The Toolbox of Functions: Teaching Code Reuse in Schools

Luca Chiodini
Software Institute - Università della Svizzera italiana
Lugano, Switzerland
luca.chiodini@usi.ch

Joey Bevilacqua
Software Institute - Università della Svizzera italiana
Lugano, Switzerland
joey.bevilacqua@usi.ch

Matthias Hauswirth
Software Institute - Università della Svizzera italiana
Lugano, Switzerland
matthias.hauswirth@usi.ch

Abstract

Large programs often contain duplicate parts, known as code clones. Programs riddled with code clones become difficult to reason about and modify. To avoid code clones and enable code reuse, programmers introduce abstractions such as functions and classes. Because abstraction is so important, it should be explicitly taught in programming courses and appropriately supported by tools. Unfortunately, development environments, including some for novices, do not always encourage abstraction. Instead, they facilitate the creation of code clones and ultimately hinder code reuse. This paper presents the *Toolbox of Functions*, an approach for teaching code reuse to beginner programmers in schools. This approach helps students to develop, collect, and reuse their own functions, as a simple form of abstraction. Learners are guided in creating and using their own library, without the complexity found in other environments. We implemented the approach in a publicly-available web platform for programming in Python. We collaborated with high school teachers who adopted the approach in their mandatory programming courses. Over the course of a year, more than 800 users executed over 30 000 programs that use the Toolbox of Functions. This initial experience suggests the potential of the approach to instill the principle of code reuse effectively.

CCS Concepts

• **Social and professional topics → Computer science education**.

Keywords

Software Engineering Education, Libraries, Code Reuse, Abstraction

ACM Reference Format:
Luca Chiodini, Joey Bevilacqua, and Matthias Hauswirth. 2025. The Toolbox of Functions: Teaching Code Reuse in Schools. In *ECSEE 2025: European Conference on Software Engineering Education (ECSEE 2025), June 02–04, 2025, Seeon, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3723010.3723029

1 Introduction

More and more countries are introducing programming education in their school curricula. Due to the limited classroom time available, programming lessons often focus on writing small one-off pieces of code. The focus lies on getting students to solve specific problems, and students then throw away their solutions once they are done.

Only diligent students may keep an organized collection of their solutions to different programming exercises. They may then review their solutions when preparing for exams, but they will rarely actually reuse any of the code they wrote. Such learning experiences, involving many small but disconnected pieces of code, do not reflect the ideas of software engineering.

This paper presents the *Toolbox of Functions (ToF)*, an approach to encourage code reuse in a motivating and simple way. We implemented the ToF in a web-based Python programming platform used in programming courses in several Swiss high schools.

We proceed by illustrating the problem addressed by the ToF (Section 2), explaining the general approach (Section 3), describing an instantiation of the approach in a web platform (Section 4), and briefly reporting on its initial use in schools (Section 5).

2 Background and Related work

When they grow beyond toy examples, programs invariably end up consisting of multiple parts that accomplish similar behaviors. How are these related behaviors implemented in program code? One straightforward option is writing the same or similar code as many times as needed. These duplicate parts of program code are known as *code clones*. Code clones can be classified into different types [9], ranging from being exact duplications of identical chunks of code, to being duplications except for some identifiers, to having some additional or missing parts of code.

Producing code clones is disadvantageous because it leads to maintainability issues [6]. A bug discovered in one clone, or a change needed to accommodate a new functionality, needs to be identified and manually applied individually to each clone.

2.1 Code Clones Are Widespread

Despite the disadvantages, programmers frequently duplicate code. The adage "Copy & Paste" embodies this idea. Such an operation is sometimes considered the fastest way to achieve a certain goal.

Because abstraction is considered a fundamental but difficult concept to master in computer science [13], it is perhaps unsurprising that novice programmers avoid abstractions and frequently resort to copy-paste [7, 15].

Studies show that even code written by experienced programmers commonly contains code clones. For example, code clones are common in code examples published on Stack Overflow [3] and in code cells contained in Jupyter notebooks [8]. A large-scale study on GitHub repositories across multiple programming languages found high rates of code duplication in files both within a repository and across repositories [11].

This work is licensed under a Creative Commons Attribution International 4.0 License.

