GymInf | Informatikausbildung für Gymnasiallehrkräfte | Formation en informatique destinée aux enseignant-e-s au gymnase

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

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

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 2.1; and show how currently used development environments already help exploring such abstractions in section 2.2.

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 of Gymnasium Neufeld who have worked with my productions and given helpful feedback.

¡expand, include full motivation, product overview and results (expanded from abstract)!

2

State of the Art in Computer Science Didactics

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 discuss didactic literature on teaching computer science at high school level in section 2.1; then we review tools used for programming in such courses in section 2.2; finally, in 2.3 we introduce the motivation for this thesis.

2.1 Didactic Approaches

The abstractions encountered in programming as discussed in section 2.3, are not always handled in didactic literature.

Traditional introductions into programming focus on the language, its syntax and the available semantics. E. g. [11] introduces most of Python so that it can later be used for scientific calculations. This traditional introduction omits many of the lower abstraction levels listed in the previous section. Interestingly, it does not assume an IDE (see section 2.2 below), but instead very briefly introduces the command line and the Python REPL[1]. Even though in that way, files are not guaranteed to be UTF-8 encoded (as examples assume, cf. p. 118), encodings are not mentioned beyond pure ASCII (p. 115). And the Python interpreter is just in the preface briefly characterized as "program which translates source code into electronic instructions"[2].

In the context of most introductions to programming, that might be sufficient with relation to available time and goal of the course. In broader teaching context such as high school CS courses, didactic teachers ask for more. [18] proposes an introduction into programming to use several different programming languages along different paradigma: e. g. Prolog as a declarative language (pp. 91–104) and Python as object oriented language (pp. 157–185), in order to demonstrate to students already at the level of instructions that the language chosen comes with inherent limitations in expressibility (p. 154, comparing programming languages with natural languages and referring to Wittgenstein's philosophy of language).

[1] REPL is short for *Read-Evaluate-Print-Loop* i. e. an interactive interpreter.

[2] German original: "[...] das Programm, das den Programmcode in Elektronik-Anweisungen übersetzt."

2.1.1 *Sichtenwechsel*

Furthermore in a broader context, they propose to explicitly discuss multitier architectures – and that not only in the context of the networking stack and computer archictecture (pp. 113–116):

They introduce the notion of *Sichtenwechsel*, a change of perspective with relation to the current layer, which should help students better understand concepts of one layer by inspecting lower layers. As an example, the show a live model of a calculator app whose input is translated to both pseudo code and machine code (p. 115); an environment for inspecting live Java objects (pp. 208–209); or the Filius environment for inspecting a virtual computer network at its different layers (p. 284).

They conclude that "it was a fallacy to assume that students would be able to develop a working model of a computer [...] by designing small programs" (p. 213),[3] reinforcing the need for going beyond of what traditional programming courses (used to) do.

2.1.2 Teaching Bottom Up

Running Tetris on NANDs as described in [21], [15]; basing every layer on the layers below

2.1.3 Teaching Top Down

Working downwards from gaming, as in [22], or art as in [17]; starting at motivating examples and then extending unterstanding and capabilites

2.1.4 Exploratory and Live Programming

Letting students explore concepts on their own – with live feedback from the environment

2.2 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.[4]

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.[5]

[3] German original: "Es erwies sich als Irrtum, dass Schüler beim Entwerfen kleiner Programme ein tragfähiges kognitives Modell vom Rechner oder von Informatiksystemen im Allgemeinen entwickeln."

[4] 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.

[5] 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.

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.

2.3 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 2.1). 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.

Besides the above rather easily observable abstraction leaks, the issue might also have to be discussed itself, since one class of leaky abstractions has been shown to be security critical: timing attacks. Since programs might be compiled differently and optimized differently and run on different hardware, runtime timing is not considered to be inherent to a particular source code.[6]

In cryptography, timing attacks have been successfully used for extracting passwords from insufficiently protected webservers [16]. More recently, another class of timing attacks taking advantage of modern CPU's branch prediction optimizations has been demonstrated [13]. In the latter case, an implementation detail of the CPU managed to leak. And in both cases, at least implementors of cryptographic programs must be aware of lower abstraction layers.

[6] At least beyond generic complexity considerations on an algorithmic level.

3 Technical Background

Background knowledge required for understanding the following chapters.

