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

Revealing Programming Language Abstractions

Some Aspects of nand-to-tetris – in Reverse – Using Smalltalk

GymInf Individual Project

Simon Bünzli Straub
from
Bern, Switzerland

Philosophisch-naturwissenschaftliche Fakultät
der Universität Bern

15. August 2025

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

Software Engineering Group
Institut für Informatik
University of Bern, Switzerland

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

This thesis presents a tool for teaching programming classes at high school level that allows students to experience and explore interactively various abstraction layers involved in running a program.

Didactic literature suggests that exploring different abstraction layers improves students' understanding of both programming and the inner workings of a computer system. Such multi-layered exploration is also considered a foundational idea of computer science and has to be taught, among other reasons because many lower abstraction layers tend to leak through interfaces anyway.

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 into interactive teaching material.

This product has been used for various lessons for which both plans and a brief evaluation are included. The evaluation shows on the basis of the limited data available 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. However, understanding of the various layers has not improved significantly in the short period of time the tool could be tested.

Contents

1 Introduction **1**

2 State of the Art in Computer Science Didactics **3**
2.1 Didactic Approaches . . . 3
2.1.1 Foundational Ideas . . . 4
2.1.2 *Sichtenwechsel* . . . 4
2.1.3 Computational Thinking . . . 5
2.1.4 Teaching Bottom Up . . . 5
2.1.5 Teaching Top Down . . . 6
2.1.6 Manageability . . . 6
2.1.7 Exploratory and Live Programming . . . 7
2.2 Multitier Architectures / Abstractions . . . 8
2.2.1 Leaky Abstractions . . . 9
2.2.2 Abstractions in IDEs . . . 10

3 Technical Background **12**
3.1 Processing . . . 12
3.2 Moldable Development . . . 14
3.3 Glamorous Toolkit . . . 15
3.3.1 Smalltalk VM . . . 15
3.3.2 Moldable Interface . . . 16
3.3.3 Bleeding Edge Issues . . . 18
3.3.4 Historical Remarks . . . 18

4 Proposed Solution: A New Teaching Environment for Programming **20**
4.1 Overview of "Processing Abstractions" . . . 20
4.2 Exploring Abstraction Levels . . . 22
4.2.1 Source Snippet . . . 23
4.2.2 Output View . . . 23
4.2.3 Compilation Views . . . 23
4.2.4 Run Step Views . . . 24
4.2.5 Encoding Views . . . 27
4.2.6 Other Views . . . 27
4.3 Implementation Details . . . 29
4.3.1 Processing Compiler . . . 29
4.3.2 Processing Runtime . . . 30
4.3.3 Views . . . 30
4.3.4 Snippet . . . 30
4.3.5 Other approaches considered . . . 30
4.4 Potential Drawbacks . . . 31

5 Implementation: Lesson Plans **32**
5.1 Introduction to Programming . . . 33
5.1.1 Educational objective . . . 33
5.1.2 Prerequisites . . . 33
5.1.3 Introduction to Glamorous Toolkit . . . 33
5.1.4 Lesson Plan . . . 34
5.2 Lesson on Computer Architecture . . . 36
5.2.1 Educational objective . . . 36
5.2.2 Prerequisites . . . 36
5.2.3 Lesson Plan . . . 36
5.3 Lesson on Compilers . . . 38
5.3.1 Educational objective . . . 38
5.3.2 Prerequisites . . . 38
5.3.3 Lesson Plan . . . 39
5.4 Further Lesson Ideas . . . 40

6 Validation **41**
6.1 First Round . . . 41
6.1.1 Setting . . . 41
6.1.2 Observations . . . 42
6.1.3 Student Feedback . . . 42
6.1.4 Learnings . . . 43
6.2 Second Round . . . 44
6.2.1 Setting . . . 44
6.2.2 Observations . . . 44
6.2.3 Student Feedback . . . 45
6.2.4 Learnings . . . 45

7 Conclusion **46**
7.1 Future Work . . . 47

A Installing and Using "Processing Abstractions" **49**

B GT Processing API **51**
B.1 Rendering . . . 51
B.1.1 Setup . . . 51
B.1.2 Shapes . . . 51
B.1.3 Colors . . . 52
B.1.4 Transforms . . . 52
B.2 Events . . . 53
B.3 Mathematics . . . 53
B.4 Lists . . . 54
B.5 Miscellanea . . . 54

C Technical Implementation of "Processing Abstractions" **56**
C.1 Classes . . . 56
C.2 Views . . . 56

D Questionnaires **66**
D.1 Questionnaire for 6.1 66
D.2 Questionnaire for 6.2 68

Abbreviations **70**

Bibliography **71**

1 Introduction

In our 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 for decades, broader and more theoretical courses have only 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 at high schools usually contain introductions not only to algorithms and programming but among other topics also to encodings, computer architecture, networking and social ramifications such as privacy and security (see *e.g.* [14]). Students are therefore not only taught a high-level programming language such as Python but should also develop insights into what happens at various other abstraction levels when such a program is stored and run.

One traditional approach to teaching computer architecture consists in teaching a separate assembly-like language during the introduction to computer architecture. This can happen in a more gamified fashion *e.g.* with the app Human Resource Machine [4], closer to theory with the Von Neumann Simulator [31] or even without mnemonics using the Little Man Computer architecture [5]. While all of these approaches help to show how a microprocessor might work approximately, none of them offer a direct, explorable connection to a high-level language.

In our experience, high-level programming has been quite well received with students, whereas the teaching sequence on computer architecture was less so. Programming and computer architecture also hadn't fit together as nicely as we'd have liked, and processor and memory have remained a mystery for too many students. This has led us to look into a better way of combining the two:

One suggestion for such a direct connection between high-level language and machine code will be presented in this thesis, in the form of a teaching environment targeted at high school students and implemented using the Glamorous Toolkit platform (introduced in section 3.3) for the Processing programming language based on Python syntax (which will be introduced in section 3.1).

We deem such a teaching environment as being called for, if didactic literature (summarized in 2.1) and currently existing Integrated Development Environments (IDEs) are considered. In particular, we propose to use this as a basis for explicitly discussing the foundational idea of multitier abstractions with students, which appear in multiple places all throughout their curriculum within computer science –

most prominently in networking and in information encoding – but also in other subjects such as natural sciences, psychology, *etc.* This discussion is important insofar as the clean separation of abstraction levels may unexpectedly fail or "leak" (a concept introduced in 2.2.1), exposing lower levels through seemingly irrelevant side effects such as timing or physical constraints.

The teaching environment introduced in chapter 4 of this thesis, named "Processing Abstractions", will focus in particular on abstractions encountered during programming (see *e.g.* figure 4.1 on page 21), allowing students to have a *Sichtenwechsel* on their own programs.

For teachers, suggestions for how to include the environment in the classroom are provided in chapter 5, with a sequence on computer architecture at the center, but flanked by two sequences on programming and compilers for embedding it. All these sequences build upon the teaching environment provided and rely on students being able to get multiple, varied views and insights into the same program, in order to experience the behind-the-scenes work or rather the details usually abstracted away in an interactive way. The students' engagement is ensured due to all input being readily modifiable with changes being immediately visible, allowing for almost frictionless exploration.

Parts of the lessons suggested have already been tested with two classes at the Gymnasium Neufeld in Berne for collecting valuable feedback from students. While the sample size was too small to get statistically significant results, observations and student feedback (discussed in chapter 6) have shown that the environment works and that students are motivated by its liveness to explore the concepts provided. Whether their understanding of the abstraction levels involved have improved, could, however, unfortunately not (yet) be shown.

All of this wouldn't have been possible without the very helpful support of Prof. em. Oscar Nierstrasz who has finally managed to introduce me to Smalltalk and Prof. Timo Kehrer who has taken this project together with his predecessor under his wing. I also want to thank my students from the classes 27Ga and 28Ga of Gymnasium Neufeld who have worked with my prototype and given helpful feedback. Finally, many thanks go to my kids for their understanding of me having to work even during their holidays and to my wife for her endless support, which made this thesis possible in the first place.

2
State of the Art in Computer Science Didactics

With relation to systems, didactic literature recommends examining multiple layers of a system, for students to better understand the material. This concept (later called *Sichtenwechsel* for lack of a better fitting English term) is introduced in 2.1.2 and is considered a foundational idea that should be exploited in teaching as well as also taught explicitly (elaborated in 2.1.1).

Learning to handle complex systems is at the core of what happens in many subjects in school (high school and beyond). With relation to computer science, this is asked for in particular when teaching students computational thinking (see 2.1.3). One aspect that we feel is particularly important in this regard is the fact that systems can rarely be separated into independently investigable layers in a clean way – or in other words: many abstractions involved in layering complex systems leak (see 2.2.1).

When multiple abstraction levels are taught together, common approaches consist in doing so either bottom up (2.1.4) or top down (2.1.5), showing how to potentially proceed.

In our experience, one thing high school students like particularly well during their computer science curriculum is programming (which is reflected in the student feedback discussed in chapter 6). Therefore, we want to merge the discussion on multitier abstractions and abstraction leaks with programming. However, existing IDEs as discussed in 2.2.2 only support this up to a point we found insufficient. Thus, we set out to create our own teaching environment (presented in chapter 4).

Finally, this environment for combining programming with a view on different abstraction levels should also adhere to current didactic best practices, mainly manageability (2.1.6), liveness and explorability (2.1.7).

2.1 Didactic Approaches

Traditional introductions to programming usually focus on a single programming language, its syntax and the available semantics – introducing them iteratively and practicing each element with basic exercises. Hartinger *e.g.* introduces most of Python in this way, so that it can later be used for scientific calculations [36].

This traditional introduction works mainly on the level of the programming language, just barely mentioning a simplified memory model (p. 31) and binary encoding (pp. 114–115). Interestingly, it does

not assume an IDE but instead very briefly introduces the command line and the Python Read-Evaluate-Print-Loop (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 briefly characterized as "the program that translates source code into electronic instructions"[1] in the preface.

While the needs of academic teaching and the form of semester courses with lectures and separate lab work tend to suggest such an approach, this is not a good fit with suggestions from current didactic literature [49, 56] (and even past texts [28]), as described in the following subsections.

2.1.1 Foundational Ideas

Schubert and Schwill [56] base their didactic approach around the notion of "foundational ideas" derived from Jerome Bruner and Alfred Schreiber. A foundational idea is a concept that is deeply ingrained in a specific subject and without which the subject would lose part of its core. Additionally, they ask for a foundational idea to have the following properties (pp. 64–65):

- Breadth: The concept is applicable not just in one specific context but can be used more generally.
- Abundance: The concept encompasses sufficient substance, so that it can be taught to beginners and experts alike.
- Meaning: The concept is meaningful to the learner beyond the scope of a course.
- Historical Relevance: The development of the concept can be observed historically, and the concept has remained relevant through time.

As such foundational ideas they list among many others the idea of modularization, the idea of layered architectures and the idea of encoding information (and as a special case: instructions).

As a consequence, they propose an introduction to programming by using several different programming languages along different paradigms: *e.g.* Prolog as a declarative language (pp. 91–104) and Python as an 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).

Interestingly, they don't mention any lower-level language as an alternative, considering translations between different high-level languages to be sufficiently plausible to students, although the limitations of languages such as an Assembly variation might be just as plausible, if not more, due to a far less expressive command set.

2.1.2 *Sichtenwechsel*

Consequently, Schubert and Schwill [56] propose to explicitly discuss the foundational idea of multitier architectures – and that not only in the context of the networking stack and computer architecture (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, [32]).

