBlock uses exactly the same block categories and shapes as Scratch extended with new blocks for controlling the robots' actuators and reading the sensors when connected to the computer.

Two popular types of robots compatible with mBlock are the Codey Rocky and the mBot. Both robots have two motors to move each side separately. Also, with both robots have sensors to detect the intensity of the ambient light, display information on an LED matrix and turn LEDs off and on. Furthermore, the mBot has an ultra-sonic sensor for measuring distances and a line following sensor to detect if the robot is driving over dark or bright ground. The Codey Rocky, on the other hand, has a gyroscope, additional lights, a colour sensor as well as a potentiometer.

Since both robots have their own challenges rooted in the robot specific programming based upon sensors and

1. <http://mblock.makeblock.com/>

actuators, bug patterns and code smells fitting for this learning scenario have been defined (Obermüller et al., 2022). This naturally also results in automated feedback about problematic code that can be received by LitterBox. However, positive feedback on good code parts should also be considered.

## Feedback and Learning

Feedback generated by program analysis tools is generally assumed to be effective, in particular, because the feedback is returned regardless of other characteristics of the student and is accordingly perceived as less threatening (Hattie, 2009). Teachers face the challenge of not only having to assess, but also having to support students with their individual problems when programming (Michaeli & Romeike, 2019; Sentence & Cszimadia, 2017; Yadav et al., 2016) and tools can be of help in analysing their students' misunderstandings and currently lacking skills.

Albeit effects of feedback on intrinsic motivation are generally assumed to be small, but nonetheless depend on the type of feedback (Wisniewski et al., 2020). On one hand, negative feedback tends to reduce the perceived autonomy and self-efficacy both influencing factors on the intrinsic motivation (Ryan & Deci, 2000; Wisniewski et al., 2020). Positive feedback on the other hand, might lead to a higher intrinsic motivation (Hattie, 2009). Especially for learners and novices, positive feedback is effective to keep up motivation (Fishbach et al., 2010). There is also evidence that more motivated learners process feedback better (DePasque & Tricoli, 2015), so combined feedback containing positive and negative aspects alike may be better than only negative feedback. This supports the need for positive feedback in automated program analysis for novices.

# Code Perfumes

## Notion of a Code Perfume

Quality problems in source code is considered to 'smell bad'. Based on this metaphor we consider good code to smell code, like a perfume. Therefore, we call code idioms indicating the correct application of programming concepts code perfumes. As with code smells, the presence of code perfumes does not give information about desired functionality: You can implement the wrong functionality using correctly applied programming concepts and vice versa. Furthermore, code perfumes are not intended to be the single indicator for automated grading of student code, as the number of perfumes can be easily increased by adding correctly used coding concepts that do not contribute to the functionality of the project.

## Code Perfumes in Scratch

Code perfumes for Scratch are aspects of code that correctly apply concepts related to use of Sprites and their interactions, in particular correct use of initialisation, collisions and user interaction. The following description of code perfumes is taken from our previous work (Obermüller et al., 2021).

- **Backdrop Switch:** Changing to a backdrop in Scratch games or animated stories should likely induce some state alterations in one or more sprites or backdrops. This can elegantly be implemented using appropriate switch backdrop to options together with when backdrop switches to event handlers to start the desired actions.

Backdrop Switch is inspired by a possible fix of the Missing Backdrop Switch bug pattern (Frädrich et al., 2020).

