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. [8]). 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 [18] 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 2.2.2.

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)!

- ☐ Computer Science in High School
- ☐ based on introduction in primary school
- ☐ broad overview, not just programming, but also hardware, networking, encoding, *etc.* (see e. g. [8])
- ☐ here focus on various abstractions involved in programming

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 introduce the motivation for this thesis in 2.2; finally we review tools used for programming in such courses in section 2.2.2.

2.1 Didactic Approaches

- ☐ Other requirements for teaching programming: Manageable Complexity, Meaningful Engagement, Clean Problem Decomposition [13]
- ☐ Base requirements: Motivational learning, transparency, activating prior knowledge, multiple methods, *etc.* [17, p. 998]

Traditional introductions into programming focus on a single programming language, its syntax and the available semantics – introducing them iteratively and practising each element with basic exercises. E. g. Hartinger introduces most of Python in this way so that it can later be used for scientific calculations [20].

While the needs of academic teaching and the form of semester courses with lectures and separate lab work tends to suggest such an approach, this is not a good fit with suggestions from current didactic literature [26, 32] (and even past texts [15]):

2.1.1 *Fundamentale Ideen*

Schubert and Schwill [32] base their didactic approach around the notion of "foundational ideas" derived from Jerome Bruner and Alfred Schreiber. A foundational idea is for them a concept with the following properties (pp. 62–63):

- Breath: the concept is applicable not just in one specific context but can be used more generally.

- Abundance: The concept can be applied in different ways.
- Meaning: The concept is meaningful to the learner beyond the scope of a course.

Such foundational ideas are among many others the idea of modularization, the idea of layered archictectures or the idea of encoding information and instructions.

As a consequence, they propose 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 better teach the foundational idea of 'language' and 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).

2.1.2 *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),[1] reinforcing the need for going beyond of what traditional programming courses (used to) do.

2.1.3 Manageability

Modrow and Strecker [26] follow Schubert and Schwill [32] in also building upon the concept of foundational ideas. They do however propose to significantly reduce their number and focus on a few very general such ideas such as modelling (*Modellierbarkeit*), connectivity (*Vernetzbarkeit*), digitization (*Digitalisierbarkeit*) and algorithms (*Algorithmisierbarkeit*) (pp. 27–37). As a consequence, they do propose to focus programming exposure for high school students to block based programming languages such as MIT's Scratch or Berkeley's variant Snap! (p. 125).

Their reasoning for this is that students should not (yet) have to deal with syntax errors (as opposed to teaching them as suggested by Bouvier *et al.* [12]) and only have a manageable command palette. Furthermore, they note that block based languages with free form layouting encourages to build complex behavior from simple building blocks which can always be run and inspected individually (pp. 184–185). This allows for a bottom up development approach in which abstractions are incrementally developed out of basic command blocks which they deem more suiteable for students than starting development at the abstract algorithm.

While Modrow and Strecker do suggest also including digital circuits in the curriculum as concrete examples of digitization (p. 118), they seem content in programming them without inspecting the layers in between. From their foundational idea of connectivity, treating these layers could however still follow: if students are to see how hardware and software are connected – and that such a connection is even possible –, intermediary steps between program and hardware must be explorable by students.

Chiodini *et al.* [13] similarly state as requirements for programming classes:

[1] 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."

- Manageable complexity: IDEs and APIs must not be unnecessarily complex so as to not confuse students.
- Meaningful engagement: Samples and exercises should be introduced such that it's clear what they refer to outside of class. Their usefulness shouldn't end at the exam.
- Clean problem decomposition: Bottom up development should be effortless and code should compose with as little refactoring as possible.

2.1.4 Teaching Bottom Up

Beyond suggesting to connect multiple abstraction layers in a *Sichtenwechsel* (see 2.1.2), general didactic literature does not offer more specific suggestions. There are however two readily available ways to deal with abstraction layers: Either starting at the bottom and building abstractions on top; or starting at the most abstract and disecting it into its more fundamental forms.

Starting at the bottom conceptually seems to be the more sound way and is e. g. how Mathematics are taught. One rigorous implementation of this approach is offered by Nisan and Schocken [28]: They offer a course which starts with basic logic gates and builds out of them first the parts of and later a fully functioning, basic CPU for which they continue to develop a low level and a high level programming language until reaching the point where applications can be run on the developed hardware (this was originally dubbed as "NAND to Tetris").

Their motivation is the same as the motivation for this thesis: "The most fundamental ideas [...] are now hidden under many layers of obscure interfaces" (p. ix). Since the course is taught at university with hundreds of students, once concession is hardware virtualization: The original logic gates are not built out of silicon or electronics but instead simulated in a portable Java app. This does allow skipping intermediary steps and allows to start the course at any desired level.