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

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 combining programming with computer architecture.

The same conclusion has been reached repeatedly, also *e.g.* by Jaokar [39, p. 51] or Zhang [62, p. 407].

2.1.3 Computational Thinking

Since other disciplines have started to rely on computers as more than glorified digital typewriters and filing cabinets, the notion of preparing students for work in academia and industry has shifted from teaching applications to teaching computational thinking. Lee *et al.* [45] have assembled case studies from schools where programming is used in physics, biology, chemistry and other sciences, linking it to the role that mathematics has had in the past centuries.

If students are to be able to solve problems in other domains using programming, simulations or data transformations, they must also be versed in dealing with different abstraction levels in order to connect the abilities of a computer with the subject at hand. Buitrago *et al.* [30] point out that students must in particular know the limits of computers and not attribute understanding and intelligence to them.

This has become an even more important matter with the recent rise of Large Language Models (LLMs) to being ubiquitously available. With LLMs available, computational thinking even refers back to programming: Martini [46] argues that with LLMs being able to write programs on their own, programming curricula have to be adjusted accordingly. However, 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 [19] nor to better code [50], 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 LLMs must not only be taught as part of computational thinking but also be used as another example for examining different abstraction levels in a computer system.

2.1.4 Teaching Bottom Up

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

Starting at the bottom seems to be the conceptually sounder way and is *e.g.* how Mathematics are taught. One rigorous implementation of this approach is offered by Nisan and Schocken [52]: They offer a course that 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 similar to the motivation for this thesis: "The most fundamental ideas [...] are now hidden under many layers of obscure interfaces" (p. ix), which they set out to reveal. Since the course is taught at university with hundreds of students, one 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 allows skipping intermediary steps and makes it possible 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, which also serves as the table of contents (pp. 1–4, represented

[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."

in figure 2.1 on page 8). With this, students keep in sight what they are 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 with the students' experience, *e.g.* at gaming [60], art [54] or even toy houses requiring intruder alerts [49], and starts exposing lower abstraction levels as required to gain more control (and understanding) as well as extending available capabilities.

One approach, which nowadays might already be considered rather traditional, starts with games: Weintrop and Wilensky [60] discuss a variety of games where programming plays a role in either shaping an avatar, improving its available actions or making 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 also be sufficiently motivating.

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

An alternative approach, which originally targeted art students but works well at high school as well, is proposed by Reas and Fry [54]: Starting from visual arts and extending the capabilities of the artist through digital means. While the involved programming language, Processing, will be discussed on its own merit in 3.1, their didactic approach is notable as well: A work of art on its own is something abstract that not only can be interpreted but also created. To (re)create it, various painting techniques are needed, which can be further broken down to basic movements. At this abstraction layer, they start 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 of how to achieve more complex output is naturally motivated by the more complex works of art already discussed.

To achieve this, Reas and Fry [54] feature several chapters focused on exhibits of digital or hybrid art such as Manfred Mohr's *Une esthétique programmée* or Steph Thirion's *Eliss*.

Additionally, as art can be considered as individual expression, the parallel to programming as individual expression (as also proposed by Modrow and Strecker [49]) 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 to 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.

As a side note: In high school courses spanning over one or even multiple years, either a top-down or a bottom-up approach could be implemented consistently. At university, with independent semester lectures being the standard, the focus usually lies on a single abstraction layer or picking out just a few. Courses as the one held by Nisan and Schocken [52] seem to be the exception. Arguably, computer science students should be able to better cope with such disparities, whereas a more coherent approach might be desirable at high school.

2.1.6 Manageability

Modrow and Strecker [49] follow Schubert and Schwill [56] in also building upon the concept of foundational ideas. They do, however, propose to reduce their number significantly and focus on a few

very general such ideas such as modelling, connectivity, digitization and algorithms (pp. 27–37).[3] As a consequence, they propose to focus programming exposure for high school students on block-based programming languages such as MIT's Scratch[4] or Berkeley's variant Snap![5] (p. 125).

Their reasoning for this is that students should not (yet) have to deal with syntax errors (as opposed to teaching these as suggested by Bouvier *et al.* [23]) and only have a manageable command palette. Furthermore, they note that block-based languages with free form layouting encourages students to build complex behavior from simple building blocks that 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 suitable for students than starting development at the abstract algorithm.

While Modrow and Strecker suggest also including digital circuits in the curriculum as concrete examples of digitization (p. 118), they seem content with 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.* [26] similarly state the following requirements for programming classes:

- Manageable complexity: IDEs and Application Programming Interfaces (APIs) must not be unnecessarily complex so as to not confuse students.
- Meaningful engagement: Samples and exercises should be introduced in such a way that it is clear what they refer to outside of class. Their usefulness shouldn't end with the exam.
- Clean problem decomposition: Bottom-up development should be effortless and code should be composable with minimal changes required, if any.

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).

2.1.7 Exploratory and Live Programming

Live programming refers to output and other intermediary products being adapted or recalculated every time source code changes, without requiring any explicit saving and/or rerunning by the programmer [55]. This gives students the quickest results, as else they might 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 exploring code and its effects by running and modifying it in order to build a mental model.[6] Both Schubert and Schwill [56, p. 367] and Modrow and Strecker [49, p. 167] suggest exploratory programming to allow students to work at their own speed and depth: With shared examples, less experienced students can stay closer to the given – simple but working – code, while more experienced students can use their knowledge to test more complicated hypotheses.

[3] German original: "Modellierbarkeit", "Vernetzbarkeit", "Digitalisierbarkeit", "Algorithmisierbarkeit".

[4] *Cf.* https://scratch.mit.edu/.

[5] *Cf.* https://snap.berkeley.edu/.

[6] Usually, exploratory programming more broadly refers to programmers working without fully specified requirements and thus having to discover these through exploration of the problem space. In this thesis, we will, however, restrict usage of the term to the pedagogical setting.

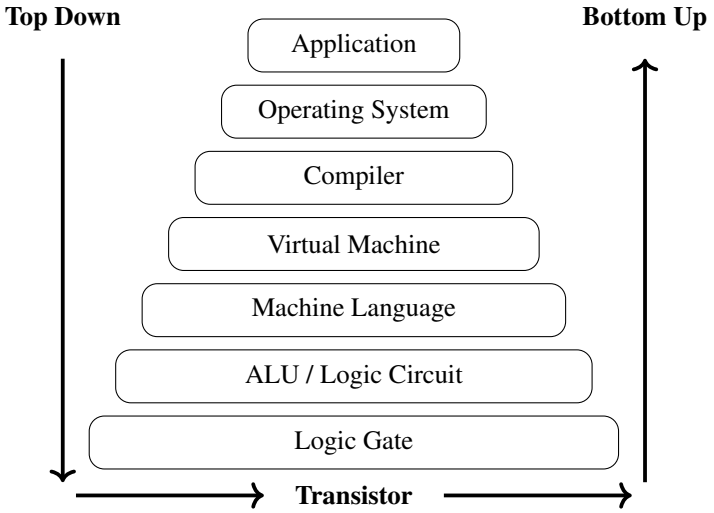

Figure 2.1: Multitier model of a computer system [52, p. xii].

While live programming requires explicit support from the programming environment (see 2.2.2), exploratory programming is possible without it. In fact, Rein *et al.* [55] note that in literature these concepts are usually treated separately, with barely any intersection between the two concepts. Apps can better support exploration by at least providing a form of REPL, which most scripting languages do, an interactive notebook (such as Jupyter or Lepiter, see 3.3), or a way of directly running any part of a program, as Scratch does (see 2.1.6).

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

Common multitier architectures taught at high school level are the networking stack (for manageability reasons rather the simplified four layered DoD architecture than the seven layered OSI model) or the software-hardware stack ranging from apps and hardware abstracting OS down to transistors consisting of *e.g.* silicium atoms (see [1] and figure 2.1).

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.

In this regard, programming should be possible independently of either hardware or operating system, in the same way that natural languages can be taught independently of a specific body or mind of the students.

However, this analogy 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” [27, 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 current on the one hand and a running program on the other is decomposable all the way through.

And as has been shown above, 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", Spolsky [57] introduces the concept of *leaky abstractions*, claiming that for all non-trivial 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 Representation (IR) translated into machine code, which runs on a virtual or actual machine.
- When programming instructions are executed through the above abstractions, variable values are encoded and stored in memory, function calls are traced 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 turning 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-level 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 the 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 an 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 rather easily observable abstraction leaks mentioned above, the issue itself might also have to be discussed, 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 piece of source code.[7]

In cryptography, timing attacks have been successfully used for extracting passwords from insufficiently protected webservers [53]. More recently, another class of timing attacks taking advantage of modern CPU's branch prediction optimizations has been demonstrated [42]. 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 levels.

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

Further examples of leaky abstractions are discussed by Egger [29] and many others [12, 40]. These show that knowledge of lower layers is 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 within block-based languages such as Scratch –, they have to be discussed in any case. Instead of tackling them one by one as separate exceptional cases, the literature discussed above suggests using 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 *e.g.* offers step-by-step debugging with variables and the call stack listed through extensions [8]. This is mirrored in most other full-fledged IDEs such as PyCharm [2] or Eclipse [7]. And while such IDEs even allow inspecting Python bytecode through appropriate extensions, the respective views are usually overwhelming for programming novices and thus rather targeted at professional developers than high school students.

As a remedy, several teaching-oriented IDEs have been developed, such as the Code with Mu IDE, which offers a minimal command set and still allows runtime inspection [59], or Thonny, which had the goal to visualize runtime concepts beyond what IDEs offered at the time [18, 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.[8] 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 in a separate view, which abstracts the call stack into a list of method names and line numbers.[9] Finally, Thonny distinguishes between values on the stack and values 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 sets out to and indeed nicely visualizes several concepts on lower runtime layers.

Jalalitabar and Wang [38] have assembled a list of tools targeted at visualizing some of these concepts outside of an IDE. One notable such alternative approach is taken by Python Tutor [6], which combines a visualization of stack frame variable values as pointers and deconstructed objects.

As an additional IDE feature, Sychev [58] suggests hints for syntax errors that show students a side-by-side view of their entered code and the corrected code with all the required transformations highlighted. They have implemented this feature for Moodle.

Bouvier *et al.* [23] ask for an extension of this: a view to show details about any form of error, helping students to better understand the issue at hand. In particular, they suggest including an LLM assistant that can further help explain an error to a novice student. How effective such an assistant would be remains to be seen (see 2.1.3).

Compared to IDEs, the web development consoles offered by modern web browsers also provide a variety of views of 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.

A different introduction to lower system layers is suggested by Wörister and Knobelsdorf [61], who also stick to block-based programming languages for manageability (see 2.1.6). They propose teaching

[8] 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.

[9] As a compromise, Glamorous Toolkit (GT) presented in 3.3 displays the call stack as a list of expandable method sources with the call location highlighted.

lower-level concepts based on their newly developed block-based environment Blocksambler, whose block structure is translated live into pure text-based assembly language and whose debugging view exposes program counter, registers and memory. What is missing from Blocksambler is a way to directly connect it with Scratch or any other high-level language.