ECSEE 2025, Seeon, Germany
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1282-1/25/06
https://doi.org/10.1145/3723010.3723029

The recent diffusion of powerful language models for code, and their integration in Integrated Development Environments (IDEs), reduced the time needed to duplicate a piece of code even further. This applies also to non-exact clones, such as those with replaced identifiers. The language model can quickly recognize the desired pattern even just after typing the first replacement, and can instantly complete the rest of the code.

2.2 Avoiding Code Clones With Code Reuse

Software engineering as a discipline realized a long time ago the possibility of writing programs in a modular sense [12, 16]. These "modules" have been variously called subroutines, procedures, or functions; fundamentally, they are abstractions. Modern programming languages support several forms of abstractions, including some more elaborate than the ones mentioned above, such as classes.

In this work we will focus on functions, as they are a simple but powerful form of abstraction that is suitable for novices in a school context. Functions can offer "configuration options" [16] through parameters, to accommodate the differences in behavior that are required in different parts of the program.

When the code to achieve a certain functionality is abstracted as a function, it can be *reused* by calling that function in multiple places within the program, every time one needs that functionality.

The next logical step is to reuse code across programs. Indeed, programmers are familiar with the idea of using functions from a *library*. Programming languages come by default with libraries containing several functions deemed useful in many contexts (a library for operations with dates and times, for example).

2.3 Environments Not Always Favor Code Reuse

Code reuse is however not on the path of least resistance: features of existing IDEs can discourage reuse and instead lead to code clones.

2.3.1 Code Snippets. Programmers, both novice and experienced ones, can find themselves not having a clear idea of how to solve a problem. In most cases, that specific problem—or a very related one—has already been solved by someone else, and thus the programmer can try to obtain a fragment of code, either to include as is or to adapt with minor modifications [2].

These code snippets can be obtained in different ways from several different sources, such as code-repository sharing platforms (e.g., GitHub), Q&A platforms (e.g., StackOverflow), code-snippets sharing platforms (e.g., GitHub Gist), and language models, that generate code fragments based on the code they have been trained on (which was in turn sourced from the aforementioned platforms).

Moreover, some environments for notebooks, such as Google Colab, allow users to directly inject pieces of code from a collection of "snippets": fragments of code to solve recurring programming problems. The environment comes with a selection of pre-written snippets, but also allows users to save and retrieve their own.

2.3.2 Code Templates. Online platforms are not the only source of fragments of code: IDEs also provide features to facilitate the repeated insertion of specific *code templates*. Templates are more powerful than snippets because they may contain *holes* to be filled in by the programmer when they want to include them. When code templates become fully concrete, they effectively turn into snippets.

Some IDEs use code templates to assist in writing "boilerplate code", i.e., repetitive patterns of code. For example, the `for` keyword can be automatically expanded to the full skeleton of the for loop statement in a language like C, and the keyword `class` can be turned into a complete class declaration in Java.

More advanced environments also enable developers to define their own custom templates. These features go under different names, such as IntelliJ's Live Templates and VSCode's Snippets.

Code templates can go even further: some IDEs employ static analysis techniques to generate code leveraging information extracted from existing code. For instance, IDEs such as Eclipse or IntelliJ IDEA include a widely used feature that generates an implementation of the `equals` and `hashCode` methods for a class by inspecting the fields declared by the programmer.

2.3.3 Remixing in Scratch. Code clones are not exclusive to text-based programming languages. A study on Scratch, a popular choice to teach programming in schools using blocks, showed that Scratch projects contain code clones quite pervasively, and that functions as abstractions are rarely used [1].

The Scratch platform promotes taking an entire published project and "remixing" it. This is yet another example of an environment that favors code duplication over reuse: instead of being able to import a well-defined abstraction, such as a sequence of blocks packaged in a function, learners are nudged to "fork" [9] an entire project, ending up with code clones.

2.3.4 Complexity of Multi-File Projects. School teachers use different kinds of IDEs: some choose to work with novice-oriented IDEs such as Thonny or BlueJ, some work with full-fledged ones such as Visual Studio Code, and others adopt web-based programming platforms, which range from barebone ones like Ideone to sophisticated ones like GitHub's Codespaces.

No matter the setup, it is non-trivial to reuse code beyond one single file. IDEs may structure a project into several files, potentially divided into several folders. This complexity is one of the reasons behind the need for a build system: a tool that helps programmers develop programs consisting of multiple files. Reusing code across files requires configuring the environment to handle this setup. This may require configuring the IDE, adopting a precise structure for files and folders, and following the non-obvious importing rules that deal with relative or absolute paths.