- **Boolean Expression:** The presence of combinations of expressions (i. e., <, = and > blocks) and Boolean operators (i. e., and, or and not blocks) can be indicative of attempts to properly simplify control flow (Seiter & Foreman, 2013). Note that only instances without Comparing Literals patterns (Frädrich et al., 2020) will be reported.
- **Collision:** Continuous collision checks (sprite touches edge or other sprite) that implicate adapted reactions (e. g., move, change look) are used to implement basic game and animation behaviour in Scratch (Talbot et al., 2020; Werner et al., 2020).
- **Conditional Inside Loop:** Considered as an advanced code structure (Talbot et al., 2020), this perfume is checking for loops that contain at least one conditional construct. For example, an if else statement within a repeat until block.
- **Controlled Broadcast Or Stop:** The timing and conditions for when to start other scripts via a broadcast, or when to stop scripts, must be correct for a right program behaviour. So, a check for a condition, which must be met before broadcasting or stopping, to control both these actions is useful (Amanullah & Bell, 2018). Furthermore, to ensure correct timing, the block responsible for this must be within a loop.
- **Coordination:** The existence of a wait until statement in a Scratch program might be a sign of an effort to adapt the coordination of scripts to changing control flows (Seiter and Foreman, 2013).
- **Correct Broadcast:** Properly implemented message broadcasts should at least consist of matching sending and receiving blocks. Correct Broadcast is inspired by fixes of the Message Never Received and the Message Never Sent bug pattern (Frädrich et al., 2020).
- **Custom Block Usage:** To identify solutions of subtasks that

might be reusable and to implement appropriate custom procedures is considered to be good programming practice. This finder is inspired by fixes of the Call Without Definition bug pattern (Frädlich et al., 2020). It detects the presence and use of custom blocks.

- Directed Motion: Controlling sprite movement by keyboard inputs is a common task in games and animated stories. A simple implementation consists of a when key pressed event handler followed by point in direction and move steps statements (Talbot et al., 2020; Werner et al., 2020).
- Gliding Motion: This finder reports another simple implementation to manipulate sprite movement: a when key pressed event handler followed by one or more glide secs to statements (Talbot et al., 2020; Werner et al., 2020).
- Initialisation of Looks: Defining the start state of games and animated stories is especially useful, since Scratch does not perform any default resetting of attributes automatically. Furthermore, it is considered a good programming practice to think about desired initial states of program executions. Look blocks like costume or backdrop setter statements being present in when green flag clicked scripts are reported by this finder. This might indicate that learners tried to solve a subtask of the defining a start state problem. The presence of this and the next pattern is used by Seiter and Foreman (Seiter & Foreman, 2013) to measure computational thinking skills of students.
- Initialisation of Positions: This perfume finder reports position setter statements being present in when green flag clicked scripts possibly indicating that learners tried to solve another subtask of the defining a start state problem.
- List Usage: The existence of list-statements in Scratch programs might be a sign of an effort to hold and process a number of values efficiently.
- Loop Sensing: Continuously checking for touch or key



events inside a forever or repeat until loop is a useful pattern to implement event processing in Scratch. This perfume is inspired by a possible fix of the bug pattern Missing Loop Sensing (Frädrieh et al., 2020).

- Matching Parameter: Properly implemented custom blocks consist at least of a signature containing a complete parameter list: all parameters, that are used inside a custom block, are to be present in the list. This perfume finder is inspired by fixes of the Orphaned Parameter bug pattern (Frädrieh et al., 2020). It detects whether all parameters used are also declared in the custom block.
- Mouse Follower: Sprite movement can be controlled by mouse input. This behaviour can be implemented in Scratch by a loop containing either a go to mouse-pointer statement or a combination of point towards mouse-pointer and move steps statements (Talbot et al., 2020).
- Movement In Loop: To avoid Stuttering Movement (Frädrieh et al., 2020) when controlling sprites by keyboard input it is recommended to use a loop with a conditional containing a key pressed, expression and appropriate actions.
- Nested Conditional Checks: Nested conditional checks (i. e., nested if then and if else blocks) can be seen as advanced code structures (Amanullah & Bell, 2018; Talbot et al., 2020).
- Nested Loops: The presence of nested loops, where the inner one is accompanied by other blocks preceding or following it to not have a Nested Loop smell (Fraser et al., 2021), might be indicative for attempts to implement advanced control flow (Talbot et al., 2020).
- Object Follower: In some games or animations one sprite follows another for at least a certain time (Talbot et al., 2020). This can be implemented using a loop containing a point towards statement targeting the other sprite, followed by a move steps statement.
- Parallelisation: The presence of two scripts with the same

hat block can be indicative of attempts to implement independent subtasks more clearly and readably (Seiter & Foreman, 2013).