As an intermediary step between block-based and purely text-based languages, Kyfonidis *et al.* [44] propose a frame-based interface that adds a colored background to blocks and statements. While this is meant to mimic a block-based view, it could just as well be used as an inline view of a program's AST. Their solution for preventing parser errors does, however, rely on heavily restricting possible student input by having students first choose a statement type, inserting the respective template node and have them fill in the blanks, which effectively introduces a new, two-stepped input mode.[10]

One kind of IDE where a close connection to lower architectural layers would be expected are didactic IDEs targeted at microcontroller programming. While common microcontroller platforms such as Arduino or micro:bit are supported in many common IDEs (including didactic ones such as Thonny, mentioned above), specialized IDEs would be the ideal place to inspect the inner workings of a (virtual) machine. However, neither the official Arduino Lab for MicroPython editor[11] nor micro:bit's Python editor[12] offer any additional views, not even matching Thonny's.

[10]The original proponents of frame-based editing, Kölling *et al.* [43] also noted as much.
[11]*Cf.* https://labs.arduino.cc/en/labs/micropython, documented at https://docs.arduino.cc/micropython/environment/code-editor/.
[12]*Cf.* https://python.microbit.org/v/3.

3
Technical Background

Before delving into this thesis' product, an overview of the technologies involved is given in this chapter: The environment described is implemented on the Glamorous Toolkit platform, which will be introduced in 3.3, using Moldable Development patterns (a concept introduced in 3.2). Finally, as a teaching language, we've chosen Processing for which an overview is given in 3.1.

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 as a continuation of the Design By Numbers project with the goal of creating a unified environment for teaching art students the fundamentals of programming as a basis for creating digital, visual art.

While the original Design By Numbers integrated the language into an IDE, having input and output side by side, it used its own, simplified programming language [13]. Processing's authors, Reas and Fry, based their 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).

Inside the IDE, Processing code is compiled to Java bytecode and run inside the same Java Virtual Machine (JVM) as the IDE. The Processing API was thus provided in the form of compiled Java code, and this hasn't changed for the official Processing IDE to this day.

```
// 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);
```

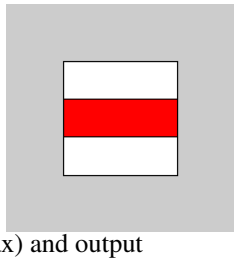

Figure 3.1: Example code (with Java syntax) and output

```
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
```

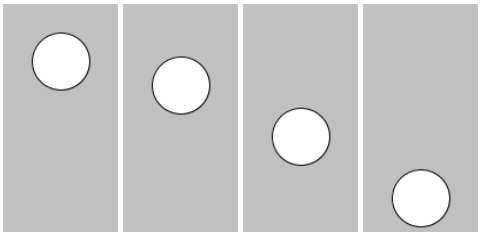

Figure 3.2: Example code (with Python syntax) and four output frames

Apart from graphical primitives, Processing features an implicit event loop, which allows for creating (interactive) animations within a dozen lines of code (see figure 3.2). Reacting to input happens either by polling while painting a frame (for this there are implicit global variables such as `mousePressed`, `keyCode`, *etc.*) or by defining event handlers alongside `setup` and `draw` (such as `mouseClicked(event)` or `keyTyped(event)`).

Since Python has become the prevalent teaching language [15], Processing has been extended with a Python mode, which uses Python as a basis, with the Processing API being available by default and the animation loop still being implicit [9].

As the official Processing IDE remains implemented in Java, Processing's official Python mode uses the Jython library to compile the code to Java Bytecode, so that it can be run in the same way as Processing programs written in the original Java mode [11]. This also gives access to most of Python's vast standard library, with the exception of a few modules that were precompiled to native code for the various platforms for speed reasons and thus had to be rewritten for or left out of Jython.

In fact, as the JVM is sufficiently generic to be the target for a wide variety of other languages, further modes for JavaScript[1] or R[2] have been added. This allows Processing and its dedicated IDE to be used as a starting point for programming and later seamlessly transitioning to the desired language such as pure Python, which remains part of the motivation for students: learning an "actually useful" language.

Since developers have started moving away from the JVM, there are now several reimplementations of Processing such as p5.js for running Processing on top of JavaScript in a web environment,[3] p5.py for running Processing in a pure Python environment[4] or a version of Processing for microcontrollers such as Arduino.[5] With this thesis, a limited version for a Smalltalk environment is also available (see chapter 4 and appendix B for an API overview).

In fact, when we started teaching programming in high school classes, we initially ran our own IDE based on web technologies and p5.js with custom error handling and support for live programming[6] before changing to the official IDE for its Python mode. Our experience of working with Processing with ninth and tenth graders over the past decade has shown that it allows novice programmers to learn enough of the language within a month that they are able to write a clone of a game like "Pong", "Flappy Bird" or

[1] Based on the Rhino compiler from https://rhino.github.io/.

[2] Based on the Renjin interpreter from https://renjin.org/.

[3] *Cf.* https://p5js.org/ and try it out at https://editor.p5js.org/.

[4] Requiring two additional lines: `from p5 import *` at the top and `run()` at the bottom; *cf.* https://github.com/gromko/p5-python.

[5] *Cf.* https://www.arduino.cc/education/visualization-with-arduino-and-processing/.

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

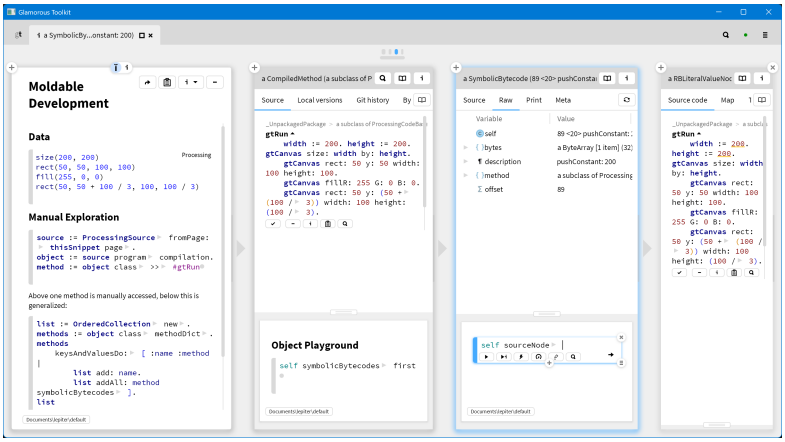

Figure 3.3: The moldable GT environment with data structures being explored for creating a view of an aspect (here: bytecode for a Smalltalk method)

"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 (and remains so, as we'll see in chapter 6).

Reasons to use Processing are thus manyfold: Processing allows a top-down approach starting with visual arts, which allows teachers to motivate students with less interest in mathematics and natural sciences. Furthermore, it quickly yields pleasant looking results, which also adds to the initial motivation [26]. Additionally, Processing has a large community sharing sketches and ideas, which can be used as inspiration for both students and teachers. Then, it can be based on currently widely used languages such as Python, which allows using it as a stepping stone and makes it a "real" programming language in the eyes of novices. In contrast, Processing itself is sufficiently unknown that even students already experienced with programming will have something new to discover. Finally, it has proven itself in our own experience over the years.

3.2 Moldable Development

Moldable development is a term coined by Chiş, Nierstrasz and Gîrba [24, 25, 33] for a software development approach that 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. Moldable development works in 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 developers to refactor such throw-away code into reusable components when needed.

Moldable development is thus a form of exploratory programming (cf. 2.1.7) on live objects where tools, whether one-off or reusable, are created in a bottom-up approach with immediate feedback available at every step. See *e.g.* figure 3.3 where the compilation of a Processing program is explored for what information about the produced bytecode to show with the goal of having one-off code sufficiently generalized that it can be added as reusable view for all objects of this type (as eventually seen in figure 3.4). Such exploration code could also later be extracted into tests, ensuring that what worked once will continue to work.

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

Nierstrasz and Gîrba [51] identified several development patterns which are common to or supportive for moldable development. 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 initially 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. Extending objects iteratively based on actual needs should also ensure that code is cleanly separated, as during the exploration phase it should become clear where code fits best.

Having a moldable environment also allows for working on code and documentation intertwinedly, similar to literate programming [41]. In contrast 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[7]), for project exploration (similar to Jupyter notebooks[8]) or for project documentation. Another useful pattern for teaching is the "Composed Narrative" that 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), consisting of a Smalltalk Virtual Machine (VM) and runtime environment, a custom user interface and the source code of the Smalltalk code for everything running within it. By default, it persists its entire state into a system image, so that live objects don't have to be recreated at restart and may outlive their original source [34].

3.3.1 Smalltalk VM

Smalltalk is a fully object-oriented language based on message passing[9] that was originally designed for educational use and as a consequence has rather minimalist syntax that is supposed to read more naturally. As a consequence, its syntax limits the use of parentheses, aligns punctuation with natural language (using full stops to end a statement and semicolons to continue a statement by sending another message to the same object) and interweaves a message's name with its arguments[10] [35].

One potential issue for programmers experienced in ALGOL-68-derived languages is operator precedence, which in Smalltalk is limited for simplicity to just three different levels: (1) messages without arguments; (2) binary operators (which in contrast to mathematics and most other languages discussed in this thesis all have the same precedence and left associativity, and are of course also implemented as messages); and (3) all other messages.

[7] *Cf.* https://www.microsoft.com/de-ch/microsoft-365/onenote/digital-note-taking-app.

[8] *Cf.* https://docs.jupyter.org/.

[9] One of many characteristics that Smalltalk shares with Java.

[10] *Cf. e.g.* the `#ifTrue:ifFalse` message in figure 3.5 where each argument follows part of the name. Note that these are not the argument's names, these are declared separately in the definition of `#ifTrue:ifFalse`.

At GT's core is the OpenSmalltalk Cog VM.[11] The Cog VM is open sourced (MIT licensed) and shared with other Smalltalk based environments, in particular GT's predecessors (see 3.3.4). Its source code is written in a subset of the Smalltalk language [37] which is transpiled to C both for performance and for achieving cross-platform compatibility by relying on the various available C compilers. As a consequence, GT runs on Unix systems just as well as under Microsoft Windows.

Smalltalk and the Cog VM are highly reflective, allowing access to all but the most fundamental built-ins. In fact, all messages passed are primarily implemented in Smalltalk, but common operations can be forwarded to native code with a `<primitive:...>` pragma annotation with a fallback being provided in Smalltalk in case the native implementation fails. In particular, the execution context and the compiled bytecode of any message are available for inspection and modification. This allows users to customize the environment entirely to their liking.

For performance, the Cog VM includes a Just-In Time Compiler (JIT) for compiling methods called multiple times to native code on the fly [48]. More recently, for further optimizations, an adaptive optimizer called "Speculative Inlining Smalltalk Architecture" (SISTA) has been introduced by Clément Béra [20], which also enables saving the optimized methods into the image, thus persisting them between restarts of the VM. The bytecodes used for GT are thus those proposed by Béra and Miranda [21] and diverge to some extent from the original Smalltalk-80 bytecode format [35, p. 596], in particular by enabling (more) multi-byte instructions that allow compilers to inline more common objects and code.

With GT being based on a Smalltalk VM, Smalltalk is GT's primary language. Support for other popular languages such as Python, JavaScript or Java is, however, possible by connecting to an external runtime through the `LanguageLink` protocol, *i.e.* by passing serialized objects over sockets [17].[12] Obviously, objects in the other runtime can't be persisted there, however, transferred data and objects can be recreated from persisted objects within the Smalltalk VM.

3.3.2 Moldable Interface

While Smalltalk and the VM are inherited from Pharo,[13] the user interface has been written afresh based on the cross-platform Skia Graphics Engine, which also powers most modern web browsers.[14] Every window is rendered according to a dynamic rendering tree where every element involved (being a Smalltalk object) indicates how it wants to be laid out and the layout is recalculated for all size changes.