Since working with logic gates without seeing their eventual purpose may lack motivation, the course starts with an overview from the top (pp. 1–4) which also serves as the table of contents. With this, students keep in sight what they're working towards and can start connecting their own preexisting notions with the new material.

2.1.5 Teaching Top Down

An alternative teaching approach starts directly at the students' experience, e. g. at gaming [37], art [30] or even toy houses [26] and starts exposing lower abstraction layers as required for gaining more control (and understanding) and extending available capabilities.

At this point a rather traditional approach starts with games: Weintrop and Wilensky [37] discuss a variety of games where programming plays a role in either shaping an avatar, improving its available actions or make it move altogether. Whereas such games are specifically created as 'pedagogical' games, many other games have at least part of their logic implemented in scripting languages such as Lua [7] which allows them to be modified by players. While the content involved is more complex, modifying or creating a game within still popular 'Roblox' might be sufficiently motivating.

While gaming works as an approach into programming, it's rarely used for inspecting further layers in an educational context. Motivation for doing so is usually constricted to performance optimizations for game platform developers.

An alternative approach which was originally targetted at art students but works well at high school as well is proposed by Reas and Fry [**?**]: Starting from visual arts and extending the capabilities of the artist through digital means. While the involved programming language, Processing, will be discussed in its own merit in 3.1, their didactic approach is notable as well:

A work of art on its own is something abstract which can not only be interpreted but also created. For (re)creating it, various painting techniques are needed which can be further broken down to basic movements. At this abstraction layer, they set in with high-level programming primitives for creating basic shapes. This allows them to achieve pleasing visual results with just a few commands and initially barely any programming knowledge. Afterwards, the question as to how to achieve more complex output is naturally motivated by already discussed more complex works of art.

Additionally, as art can be considered as individual expression, the parallel to programming as individual expression (as also proposed by Modrow and Strecker [26]) and programming as art is easily drawn: "To use a tool on a computer, you need do little more than point and click; to create a tool, you must understand the arcane art of computer programming." (p. 3). Finally, programmed art must not remain purely digital and can be extended into physical sculptures, for which at least some considerations about hardware can be introduced.

While in both approaches – gaming and art –, stepping further down to lower abstraction levels is a possibility, only one or two such steps are naturally motivated from the source material. Nonetheless, in both cases a *Sichtenwechsel* is possible.

2.1.6 Exploratory and Live Programming

Live programming refers to output and other intermediary products being adapted or recalculated every time source code changes, without any explicit saving and/or rerunning required by the programmer [31]. This gives students the quickest results, as otherwise they might tend to work on code too long without occasionally testing it, if doing so requires additional interactions (for the same reason, many applications have switched to auto-saving instead of relying on users to do so manually from time to time). Live programming also gives students immediate feedback about their code and modifications.

Exploratory programming on the other hand refers to students being given sample code and then modifying said code in order to figure out what effects their modifications have, building like this a mental model of what the code performs.

Both Schubert and Schwill [32, p. 367] and Modrow and Strecker [26, p. 167] suggest exploratory programming for allowing students to work at their own speed and depth: With given examples, less experienced students can stay closer to the given – simple but working – code, while more experienced students can use their knowledge for testing more complicated hypotheses.

While live programming requires explicit support from the programming environment (see 2.2.2), exploratory programming can be done without. Apps can still support exploration better by providing a form of REPL[2] which most scripting languages do, an interactive notebook (such as Jupyter or Lepiter, see 3.2) or a way of direclty running any part of a program, as MIT's Scratch does (see 2.1.3).

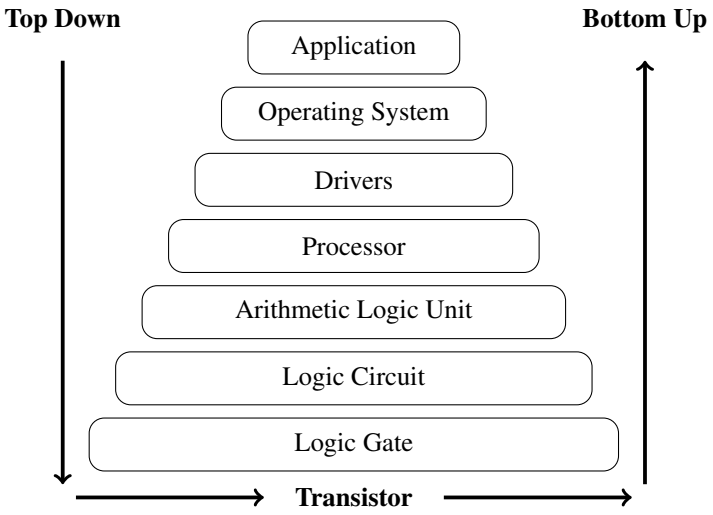
The didactic reasons for using both exploratory and live programming also apply outside or pure programming, such as when inspecting and modifying live systems, networks, *etc.*.