- **Say Sound Synchronisation:** A nice way to enhance interaction between program and player is to use both say and play sound blocks in a synchronous way to let sprites talk. However, this say sound synchronisation is not straightforward in Scratch. It can be implemented by placing a play sound file block, playing a message, right after the say block that shows the message in a speech bubble. As soon as the sound file ends, the speech bubble must be cleared by using an empty say block afterwards (Boe et al., 2013).
- **Timer:** Timing durations is a useful subtask in many Scratch programming problems. This finder reports the usage of a variable that is changed repeatedly (inside, e. g., a forever loop) by a fixed value in combination with a wait seconds statement. (Talbot et al., 2020; Werner et al., 2020).
- **Useful Position Check:** Checking position and distance values can be quite error-prone since floating point values are used and have to be compared. A bigger-than or less-than operator to compare values can be a fix to the Position Equals Check bug pattern (Frädrich et al., 2020).
- **Valid Termination Condition:** The repeat until statement requires a termination condition, otherwise the loop will run forever and code following the loop will never be executed. This perfume is inspired by a possible fix of the Missing Termination Condition bug pattern (Frädrich et al., 2020).

## Code Perfumes in mBlock

Code perfumes for mBlock are aspects of code that correctly apply concepts related to robot use, in particular correct use of

sensory data and actuators. The following description of code perfumes is taken from our previous work (Obermüller et al., 2022).

- Colour Usage: Appropriate values for colours are integers from 0 to 255.
- Correct Sensing: Queries involving a sensor should use a valid value range. This indicates comprehension of the sensing concept, which is frequently used in robot programming. Depending on the sensor used, we distinguish several variants of this code perfume:
  - Battery Sensing: The value range for Battery Sensing is from 0 to 100.
  - Colour Sensing: The value range for Colour Sensing is from 0 to 255.
  - Distance Sensing: The value range for Distance Sensing is from 3 to 400.
  - Light Sensing: The value range for Light Sensing is from 0 to 100 on the Codey Rocky and from 0 to 1020 on the mBot.
  - Line Sensing: The values for Line Sensing are the integers from 0 to 3.
  - Loudness Sensing: The value range for Loudness Sensing is from 0 to 100.
  - Pitch Angle Sensing: The value range for Pitch Angle Sensing is from -180 to 180.
  - Potentiometer Sensing: The value range for Potentiometer Sensing is from 0 to 100.
  - Roll Angle Sensing: The value range for Roll Angle Sensing is from -90 to 90.
  - Shaking Sensing: The value range for Shaking Sensing is from 0 to 100.
- Correct Actuator Deactivation: When using blocks that activate an actuator, one must be aware of the necessity of also deactivating them. Writing a separate script for

turning the actuators off is often useful and shows that this robot specific usage of actuators has been understood. We define several versions of this code perfume depending on the specific actuator used:

- LED Off
- Light Off
- Matrix Off
- Motor Off
- Loop Sensing: In order to make the robot react to a specific change of a sensor's value, one must continuously read the corresponding sensor values. Using queries concerning sensors within a loop indicates the comprehension of this robot typical concept of sensing.
- Motor Usage: The motors of the robots can be controlled with a minimum of 0% and a maximum of 100%. When using the mBot robot, values beneath 25% have the same effect as 0%. Therefore, appropriate values range from 0 to 100 for the Codey Rocky robot, and from 25 to 100 for the mBot robot.
- Parallelisation: Writing several scripts with the same hat block can be indicative of attempts to implement independent subtasks at a higher readability level.

## Evaluation

This evaluation summarises parts of the evaluation done in our prior work (Obermüller et al., 2021; Obermüller et al., 2022).

## Experimental Setup

To evaluate the different types of code patterns for Scratch

and mBlock projects, we empirically investigated the following research questions:

**RQ1:** How common are code perfumes in Scratch programs?

**RQ2:** How common are code perfumes in mBlock programs for Codey Rocky and mBot?

## Analysis Tool

In order to study the occurrence of the patterns listed in [Section 3](#) in mBlock programs, we utilise the LitterBox tool. LitterBox handles the analysis of Scratch and mBlock programs by automatically converting a project into an abstract syntax tree (AST), where each block is represented by a node. After conversion the tree is checked for the presence of block combinations indicating a code perfume utilising a visitor pattern. We added new finders for these perfumes according to the guidelines of prior work (Fraser et al., 2021).

