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

This thesis develops an empirically validated tool for programming classes at high school level which allows students to experience and explore interactively various abstraction layers involved in running a program.

Didactic literature suggests that exploring different abstraction layers (called *Sichtenwechsel*) improves students' understanding of both programming and the inner workings of a computer system. Such multi-layered exploration is even considered a foundational idea of computer science and has to be taught, among others because many lower abstraction layers tend to leak through interfaces anyways.

On the basis of the Smalltalk environment "Glamorous Toolkit", an interpreter for the Processing programming language in its Python form was developed and molded with many different views: Tokenization, syntax tree building, transpilation of Processing into Smalltalk, translation to an intermediary language and eventually into Smalltalk bytecode and finally the program's actual output. These views are tied to source code and updated live for seamless exploration and can be composed in interactive teaching material.

This product has been used for various lessons for which both plans and a brief evaluation is included. The evaluation shows on very limited data that the live environment does encourage experimentation and allows for students to work at their own speed and depth, allowing them to profit at their pace. Understanding of the various layers has however not improved significantly in the short period of time the tool could be tested.[1]

[1] A typographical note: Text written like this: **¡delete this!**

□ or text prepended by such a checkbox is a temporary note for the author.

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. [11]). 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 "Von Neumann Simulator" [21] or in a more gamified fashion e. g. with "Human Resource Machine" [3] or even without mnemonics as using the Little Man Computer architecture [4]. 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 3.3).

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. Finally, many thanks go to my kids for letting me work even during their holidays and to my wife for her endless support when morale was low.

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

- ☐ Scoping: 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. [11])
- ☐ 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, introducing the notions of "foundational idea" one of which is a *Sichtenwechsel* – a change of perspective in a multitier architecture; then we show the relevance and limits of abstractions in programming and motivate why students should look beyond a single abstraction layer in 2.2.

2.1 Didactic Approaches

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 [24].

This traditional introduction works mainly on the level of the programming language, only barely mentioning a simplified memory model (p. 31) and binary encoding (pp. 114–115). 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. And the Python interpreter is just in the preface briefly characterized as "program which translates source code into electronic instructions"[1].

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 [32, 39] (and even past texts [19]):

2.1.1 Foundational Ideas

Schubert and Schwill [39] base their didactic approach around the notion of "foundational ideas" derived from Jerome Bruner and Alfred Schreiber. A foundational idea is a concept which is deeply ingrained in

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

a specific subject such as computer science and without which the subject would loose part of its core. Additionally, they ask for a foundational idea to have 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.

In a later refinement, the property of Historical Relevance is added by requiring a foundational idea to also have persisted through time [32, p. 17].

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*

Consequently, they propose to explicitly discuss the foundational idea of 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, they 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),[2] reinforcing the need for going beyond of what traditional programming courses (used to) do.

This conclusion is repeated again and again, e. g. by Jaokar [27, p. 51] or Zhang [44, p. 407].

2.1.3 Manageability

Modrow and Strecker [32] follow Schubert and Schwill [39] 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 [8] or Berkeley's variant Snap! [9] (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.* [16]) 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.

[2] 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."

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.

Following in these footsteps, Chiodini *et al.* [17] similarly state as requirements for programming classes:

- 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.

The last point explicitly asks for a clean separation of abstraction levels which might be an ideal to strive for but might not be realistically achievable (cf. 2.2.1).

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 [35]: 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 [43], art [37] or even toy houses requiring intruder alerts [32] 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 [43] 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 [10]

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 [32]) 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 [38]. 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 [39, p. 367] and Modrow and Strecker [32, 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[3] which most scripting languages do, an interactive notebook (such as Jupyter or Lepiter, see 3.3) or a way of direclty running any part of a program, as 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.*.

