Revealing Programming Language Abstractions

An Excerpt of nand-to-tetris – in Reverse – Using Smalltalk

GymInf Individual Project

Simon Bünzli
from
Bern, Switzerland

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

01. August 2025

Prof. Dr. Timo Kehrer, Prof. Dr. Oscar Nierstrasz

Software Engineering Group
Institut für Informatik und angewandte Mathematik
University of Bern, Switzerland

Abstract

Not an *abstract* yet, but the original project description:

Ziel des Projekts ist, ein empirisch abgestütztes Instrument für den Programmier-Unterricht am Gymnasium zu entwickeln, in welchem Schüler:innen verschiedene Abstraktionsebenen interaktiv erleben können.

Auf der Basis von Processing mit Python Syntax (https://py.processing.org/) soll der einerseits der visuelle Ablauf eines Programms, aber auch die Parsing-Schritte und die Übersetzung in Byte-Code Seite-an-Seite sicht- und untersuchbar gemacht werden, damit Schüler:innen die Auswirkungen ihres Programmcodes auf die Maschine live erleben können.

Die Entwicklung des Produkts wird theoretisch begleitet und das Produkt selbst empirisch geprüft werden.

Als Basis der Umsetzung dient Glamorous Toolkit, eine Entwicklungsumgebung basierend auf Smalltalk/Pharo, welche u. a. von Oscar Nierstrasz für Master- und Doktoratsstudiengänge weiterentwickelt worden ist.

Contents

1
Introduction

In modern digitized society, the importance of computer science has grown to the point where some of its subjects are taught at schools of all levels. Whereas elementary schools focus on introducing digital, connected devices and their applications, high schools also teach fundamentals. And while programming or application use courses have been implemented since decades ago, broader and more theoretical courses have recently become standard. E. g. in Switzerland, computer science has become an obligatory subject for all high school students similar to more traditional sciences starting in 2019.

The curricula used usually contain introductions not only to algorithms and programming, but among others also into encodings, computer architecture, networking and social ramifications such as privacy and security (see e. g. [7]). As such, students not only are taught a high level programming language such as Python, but should also have insights into what happens at various other abstraction layers when such a program is stored and run.

One traditional approach to this consists in teaching a separate assembly-like language during the introduction to computer architecture. This can happen closer to theory like [9] or in a more gamified fashion e. g. with [2] or even without mnemonics as using the Little Man Computer architecture [3]. While all of these approaches help to show how a microprocessor might approximately work, none of them offer a direct, explorable connection to a high level language.

One suggestion for such a direct connection between high level language and machine code will be presented in this thesis. As high level language, ”Processing” based on Python syntax is chosen (which will be introduced in section 3.1), and the implementation is based on ”Glamorous Toolkit” (which will be introduced in section 3.2).

Before going into the technical details, we'll first introduce the notion of ”leaky abstractions” in chapter 2; motivate why having a direct connection over multiple abstraction layers is helpful from a dictactic point of view in section 2.2; and show how currently used development environments already help exploring such abstractions in section 2.3.

The tool introduced in this thesis is called `Processing Abstractions` and will be introduced in chapter 4. In chapter 5, we'll offer suggestions for how to employ it in the high school classroom, and in chapter 6 student feedback from two trial runs is discussed.

All of this wouldn't have been possible without the very helpful support of Prof. em. Oscar Nierstraz who has finally managed to introduce me to Smalltalk and Prof. Timo Kehrer who has taken this project

below his wing. I'd also want to thank my students from the classes 27Ga and 28Ga who have worked with my productions and given helpful feedback.

2
Leaky Abstractions when Teaching Programming

Before introducing the product of this thesis in chapter 4, we first introduce the problem it should help solve: How abstractions involved in programming are taught.

In detail, we first introduce the concept of (leaky) abstractions in multitier architectures in section 2.1; show how these are discussed in didactic literature in sections 2.2; and how common IDEs already support handling these difficulties (in section 2.3).

2.1 Multitier Architectures and (Leaky) Abstractions

In order to handle complexities arising in both theoretical and practical computer science, subjects are split into multiple layers or tiers to be described, investigated and used separatedly. **¡citation needed?!**

Common such multitier architectures taught at high school level are the networking stack (either the seven layered OSI model or the simplified four layered DoD architecture) or the software-hardware stack ranging from apps and hardware abstracting OS down to transistors consisting of e. g. silicium atoms.

¡diagram of such an architecture?!