In its windows, GT by default provides a tabbed interface that can show one of several tools: an object viewer, a notebook (dubbed "Lepiter"), 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 simply annotating an object method that returns a `GtPhlowView` object with the `<gtView>` pragma as shown *e.g.* in figure 3.5).

In this example, 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 methods 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 function as tested and thus guaranteed to work examples for documentation.

Since one of GT's stated goals is to make systems explainable [34], it provides ample packages for

[11] *Cf.* https://github.com/OpenSmalltalk/opensmalltalk-vm.

[12] The serialization format chosen is either JSON or the more compact binary representation "MessagePack" (see https://msgpack.org/).

[13] *Cf.* https://www.pharo.org/features.

[14] *Cf.* https://skia.org/docs/.

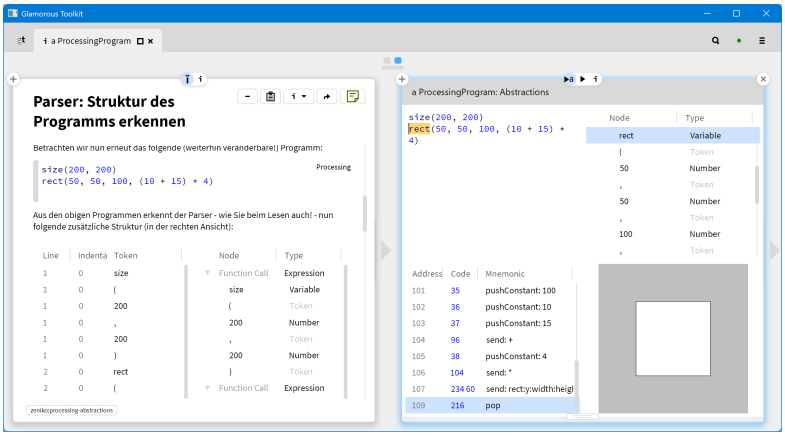

Figure 3.4: GT with a live notebook page (left) and inspectable object view (right)

```
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 ]
]
```

Figure 3.5: Smalltalk source required for creating a custom view.

loading, transforming and visualizing data in various forms such as the SmaCC parser generator,[15] a graph builder [47], *etc.*, but also a built-in explanation system, allowing developers to visually connect arbitrary visual elements by annotating them.[16]

What might take some getting used to: All Smalltalk code and all live objects are stored in GT's `.image` file, which is updated whenever GT is quit with saving.[17] This means that there are no easily accessible source files outside of GT's interface. Synchronization of Smalltalk code thus happens best through GT's built-in git client. Preexisting notebook pages are also stored within one of GT's subdirectories. Users can, however, create new pages in the "Local knowledge base",[18] which can be backed up separately and which are stored even when GT is quit without saving. All notebook pages indicate where they are stored in their footer and can be moved between databases through that footer. This allows students to take an existing page from teaching material and move it locally where it is separately backed up, in case they later delete or update GT.

GT was thus chosen for its moldable environment: different views are easily implemented and can be combined freely with interactions and updates between them.

3.3.3 Bleeding Edge Issues

The developers of GT follow a trunk-only development style without release branches. This means that the release version changes almost daily, with new features being introduced gradually. This also means that subtle issues might be unexpectedly introduced in a release by or as a side-effect of some partially implemented feature. As a consequence, 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, verifying that it works and then distributing *this known good* image.

When GT is used heavily, occasionally some lesser tested code paths might be hit. We've occasionally had some modifier keys apparently lock up, requiring app switching to get keyboard shortcuts working again; we've sometimes hit a cascade of error messages, spawning dozens of debug windows, which had to be closed without other consequences; and occasionally GT seemingly stopped responding, with even the `Ctrl+.` keyboard shortcut not interrupting the running code (luckily, code modifications are backed up and restorable through the "Code changes" tool). Most of these are small annoyances, which more restrained users – such as students – shouldn't encounter often.

Finally, GT is mainly developed under macOS and makes some platform assumptions with respect to its host operating system. This isn't noticeable when working purely within GT but occasionally shows at its seams, with external executables not being located reliably when establishing a link to other runtimes,[19] knowledgebase names containing path separators,[20] or pasting source code from third party apps leading to visual bugs in GT's code editor.[21] We assume that most of the reported issues will have been fixed at the time of reading, though.

3.3.4 Historical Remarks

Smalltalk environments have been image-based and resumable 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 VM, Ingalls, Kay *et al.* started developing a new VM and development

[15] *Cf.* https://refactory.com/smacc/.

[16] In contrast to methods, objects are annotated by sending corresponding objects, in this case a `GtExplainerTargetAptitude` or a `GtExplainerExplanationAttribute` respectively.

[17] Source code changes are in addition tracked separately in the `.changes` journal.

[18] By default, this is located in the `lepiter` subdirectory of the user's documents or home folder.

[19] *Cf.* GitHub issues feenkcom/gtoolkit#4608 for Linux and feenkcom/gtoolkit#4633 for Windows.

[20] Which can be worked around by renaming the database, see GitHub issue feenkcom/gtoolkit#3036.

[21] This applies under Windows, see GitHub issue feenkcom/gtoolkit#4634.

environment at Apple that had the goal to also be customizable by non-programmers [37]: “Squeak” inherited its built-in capabilities for live and exploratory coding from the original Smalltalk and it is back to this point that GT’s heritage is directly tied.

While Squeak was further developed at Walt Disney Media Labs and among other things included in the “One Laptop per Child” laptops, it remained a niche product – likely due to missing interoperability between the live environment inside its VM and outside code. Still, Squeak and its later fork Pharo continued to be worked on and were actively used in academia and related spin-offs. Eventually, a team around Tudor Gîrba set out to implement their idea of a moldable environment on the basis of Pharo, thus creating GT [3]. Version 1.0 has been released in 2023 and is still being actively worked on.

GT thus has an illustrious lineage and has achieved support for many concepts asked for by literature: It is a moldable environment, supports a clean object-oriented 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 its VM.

4

Proposed Solution: A New Teaching Environment for Programming

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

Within GT, 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 AST and resulting bytecode for the GT VM side by side.

In this chapter, we document the architecture of this environment and the reasons for the approaches chosen. If you want to inspect the environment yourself, see appendix A for how to install all referenced code.[1]

4.1 Overview of "Processing Abstractions"

"Processing Abstractions" consists of a transpiler for translating Processing source code into executable objects, a runtime environment, a large collection of views of various aspects of the program, and teaching materials using them.[2] Tools and views as well as materials are all implemented within Glamorous Toolkit (GT), the former in Smalltalk code and the latter as Lepiter notebook pages.

Figure 4.1 shows an excerpt from materials for students: The program visible at the top resides in a Processing-specific snippet with commands for running and inspecting the program independently of the page it resides on at the bottom; below a combined view for one execution step is shown with (clockwise) the step in question highlighted in source code; the corresponding bytecode visible; the program's output up to and including the current step; the contents of the execution stack of the Smalltalk VM; and a list of all variable states.

[1] Remove the line `GtExplorationHomeSection studentMode: true.` in order to also see our implementation notes.

[2] The teaching materials currently provided are in the language used for teaching at the location of writing: German.

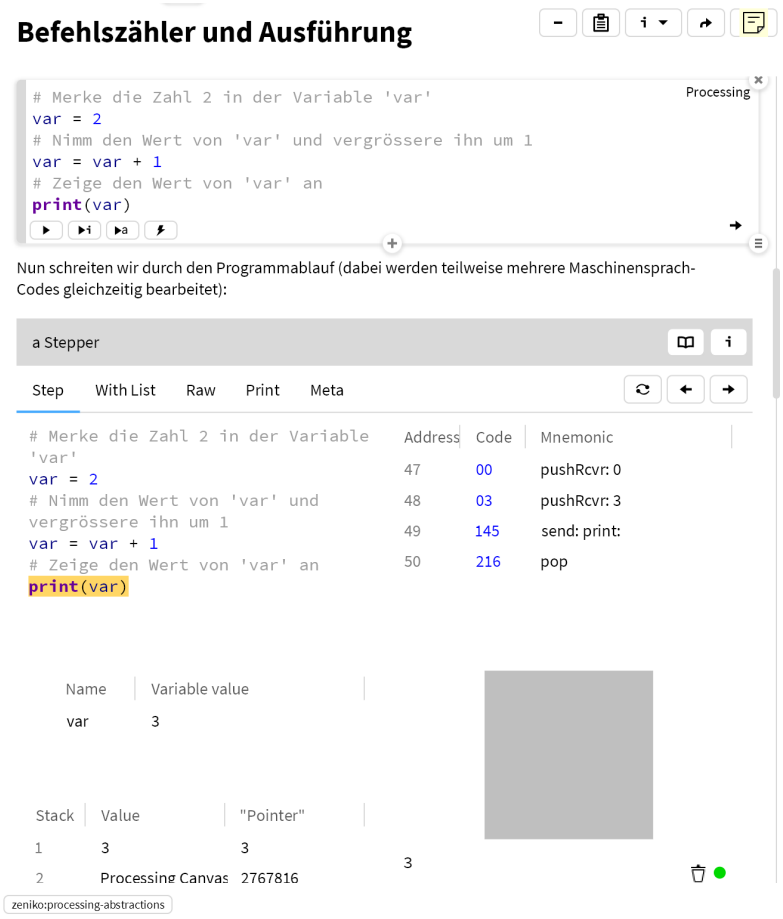

Figure 4.1: Excerpt from an interactive notebook page on program execution in a VM.

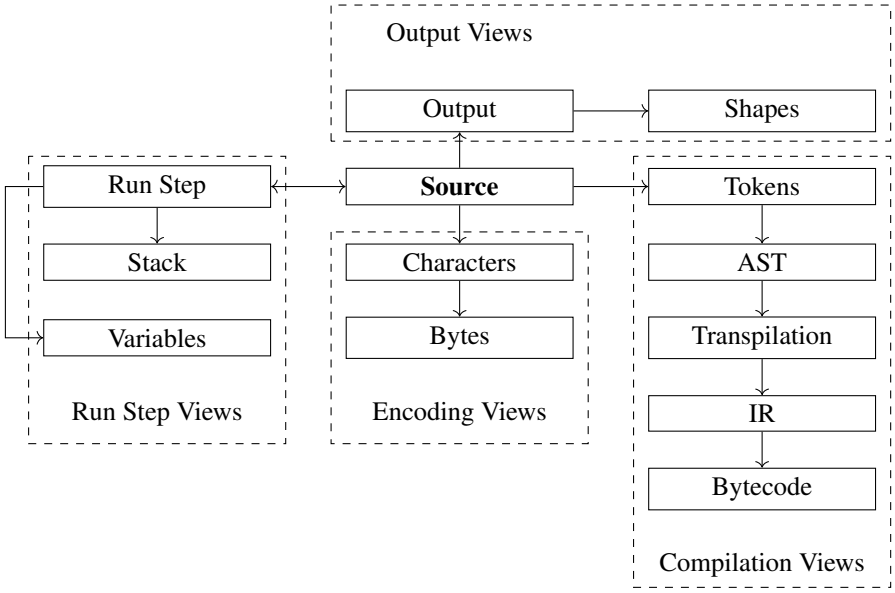

Figure 4.2: An overview of all the views and the order students will step through them

Many such views are updated live whenever the source code is modified, without any other action required by students.[3] The five views shown in figure 4.1 can also be used on their own or in other arrangements (*e.g.* only source and a full bytecode view where interacting with one automatically highlights the corresponding items in the other).