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

2.1.7 Computational Thinking

Since other disciplines have started to rely on computers as more than a glorified typewriter and filing system, the notion of preparing students for working in academia and industry has shifted from teaching applications to teaching "Computational Thinking". Lee *et al.* [30] have assembled cases from schools where programming is used in physics, biology, chemistry and other sciences, linking it to the role that mathematics have had in the past centuries.

As a basis for employing programming, simulation or data transformations to solve problems in other domains, students must be versed in dealing with different abstraction levels in order to connect the abilities of a computer with the subject at hand. Flórez *et al.* [**?**] also point out that students must know the limits of computers and not attribute them understanding and intelligence.

With the sudden rise of large language models in the past decade, computational thinking even refers back to programming: Martini [31] argues that with LLMs being able to write programs on their own, programming curricula have to be adjusted accordingly. Students will still have to be able to understand the basics, in order to be able to instruct an LLM towards a desired result.

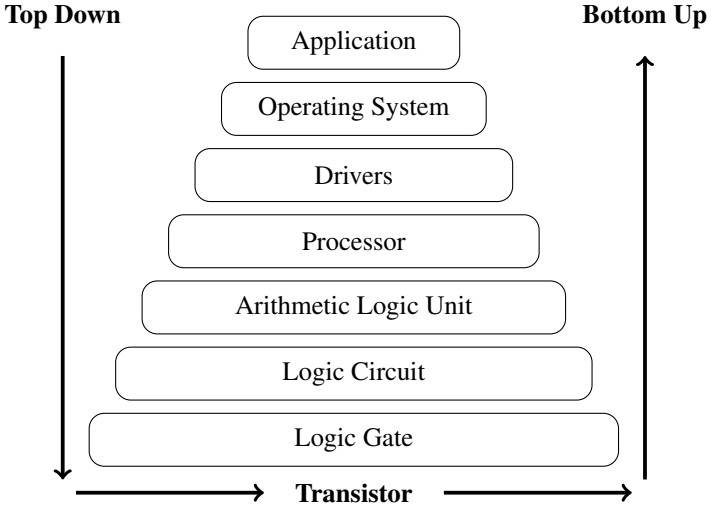
Also, recent studies seem to suggest that relying on LLMs for coding does not lead to better developer performance [14] nor to better code [33], as cognitive offloading has been observed. This effect is likely even more pronounced for students, as getting quick results for simple tasks might lull them into overconfidence with relation to the available LLM.

As a consequence, the functioning and limits of large language models must not only be taught as part of computational thinking but also used as another example for examining different abstraction levels in a computer system.

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:



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.

This analogy however shows that even in computer science, the philosophical mind-body problem persists, albeit in a different form: “At the grossest physical level, a computer process is a series of changes in the state of a machine” [18, p. 12]. In contrast to the human mind, where the interaction between objectively observable brain matter and subjective thought is at the core of an ongoing philosophical and neuroscientific debate, in computer science this duality of having electrical currenty on the one hand and a running program on the other is decomposable all the way through.

And as has been shown in 2.1, didactic literature suggests that this feature should be discussed, as this is a foundational idea of computer science.

2.2.1 Leaky Abstractions

In his article “The Law of Leaky Abstractions” [40] 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.

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 recently 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.[4]

In cryptography, timing attacks have been successfully used for extracting passwords from insufficiently protected webservers [36]. More recently, another class of timing attacks taking advantage of modern CPU's branch prediction optimizations has been demonstrated [29]. 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.

Further examples of leaky abstractions are discussed by Egger [20]. These show that knowledge of lower layers are particularly important for developers of compilers and other performance-critical programs: In order to optimize a program, the specifics of the platform architecture and the implementation of processors and networking become crucial.

Since abstraction leaks are unavoidable in programming – even with block based languages such as Scratch –, they have to be discussed anyway. Instead of tackling them one by one as separate exceptional cases, literature discussed above suggests to use this opportunity to work out the foundational idea of a multitier architecture and of the interconnectivity of computer systems.

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 [7]. This is mirrored in most other full fledged IDEs such as PyCharm [1] or Eclipse [6].