Ideally, in such architectures all layers above the layer to be investigated can be ignored (beyond what the layer will be used for) and all the layers below can be abstracted away into a nicely defined interface.

As such, programming should be possible to be done independently of hardware and even the operating system, in the same way that natural languages can be taught independently of body or mind of the students.

In his article ”The Law of Leaky Abstractions” [19] introduces the concept of *leaky abstractions*, claiming that for all non-trivial such architectures, details of lower layers are to some degree bound to bleed through to upper layers. In other words, in practice complex interfaces tend to be incomplete or ’leaky’.

In teaching computer science, such leaky abstractions occur repeatedly, e. g. when an app doesn’t run on a different device (with either the OS or the processor architecture leaking); or when a document seemingly can’t be saved (with either the file system or differences between apps leaking).

More specifically, in programming there are several ways of abstracting away technical details:

- Programming instructions consist of source code which consists of encoded bits which are stored in memory or on a drive.
- Source code consists of tokens which are usually parsed into an abstract syntax tree (AST) which are either directly or via intermediary representations translated into machine code to be run on a virtual or actual machine.
- When programming instructions through the above abstractions are executed, variable values are encoded and stored in memory, function calls are tracked through a call stack, input state is continually mapped into memory and output is generated in several forms – where e.g. textual output causes a font renderer to interpret glyph instructions for every character; or graphical output is anti-aliased before any pixel data is produced.

¡citation needed?!

Of these different layers, students usually focus on turining instructions into source code and then checking the program's output – or any error messages produced by the compiler or interpreter (see section 2.2). Still, several of the lower layered abstractions might leak through, such as:

- Missing a stop condition in a recursive function leads to a cryptic "Stack overflow" error – leaking information about the call stack.
- If a program outputs emojis, they might look notably differently in source code and output – leaking font rendering.
- Similarly, programs containing emojis might have emojis garbled depending on the app used for inspecting the source code – leaking text encoding.
- If a program contains an endless loop, there might be neither error message nor output, so that it might wrongly seem that the computer isn't doing anything. This isn't an abstraction leak in the above sense but a related student misconception.

2.2 Didactic Approaches

See e. g. [18], [14], [11] or [13] only focusing on one aspect

2.2.1 Teaching Top Down

Working downwards from gaming, as in [22]

2.2.2 Teaching Bottom Up

Running Tetris on NANDs as described in [21], [16]

2.3 Abstractions in IDEs

Integrated development environments used for programming offer a variety of different views on a program beyond its source code and its runtime output. The popular Visual Studio Code offers e. g. through extensions step-by-step debugging with variables and the call stack listed [6]. This is mirrored in most other full fledged IDEs such as PyCharm [1] or Eclipse [5].

And while such IDEs through appropriate extensions even allow inspecting Python bytecode, the respective views are usually overwhelming for programming novices and thus rather targetted at professional developers than high school students.

As a remedy, several teaching oriented IDEs have been developped, such as ”Code with Mu” which offers a minimal command set and still allows runtime inspection [20]; or Thonny which had the goal to visualize runtime concepts beyond what IDEs offered at the time [8, p. 119]:

On the one hand, Thonny shows intermediary steps during expression evaluation. This demonstrates that statements are not evaluated in one go, but indeed in a predetermined order operation by operation.[1]

On the other hand, Thonny visualizes recursion by showing code in a new pop-up for every function call, so that multiple recursive function calls lead to an equivalent number of visible pop-ups. Most other IDEs rather show a call stack as in a separate view, which abstracts the stack into a list.[2]

Finally, Thonny distinguishes between values on the stack and on the heap, showing the pointer to the heap as the value actually pushed on the stack and in a separate view the actual object on the heap at the given address.

Thus, the Thonny IDE set out to and indeed nicely visualizes several concepts on lower runtime layers.

[12] has assembled a list of tools targetted at visualizing some of these concepts outside of an IDE. One noteable such alternative approach is taken by Python Tutor [4] which combines a visualization of stack frames variable values as pointers and deconstructed objects.

[1] In professional IDEs, intermediary results are usually available by hovering over a specific operator with the order of evaluation being left to the user to determine.

[2] As a compromise, Glamorous Toolkit presented in chapter 3 displays the call stack as a list of expandable method sources with the call location highlighted.

3
Technical Background

Background knowledge required for understanding the following chapters.

3.1 Processing

Brief overview over the ”Processing” programming language (along [17]) and reasons for using it.