The problem is exacerbated when one wants to reuse code across different projects. This requires refactoring code into a separate *library*, which can then be imported into multiple projects. Importing libraries outside those that come standard with a programming language is often a source of pain, even for experienced developers.

Understandably, teachers want to eschew all these problems and have their students spend the limited time available on actual programming, as opposed to configuring an environment. As a result, students may be instructed to limit themselves to using only a single file, be that a real file on their device or a virtual one in a web-based text editor.

Unfortunately, this deprives learners of the opportunity to get acquainted with code reuse in the simple context of programs with a modest size. Having students practice code reuse early on teaches them the right practices, which they can then apply to larger software engineering projects.

2.4 Assignments Not Always Favor Code Reuse

Teachers of programming courses commonly design assignments for their students to practice programming skills. At the beginning of an introductory course, a programming assignment typically consists of small, independent exercises. As the course progresses, an assignment can become a small project. Still, most assignments do not reuse solutions from earlier work.

Students can find themselves re-implementing the same functionality multiple times, effectively writing code clones across assignments. Instead, learners should be guided to decompose the solution of each assignment into functions, identify the ones that are general and potentially useful for other problems, and use these functions again in the subsequent assignments. This way, students could learn and practice code reuse in a controlled context.

3 An Approach to Promote Code Reuse

> A wise handyperson knows the advantage of having the right tool ready to deal with a certain situation, and always carries a *toolbox* containing various useful items. Based on this analogy, we propose that beginner programmers should keep a *Toolbox of Functions (ToF)* at their disposal: an always-available library of functions that are potentially useful to solve future tasks.

When students recognize the potential general applicability of a function they defined (or the instructions in an assignment recommend doing so), they should add that function to their personal ToF. Applying on a small scale what a good software engineer would do, adding a function to the ToF requires some polishing, to become a reusable abstraction. The learner should identify all the dependencies of the function, such as other functions called or global constants used, and keep them alongside the function they intend to save. Then, students should consider improving the names of the function and its parameters to ensure they are descriptive, annotate the signature with types (for statically typed languages), and add documentation that reminds their future selves of what the function is supposed to do. Optionally, they could also add some tests to ensure that the function they extracted works as intended.

As the ToF grows, teachers can ask students to implement more elaborate programs, counting on the fact that they have already implemented certain functions which are ready to be used. Students can quickly solve parts of a larger assignment by importing functions from their ToF, call them with the right arguments, and then focus on the rest of the program.

This practice empowers students with a sense of satisfaction and purpose that derives from reusing—instead of throwing away—the code they had to implement earlier for a different problem. And this is achieved without any code clone or copy-paste activity.

4 Toolbox of Functions: Instantiation

The ToF approach we described is independent of both the programming language and the environment. In this section, we describe one specific instantiation of the approach, as it is implemented in PyTamaro Web, a publicly available web-based Python programming environment at https://pytamaro.si.usi.ch and centered around the PyTamaro graphics library [5].

4.1 Context

The PyTamaro Web platform offers a rich environment, with web pages offering guided programming activities that contain "code cells" interleaved with text and pictures, similar to a Jupyter notebook. The platform includes over 200 guided activities, many of which were contributed by high school teachers.

Activities can be organized in a curriculum. A curriculum is a sequence of activities that students may need to go through in order. Some activities may be optional, and sometimes students can select one of multiple alternative activities. While some curricula consist of only a handful of activities, some teacher-designed curricula cover an entire semester of programming activities.

Through a curriculum and its activities, the teacher can structure the learning process. They can gradually introduce language constructs and library functions, providing an increasingly richer programming experience.

As a small-scale running example for demonstration purposes, we will consider a tiny fictional curriculum with two activities, which ask students to draw an "eye" and a "no entry" traffic sign.

4.2 Introducing Functions

When a student solves a programming activity, code clones may appear, possibly due to copy-paste. This often occurs when working with a minimal educational library such as PyTamaro, which does not include functions like `square` or `circle` to encourage students to define their own starting from `rectangle` or `ellipse`.