2.2 Multitier Architectures / 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.

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.

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



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.

2.2.1 Leaky Abstractions

In his article "The Law of Leaky Abstractions" [33] 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.[3]

In cryptography, timing attacks have been successfully used for extracting passwords from insufficiently protected webservers [29]. More recently, another class of timing attacks taking advantage of modern CPU's branch prediction optimizations has been demonstrated [23]. 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.

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

□ Further examples [16]

□ Philosophical problem of duality [14]: "At the grossest physical level, a computer process is a series of changes in the state of a machine" [14, p. 12]

2.2.2 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 [35]; or Thonny which had the goal to visualize runtime concepts beyond what IDEs offered at the time [10, 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] At least beyond generic complexity considerations on an algorithmic level.

[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.

[21] 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.

- ☐ Other helpful IDE views: syntax hints ([34]), Programming Error Message interpretation ([12]), LLM interpretation ([12, p. 28]), algorithm visualization ([21])

[20] This traditional introduction omits many of the lower abstraction levels listed in the previous section. Interestingly, it does not assume an IDE (see section 2.2.2 below), but instead very briefly introduces the command line and the Python REPL. 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"[6].

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 – as seen above.

- ☐ [22, p. 51] combine abstractions with non-abstractions
- ☐ Computational thinking: [24]
- ☐ [**?**] complex world, IT abstractions required in other fields, focus on CT skills
- ☐ [25] adjustments required due to LLMs?
- ☐ [36] peer assessment
- ☐ [38] Neuroeducation: explaining on different abstraction levels for better understanding

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

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 [30]) and reasons for using it.

- ☐ Processing is taught using a top down approach (cf. 2.1.5)
- ☐ 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 ([9]), 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. [19]

- ☐ 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>`, ...
- ☐ Currently tied to Smalltalk, with other languages such as Python, JavaScript, Java supported through bridges
- ☐ Development happens mainly under MacOS, thus Windows integration lags behind (visible e. g. when using PythonBridge)

3.3 Moldable Development

Referring to Nierstrasz and Gîrba [27].

- ☐ Alternative to inspecting static source code or live runtime objects
- ☐ Tool should be adaptable to data
- ☐ Live exploration by writing throw-away code
- ☐ Compare to web development through Web Developer console
- ☐ Quick refactoring for keeping useful code around
- ☐ Development patterns: Moldable Tool, Moldable Object, Throwaway Analysis Tool, Custom View, ...
- ☐ Allows bottom-up development (cf. 2.1.4?) and thus e. g. implementing support for new languages
- ☐ Languages implemented within GT are inspectable and moldable as far as they're implemented: parser state, tokens, parse tree, bytecode, *etc.*
- ☐ Custom views are cheap to implement, given knowledge of Smalltalk+GT
- ☐ Composed Narrative: Visualize object relations through side-by-side views – useful as pedagogic tool
- ☐ Moldable Object: Incremental development of objects with state and prior views always available; forces transparency and clean separation
- ☐ Project Diary: Lepiter notebook page with runnable code and live views; document progress – useable as learning journal for students (similar to Microsoft OneNote), or as project basis (similar to Jupyter)
- ☐ Was very useful for quick prototyping, reusable code, ... (cf. chapter 4)

4

Proposed Solution: A New Teaching Environment

4.1 Development of ”Processing Abstractions”

Excerpts from gt-exploration Lepiter pages

- ☐ Adaptable foundation: GT
- ☐ Student language: visual language, own language (e. g. like Grace [11]), common language?
- ☐ Decision: Processing/Python (cf. 3.1) – based on popular Python, visual primitives for quick results (which is motivating cf. [13])
- ☐ Critics from [13]: limited API (`square`, `circle` could be removed; see also 2.1.3), absolute coordinates initially more intuitive, Turtle and other approaches implementable through transparent functions or libraries
- ☐ 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

```
ProcessingCanvas >> ellipse: x y: y dx: dx dy: dy [
   self
      ellipse: dx
      by: dy
      at: x @ y
]
```

```
ProcessingCanvas >> endFrame [
   presenter updateOutput.
   (1 / frameRate) seconds wait. "The frame rate is adjustable through `frameRate()`"
   frameCount := frameCount + 1.
   transform := #yourself  "Transforms are reset at the end of a draw-cycle"
]
```

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 5.3); an introduction into the inner workings of a compiler (section 5.2); and a plan for a general introduction to programming (section 5.1). 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 Introduction to Programming

Using PA as a live programming environment.