As will be shown below, dozens of premade views are available. Any of them can be embedded in a notebook page by adding an Element snippet with one line of Smalltalk code. The following code *e.g.* connects the first Processing snippet on the current page with a live `gtTreePlusSourceFor:` view:

```
(ProcessingSource fromPage: thisSnippet page at: 1) renderLiveView: #gtTreePlusSourceFor:
```

From these views, interactive teaching materials related to programming, compilers and code execution in a (virtual) machine can be composed, allowing students to combine their preexisting knowledge from programming with concepts of different abstraction levels.

4.2 Exploring Abstraction Levels

Any of the views of a program require its source to be written in Processing with Python syntax[4] available either as a single file or as snippet in a GT notebook. We recommend the latter, as views are then generally live and effects of changing the source code can be more easily explored by students. An overview of available views is listed in figure 4.2.

Screenshots of all views can be found in appendix C. Note that most views show a molded collection of objects that can be inspected individually by double-clicking the corresponding item.

[3]The one exception is the combined run-step view, which for longer running programs would be too resource intensive to regenerate on the fly. Refreshing it manually is still possible.

[4]Restricted to the implemented API as documented in appendix B.

4.2.1 Source Snippet

The Processing/Python snippet used in GT's notebook pages is the only place where Processing source code can be modified. It also provides several ways to run the program:

- ▶ (or its shortcut `Ctrl+R` for "Run") runs the program and either displays its Output view or an inline error message.
- ▶`i` (or its shortcut `Ctrl+G` for "debuG") runs the program, recording all individual steps at the level of Python (sub)expressions – allowing programmers to step through the program's runtime views and inspecting among other aspects the values of variables and the current state of the output.
- ▶`a` (or its shortcut `Ctrl+D` for "Details") shows the Abstractions view, *i.e.* the program's main decomposition states: source code, AST, bytecode, and output.
- ϟ (or its shortcut `Ctrl+Shift+D`) opens GT's Smalltalk debugger at the `gtRun` entry point of the transpiled code for live debugging for either advanced students or for looking under the hood of the API calls.

With just one source snippet, students can write and inspect programs with the opened views updating live as the program changes. This should give about the same experience as the Processing IDE with the main distinction that it is a live environment.

In contrast to the snippet, views are static in the sense that the source code can't be modified there.

4.2.2 Output View

The only views students should already know are the traditional Source and Output views, which show the source code and the result of running the program respectively. Programs containing an animation loop will show the animation and allow user interaction in any output view.[5]

The Source view is meant to be linked to any other view, allowing for interacting between them by selecting source expressions and having the corresponding item(s) in the other view(s) also selected.

The Output view can be used at any state to quickly verify visually that a program behaves as expected, be it during an introduction to programming or when exploring part of a program's execution or decomposition.[6]

4.2.3 Compilation Views

One way to show students what happens to a program before it can be executed on a (virtual) machine is to take them through the steps involved:

- The Tokens view (see figure 4.3) shows each of the tokens the lexer has encountered, including additionally required information such as a line number and line indentation.[7] This view can be used for students to see what a compiler is looking at in their program with whitespace and comments in particular removed.[8]

[5] Interactions are currently limited to mouse movements and clicks.

[6] All these views are directly available for a Processing program when running it using ▶`a` or `Ctrl+D` and then selecting the `i` selector at the top (see figure 3.4 on page 17).

[7] Indentation is relevant, as Python and as a consequence Processing with Python syntax is a language with significant whitespace, using common indentation for denotating code blocks.

[8] As a limitation, tokens are currently extracted in reverse from the parser, which prevents inspecting tokens of syntactically invalid programs.

- The AST view (see figure 4.3) shows the resulting parse tree in a form that differentiates semantically relevant (sub)expressions from purely structural tokens. Combined with the Tokens view, this view can be used for hypothesizing and exploring how and what tokens are grouped together. In order to allow better interaction between Tokens and AST views, the latter is a tree-list. A proper AST Tree view is, however, also provided where the tree structure is more obviously visible, in particular also for larger programs.

- The Transpilation view shows the result of translating the AST to Smalltalk. Since the AST needs to be barely modified for this translation step, Smalltalk code should be relatively easy to understand, at least when the original Processing source is displayed in parallel.

  The Transpilation view also allows students to see what Processing does implicitly behind the scenes: setting and updating implicit variables, such as `width`, `mousePressed`, *etc.*, calling `setup` once and `draw` repeatedly, and running top-level code before entering the animation loop (see figure 4.4). Since GT's code component is used for displaying the transpiled code, this view has syntax highlighting and also allows users to explore the source code of called methods, letting students see that even presumably primitive commands can be implemented out of blocks (or in the case of `whileTrue:` in figure 4.4 on the basis of recursion).

  In order to discuss programming language syntax, two additional views Prefix and Postfix show transpilations into a Lisp-like language using S-expressions and a Postscript-like language with reverse polish notation respectively. These may also serve as a basis for students' own parser projects, translating these pseudo languages back into Processing or Smalltalk.

- The IR view shows the IR generated by the Smalltalk Opal compiler from the transpiled Smalltalk code. With function and variable names still showing and optimizations still missing, this allows students to make more sense of the eventual bytecode, in particular when both views are displayed side by side.

- The Bytecode view finally shows a list of the bytecodes that will be run by the Smalltalk VM[9] with instruction addresses[10] and mnemonics added, so that jumps can be understood and the code can be more easily connected to either IR or any other form of Assembly language. The bytecode of multiple methods is separated with the method name used as separator, as in the Smalltalk VM addresses are counted starting at zero for every method.

All these views can be linked together, so that selecting an item in one view will highlight the corresponding item(s) in the linked views. The default Abstractions view, available directly from every Processing snippet, *e.g.* combines the source with AST, Bytecode and Output (visible on the right-hand side in figure 3.4 on page 17, with a source expression and its corresponding items highlighted).

Being able to walk through all of these various compilation steps is something we haven't seen any IDE offer. Furthermore, being able to connect this with an actually working program – in the form of the Output view – should offer students an explorable and interactive insight into a manageable package.

4.2.4 Run Step Views

Since in the background we have an actual translation of the Processing program to bytecode, which can be executed, we can also inspect the resulting execution. In contrast to the other views, this isn't live but a recording of a run, where we collect and show the following for every Processing (sub)expression being executed:

[9] Unless it is later translated to native code by the JIT.

[10] These are actually byte indices in the method's binary layout as used by the Smalltalk VM.

Parser: Struktur des Programms erkennen

Betrachten wir nun erneut das folgende (weiterhin veränderbare!) Programm:

Processing

```
size(200, 200)
rect(50, 50, 100, (10 + 15) * 4)
```

Aus den obigen Programmen erkennt der Parser - wie Sie beim Lesen auch! - nun folgende zusätzliche Struktur (in der rechten Ansicht):

Line	Indenta	Token
1	0	size
1	0	(
1	0	200
1	0	,
1	0	200
1	0	)
2	0	rect
2	0	(
2	0	50
2	0	,
2	0	50

Node	Type
Function Call	Expression
size	Variable
(	Token
200	Number
,	Token
200	Number
)	Token
Function Call	Expression
rect	Variable
(	Token
50	Number

Aufgaben

1. Welche zusätzliche Information hat der Parser hier nun gewonnen?

2. Schreiben Sie das obige Programm um (oder kopieren Sie ein kompliziertes Programm von einer der vorher angetroffenen Seiten) und stellen Sie sich folgendes Vorgehen des Parsers vor:

   Er liest ein *Token*.

zeniko:processing-abstractions

Figure 4.3: Screenshot of a page of student content showing modifyable Processing source with live views for tokenization (left) and abstract syntax tree (right).

```
_UnpackagedPackage > a subclass of ProcessingCodeBase
draw
    gtCanvas backgroundGray: 192.
    gtCanvas rect: var y: var width: 50 height: 50.
    var := (var + 20).
    (var > (height - 50)) ifTrue: [
        var := 0.
    ].
```

as yet unclassified | instance

```
_UnpackagedPackage > a subclass of ProcessingCodeBase
gtRun
    var := 0.
    self setup .
    [ gtCanvas frameRate > 0 ] whileTrue:
```

```
Kernel > BlockClosure
whileTrue: aBlock
    "Ordinarily compiled in-line, and therefore not overridable.
    This is in case the message is sent to other than a literal block.
    Evaluate the argument, aBlock, as long as the value of the receiver
is true."

    self value  ifTrue: [ aBlock value. self whileTrue:  aBlock ].
    ^ nil
```

controlling | instance

```
        mouseX := gtCanvas mouseX. mouseY := gtCanvas mouseY. mousePressed :=
gtCanvas mousePressed.
        self draw .
        gtCanvas endFrame.
    ].
```

as yet unclassified | instance

Figure 4.4: Screenshot of a transpiled `draw` method and the implicit animation loop

- The Source view shows the full source code with the (sub)expression in question being highlighted. This view is meant to be combined with any of the following views.
- The Bytecode view shows only the bytecodes relevant for executing the current (sub)expression. This will show students not only into how many instructions a perceived single Processing instruction is compiled but also allow them to find that there's a calling convention that involves multiple instructions for every function call.[11]
- The Stack view shows the VM's value stack at the moment of a function call (*i.e.* with the arguments already on the stack). For every stack argument a "pointer" value is also shown as a pseudo address, as effectively that's what the stack will contain. Since actual pointers aren't available from the Smalltalk VM, these "pointers" are, in fact, hashes (except for small integers, which students will have to discover can up to a point be used as themselves).
- The Variables view shows the current values of all local and global variables (except for implicit variables for manageability reasons).
- The Output view in this case shows the output state *after* the step's execution, allowing students to better judge the effects of a function call.[12]

Since a single program run usually takes many run steps, these views are meant to be displayed together in an interface allowing observers to step through the execution. For manageability, when accessed directly from the Processing snippet, this combined view will not contain the Bytecode and Stack views. The full view will thus have to be shown explicitly to students once they have been introduced to the relevant concepts, whereas the reduced view can also be used for debugging the execution of a program during an introductory programming course.

4.2.5 Encoding Views

For a different connection, source code can also be viewed as text that has to be stored somehow. To show this, there are two additional views to source code:

The Characters view lists the source code split into individual characters and shows characters with their Unicode value in decimal and hexadecimal. The hexadecimal notation is for a later comparison with either a binary file viewer[13] or with the Bytes view.

The Bytes view shows the UTF-8 encoding[14] of the source characters, as they are actually written to the disk. The bytes are again shown in their decimal and hexadecimal but also in binary notation. This is meant for students to combine programming also with text encoding lessons.

4.2.6 Other Views

Two further views are provided for delving deeper into the implementation of compiler and runtime:

The Slices view shows a list of all parsed Processing (sub)expressions with the corresponding transpiled expression. This list is used internally for matching Processing code to derivates of the transpiled Smalltalk.

The Shapes view shows a list of all shapes produced by a Processing program, demonstrating that in an object-oriented environment such as GT, shapes aren't just collections of pixels but also objects with their properties, which can be handled independently.

[11] In the case of the Smalltalk VM, this convention consists in pushing the receiver, *i.e.* the object being sent a message, to the stack and in the end cleaning the stack up if the call's result isn't needed.

[12] Else they would have to switch between states to verify that the expected result has actually happened.

[13] Such as *e.g.* https://hexed.it/.

[14] UTF-8 is the default encoding of most modern software, and in particular it is the encoding of both GT's notebook pages and source files produced by the Processing IDE.