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 [42]; or Thonny which had the goal to visualize runtime concepts beyond what IDEs offered at the time [13, 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.[5]

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.[6]

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.

Jalalitabar and Wang [26] have 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 [5] which combines a visualization of stack frames variable values as pointers and deconstructed objects.

Sychev [41] suggests as an additional IDE feature hints for syntax errors which show students a side-by-side view of their entered code and the corrected code with all required transformations highlighted.

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

[5] 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.

[6] 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.

They have implemented this feature for Moodle.

Bouvier *et al.* [16] ask for an extension of this: a view to show details about any form of errors, helping students to better understand the issue at hand. In particular, they suggest including an LLM assistant which can further help explain an error to a novice student. How effective such an assistant would be, remains to be seen (see 2.1.7).

Comparing to programming IDEs, the web development consoles offered by modern web browsers also provide a variety of views into the various layers of the network stack. These views are however tailored to answer the most common questions of a web developer instead of providing a coherent overview of how the network layers interact.

□ Conclusion

3 Technical Background

The product of this thesis is implemented in Glamorous Toolkit and bases teaching material on the "Processing" programming language. A brief overview of both is given in this chapter for readers unaware of either of them.

3.1 Processing

"Processing" is a programming language consisting of a graphics API built upon a mainstream language as a base. Development started between 1997 and 2004 at the MIT Media Lab with the goal of creating a language for teaching art students the fundamentals of programming as basis for creating digital, visual art.

Its authors, Reas and Fry [37], wanted to create a unified teaching system consisting of art, language and a matching IDE. They based the language upon then popular and portable Java, removing much of the boilerplate required for object orientation, enhancing it with visual primitives and implicitly showing an output window, allowing for quick results (see figure 3.1).

As elaborated in 2.1.5, Processing was meant to be taught top down, starting from art and then decomposing it. As such, its main introduction features several chapters focused on exhibits of digital or hybrid art such as Manfred Mohr's *Une esthétique programmée* or Steph Thirion's *Eliss*.

Apart from graphical primitives (see appendix B for the API subset implemented for this thesis), Processing features an implicit event loop which allows for creating (interactive) animations within a

```
// Output canvas dimensions
size(200, 200);
// (Default white) square
rect(50, 50, 100, 100);
// Red inner rectangle
fill(255, 0, 0);
rect(50, 50 + 100 / 3, 100, 100 / 3);
```

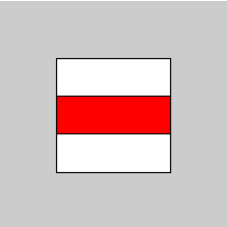


Figure 3.1: Example code (with Java syntax) and output

```
y = 50; dy = 0

# called once after global code
def setup():
    size(100, 200)

# called repeatedly for every frame
def draw():
    global y, dy
    background(192)
    circle(50, y, 50)
    y += dy; dy += 1
    if y > height - 25:
        dy = -0.9 * dy
```

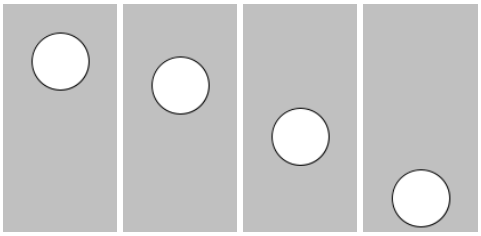


Figure 3.2: Example code (with Python syntax) and one output frames

dozen lines of code (see figure 3.1).

Reacting to input happens either by pulling state while painting a frame (implicit global variables `mousePressed`, *etc.*) or by defining event handlers alongside `setup` and `draw`.

Since Python has become the prevalent teaching language ([12]), Processing has been extended with a Python mode which leads to Processing code reading like Python code with the additional Processing API calls and exposes most of Python's libraries to Processing code. This allows using Processing as a starting language and later seamlessly transitioning to pure Python code, which remains part of the motivation for students: learning an "actually useful" language.