3.1 Processing

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

- Processing is taught using a top down approach (cf. 2.1.3)
- Processing is an imperative language with visual primitives, allowing for quick visual results
- Developed in the early 2000s at MIT Media Lab, based on then popular Java
- Since Python has become the prevalent teaching language (**¡citation needed!**), Processing has been extended with a Python mode
- This allows a seamless transition to using all of Python
- The original IDE is still based on the JRE and transpile code to Java
- Built-in structure for animations, interaction, *etc.*
- Some basic code examples: ...
- Quick results possible: Flappy Bird, Pong, ...
- Own experience since the 2010s, originally using `https://software.zeniko.ch/ProcessingIDE.zip`
- Newer alternatives are p5.js, p5.py, *etc.*

3.2 Glamorous Toolkit

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

- GT is a fully programmable environment (similar to Wirth's Oberon)
- Origins of GT in Smalltalk, Squeak, Pharo
- Easy to inspect, adapt, extend
- Developed by feenk (nod to Oscar)
- Base concepts: ¡gtView¿, ¡gtExample¿, ...

3.3 Moldable Development

Referring to Nierstrasz and Gîrba [14].

- Tool should adaptable to data
- Live exploration by writing throw-away code
- Quick refactoring for keeping useful code around

4

Proposed Solution: A New Teaching Environment

4.1 Development of ”Processing Abstractions”

Excerpts from gt-exploration Lepiter pages

- Adaptable foundation: GT
- Various approaches to run Processing: PythonBridge, interpreter, compiler, transpiler
- Class hierarchy

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 Implementation: Lesson Plans

In its current form, `Processing Abstractions` as presented in chapter 4 is mainly targetted at the obligatory introduction to computer sciences at high school level.

Before going into empirical results from using `PA` in two courses, three lesson plans will be presented for which `PA` has been developed: a *Sichtenwechsel* in computer architecture (section 5.1); an introduction into the inner workings of a compiler (section 5.2); and a plan for a general introduction to programming (section 5.3). Some ideas for how to expand it for other school levels will be presented in section 5.4.

For all the lessons, students will need a local environment of `Processing Abstractions` installed on a computer available to them. See appendix A for how to set it up. Additionally, for non-German speaking students the contents will have to be translated to the teaching language.

5.1 Lesson on Computer Architecture

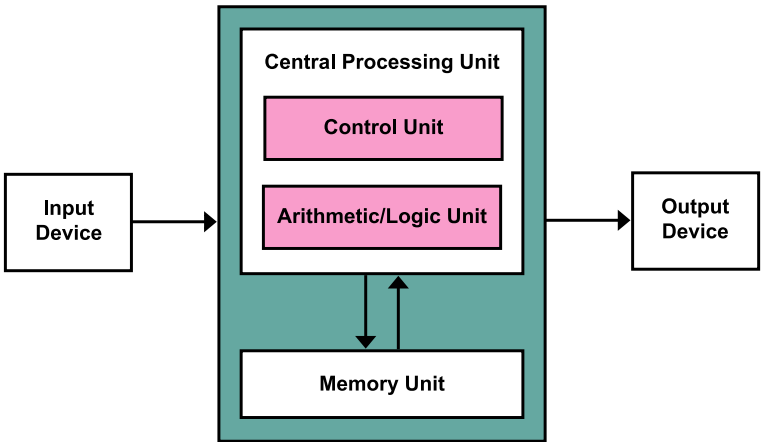
Introductions to computer science which extend beyond a pure programming course often contain lessons on computer architecture. E. g. the curriculum [7, p. 145] asks for students to "know how computers and networks are structured and work".

Now a sequence of lessons on the subject might be ordered either bottom up (as elaborated in subsection 2.1.2) or top down (2.1.3). In either case, this proposed lesson will go towards the middle or can be used at the end as part of a repetition sequence.

5.1.1 Prerequisites

Students must already know basic programming skills in a high level language such as Processing (see section 3.1). In particular, they must know about variables and loops. An introduction to programming could also be done using `PA` as outlined in 5.3 below.

The more students are supposed to work on their own, the more they'll need an overview over the different layers prior to combining them. As a prerequisite, it it recommended to at least introduce the Von Neumann architecture and its split of the CPU into control unit and arithmetic unit:



¡replace or properly attribute: Kapooht, 2013, CC BY-SA 3.0!

In a bottom up approach, this might also include the introduction of transistors, logic gates and circuits. In a top down approach, these could also be treated afterwards.

5.1.2 Lesson Plan

The goal of the lesson is for students to have connected their knowledge of high level programming with what happens within their machine when a program is executed.

If this is the student's encounter with Glamorous Toolkit, at least a brief introduction is in order (see 5.3.1). Else we can directly start with a reminder of what they already know about programming.

5.2 Lesson on Compilers

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

5.3 Introduction to Programming

Using PA as a live programming environment.

5.3.1 Introduction to Glamorous Toolkit

5.4 Further Lesson Ideas

Connecting PA with Smalltalk; extend it to object oriented programming; mould the environment to questions developed during the course; ...

6 Validation

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] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

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

[15] 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.

[16] Thomas Pornin. Constant-time crypto, 2018. https://www.bearssl.org/constanttime.html.

[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.