5.1.1 Educational objective

- ☐ students are able to write programs producing given outputs
- ☐ students are able to read and understand programs with a limited, given command set
- ☐ students learn from their mistakes, correct themselves, aren't afraid to break things
- ☐ students have a solid foundation for taking on a task of writing a basic but still interesting app/game

5.1.2 Prerequisites

Just the basics:

- ☐ using own computer
- ☐ downloading and installing, handling (ZIP) archives and virus scanners (under Windows at least)

- ☐ curiosity, ...

5.1.3 Introduction to Glamorous Toolkit

- ☐ Distribute GT/PA
- ☐ After extracting GT, a brief overview is needed before starting
- ☐ Introduction to GT as an interactive notebook (compare to previously known software such as OneNote or Jupyter)
- ☐ GT is bleeding edge (development on trunk), introduce most pressing issues (navigation, keyboard issues, saving, scrolling, zooming)
- ☐ Some quick tasks for getting the hang of it and identifying students with more supporting needs (let them help themselves)
- ☐ Point more advanced students towards inspectability

5.1.4 Lesson Plans

- ☐ stepwise introduction to Processing (introduced in 3.1)
- ☐ imperative, few commands: `size`, `rect`, `ellipse`, `fill`
- ☐ teaches importance of order
- ☐ tasks: produce given output
- ☐ debugging consists in modifying values (result is immediately visible)
- ☐ quicker and more proficient students can easily skip ahead (loops, animations, recursion)
- ☐ introduce variables, loops, animation, interaction
- ☐ available tools: output, step-by-step debugger (for now, more later)

5.2 Lesson on Compilers

Using PA to demonstrated the steps of lexing, parsing, transpiling, compiling and optimizing.
Part of this has been validated (cf. 6.2)

5.2.1 Educational objective

- ☐ students can explain the difference between high and low level language
- ☐ students can enumerate the steps required for compiling a program
- ☐ students have an understanding of the roles a lexer, parser, transpiler and compiler play

5.2.2 Prerequisites

- ☐ programming with Processing (e. g. from 5.1)
- ☐ GT/PA installed (e. g. from 5.1.3)
- ☐ Stacks and registers

5.2.3 Lesson Plan

- ☐ Repetition high level programming (see tasks in PA)
- ☐ Comparision with low level programming (e. g. [2]): levels 1 to 6 (introduces jumps, memory access, arithmetic)
- ☐ Presenting/reading overview, compare with natural language
- ☐ Lexer: compare given example with mainly different whitespace; what are tokens?
- ☐ Parser: describe AST in own words, compare with sentence structure from natural languages; develop simple parsing model (**¡better views?!**)
- ☐ Transpiler (optional): compare Processing and Smalltalk
- ☐ Compiler: compare AST with intermediary representation; compare Program with intermediary representation; compare intermediary representation with [2]
- ☐ Optimization: naive examples

5.3 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 [8, 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 2.1.4) or top down (2.1.5). In either case, this proposed lesson will go towards the middle or can be used at the end as part of a repetition sequence.

Part of this has been validated (cf. 6.1)

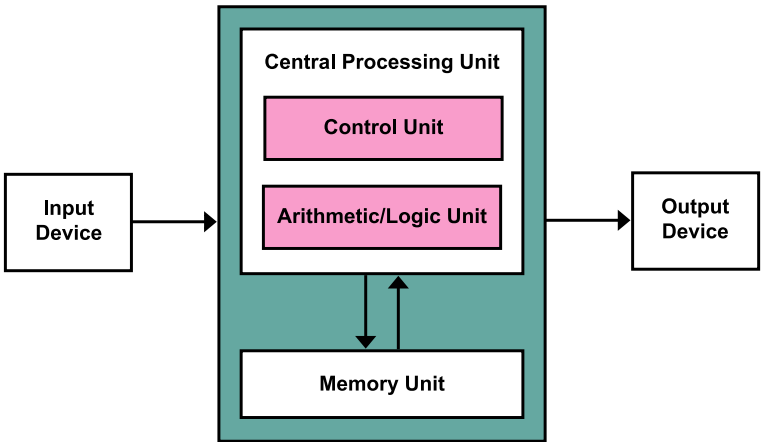
5.3.1 Educational objective

- ☐ students can explain how a program might actually be run on hardware

5.3.2 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 5.1 above.