As the official Processing IDE continues to be written in Java, Processing's official Python mode uses Jython for compiling the code to Java Bytecode. Since developers have started moving away from Java, there are now several reimplementations of Processing such as p5.js for running Processing on top of JavaScript in a web environment, p5.py for running Processing in a pure Python environment[1] or a version of Processing for microcontrollers such as Arduino. With this thesis, a limited version for a Smalltalk environment is also available.

Initially in the early 2010s, we ran our own IDE based on p5.js with custom error handling[2] before changing to the official IDE for its Python mode.

Our experience of working with Processing over the past decade has shown that it allows novice programmers in the first year of high school to learn enough of the language within a month that they're able to write a clone of a game like "Pong", "Flappy Bird" or "Geometry Dash" as a group project. Feedback from the various student groups about this part of the computer science curriculum has always been positive to very positive. This aligns with what didactics states about student motivation (see 2.1).

3.2 Moldable Development

"Moldable development" is a term coined by Nierstrasz and Gîrba [22, 34] for a collection of development patterns which should make it easier to understand a computer system by extending ('molding') it with views and features. The goal of moldable development is to quickly get feedback on code and objects being worked on so that a programmer can confidently make appropriate changes.

In traditional IDEs, a running system is inspected either through its source code or its live runtime objects. Available views (see 2.2.2) are static and new views are added through non-trivial extensions.

[1] Requiring two additional lines: `from p5 import *` at the top and `run()` at the bottom.

[2] This is still available from `https://software.zeniko.ch/ProcessingIDE.zip`. Note that it's targeted at `mshta.exe` and as such runs best under Windows.

Moldable development asks for an environment in which a tool is more easily adaptable to data, making it simple to write either one-off throw-away views and tools but also allowing to refactor such throw-away code into reusable components when needed.

Moldable development is thus a form of exploratory programming (cf. 2.1.6) on live objects where tools, whether one-off or reusable, are created in a bottom up approach with immediate feedback available at every step.

In order to support this, a moldable environment must have extensibility in its core, allowing to register tools and views e. g. through a simple code annotation of a few characters which the environment can use to detect and include it (instead of having to write a lot of configuration boilerplate and overhead which IDE extensions meant for independent distribution usually involve).

One core pattern of moldable development is the "Moldable Object": Objects should be implementable incrementally with live object states and previously developed views remaining available throughout the whole process. An object consisting of little more than a data wrapper is thus extended with new functionality as it fits the available live data – instead of designing an object on a clean slate or along tests. Exploration code can then be extracted into tests, ensuring that what worked once will continue to work. Extending objects iteratively based on actual needs ensures that they remain transparent and that code is cleanly separated.

Having a moldable environment also allows for working on code and documentation intertwined, similar to literate programming [28]. Opposed to literate programming where code has to be extracted first, in moldable development every code snippet should be runnable on its own and beside code and documentation also live results can be included. This allows a moldable environment to be used to either first document ideas and then add matching code but also to document progress or explain written code (which can then easily be extracted into a test case).

For students, such a "Project Diary" pattern could be used as a learning journal (similar to Microsoft OneNote), for project exploration (similar to Jupyter notebooks) or for project documentation. Another useful pattern for teaching is the "Composed Narrative" which visualizes object relations through side-by-side views tailored towards explaining a relation or interaction.

3.3 Glamorous Toolkit

"Glamorous Toolkit" (GT) is a fully programmable environment optimized for moldable development (see 3.2). It is programmed in Smalltalk and by default persists its entire state into an `.image` file when shut down so that live objects don't have to be recreated at restart [23].

Smalltalk environments have had that property since the early days in the 1970s, when Alan Kay sketched out the original Smalltalk which he eventually standardized at Xerox into Smalltalk-80. Based on a Smalltalk-80 virtual machine by Apple, Ingals, Kay *et al.* started developing a new virtual machine and development environment, Squeak, which had the goal to also be customizable by non-programmers [25]. Squeak inherited its built-in capabilites for live and exploratory coding from the original Smalltalk and its back to this point that GT's heritage is tied directly.