Code clones can occur within a single code cell or can be scattered across multiple cells. Students can be instructed to observe the similarities between multiple clones and identify the few differences. They can then define a function with a parameter for each of these differences, and with the body containing the clone, where the differences are replaced by the parameters. Finally, each clone can be replaced with a simple call to the newly defined function.

Defining functions enables code reuse within a single activity. The result of this abstraction process is shown in Figure 1a for the "eye" activity, in which students can extract a function to create a circle of a given radius and use it twice to draw the pupil and the iris of the eye. Teachers can scaffold this process by providing interleaved explanations and setting up code cells accordingly.

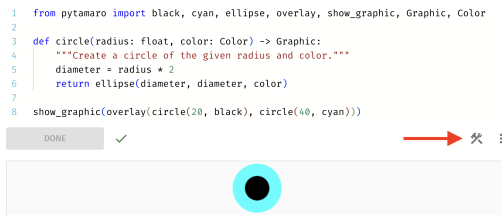
4.3 Adding to the Toolbox of Functions

Once the student has defined their own function, they may want to save it for later use by adding it to the ToF.

When the code written by a student contains the definition of at least one function, a button with the icon of a handyperson's tools appears automatically (Figure 1a). This automatism is made possible by statically analyzing the Python code directly in the browser, using a Rust library compiled into WebAssembly.

Clicking on that button opens a popup with instructions on how to save a function to the ToF (Figure 1b). Following the approach described in Section 3, learners should:

(1) Identify all the dependencies of the function they want to save. These may include `import` statements or global constants to define the names that are used in the function body and other functions that are called by the function to be saved. Other pieces of code are unnecessary and should be



(a) Students define the `circle` function to solve the "eye" activity. They can save it by clicking on the ToF button (red arrow).

Implementation

Clean up your code before saving a function to the toolbox. Here are some tips:
- Remove any code that is not needed by the function you want to save
- Remove calls that produce output (e.g., calls to print or show_graphic)
- Remove unnecessary imports (move them all to the top, combine them and get rid of unneeded ones)

```
1 from pytamaro import black, cyan, ellipse, overlay, show_graphic, Graphic, Color
2
3 def circle(radius: float, color: Color) -> Graphic:
4     """Create a circle of the given radius and color."""
5     diameter = radius * 2
6     return ellipse(diameter, diameter, color)
```

(b) A popup opens. In the first part of the popup, instructions guide students in cleaning up their code to make the function reusable.

Function to save | Function: circle | Create a circle of the given radius and color.

Document what this function does

Example that uses this function

```
1 from toolbox import circle
2 from pytamaro import show_graphic, green
3
4 show_graphic(circle(10, green))
```

DONE

SAVE TO TOOLBOX

(c) In the second part of the popup, students have to write minimal documentation, implement and run an example program before being allowed to add the function to the ToF.

Figure 1: Adding `circle` to the ToF in the "eye" activity.

removed, such as possible calls to the function being saved, as well as any code used for debugging (e.g., `print`).

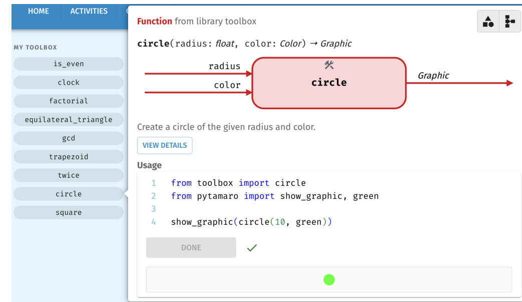
(2) Write a description to remember what the function does, as a minimal form of documentation. When a function contains a docstring written following the PEP 257 convention, the ToF uses that string as documentation by default.

(3) Implement and execute a short program that calls the function. This serves a dual purpose. First, it serves as a lightweight, non-automatic form of testing, to detect, for example, leftover side effects in the function body, such as debugging statements. Second, it serves as a form of documentation on how to call that function in the future (Figure 1c).

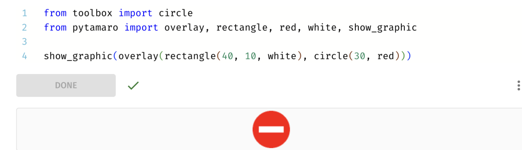
Students can then add the function to their ToF, without worrying about managing Python files to set up and maintain a library.