## Dataset

- Scratch dataset: We use the dataset by Frädriich et al. created to study bug patterns (Frädriich et al., 2020). It consists of 74,907 Scratch projects mined over the course of 3 weeks.
- mBlock dataset: We created a dataset of 28,192 mBlock programs by mining all publicly shared projects from the mBlock website until the April 2021. Out of these programs, 16,569 contain a Codey Rocky or mBot robot (and sometimes more than one). The remaining projects contain code for other robots. Continuing the filtering process, we removed all programs containing no code inside the robots. After this step 3,540 relevant projects

with a total of 529 Codey Rocky robots and 3,023 mBot robots, including 27 projects utilising both robots, remain.

## Methodology

To answer RQ1, we applied LitterBox to the dataset containing random Scratch projects. For checking how common code perfumes are we consider the total number of code perfumes found, the instances found for each type of perfume, as well as the number of projects containing at least one perfume.

To answer RQ2, we consider the LitterBox findings reported for all the mBlock specific code perfumes on the mBlock dataset. For each code perfume, we inspect the total number of instances found and how many programs are affected.

## Threats to Validity

We used large datasets for both Scratch and mBlock programs, but results may not generalise to other programs. In particular, the data mining can only download publicly shared projects, and incomplete programs may not be shared having different properties. While we analysed how frequent code perfumes are, we generally did not evaluate their effects on learners.

## **How common are code perfumes in Scratch?**

We found instances of all 25 Scratch code perfumes in the dataset. In total, there are 4,712,055 code perfume instances, and 73,787 projects contained at least one code perfume. The exact numbers for each type of perfume are summarised in

Table 1. Note that a project may contain more than one type of code perfume and also multiple instances of the same perfume type, so the numbers of projects containing one type of perfume do not add up to the total number of projects inspected.

Bug Pattern	# Patterns	# Projects	Avg. WMC
Action Not Stopped	117	77	34.34
Colour Out Of Range	9	7	4.86
Interrupted Loop Sensing	1,287	319	14.34
LED Off Missing	1,516	675	11.37
Light Off Missing	12	9	5.56
Low Motor Power	348	90	11.38
Matrix Off Missing	694	478	13.34
Missing Loop Sensing	278	126	12.20
Motor Off Missing	491	364	11.55
Motor Out Of Range	230	68	9.53
Parallel Actuator Use	887	473	16.21
Query In Loop	26	8	49.88
Sensor Equals Check	70	34	17.71
Several Launches	52	52	18.85
Stuttering Action	89	32	13.02
Useless Battery Sensing	0	0	
Useless Colour Sensing	0	0	
Useless Distance Sensing	10	7	8.20
Useless Light Sensing	1	1	12.00
Useless Line Sensing	3	3	12.00
Useless Loudness Sensing	0	0	
Useless Pitch Angle Sensing	0	0	
Useless Potentiometer Sensing	0	0	
Useless Roll Angle Sensing	0	0	
Useless Shaking Sensing	0	0	
Waiting Aborted	9	9	10.11
Total	6,129	1,903	11.39

Table 1. Number of perfume instances found in total and number of projects containing the perfume.

Code Perfume	# Patterns	# Projects	Avg. WMC
Battery Sensing	1	1	9.00
Colour Sensing	0	0	
Colour Usage	829	291	12.63
Distance Sensing	1,204	548	11.90
LED Off	9	9	7.56
Light Off	0	0	
Light Sensing	250	178	12.89
Line Sensing	584	138	25.72
Loop Sensing	573	272	9.36
Loudness Sensing	12	8	13.75
Matrix Off	2	2	9.50
Motor Off	43	43	9.19
Motor Usage	10,896	1,815	9.64
Parallelisation	72	49	14.65
Pitch Angle Sensing	1	1	40.00
Potentiometer Sensing	5	3	11.67
Roll Angle Sensing	2	1	102.00
Shaking Sensing	3	3	22.00
Total	14,495	2,284	9.44

Table 2. Number of perfume instances found in total and number of projects containing the perfume.