While Squeak was further developed at Walt Disney Media Labs and among others included in the "One Laptop per Child" laptops, it remained a niche product – likely due to missing interoperability between the live environment inside its virtual machine and outside code. Still, Squeak and its later fork Pharo continued being worked on and were actively being used in academia and related spin-offs. Eventually, a team around Tudor Gîrba – including this thesis' supporter Oscar Nierstrasz – set out to implement their idea of a moldable environment on the basis of Pharo, thus creating Glamorous Toolkit [2]. Version 1.0 has been released in 2023 and is still being actively worked on.

Glamorous Toolkit thus has an illustrious lineage and has achieved support for many concepts asked for by literature (as outlined in 2.1 and 3.2): It's a moldable environment, supports a clean object-oriented

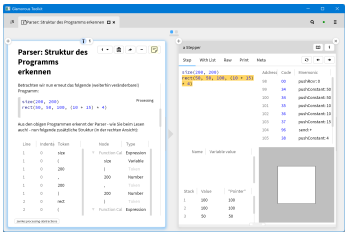


Figure 3.3: GT with a live notebook page (left) and inspectable object view (right)

language, allows for live and exploratory programming, still remains comparatively manageable and – particularly relevant for this thesis – allows for reflection at various levels, including for every object access to its method's source code, its compiled form and even its memory layout inside the virtual machine.

GT provides a tabbed interface which can show one of several tools: an object viewer, a notebook, a code browser, a git interface and many more. While such tools are about as difficult to implement as an IDE extension, the object viewer – a tabbed interface itself – is extended by annotating an object method which returns a `GtPhlowView` object with `<gtView>`:

```
ProcessingCodeBase >> gtOutputFor: aView [
   <gtView>
   ^ aView explicit
      title: 'Output' translated;
      priority: 40;
      stencil: [ (ProcessingRunner new
            limitTo: (self gtIsAnimation ifTrue: [ 30 ] ifFalse: [ 2 ]) seconds;
            run: self clone;
            canvas) asElement ]
]
```

The element passed to the `stencil:` message – here the canvas resulting from running a Processing program – could instead also be displayed inside a notebook page, with no annotations needed at all.

Annotations are thus only required to allow GT to discover messages of a certain type. Similarly, methods annotated with `<gtExample>` are considered tests and can be collectively inspected and run for a class or an entire package. This achieves several goals of moldable development: What starts as throw-away code can be extracted into a method, annotated and remains then permanently available for repeated testing. Examples are also includable by name in notebooks, where they do function as (tested and thus guaranteed working) examples for documentation.

While GT is based on a Smalltalk virtual machine, support for other modern languages such as Python, JavaScript or Java has been added through a language bridge connecting to an external virtual machine and allowing for inspection and visualization of objects originating there. Unfortunately, these language bridges are one example where one of GT's major drawbacks starts showing:

GT is mainly developed under macOS and makes some platform assumptions with relation to its host operating system. This isn't noteable when working purely within GT but occasionally shows at its seams. Also, GT follows a trunk-only development style without release branches. This means that downloading GT on two different days might result in subtle differences. If GT with an app is to be distributed, the best way to do this is by downloading the latest version, loading the app into it, testing it and then distributing *this known good* image (as is described in A).

4
Proposed Solution: A New Teaching Environment

In order to let students have a *Sichtenwechsel* with relation to programming, i. e. have them experience several different abstraction layers involved between a program's source code and its execution, a new teaching environment dubbed "Processing Abstractions" is proposed:

Within Glamorous Toolkit, we've implemented support for the Processing programming language and molded views for every implementation step along the way. This allows for creating interactive notebook pages containing source code and a variety of these views, showing e. g. the abstract syntax tree (AST) and resulting bytecode for the GT virtual machine side-by-side (see figure 3.3).