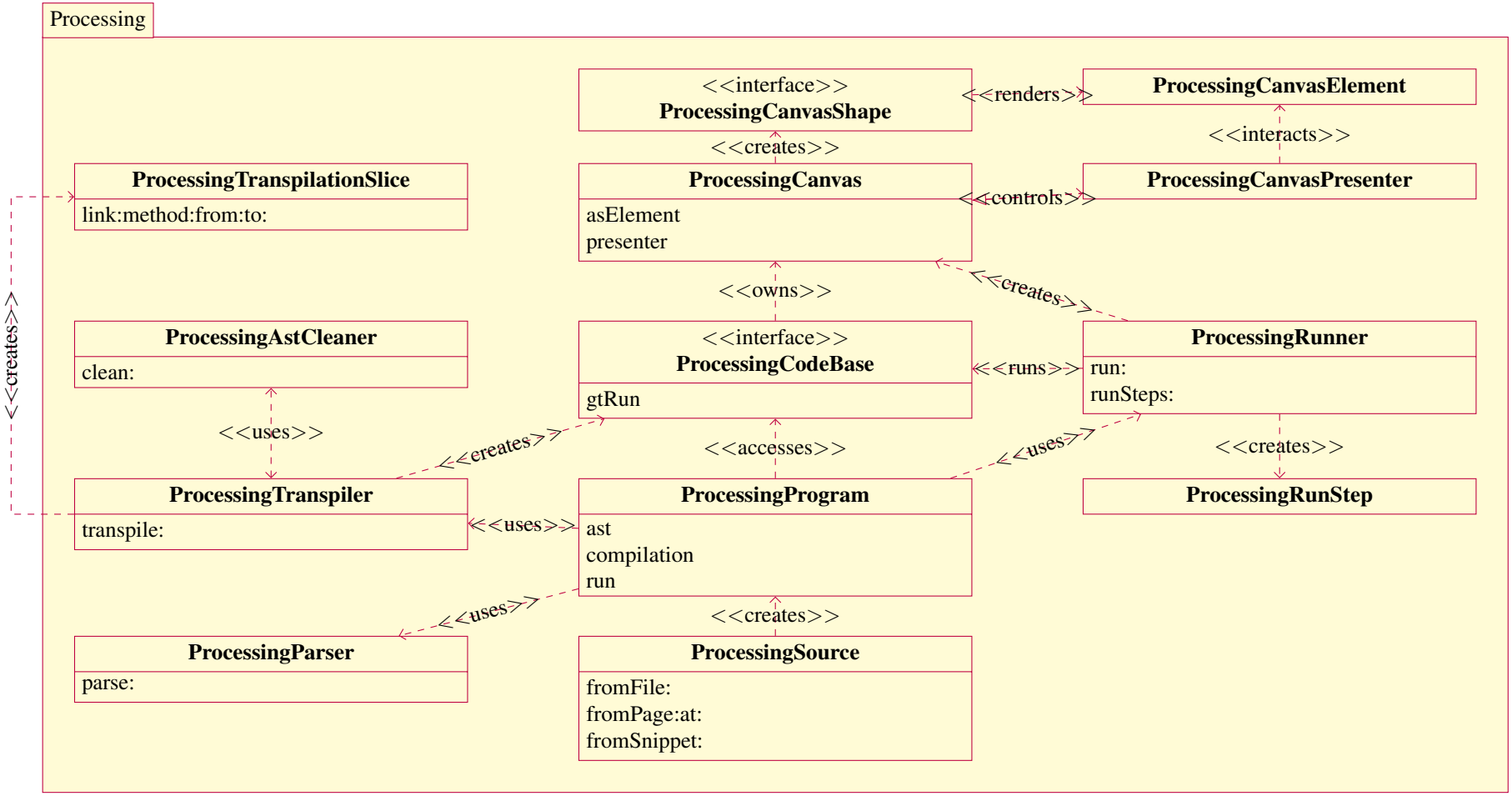

Figure 4.5: Diagram of key classes involved in running Processing code within GT

4.3 Implementation Details

The environment consists of a Processing compiler and a runtime environment both implemented in Smalltalk inside GT. Views were primarily implemented as close to the required data as possible. They are, however, also mirrored inside the main class `ProcessingProgram`, with a subset of these being shown for `ProcessingSource`, which serves as entry class (see figure 4.5).

The implemented code can be found within GT's Coder (package manager) or by name through the Spotter (search engine (🔍)).[15]

4.3.1 Processing Compiler

A Processing program is compiled automatically when loading a source as a `ProcessingSource` from snippet, file or string and then sending the source a `program` message. Alternatively, the following steps can also be performed manually:

The Processing compiler mainly uses GT's builtin `PythonParser` class for parsing the code, which is based upon the SmaCC parser generator.[16] `ProcessingParser` is a transparent subclass, allowing us to intermittently fix issues encountered in the Python parser.[17]

The resulting AST is then passed to `ProcessingTranspiler`, which first rewrites the constructs that Smalltalk doesn't support: compound assignments (*e.g.* `a += 2` is expanded to `a = a + 2`) and unreachable statements after a `return`, which have to be removed. Additionally, logical `and` and `or` operations are converted from being left associated (as per Python's precedence table) to being right associated, which GT suggests for more efficiently handling early exits.[18] This rewritten AST is only used internally and not visible to users.

Transpiling is performed by a tree visitor, translating the individual AST nodes to corresponding Smalltalk with no optimizations. For each node, a `ProcessingTranspilationSlice` is recorded that contains a reference to the AST node and the text span in the transpiled code.

Finally, an anonymous subclass of `ProcessingCodeBase` is created and all Processing functions are then compiled from the transpiled code by the `OpalCompiler` to executable methods. Global Processing code is compiled into the `gtRun` entry point method and, if `setup` and/or `draw` functions are defined in Processing code, the implicit animation loop is also added.[19] Processing API calls – if not overwritten by user code – are translated through messages of the form `ProcessingTranspiler>>>emit_...:`, which the transpiler detects through reflection.

The compiled object can be used as any object. In particular, GT's reflective capabilities can be used, *e.g.* for extracting bytecodes of a method through `CompiledCode>>>symbolicBytecodes`. `ProcessingCodeBase` thus implements all views related to the Smalltalk VM.

`ProcessingProgram` mirrors those views and only implements views related to Processing code. In order not to overwhelm students with too many views, `gtDefaultInspectorTool` has been implemented on `ProcessingProgram` to hide all but the main Abstractions combination view behind the same symbols as used by the snippet.

During compilation, most common exceptions are `SmaCCParserError` during parsing, during transpilation `ProcessingCompileTimeException` and `ProcessingRunTimeException` during execution.

[15] After it has been imported as described in appendix A. Note that a stable snapshot of the code discussed here is available at https://github.com/zeniko/gt-exploration/tree/thesis.

[16] *Cf.* https://refactory.com/smacc/.

[17] It should be noted: All reported parser issues have been fixed within a day.

[18] The Smalltalk compiler doesn't seem to optimize this case. We have not been able to verify whether the optimizing JIT does.

[19] In order to prevent overriding of `gtRun` and the different view messages, Processing names starting with `gt` are renamed to starting with `gt_` during transpilation.

4.3.2 Processing Runtime

The Processing runtime consists of `ProcessingRunner` and `ProcessingCanvas`:

The runner moves code execution to a worker thread, leaving the UI thread available for output and interaction, and performs a simple heuristic for detecting accidental endless loops by simply terminating any program with too long a runtime. This is necessary in a live programming environment, as during composition students will almost inevitably write endless loops. The runner can also be used for extracting runtime steps. In order to achieve this, GT's debugging facilities for stepping through code are called and a debugging session is executed step-by-step in the background. For both use cases, the runner is automatically called by `ProcessingProgram`.

The canvas provides the implementation of most of the Processing API: Every compiled Processing program is assigned a canvas by the runner and all Processing API calls are forwarded to the canvas for rendering. The canvas is implemented using the model-view-presenter pattern, allowing for multiple views for a single canvas. Shapes are thus also only abstractly created by the canvas, being instantiated by the presenter in every single view instance.

4.3.3 Views

All views in GT are by convention provided by methods with names of the form `gt...For:` and are categorized as "views" in the class browser.[20]

With any view visible in GT, `Alt`+clicking on a view's name shows its source method(s), revealing the view's method name and implementation. The method name will be required for embedding a view in a notebook page as described on page 22 above.

For comparing the same views of different programs, the `ViewComparison` helper class is provided, which can be used in Element snippets as follows:

```
ViewComparison newFor: {
   (ProcessingSource fromPage: thisSnippet page at: 1) program -> #gtSourceCharsFor:.
   (ProcessingSource fromPage: thisSnippet page at: 2) program -> #gtSourceCharsFor:.
}
```

4.3.4 Snippet

The Processing/Python snippet is implemented through `LeProcessingSnippet` and associated classes.[21] This is mostly based on the provided `LePythonSnippet`, replacing the use of the Python language bridge with calls to our own Processing compiler and runtime.

The snippet can either create `ProcessingSource` instances by itself, showing either the Output, Runsteps or Abstractions views; or its code can be programmatically loaded in a notebook Element snippet as shown on page 4.1.[22]

4.3.5 Other approaches considered

Since Processing implemented on top of Python is a strongly but dynamically typed language, it maps well onto Smalltalk. Still, initially three other approaches were considered:

Processing could be run either in the original JVM and then accessed through Python or directly run using one of several Python libraries [16]. In all cases, its objects would be accessed through `PythonBridge`.

[20] Actual requirement is only the `<gtView>` pragma annotation and the method signature (`GtPhlowView` → `GtPhlowView`).

[21] *Cf.* figure C.1 on page 57.

[22] The `at: 1` part of the message may also be omitted, if there's only one snippet on a notebook page. Inspect the "views" category of `ProcessingProgram` in the coder or consult appendix C for a full list of availble view names.

Unfortunately at the time of writing, support for PythonBridge under Windows was difficult to achieve in a portable manner (*i.e.* without requiring students to install multiple different packages, which increases the risk of accidental breaking and thus potential support issues). Additionally, `PythonBridge` only gives access to dictionaries of serialized Python object properties, which would have required a potentially slow level of indirection involved when running and inspecting animations.

Alternatively, Processing could have been implemented through an interpreter in GT.[23] This would have required writing a separate compiler for creating bytecode just for demonstration purposes.

As a third option, a compiler from Processing to Smalltalk bytecode could have been written.[24] While this would have allowed for closer control over optimizations, it would effectively have become a reimplementation of most of `OpalCompiler`.

4.4 Potential Drawbacks

Since GT is based on Smalltalk, an initial effort is required to learn language and environment before their benefits can be used. This is helped by Smalltalk's regular syntax and GT's reflective capabilities.[25]

Modrow and Strecker [49] prefer a block-based language in order to prevent students from getting lost in syntax errors. This issue has at least been remedied partially by having a live environment. Error messages are, however, not in optimal shape yet, sometimes rather hinting at an issue than explaining it as Bouvier *et al.* [23] call for.

Chiodini *et al.* [26] also propose starting with visual programming but have different requirements: In order to keep an introductory language manageable, they ask among other things for a limited API that should be expandable by students (see also 2.1.6). And the full Processing API can indeed be quite overwhelming, so only a subset must be introduced at the start. Indeed, also for this reason only a subset has been implemented in GT (see appendix B).