The Parallelisation code perfume shows most perfume instances (1,142,319) overall. This can be attributed to the parallel nature of Scratch programs. The other code perfume with an outstanding high total number of instances found is Boolean Expression (1,037,703). This again is natural as the Boolean operators in Scratch are a crucial part to regulate the control flow without having to nest multiple if then blocks. This is a simple way to prevent a Nested Loops code smell.

There is a much bigger difference between projects using Initialisation of Looks (54,065) to total number of perfumes

(473,814) than for Initialisation of Positions with 38,824 projects containing a total of 131,560 perfumes. This is because the perfume checking for the looks blocks has multiple possible initialisations (e. g., visibility, size and costume) whereas the other one just looks at the position. Lists are the least used Scratch Feature (Amanullah & Bell, 2020), but users knowing how to handle lists use them frequently, as indicated by the 130,290 instances of List Usage in only 4,082 projects.

Considering the number of projects containing code perfumes, the most common perfume is Parallelisation occurring in 70,519 projects. Again, this is not surprising due to the parallel Scripts used as basis for the event-driven paradigm used in Scratch. The low weighted method count (WMC) of 49.74 of the projects containing Parallelisation furthermore suggests that this concept is already used in quite small projects.

The frequent occurrence of Initialisation of Looks (54,065) and Initialisation of Positions (38,824) is also intuitive, since initialisation of sprites in Scratch is usually necessary for programs to work correctly. For both the WMC is comparatively low, showing that initialisation of looks and position is also needed and important in early stages of programming.

The rather low occurrences of the motion related Gliding Motion (1,810), Directed Motion (2,116) and Movement in Loop (11,665) perfumes is supported by prior research (Frädrieh et al., 2020) which found frequent occurrences of the Stuttering Movement bug pattern. This suggests that learners prefer to follow the event-driven but simpler approach of detecting events with the dedicated blocks, even though this results in stuttering movement. The least common perfume is Say Sound Synchronisation appearing in only 41 projects.

This perfume is not directly related to any programming concept and likely only useful in specific types of animation projects, therefore lower numbers can be expected.



## How common are code perfumes in mBlock programs for Codey Rocky and mBot?

We found instances for 16 of the 18 code perfumes defined in [Section 3.2](#). In total there are 14,495 code perfume instances, and 2,284 projects containing at least one code perfume. Table 2 shows the number of code perfume instances found for each type, the number of projects containing at least one instance of the respective code perfume, and the average weighted method count of these programs. A project may contain more than one type of code perfume and also multiple instances of the same perfume type.

As visualised by the table two code perfumes were not found in the dataset: The Colour Sensing code perfume depends on the Colour Detection sensor, which is rarely used. For Light Off the likely reason is that there are only 529 Codey Rocky projects in the dataset.

The most frequent code perfume is Motor Usage with 10,896 instances. It is followed by Distance Sensing (1,204) and Colour Usage (829). Both perfumes concerning the correct usage of actuators represent the easiest and most basic way of working with robots. Distance Sensing is also to be expected as it relates to the most used sensor.

Battery Sensing (1), Pitch Angle Sensing (1), Roll Angle Sensing (2), Shaking Sensing (3) and Potentiometer Sensing (5), all are based on sensors which are not used frequently in the dataset. Matrix Off (2) and LED Off (9) are also quite rare, demonstrating that returning the robot to a neutral state after the program has finished execution is not frequently done. However, the low complexity of projects exhibiting LED Off, Motor Off, and Matrix Off shows that it is not difficult to correctly turn off these actuators. This leads to the notion that

actuators are rarely turned off because of the difficulty, but because users may not be aware it should be done.

The fact that Line Sensing is used in more complex projects (25.72) seems surprising at first, as it is a common and easy task. However, line following tasks tend to require several control structures, i. e., at least one loop and then one if block for each of the four states of the sensor; this may explain the higher complexity. The average complexity of projects containing Loop Sensing (9.36) is low because most robot programs need a sensing loop in order to react to the real world.