3.2 Glamorous Toolkit

Brief introduction into GT for the uninitiated and reasons for using it. ([10])

3.3 Moldable Development

Referring to [15].

4 Processing Abstractions

4.1 Development of ”Processing Abstractions”

Excerpts from gt-exploration Lepiter pages

4.2 Abstraction Levels

For each a short problem description and a presentation of the chosen approach:

4.2.1 Source Code

4.2.2 Abstract Syntax Tree

4.2.3 Transpilation/IR

4.2.4 Machine Code

4.2.5 Output

5 Teaching with PA

How to use it

5.1 Lesson on Computer Architecture

Using PA to demonstrate what happens under the hood when running a program in a high level language.

5.2 Lesson on Compilers

Using PA to demonstrated the steps of lexing, parsing, transpiling, compiling and optimizing.

5.3 Lesson on Introduction to Programming

Using PA as a live programming environment.

6

PA in Practice

PA has been used twice with students (on 2025-05-12 and 2025-06-30).

6.1 First Round

6.1.1 Setting

6.1.2 Observations

6.1.3 Student Feedback

6.1.4 Learnings

6.2 Second Round

6.2.1 Setting

6.2.2 Observations

6.2.3 Student Feedback

6.2.4 Learnings

7
Conclusion

7.1 Future Work

A
Installing and Using `Processing Abstractions`

B
Data from Questionnaires

Bibliography

[1] Debug your python code with pycharm. https://www.jetbrains.com/pycharm/features/debugger.html.

[2] Human resource machine: Hour of code edition. https://tomorrowcorporation.com/human-resource-machine-hour-of-code-edition.

[3] Little man computer. https://oinf.ch/interactive/little-man-computer/.

[4] Online compiler, ai tutor, and visual debugger for python, java, c, c++, and javascript. https://pythontutor.com/.

[5] Pydev - python ide for eclipse. https://marketplace.eclipse.org/content/pydev-python-ide-eclipse.

[6] Python debugging in vs code. https://code.visualstudio.com/docs/python/debugging.

[7] Lehrplan 17 für den gymnasialen bildungsgang, 2016.

[8] Aivar Annamaa. Introducing thonny, a python ide for learning programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 117–121, New York, NY, USA, 2015. Association for Computing Machinery.

[9] Lorenzo Ganni. Von neumann machine simulator, 2023. https://vnsim.lehrerlezius.de/.

[10] Tudor Gîrba, Oscar Nierstrasz, et al. *Glamorous Toolkit*. feenk.

[11] Stefan Hartinger. *Programmieren in Python*. Universität Regensburg, 2020.

[12] Maryam Jalalitabar and Yang Wang. Demystifying the abstractness: Teaching programming concepts with visualization. In *Proceedings of the 23rd Annual Conference on Information Technology Education*, SIGITE '22, pages 134–136, New York, NY, USA, 2022. Association for Computing Machinery.

[13] Irene Lee, Shuchi Grover, Fred Martin, Sarita Pillai, and Joyce Malyn-Smith. Computational thinking from a disciplinary perspective: Integrating computational thinking in k-12 science, technology, engineering, and mathematics education. *Journal of science education and technology*, 29(1):1–8, 2020.

[14] Eckart Modrow and Kerstin Strecker. *Didaktik der Informatik*. De Gruyter Studium. De Gruyter Oldenbourg, München, 2016.

[15] Oscar Nierstrasz and Tudor Gîrba. Moldable development patterns. 2024.

[16] Noam Nisan and Shimon Schocken. *The elements of computing systems : building a modern computer from first principles*. The MIT Press, Cambridge, Massachusetts, second edition edition, 2021 - 2021.

[17] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. The MIT Press, 2 edition, 2014.

[18] Sigrid Schubert and Andreas Schwill. *Didaktik der Informatik*. Spektrum Akademischer Verlag, Heidelberg, 2. auflage edition, 2011.

[19] Joel Spolsky. The law of leaky abstractions, 2002. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

[20] Nicholas H. Tollervey. The visual debugger, 2023. https://codewith.mu/en/tutorials/1.2/debugger.

[21] Ünal Çakıroğlu and Mücahit Öztürk. Flipped classroom with problem based activities: Exploring self-regulated learning in a programming language course. *Journal of Educational Technology & Society*, 20(1):337–349, 2017.

[22] David Weintrop and Uri Wilensky. Playing by programming: Making gameplay a programming activity. *Educational Technology*, 56(3):36–41, 2016.