Another requirement by Chiodini *et al.* is for problems to be transparently decomposed and solutions recomposed. This is indeed an issue with Processing: Moving a composed shape to a different location requires adjusting the coordinates of all basic shapes involved, therefore variables and even functions have to be introduced sooner rather than later to allow the examples shown [26] to work. Similar to how they introduce a library to achieve their desired API, the same functionality could be implemented on top of Processing at a later stage if desired.[26]

The main reason for not introducing a new API as proposed by Chiodini *et al.* is the same as the reason for not introducing an entirely new programming language optimized for teaching (as done *e.g.* by Black and Bruce [22]): This prevents benefiting from the large community and preexisting documentation and example code.

Using GT for students also means introducing a new environment, which might introduce new pitfalls and adds additional cognitive tax on students.

[23] Remnants of which are available as `ProcessingInterpreter`.

[24] A compiler for a tiny subset of Processing is included as `ProcessingCompiler`.

[25] For Smalltalk and GT's ancestor Pharo, there are sufficient resources available online, for GT itself, there's the Glamorous Toolkit Book [34] and a Discord server.

[26] In the teaching materials provided, an example of how to implement a simpler Turtle based API is provided (see "Schildkröten und Rekursion"), however even Turtles have state, which makes composition non-trivial.

5
Implementation: Lesson Plans

With “Processing Abstractions”, high school teachers get a toolkit for teaching programming and systems at various levels:

At the start of a course, the environment can be used as an introduction to programming (see the suggested lessons in 5.1). When the limitations of the environment are reached, the move over to the official Processing IDE should be seamless: code copied over and run (with the same icon and shortcut) yields the same output and can then be further modified with the full Processing API once students are ready.

When teaching computer architecture, the environment can be used for either exploring the various abstraction levels involved, or even again as a full course embedded in notebook pages (see 5.2). In case of both usages, this better ties together programming and system architecture. And if programming has happened in Processing, students can investigate their own programs instead of mainly relying on those given by the teacher.

In an in-depth course for students specializing in computer science, this environment can further be used to teach the inner workings of a compiler from lexer to optimizer (see 5.3).

Finally, this environment could also be used as a stepping stone for introducing Smalltalk as a different programming language and GT as the moldable environment it is, leading students to molding the provided materials further by *e.g.* extending the Processing API, developing new views or starting to work on a language of their own (see 5.4).

In all cases, GT can also be used as a digital notebook by students, where they solve tasks directly in the page, add their own notes and keep their modified examples.

While this environment is mainly targeted at high school students, it could also be used in middle school. For middle schoolers, the exposed user interface would, however, have to be reduced as far as possible to keep it manageable. It could nonetheless work at least in a smaller group with interested and motivated students wanting to step beyond block-based programming languages. For university students, there’s currently not enough depth available.

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 content will have to be translated to the teaching language.

5.1 Introduction to Programming

While the "Processing Abstractions" environment has been developed for linking programming and computer architecture, it can also be used for an introduction to programming. This even helps linking programming and computer architecture for students, since they can start at the same point and reuse the experience already gained.

5.1.1 Educational objective

After the introduction, students should be able to ...

- work within GT.
- read, write and understand programs with a limited command set, including working with the implicit animation loop.
- learn from their mistakes, correct themselves and not be afraid of breaking things.

In the end, students should have a solid foundation for taking on the task of writing a basic but still interesting app or game.[1]

5.1.2 Prerequisites

Students need experience in using their own computer, including ...

- downloading and extracting archives, and
- dealing with their virus scanner.[2]

Additionally, the teacher must prepare their content in GT *e.g.* as described in A and distribute them. Ensure that the first lesson page will be displayed on first startup.

5.1.3 Introduction to Glamorous Toolkit

Since GT will be a new environment altogether for all students, some basics on its usage have to be introduced first within ten minutes:

→ Ask students to download and extract your GT distribution as homework for the first lesson. Under Windows, they might run into a first issue with their virus scanner blocking the download, so remind them about what to do in that case.[3]

→ Tell students to start GT and open the notebook page about working with GT ("Arbeiten mit Glamorous Toolkit" in the included teaching materials). Also ask them to help their neighbours, if they see them struggling. Use the time while students are reading and doing the first tasks for ensuring that everyone has managed to get GT running.

[1] Possible ideas, which all have been implemented by students, include: "Flappy Bird", "Geometry Dash", "Pong", "Doodle Jump", "Snake", a quiz, a labyrinth, *etc.* For most of these, "Processing Abstractions" provides sufficient support, the main deficit being the lack of keyboard input.

[2] At least under Windows, many scanners flag GT as untrustworthy due to its executable lacking a valid signature. Some virus scanners even block the entire download if the archive is distributed over a network.

[3] In most virus scanner settings, the warning can be overruled by the user; else they might have to *temporarily* disable scanning for this one download.

→ Introduce GT as an interactive notebook similar to what students already know from class.[4] One main difference will be that additional views will open to the side, hiding the table of contents. So show them, how to get back by selecting the left-most view (through the blue dots at the center top) and expand it (through the ⊕ at the top left).

→ If you want students to be able to take notes of their own inside GT notebooks, show them how to move a page to their local knowledge base (where it can be backed up individually) and how to create and structure new paragraphs (`Ctrl+Enter`, `Alt+Shift+Arrow`). Markdown syntax for formatting does not have to be introduced explicitly, as that can be replicated by students by inspecting your content.

→ Finally, keep in mind GT's ability to be inspected at every level and point more advanced students towards using these (*e.g.* by `Ctrl+Shift+Alt+Click` anywhere, by double-clicking on list items, or by going through the different views of an object).

Please note that as described in 3.3, GT is bleeding edge technology and might not always behave the way you and your students have become used to: Since notebook pages are rendered progressively, scrolling won't always work smoothly (and will scroll inner content, if the mouse cursor isn't positioned at a page's border); occasionally unexpected error messages cause a debugger window to pop up, which has to be closed again; and sometimes the keyboard modifiers may get stuck, resulting in shortcuts no longer working (which is resolved by switching to a different app and back).

5.1.4 Lesson Plan

Since this isn't the focus of this thesis, we won't go into much detail about introducing programming but just summarize what might work within the provided environment:

Start top down as suggested in 2.1.5 from either abstract art or games[5] and start dissecting how this concrete entity might be described, first in natural language and then in formalized language.

Then follow Reas and Fry [54] and introduce the Processing language (see 3.1), by starting with a sequence of statements with few commands (*e.g.* `size`, `rect`, `ellipse` and `fill`), and then sequentially introducing comments, colors, variables and arithmetic, the animation loop, conditions and interactivity.

Every concept can be described on a notebook page with given examples for first exploring the effects of a command and its arguments[6] and then later using the concept for solving problems by starting from a skeleton program (see figure 5.1 for an example of both).

At the beginning, debugging will only consist in modifying values and observing live changes until the desired output is reached. At later stages, the step-by-step execution views will be useful, where executed expression, variable values and visual output are displayed side by side and execution can be played forward and also in reverse.

Quicker and more proficient students could also already at this point start to delve deeper: The third and fourth execution icons will lead them to discover by themselves what lies under the hood.

Sample content for this is included in the *Unterrichtsmaterialien* of "Processing Abstractions" as "*Programmieren mit Processing*".

[4] Many schools have standardized their IT infrastructure and use the Microsoft 365 Office Suite, which includes OneNote.

[5] Most one-tap games should work, *e.g.* https://www.lessmilk.com/almost-pong/ has reasonably simple gameplay with minimalist visuals.

[6] Students may either describe the expected behavior and then verify that their expectation matches the outcome, or less taxingly try given or random values and then describe their observations. Variations consist *e.g.* in a student describing desired behavior and then (another student) trying to achieve this.

Runde Farben

1. Verändern Sie im folgenden Programm die Zahlen bei `fill` und führen Sie es aus. Welche Farben erhalten Sie für die folgenden Kombinationen?

```
size(200, 200)
fill(255, 0, 0)
ellipse(100, 100, 50, 50)
```

Processing

```
fill(0, 0, 0) =
fill(255, 0, 0) =
fill(0, 255, 0) =
fill(0, 0, 255) =
fill(255, 255, 0) =
fill(255, 0, 255) =
fill(0, 255, 255) =
fill(255, 255, 255) =
```

2. Erstellen Sie Programme, welche die folgenden Ausgaben erzeugen:

2a) Fast die Flagge Japans:

```
size(200, 200)
```

Processing

zeniko:processing-abstractions

Figure 5.1: Excerpt from an interactive notebook page with exploratory and live programming tasks.

5.2 Lesson on Computer Architecture

The main objective of this thesis was to demonstrate, how understanding of what happens when executing a program could be improved by having students perform a *Sichtenwechsel*. The same goes for the other way around, where computer architecture is explained in more concrete form by tying it to previous programming experience.

Introductions to computer science, which extend beyond a pure programming course, often contain lessons on computer architecture. *e.g.* the curriculum [14, p. 145] asks for students to "know how computers and networks are structured and work".

We again propose to approach this task top down (see 2.1.5) and move along the path specified in figure 2.1. This will be outlined here with validation of part of this approach following in 6.1.

With respect to manageability (see 2.1.6), it is proposed to mostly skip the inner workings of a compiler and treat that in a separate sequence (maybe only for students specializing in computer science).

5.2.1 Educational objective

After these lessons, students should be able to ...

- explain the concepts of (virtual) machine, memory, stack, machine language, and program counter (with relation to a von Neumann model).
- elaborate why machine language must be different from a high-level language such as Processing.
- explain why some commands are slower than others.
- connect their knowledge about encodings with how values and machine code are stored.
- consequently understand how one of their programs might be run on actual hardware and document their understanding with correct terms.

5.2.2 Prerequisites

Students must already have basic programming skills in a high-level language. In particular, they must know about variables and loops. If they don't know Processing or at least Python yet, a brief introduction (maybe along the ideas proposed in 5.1 above) is required.

5.2.3 Lesson Plan

Whereas Nisan and Schocken [52] introduce their bottom up course with a top-down overview, we propose doing the same in reverse for this top-down approach:

→ Start with a brief repetition on programming by giving students a few quickly solved tasks (including one about the animation loop, where at least the skeleton structure is given). If this is the students' first encounter with GT, use this opportunity to introduce it (see 5.1.3).

→ Show students the innards of a computer and ask where their programs reside in there. Use this for discussing hard drives and volatile memory, and a repetition about encodings and how everything boils down to `1`s and `0`s (or current and no current respectively).

→ Briefly start at the desired lowest abstraction layer, *e.g.* (light) switches, and explain how they are used for creating processors.

→ Ask students about games they play and have them describe possible player *actions* and compare them with actual player *input*.[7] This repeats top down abstraction decomposition as maybe already encountered in the introduction to programming lessons and allows teachers to bring the foundational idea of multitier architectures back to students' minds.[8]

→ Have students go back to a sample or one of their own Processing programs and let it (again) be displayed in machine code. Provide them with several examples, *e.g.* with variable assignment, with conditionals and loops, and let them assemble the required machine code mnemonics and byte values.[9] Suggest variations for the example, so that even with less programming experience they may find regularities.

→ Collect their findings and group their arguments: pushing to the stack/accumulator (machine codes 0–83), returning (88–94), binary arithmetic and comparison operations (96–111), common commands (112–127), message sending/function calls (128–175), jumps (176–199), storing results (200–216), combined multi-byte instructions (first byte: 224–255) [21, p. 12].[10]

→ Take a step back: What is relevant for running a program? This should yield memory, (arithmetic) operations, input and output and control flow. Let students go back to their examples and see if Processing code doing for one or multiple of these core concepts map to the found machine codes.