## Conclusion

A common ground is important for providing feedback. Negative aspects of code in Scratch and mBlock can be pointed out with bug patterns and code smells. However, in order to provide more individualised feedback, and especially for encouraging learners, providing feedback about positive aspects of the code is also essential. For defining a vocabulary about good coding practices, we introduced and empirically evaluated a catalogue of 25 code perfumes in Scratch and 18 in mBlock. Our evaluation found occurrences of all Scratch perfumes and 16 types in mBlock.

An important next step will be to study the effects of these positive code patterns and the feedback on the learning success of novice programmers. Furthermore, the benefit for teachers, who need to get an overview of their students' Scratch programs in order to provide individual feedback, is another interesting topic. Hence, another important aspect of future work will be to evaluate the impact of positive linting on help teachers provide to their students.

In order to support the use of code perfumes as means of individual feedback and further research, all our code

perfumes are implemented directly into LitterBox, which is freely available at: <https://scratch-litterbox.org>.

## References

Amanullah K. & Bell T. (2018). Analysing students' scratch programs and addressing issues using elementary patterns. 2018 IEEE Frontiers in Education Conference (FIE), 1–5. <https://doi.org/10.1109/FIE.2018.8658821>

Amanullah K. & Bell T. (2020). Teaching Resources for Young Programmers: the use of Patterns. 2020 IEEE Frontiers in Education Conference (FIE), 1–9. <https://doi.org/10.1109/FIE44824.2020.9273985>

Bau D., Gray J., Kelleher C., Sheldon J., Turbak F. (2017). Learnable Programming: Blocks and Beyond. Communications of the ACM 60, 72–80. <https://doi.org/10.1145/3015455>

Boe B., Hill C., Len M., Dreschler G., Conrad P., Franklin D. (2013). Hairball: Lint-inspired static analysis of scratch projects. SIGCSE 2013 – Proceedings of the 44th ACM Technical Symposium on Computer Science Education, 215–220. <https://doi.org/10.1145/2445196.2445265>

DePasque S. & Tricoli E. (2015). Effects of intrinsic motivation on feedback processing during learning. *NeuroImage* 119, 175–186. <https://doi.org/10.1016/j.neuroimage.2015.06.046>

Fishbach A., Eyal T., Finkelstein S. R. (2010). How positive and negative feedback motivate goal pursuit. *Social and Personality Psychology Compass* 4, 517–530. <https://doi.org/10.1111/j.1751-9004.2010.00285.x>

Fowler M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA.

Frädriich C., Obermüller F., Körber N., Heuer U., Fraser G. (2020). Common Bugs in Scratch Programs. Proceedings of the 2020 ACM Conference on Innovation and Technology in

Computer Science Education, 89–95. <https://doi.org/10.1145/3341525.3387389>

Fraser G., Heuer U., Körber N., Obermüller F., Wasmeier E. (2021). LitterBox: A Linter for Scratch Programs. 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training, 183–188. <https://doi.org/10.1109/ICSESEET52601.2021.00028>

Greifenstein L., Graßl I., Heuer U., Fraser G. (2022). Common Problems and Effects of Feedback on Fun When Programming Ozobots in Primary School. Proceedings of the 17th Workshop in Primary and Secondary Computing Education. Article 5, 1–10. <https://doi.org/10.1145/3556787.3556860>

Hansen S. & Eddy E. (2007). Engagement and Frustration in Programming Projects. Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education 271–275. <https://doi.org/10.1145/1227310.1227407>

Hattie J. (2009). Visible Learning: A Synthesis of Over 800 Meta-Analyses Relating to Achievement. <https://doi.org/10.4324/9780203887332>

Hermans F. & Aivaloglou E. (2016). Do code smells hamper novice programming? A controlled experiment on Scratch programs. 2016 IEEE 24th International Conference on Program Comprehension, 1–10. <https://doi.org/10.1109/ICPC.2016.7503706>

Hermans F., Stolee K. T., Hoepelman D. (2016). Smells in Block-Based Programming Languages. 2016 IEEE Symposium on Visual Languages and Human-Centric Computing, 68–72. <https://doi.org/10.1109/VLHCC.2016.7739666>