4.4 Using the Toolbox of Functions

The tools in a physical toolbox are always near a handyperson. Similarly, the functions in the virtual toolbox are near the programmer. On each activity page, the ToF is presented in a sidebar (left side of Figure 2a). This quick overview of the ToF realizes Victor's idea of "dumping the parts bucket onto the floor" to "encourage the programmer to explore the available functions" [14]. Students can see that they have a `circle` function that may be useful in an activity where they need to draw a "no entry" traffic sign (Figure 2b).



(a) Students explore functions in the ToF (left sidebar, always visible while solving activities). Their documentation opens in a popup.



(b) Students can import the `circle` function from the ToF just like from any other library and reuse their code in a different activity.

Figure 2: Using `circle` from the ToF in the "no entry" activity.

Moreover, by clicking on the corresponding item in the sidebar, students can quickly retrieve the documentation for each of their functions. The documentation for all the libraries, including the ToF, is shown using the novice-friendly format introduced by Chiodini et al. [4] (right side of Figure 2a). The documentation describes the various properties of the function. The function signature is detected automatically and is optionally enriched with the parameter and return types, if the original function was augmented with type annotations. The description and the usage example, instead, come directly from the learner. The example code can be executed in-place, as a reminder of the function's behavior (Figure 2a).

Once the student has identified a function to use, they can reap the benefits of code reuse with almost no effort. All that is needed is to `import` the `circle` function from the `toolbox` library and call it with the proper arguments.

4.5 Growing the Toolbox of Functions

Functions in the ToF behave exactly like all the other functions. They can be used as part of the definitions of new functions, and students can add these new functions to their ToF as well. On a small scale, this showcases how to build more powerful abstractions on top of simpler ones.

Behind the scenes, each function is stored in a separate file, to avoid possible conflicts (e.g., two functions may have been added to the ToF from two activities that defined two different constants with the same name). All the ToF functions required for a certain activity are exported from the `toolbox` Python module, which in turn imports the necessary functions. These imports follow the topological sort of the dependencies, to avoid circular imports.

All this complexity is hidden from the student, who can just focus on defining and using functions, fulfilling the goal of practicing abstraction and code reuse without any waste of time.

4.6 Managing the Toolbox of Functions

The platform also enables learners to manage their ToF. Learners can search their toolbox, remove functions from the toolbox, and modify existing functions. This is an essential feature to support students who need to fix a bug or make an improvement to their code. At the same time, it offers a sneak peek into the intricacies of library and Application Programming Interface (API) evolution—another important aspect of software engineering [10]—in a controlled setting: whenever the signature (public interface) of a function in the ToF changes, any program using it needs to be updated accordingly.

5 Classroom Experience

We implemented the ToF in the PyTamaro Web platform roughly two years ago. We first created example activities to showcase the ToF. Then we collaborated with high school teachers and explained its benefits. Over time, teachers started to integrate the ToF into their own curricula and activities they use regularly in class.

Focusing only on the last year, usage statistics show that students added more than 1 420 functions to their ToF. A total of 32 522 code executions made by 880 different users imported at least one function from their ToF in their code. The actual user count is likely underestimated, as the platform collects data only from users who give their explicit consent. However, users are simply identified by a randomly generated UUID stored in the browser's local storage, which might be reset at any time, leading to double counts.

Currently, the platform hosts more than 50 different activities that use the ToF, created by 6 different instructors. According to a teacher who actively uses the ToF in their lessons, students are keen on collecting functions and watching their ToF grow. This anecdote suggests that students can perceive curating a ToF as a benefit rather than a chore.

6 Limitations & Future Work

The ToF focuses on functions because they are both a fundamental building block to define abstractions and widely taught in schools. However, other kinds of abstractions exist, such as class definitions. The approach of the ToF could be extended to support those too.

The current model of the ToF has a flat structure. This is a deliberate choice to keep adding to and importing from the ToF as simple as possible, but it can become inadequate when the number of functions grows too large. Some form of namespacing (e.g., Python submodules) could better organize the ToF, but it would require dealing—albeit in a more controlled way—with files and folders, which can be a pain point for students (cf. Section 2.3.4).

Changing the signature of a function stored in the ToF may break all the code that depends on it. Students should be warned of the risks, making them aware of the consequences of changing a public API. An expansion of the ToF could be used to teach more advanced software engineering principles related to code reuse, such as API versioning and the concept of third–party dependencies. This could also open the possibility of introducing a code-sharing mechanism, allowing students to reuse the code already written by their peers.