In this chapter, we document the development of this environment and the reasons for the approaches chosen.

4.1 Motivation for "Processing Abstractions"

For the past decade, we've taught the introduction to programming using Processing to ninth graders. This consisted of an introduction into the language along relatively simple examples inspired by minimalist art and games (similar to the approach by Reas and Fry [37] described in 2.1.5). In a separate sequence, we've taught computer architecture in a bottom up approach (inspired by Nisan and Schocken [35] as described in 2.1.4 but much abbreviated).

While the introduction to programming was usually quite well received by students and also led to satisfying results, the sequence on computer architecture did less so. The two sequences also didn't fit together as nicely as we'd have liked and processor and memory remained a mystery for too many students. Hence the decision to look into combining the two sequences.

At this point, we've briefly evaluated whether staying with Processing was reasonable. Reasons to do so were manyfold: As seen in 3.1, Processing allows a top down approach starting at visual arts, which allows to motivate students with less interest in mathematics and natural sciences; Processing also quickly yields pleasant looking results, which also adds to initial motivation [17]; then it can be based on still popular and widely used Python, which allows using it as a stepping stone and makes it a 'real' programming language in the eyes of novices; on the other hand, Processing itself is sufficiently unknown that even students already experienced with programming will have something new to discover; additionally, Processing

has a large community sharing sketches and ideas which can be used as inspiration for both students and teachers; finally, its proven itself in our own experience over the years.

- ☐ Rejected alternative: own language (e. g. like Grace [15]) – specification would've been fun, but takes time; lack of teaching materials; less motivating
- ☐ Rejected alternative: visual language (e. g. Scratch) – differentiation from middle school; more serious; better tooling
- ☐ Critics from [17]: 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
- ☐ Reasons for GT: moldable environment leads to quick result; views are easy to create and combine; something new for all students
- ☐ Rejected alternative: web platform – many libraries, less ideal interaction, requires server infrastructure for reliable saving
- ☐ 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
- ☐ Was very useful for quick prototyping, reusable code, ... (cf. chapter 4)

4.2 Development of "Processing Abstractions"

- ☐ Various approaches to run Processing: PythonBridge, interpreter, compiler, transpiler (chosen)
- ☐ Python maps relatively well to Smalltalk
- ☐ Lack of first class functions: transpiling only known functions
- ☐ No type checking (Python and Smalltalk are both strictly but dynamically typed)
- ☐ Option to translate bytecode or intermediary code to other assembly syntaxes (Intel x86, LMA, WASM, ...)
- ☐ API reference in appendix B
- ☐ setup instructions in appendix A

4.3 Class Hierarchy

- ☐ `ProcessingCodeBase`, `ProcessingCanvas`, `ProcessingParser`, `ProcessingTranspiler`, `ProcessingProgram`, `ProcessingSource`, `ProcessingSnippet` exceptions, helpers, events

```
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"
]
```

4.4 Abstraction Levels

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

- ☐ Source Code
- ☐ AST
- ☐ Transpilation (and IR)
- ☐ Machine Code
- ☐ Output
- ☐ Others

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. [3]): 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 [3]
- ☐ 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 [11, 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)