Hovemeyer D. & Pugh W. (2004). Finding Bugs is Easy. SIGPLAN Not. 39, 92–106. <https://doi.org/10.1145/1052883.1052895>

Johnson D. E. (2016). ITCH: Individual Testing of Computer Homework for Scratch Assignments. Proceedings of the 47th ACM Technical Symposium on Computing Science Education, 223–227. <https://doi.org/10.1145/2839509.2844600>

Jung, S.E.; Won, E.-s. (2018). Systematic Review of Research

Trends in Robotics Education for Young Children. *Sustainability*, 1-24. <https://doi.org/10.3390/su10040905>

Körber N., Bailey L., Greifenstein L., Fraser G., Sabitzer B., Rottenhofer M. (2021). An Experience of Introducing Primary School Children to Programming using Ozobots (Practical Report). The 16th Workshop in Primary and Secondary Computing Education, Article 23, 1–6. <https://doi.org/10.1145/3481312.3481347>

Louridas P. (2006). Static code analysis. *IEEE Software* 23, 58–61. <https://doi.org/10.1109/MS.2006.114>

Maloney J., Resnick M., Rusk N., Silverman B., Eastmond E. (2010). The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*, 1-15. <https://doi.org/10.1145/1868358.1868363>

McGill M. M.& Decker A. (2020). Tools, Languages, and Environments Used in Primary and Secondary Computing Education. *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, 103–109. <https://doi.org/10.1145/3341525.3387365>

Michaeli T. & Romeike R. (2019). Current Status and Perspectives of Debugging in the K12 Classroom: A Qualitative Study. 2019 IEEE Global Engineering Education Conference (EDUCON), 1030–1038. <https://10.1109/EDUCON.2019.8725282>

Moreno-León J., Robles G. & Román-González M. (2015). Dr. Scratch: Automatic Analysis of Scratch Projects to Assess and Foster Computational Thinking. *RED-Revista de Educación a Distancia*, 1-23. <https://doi.org/10.6018/red/46/10>

Novak J., Krajnc A., Žontar R. (2010). Taxonomy of static code analysis tools. The 33rd International Convention MIPRO. 418–422.

Obermüller F., Bloch L., Greifenstein L., Heuer U., Fraser G. (2021). Code Perfumes: Reporting Good Code to Encourage Learners. The 16th Workshop in Primary and Secondary Computing Education, 1–10. <https://doi.org/10.1145/3481312.3481346>

Obermüller F., Pernerstorfer R., Bailey L., Heuer U., Fraser G. (2022). Common Patterns in Block-Based Robot Programs. Proceedings of the 17th Workshop in Primary and Secondary Computing Education. Article 4, 1-10. <https://doi.org/10.1145/3556787.3556859>

Peng L., Bai M., Siswanto I. (2020). A study of learning motivation of senior high schools by applying unity and mblock on programming languages courses. Journal of Physics: Conference Series. 12-37. <https://doi.org/10.1088/1742-6596/1456/1/012037>

Ryan R. M. & Deci E. L. (2000). Intrinsic and extrinsic motivations: Classic definitions and new directions. Contemporary educational psychology 25, 54-67. <https://doi.org/10.1007/s10639-016-9482-0>

Seiter L. & Foreman B. (2013). Modeling the learning progressions of computational thinking of primary grade students. Proceedings of the ninth annual international ACM conference on International computing education research, 59-66. <https://doi.org/10.1145/2493394.2493403>

Sentance S. & Csizmadia A. (2017). Computing in the curriculum: Challenges and strategies from a teacher's perspective. Education and Information Technologies 22, 469-495. <https://doi.org/10.1007/s10639-016-9482-0>

Stahlbauer A., Frädrieh C., Fraser G. (2020). Verified from Scratch: Program Analysis for Learners' Programs. Proceedings of the International Conference on Automated Software Engineering (ASE), 150-162. <https://doi.org/10.1145/3324884.3416554>

Stahlbauer A., Kreis M., Fraser G. (2019). Testing scratch programs automatically. Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 165-175. <https://doi.org/10.1145/3338906.3338910>