7 Conclusion

In this paper we presented the Toolbox of Functions (ToF), an approach for teaching code reuse in schools in a simple way.

We described how existing IDEs do not always favor code reuse and may even encourage code clones. To counter that and teach proper abstractions and code reuse, we introduced the ToF approach, which we implemented in a web platform to enable a simple form of reuse with the simplest of abstractions: functions. Multiple high-school teachers adopted the ToF in their materials for mandatory programming courses. Initial usage statistics and anecdotal feedback suggest that it can be an effective approach to help teach code reuse as an important software engineering principle.

Acknowledgments

This work was partially funded by the Swiss National Science Foundation project 200021_184689.

References

[1] Efthimia Aivaloglou and Felienne Hermans. 2016. How Kids Code and How We Know: An Exploratory Study on the Scratch Repository. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*. ACM, Melbourne VIC Australia, 53–61. doi:10.1145/2960310.2960325

[2] Sebastian Baltes and Stephan Diehl. 2019. Usage and Attribution of Stack Overflow Code Snippets in GitHub Projects. *Empirical Software Engineering* 24, 3 (June 2019), 1259–1295. doi:10.1007/s10664-018-9650-5

[3] Sebastian Baltes and Christoph Treude. 2020. Code Duplication on Stack Overflow. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '20)*. ACM, New York, NY, USA, 13–16. doi:10.1145/3377816.3381744

[4] Luca Chiodini, Simone Piatti, and Matthias Hauswirth. 2024. Judicious: API Documentation for Novices. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2024)*. Association for Computing Machinery, New York, NY, USA, 1–9. doi:10.1145/3689493.3689987

[5] Luca Chiodini, Juha Sorva, and Matthias Hauswirth. 2023. Teaching Programming with Graphics: Pitfalls and a Solution. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E (SPLASH-E 2023)*. ACM, New York, NY, USA, 1–12. doi:10.1145/3622780.3623644

[6] Cory J. Kapser and Michael W. Godfrey. 2008. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering* 13, 6 (Dec. 2008), 645–692. doi:10.1007/s10664-008-9076-6

[7] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring. 2017. Code Quality Issues in Student Programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE '17)*. ACM, New York, NY, USA, 110–115. doi:10.1145/3059009.3059061

[8] Andreas P. Koenzen, Neil A. Ernst, and Margaret-Anne D. Storey. 2020. Code Duplication and Reuse in Jupyter Notebooks. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. ACM, New York, NY, USA, 1–9. doi:10.1109/VL/HCC50065.2020.9127202

[9] Rainer Koschke. 2007. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software (Dagstuhl Seminar Proceedings (DagSemProc), Vol. 6301)*. Schloss Dagstuhl, Dagstuhl, Germany, 1–24. doi:10.4230/DagSemProc.06301.13

[10] Maxime Lamothe, Yann-Gaël Guéhéneuc, and Weiyi Shang. 2021. A Systematic Review of API Evolution Literature. *ACM Comput. Surv.* 54, 8 (Oct. 2021), 171:1–171:36. doi:10.1145/3470133

[11] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 84:1–84:28. doi:10.1145/3133908

[12] David Lorge Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15 (1972), 1053. doi:10.1145/361598.361623

[13] Kate Sanders and Robert McCartney. 2016. Threshold Concepts in Computing: Past, Present, and Future. In *Proceedings of the 16th Koli Calling International Conference on Computing Education Research - Koli Calling '16*. ACM Press, Koli, Finland, 91–100. doi:10.1145/2999541.2999546

[14] Bret Victor. 2012. Learnable Programming. https://worrydream.com/LearnableProgramming/

[15] A. Vihavainen, J. Helminen, and P. Ihantola. 2014. How Novices Tackle Their First Lines of Code in an IDE: Analysis of Programming Session Traces. In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*. ACM, Koli Finland, 109–116. doi:10.1145/2674683.2674692

[16] D. J. Wheeler. 1952. The Use of Sub-Routines in Programmes. In *Proceedings of the 1952 ACM National Meeting (Pittsburgh) on - ACM '52*. ACM Press, Pittsburgh, Pennsylvania, 235–236. doi:10.1145/609784.609816