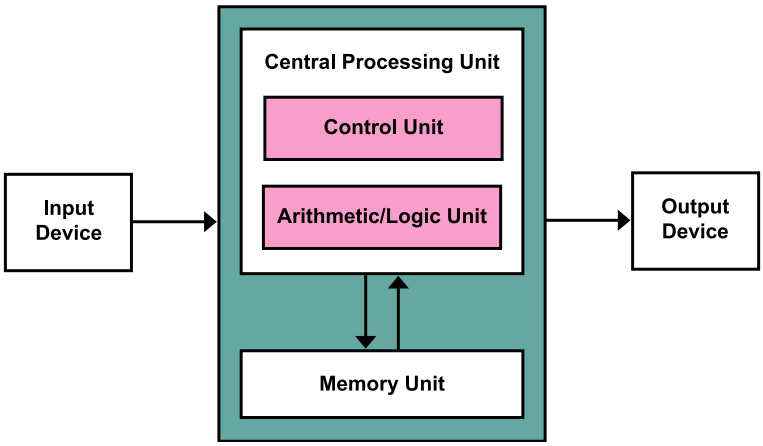
5.3.1 Educational objective

- ☐ 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

In order to set up Processing Abstractions, first download Glamorous Toolkit from https://gtoolkit.com/download/ for your platform and extract the archive's entire content.

Before running it, create a new text file called startup.st in GT's top level folder (besides GlamorousToolkit.image) with the following content:

```
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 ').
```

Finally run the GlamorousToolkit executable (under Windows and Linux it's located in the bin subfolder). The content of Processing Abstractions are now available behind the "Unterrichtseinheiten" home tile.

B
GT Processing API

This appendix lists the available API calls implemented in Processing Abstraction's Processing. This has been auto-generated from GT page "Processing API" (`db=5rgxfv9r684digmpw7ph7kbt4`):

B.1 Rendering

B.1.1 Setup

`background(r, g, b)`
Clears the canvas and changes its color (see `fill(r, g, b)`). Default is light gray (192, 192, 192).

`background(gray)`
Clears the canvas and changes its color (see `fill(gray)`). Default is light gray (192).

`size(width, height)`
Prepares an output canvas of the given dimensions. This command must always be called first (for animations: first command in `def setup`).

B.1.2 Shapes

`circle(x, y, d)`
Draws a circle with the given diameter and its center at `(x; y)`. Short for `ellipse(x, y, d, d)`.

`ellipse(x, y, dx, dy)`
Draws an ellipse with the given diameters and its center at `(x; y)`.

`image(image, x, y),image(image, x, y, width, height)`
Renders the image loaded with `loadImage(...)` at the given coordinates (and scales it to fit the given size). Arguments following after the first are identical to `rect`'s. If `width` and `height` are not given, the image's native dimensions are used.

`line(x1, y1, x2, y2)`
Draws a line from `(x1; y1)` to `(x2; y2)`.

`loadImage(pathOrUrl)`
Loads the image from the given URL or path. The returned value is to be used with `image(...)`. Paths can be absolute or relative to either `FileLocator class>>gtResource` or:

```
Element
GtInspector newOn: FileLocator documents / 'lepiter'
```

`rect(x, y, width, height)`
Draws a rectangle of the given width and height, parallel to the coordinate axes with its top left corner at `(x; y)`.

`square(x, y, side)`
Draws a square with the given side length, parallel to the coordinate axes with its top left corner at `(x; y)`. Short for `rect(x, y, side, side)`.

`text(string, x, y)`
Renders the given `string` with its baseline starting at `(x; y)`.

`textSize(size)`
Sets the size for rendering text in pixels. Default is 12px.

`triangle(x1, y1, x2, y2, x3, y3)`
Draws a triangle with its vertices at the points `(x1; y1)`, `(x2; y2)` and `(x3; y3)`.

B.1.3 Colors

`color(r, g, b)`
Generates a color object which can be stored in a variable also be used with `fill(...)`, `stroke(...)` and `background(...)`.

`fill(r, g, b)`
Selects the color to use for filling rendered shapes. The color is given as three values in the range of 0 to 255 (red, green and blue respectively). Default is white (255, 255, 255).

`fill(gray)`
Selects the gray scale value to use for filling rendered shapes. The color is given as a single value in the range of 0 to 255 (black/dark to white/light). Default is white (255).

`noStroke()`
Disables borders for future shapes. Equivalent to `strokeWeight(0)`.

`stroke(r, g, b)`
Selects the color to use for the borders of rendered shapes (see `fill(r, g, b)`). Default is black (0, 0, 0).