Sullivan A., Bers M. U. (2016). Robotics in the early childhood classroom: learning outcomes from an 8-week robotics

curriculum in prekindergarten through second grade. *International Journal of Technology and Design Education* 26, 3–20. <https://doi.org/10.1007/s10798-015-9304-5>

Talbot M., Geldreich K., Sommer J., Hubwieser P. (2020). Re-use of programming patterns or problem solving? Representation of Scratch programs by TGraphs to support static code analysis. *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. 1–10. <https://doi.org/10.1145/3421590.3421604>

Techapalokul P. & Tilevich E. (2017a). Quality Hound — An online code smell analyzer for scratch programs. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing*, 337–338. <https://doi.org/10.1109/VLHCC.2017.8103498>

Techapalokul P. & Tilevich E. (2017b). Understanding Recurring Quality Problems and Their Impact on Code Sharing in Block-Based Software. *2017 IEEE Symposium on Visual Languages and Human-Centric Computing*, 43–51. <https://doi.org/10.1109/VLHCC.2017.8103449>

Werner L., Denner J., Campe S., Torres D. M. (2020). Computational sophistication of games programmed by children: a model for its measurement. *ACM Transactions on Computing Education (TOCE)* 20, 1–23. <https://doi.org/10.1145/3379351>

Wisniewski B., Zierer K., Hattie J. (2020). The power of feedback revisited: A meta-analysis of educational feedback research. *Frontiers in Psychology* 10, Article 3087. <https://doi.org/10.3389/fpsyg.2019.03087>

Yadav A., Gretter S., Hambrusch S., Sands P. (2016). Expanding computer science education in schools: understanding teacher experiences and challenges. *Computer Science Education* 26, 235–254. <https://doi.org/10.1080/08993408.2016.1257418>

## Medien-Attributierungen

- Positive feedback given for correctly implementing a continuous check for an event © Florian Obermüller, Ute Heuer, Gordon Fraser is licensed under a [CC BY \(Namensnennung\)](#) license
- Table 1. Number of perfume instances found in total and number of projects containing the perfume. © Florian Obermüller, Ute Heuer, Gordon Fraser is licensed under a [CC BY \(Namensnennung\)](#) license
- Table 2. Number of perfume instances found in total and number of projects containing the code perfume. © Florian Obermüller, Ute Heuer, Gordon Fraser is licensed under a [CC BY \(Namensnennung\)](#) license

## About the Authors



Florian Obermüller

Florian Obermüller is a research assistant in Computer Science Education and a PhD student at the chair for Software Engineering II at the University of Passau, Germany. He passed the first state examination for secondary schools (teaching degree for computer science and mathematics) and has a master's degree in computer science. The main focus of his research is automated generation of feedback to improve programming learning scenarios.





## Ute Heuer

<https://www.ddi.fim.uni-passau.de>

Ute Heuer is an Akademische Direktorin in Computer Science Education at the University of Passau, Germany. She passed the state examinations for secondary schools (teaching degree for mathematics, physics and computer science) and worked as a teacher at several schools in Bavaria. She is interested in improving the quality of computer science learning, instruction and teacher training at primary and secondary level.



## Gordon Fraser

Gordon Fraser is a full professor in Computer Science at the University of Passau, Germany. He received a PhD in computer science from Graz University of Technology, Austria, worked as a post-doc at Saarland University, and was a Senior Lecturer at the University of Sheffield, UK. The central theme of his research is improving software quality, and his recent research concerns the prevention, detection, and removal of defects in software.

# Zitation und Lizenzhinweise

Brachmann, I., Dick, M., Heurich, B., Lukács, B. & Wöfl, E. (Hrsg.),  
Innovative Lehrkräftebildung, digitally enhanced. Multimodale  
Impulse aus dem Projekt SKILL.de. Verfügbar unter:  
<https://oer.pressbooks.pub/skilldeopenbook/>



*Innovative Lehrkräftebildung, digitally enhanced. Copyright © by Ines  
Brachmann; Mirjam Dick; Benjamin Heurich; Bence Lukács; und Eliška Wöfl  
is licensed under a [Creative Commons Nammensnennung 4.0 International](https://creativecommons.org/licenses/by/4.0/),  
except where otherwise noted.*