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.

5.3.3 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 5.1.3). Else we can directly start with a reminder of what they already know about programming.

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

- ☐ Own class, 14 students present, prepared according to prerequisites
- ☐ Two lessons, afterwards questionnaire through Microsoft Forms

6.1.2 Observations

- ☐ adapt notes from gt-explorations

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`

`startup.st`:

```
Metacello new
   repository: 'github://zeniko/gt-exploration:main/src';
   baseline: 'GtExploration';
   load.

Metacello new
   repository: 'github://zeniko/processing-abstractions:main/src';
   baseline: 'ProcessingAbstractions';
   load.

"Hide the 'Implementation and Tests' section."
GtExplorationHomeSection studentMode: true.

"Make indenting keyboard shortcuts available to non-US-English keyboard layouts."
LeSnippetElement keyboardShortcuts
   at: #IndentSnippet
      put: BlKeyCombinationBuilder new alt shift arrowRight build;
   at: #UnindentSnippet
      put: BlKeyCombinationBuilder new alt shift arrowLeft build.

"Patch unneeded addressbar out of YouTube snippet. (TODO: fix properly)"
LeYoutubeReferenceElement compile:
   ((LeYoutubeReferenceElement methodNamed: #updatePicture) sourceCode
      copyReplaceAll: '</iframe>'' ' with: '</iframe>''; removeChildAt: 1 ').
```

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] What games use lua? a quick overview. https://luascripts.com/what-games-use-lua.

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

[9] 10 best programming languages for kids of any age, 2020. https://codeweek.eu/blog/10-best-programming-languages-for-kids-of-any-age/.

[10] 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.

[11] Andrew P. Black and Kim B. Bruce. Teaching programming with grace at portland state. *J. Comput. Sci. Coll.*, 34(1):223–230, October 2018.

[12] Dennis J Bouvier, Ellie Lovellette, Eddie Antonio Santos, Brett A. Becker, Venu G. Dasigi, Jack Forden, Olga Glebova, Swaroop Joshi, Stan Kurkovsky, and Seán Russell. Teaching programming error message understanding. In *Working Group Reports on 2023 ACM Conference on Global Computing Education*, CompEd 2023, pages 1–30, New York, NY, USA, 2024. Association for Computing Machinery.

[13] Luca Chiodini, Juha Sorva, and Matthias Hauswirth. Teaching programming with graphics: Pitfalls and a solution. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E*, SPLASH-E 2023, pages 1–12, New York, NY, USA, 2023. Association for Computing Machinery.

[14] Timothy R. Colburn. Software, abstraction, and ontology. *The Monist*, 82(1):3–19, 1999.

[15] William H. Doyle. A discovery approach to teaching programming. *The Arithmetic Teacher*, 32(4):16–28, 1984.

[16] Michael Egger. The law of leaky abstractions: A guide for the pragmatic programmer, 2024. https://medium.com/mesw1/the-law-of-leaky-abstractions-a-guide-for-the-pragmatic-programmer-9bf80545c43f.

[17] M. Fikret Ercan and Dennis Sale. Teaching programming: An evidence based and reflective approach. In *2020 IEEE REGION 10 CONFERENCE (TENCON)*, pages 997–1001, 2020.

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

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

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

[21] 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.

[22] Ajit Jaokar. Concepts of programming languages for kids. *Educational Technology*, 52(3):50–52, 2012.

[23] 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.

[24] 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.

[25] Simone Martini. Teaching programming in the age of generative ai. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*, ITiCSE 2024, pages 1–2, New York, NY, USA, 2024. Association for Computing Machinery.

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

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

[28] 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.

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

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

[31] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. Exploratory and live, programming and coding. *The Art, Science, and Engineering of Programming*, 3(1):1:1–1::32, 2018.

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

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

[34] Oleg Sychev. Correctwriting: Open-ended question with hints for teaching programming-language syntax. In *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 2*, ITiCSE '21, pages 623–624, New York, NY, USA, 2021. Association for Computing Machinery.

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

[36] Xiao-Ming Wang, Gwo-Jen Hwang, Zi-Yun Liang, and Hsiu-Ying Wang. Enhancing studentsâ€™ computer programming performances, critical thinking awareness and attitudes towards programming: An online peer-assessment attempt. *Journal of Educational Technology & Society*, 20(4):58–68, 2017.

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

[38] Jianwei Zhang. Teaching strategy of programming course guided by neuroeducation. In *2019 14th International Conference on Computer Science & Education (ICCSE)*, pages 406–409, 2019.