`stroke(gray)`
Selects the gray scale value to use for the borders of rendered shapes (see `fill(gray)`). Default is black (0).

`strokeWeight(weight)`
Determines the size of drawn borders in pixels. Default is 0.5px.

B.1.4 Transforms

`rotate(angle)`
Rotates all future shapes by the given angle (in `radians`!) clockwise around the origin.

`scale(factor)`
Linearly scales all future shapes by the given factor from the origin.

`translate(x, y)`
Moves the origin `(0; 0)` for all future shapes (defaults to the upper left corner).

B.2 Events

`def draw():`
is called repeatedly (up to `frameRate` times per second) for drawing the output.

`def mouseClicked():`
is called whenever a mouse button has been clicked *and* released.

`def mouseMoved():`
is called whenever the mouse has been moved. Alternatively query `mouseX` and `mouseY` in `draw()`.

`def mousePressed():`
is called whenever a mouse button has been pressed. Alternatively query `mousePressed` in `draw()`.

`def mouseReleased():`
is called whenever a mouse button has been released.

`def setup():`
is called once as the program starts.

B.3 Mathematics

`cos(angle)`
Returns the cosine value for the given angle (measured in radians).

`int(value)`
Rounds the value to an integer.

`PI`
The value of the mathematical constant p.

`radians(angle)`
Converts the given angle (measured in degrees) into radians.

`random(limit)`
Returns a random floating point number between 0 and limit (inclusive).

`sin(angle)`
Returns the sine value for the given angle (measured in radians).

`sq(value)`
Returns the squared value. Equivalent to `value ** 2`

`sqrt(value)`
Returns the square root of the given value.

`tan(angle)`
Returns the tangent value for the given angle (measured in radians).

B.4 Miscellanea

`delay(ms)`
Waits `ms` milliseconds before continuing (mainly needed for demonstration purposes).

`frameRate(fps)`
Limits the frame rate of animations to a maximum of `fps` frames per second. Default is 30.

`height`
Contains the canvas height as set by `size()`.

`millis()`
Returns the number of milliseconds that have passed since the program has started.

`mouseX`, `mouseY`, `mousePressed`
Contains the `x`- and `y`-coordinates of the mouse cursor and whether the mouse has been pressed at the start of a `draw`-phase (undefined outside of `draw`).

`print(value)`, `println(value)`
Prints the given value into an output console (mainly for debugging and for graphic-less program).

`str(value)`
Turns the value into a string, e. g. for concatenating several values for use with `text(...)`.

`width`
Contains the canvas width as set by `size()`.

C
Data from Questionnaires

Bibliography

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

[2] feenk - about. https://feenk.com/about/.

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

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

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

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

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

[8] Scratch - imagine, program, share. https://scratch.mit.edu/.

[9] Snap! build your own blocks. https://snap.berkeley.edu/.

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

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

[12] 10 best programming languages for kids of any age, 2020. https://codeweek.eu/blog/10-best-programming-languages-for-kids-of-any-age/.

[13] 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.

[14] Joel Becker, Nate Rush, Beth Barnes, and David Rein. Measuring the impact of early-2025 ai on experienced open-source developer productivity, 2025. https://metr.org/Early_2025_AI_Experienced_OS_Devs_Study.pdf.

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

[16] 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.

[17] 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.

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

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

[20] 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.

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

[22] Tudor Gîrba. Wtf is moldable development?, 2022. https://blog.container-solutions.com/wtf-is-moldable-development.

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

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

[25] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of squeak, a practical smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, October 1997.

[26] 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.

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

[28] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 01 1984.

[29] 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.

[30] 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.

[31] 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.

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

[33] Amr Mohamed, Maram Assi, and Mariam Guizani. The impact of llm-assistants on software developer productivity: A systematic literature review, 2025.

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

[35] 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.

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

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

[38] 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.

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

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

[41] 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.

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

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

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