→ Introduce von Neumann architecture (see figure 5.2) as an abstraction of the encountered concepts. What physical parts of a computer belong to the idealized concepts?

→ Discuss VMs in comparison to physical machines[11] and the difference of stack and register machines, VMs tending to the former and physical machines to the latter.

→ Go back to Processing and inspect a combined execution view (see figure 4.1). By stepping forward and backward, students should be able to observe the role of the stack and the program counter. Let them explore execution flow and write down what role the program counter has and what the calling convention for Processing API calls seems to be.

→ As a break or maybe as homework, let students play Human Resource Machine[12] [4], which introduces students to a different architecture. The first six levels should be doable for everybody, with quicker students finding sufficient challenges later on.[13]

→ Compare the two instruction architectures with relation to expressivity and register- *vs.* stack-based design.

→ Finally, treat circuits, logic gates, transistors and maybe down to silicon outside of GT to the desired depth.

---

[7] Alternatively, if starting from art instead of games, decompose drawing *e.g.* a house into individual brush strokes.

[8] Multitier architectures might already have been discussed as part of networking lessons.

[9] Ideally, distribute the examples over small student groups, two groups per example.

[10] Students should at this point be able to explain where these seemingly odd number ranges stem from.

[11] VMs being used for portability reasons such as the Smalltalk VM, the Java VM, Microsoft's dotNet VM or any browser's VM for WASM.

[12] The Hour of Code edition is free to download for Windows and macOS.

[13] You can also come back during compiler lessons, as "Human Resource Machine" has implementing various optimizations as additional challenges.

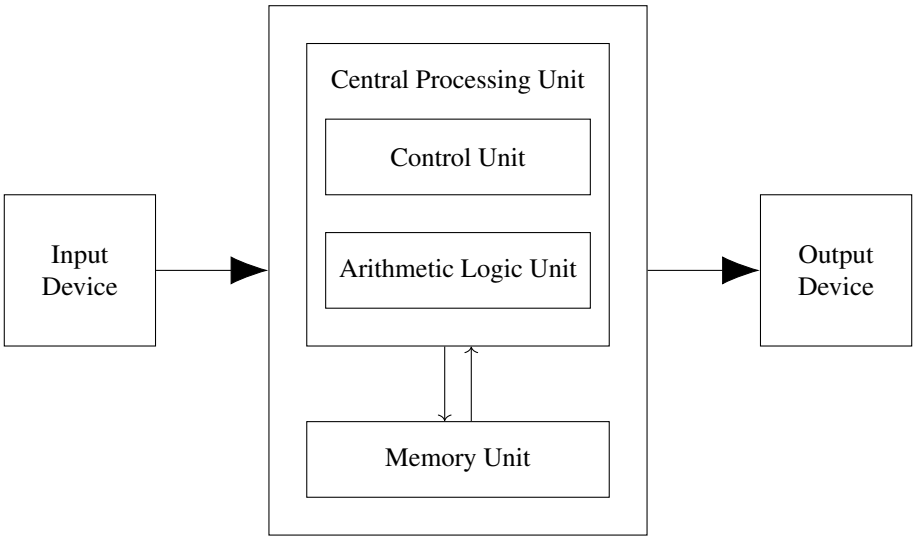

Figure 5.2: Von Neumann architecture.

5.3 Lesson on Compilers

Once students have an approximate understanding of what a program must be transformed into in order to be run on actual hardware or inside a VM, the question remains as to how the transformation from source code to machine code happens.

While this is not usually in the focus of a general introduction to computer science beyond the superficial treatment occurring during the lessons on computer architecture, this is quite relevant for students specializing in computer science.

However, using the environment presented here, this topic could just as well be used either for all students or for quicker students to work on mostly by themselves. Part of the lesson presented here has indeed been validated under these circumstances (cf. 6.2).

5.3.1 Educational objective

After these lessons, students should be able to ...

- explain the difference between high- and low-level languages.
- enumerate the steps required during compilation, using proper terminology (lexer, parser, transpiler, compiler, optimizer).
- connect this knowledge with their knowledge on computer architecture and programming.

5.3.2 Prerequisites

Students must have experience in programming and computer architecture, *e.g.* from the lessons described above (cf. 5.1 and 5.2). It helps, if their programming experience is based on Processing or at least Python. Additionally, they need to have GT installed and introduced (see 5.1.3).

5.3.3 Lesson Plan

The natural way to progress on understanding a compiler is top down, which is the same as the chronological order:

→ Before starting to fill in the content missing from the previous lesson on computer architecture, begin with a repetition on how a program is run in a VM (see figure 4.1), letting students run again a given program or one of their own and seeing into what commands their code was translated.

→ As an analogy, analyze with students how they would go about translating an unknown language using just a dictionary: splitting the text into words where punctuation yields some structure; then looking words up (where again several words might belong together for the sake of translation[14]); writing the translation down; and finally rewriting it, so that it sounds more akin to how you'd write it in the target language. This analogy yields a fitting overview over the different steps involved.

→ Now let students work through discovering the individual concepts encountered:

- Have them compare a program and the resulting lexed tokens: What modifications to a program result in different tokens, what modifications don't cause changes? What does this correspond to in the analogy about translating a natural language?
- Have them compare the abstract syntax trees resulting from their program(s). Are they able to explain how the parser structures the tokens together? How would they proceed if all they saw were just two tokens? (For this, use a view of tokens showing just two lines.)
- Discuss how a recursive descent parser works through tokens. Since Python and thus Processing based on Python uses indentation for command grouping, also highlight the need for tracking indentation at least per statement.
- Optionally have students compare given Processing source code with the resulting AST and transpiled Smalltalk: How literal is the translation? What is different between Processing and Smalltalk? Have them try to invent a syntax of their own that is easier to map to the AST (and later discuss Lisp and Forth syntaxes as examples for alternative, easier to parse syntaxes[15]).
- Since usually an IR is directly produced from an AST, collect common statement types (variable assignment, function calls, loops, ...) and have them translated live to IR. Let students work out common patterns to see what translation might take place.
- Have views for IR and actual Smalltalk bytecode side by side for students to figure out what transformations remain and why these might be required (comparing with their knowledge from computer architecture lessons).
- Should students have prior experience with "Human Resource Machine" [4], have them translate Processing to a program in that game's simplified language and translate code for a level of that game back to Processing (and from that Smalltalk bytecode with the environment's help).
- For the final step, offer some simple code examples that can obviously be simplified.[16]. What optimizations are present in the IR and which in bytecode? What patterns could be optimized? Also discuss the difference of optimizing for size, for speed and for quick compilation.[17]

[14] Compare *e.g.* the meanings of "to see", "to see to", "to see through" and "to see out".
[15] Views for variants of both are available.
[16] *e.g.* `print(1+1+1+1+1)` or `for i in range(10): pass`, both of which should be trivially optimizable
[17] For players of "Human Resource Machine", the first two are the bonus challenges for all later levels.

→ What can't be shown yet using "Processing Abstractions" are the equivalent of optimizations within a processor (or already the x86, IA64, *etc.* code produced by the Just in Time compiler). Students should at this point, however, have sufficient understanding of the various transformations that they might be able to understand what happened *e.g.* with the Spectre attack [42] and why understanding of machine code might be relevant for timing attacks.

5.4 Further Lesson Ideas

Whereas the environment presented here is mainly targeted at introductory high school classes, it might also be employed in a specialization course. Since this isn't this thesis' focus, we'll just list a few ideas:

- Smalltalk is a rather elegant language and has aged quite well. It can still be used to introduce students into pure object-oriented programming with a somewhat unusual syntax for them, and such an introduction could use transpilations from "Processing Abstractions" as a stepping stone, if students have already used it.
- Since the code provided here is run dynamically in GT, it can also be molded further, *e.g.* to implement as of yet missing support for object-oriented programming.
- The provided code might also be used as a template for students to implement their own programming language or see if they manage to write an interpreter for one of the two provided prefix and postfix pseudo languages.[18]

[18]This might already have been discussed in the compiler lessons above. Very briefly, a program such as `a = 42; print(a)` is translated to `(= a 42) (print a)` and `$a 42 = a print` respectively, which both should be simple enough to parse.

6
Validation

The content produced for this thesis has been used with two courses. Evaluation of the feedback provided by students show that they did like working with the provided environment and that their understanding of the abstractions involved in programming increased somewhat. However, due to limitations in the study setting, further analysis will be required in order to generalize these findings.

Both courses were held with classes at Gymnasium Neufeld that we have been teaching ourself for one and two years respectively. These classes consist of Swiss high school students at ninth and tenth grade (of twelve grades) respectively.

6.1 First Round

6.1.1 Setting

The first evaluation round took place in a computer science class consisting of 17 tenth grade students. This class had at this point encountered most of the base curriculum of computer science as required in the Canton of Berne [14, p. 145–146] with only the introduction to systems architecture missing.

Specifically, the introduction to programming had been taught using Processing with Python syntax inside the Processing IDE, so students had already the desired prior experience in programming with Processing. Additionally, the class had written their last partial exam five weeks prior, which had among other topics contained a repetition sequence on binary numbers and encodings.

Contrary to the suggestions in 5.2, the introduction to computer architecture was being taught bottom up, loosely following Nisan and Schocken's ideas [52] in a significantly abbreviated course of only 6 instead of 12 weeks.[1]

At the point where the environment developed for this thesis was introduced, students had already encountered some of the foundational building blocks of a modern microchip: transistors built out of semiconductors, logic gates and circuits up to an adder circuit as the basis of an Arithmetic Logic Unit (ALU).

[1] The reason for this was originally the same as Nisan and Schocken's, *i.e.* building concepts on a solid foundation.

The plan was thus to connect the already encountered foundation with their knowledge about programming by revealing and discussing several of the involved abstraction layers. Due to time constraints, only an excerpt of the sequence proposed in 5.2 could be realized.

At the day of the lessons, 14 of the 17 students were present. At the start of the first lesson, GT was distributed through the school's OneDrive infrastructure. The GT environment was then introduced similar to 5.1.3 but, as was the rest of the lessons, mostly in a self-guided way with instructions being provided in GT notebook pages.[2]

During the lessons, students were supported where needed but left to work at their own individual pace, which the reduced number of students allowed for. Afterwards, a questionnaire[3] was distributed to students for receiving their own feedback in addition to the collected observations.

At least one of the students missing the lessons was successfully able to work on the provided content on her own.

6.1.2 Observations

During the time available, students have been able to work on their own for large amounts of time with only few common issues occurring. The interactive notebook pages seemed to allow for creating an effective teaching environment.

Additionally, many students have been observed to actively tinker with the interactive elements, as was desired and was to be expected from providing an environment for live programming and exploration. Except for a few hiccups where the GT notebook pages stopped updating (for which the usual cure of "reloading" helped), the interactivity worked reliably – up to the point where students found it so engaging that they got sidetracked by writing and modifying programs for their effect instead of the changes to the views for different layers.

Nonetheless, the more active students have been able to work through the subject matter on their own, whereas less interested students had to be motivated from time to time to continue reading and interacting. With students being able to work on their own, we had ample time for supporting these students with instructions, hints and some motivating background information.

Despite their prior Processing knowledge, students were sometimes out of their depth when changes to a Processing program were asked for. In a next round, this sequence would have to be placed closer to a programming sequence with Processing or at least a brief repetition of just using Processing would be helpful.

What caused most issues was the way GT opened notebook pages from content links in a new page adjacent to the previous one, hiding the table of contents in the process, instead of opening them in place of the previous page as students were used to from web browsers. This caused students to lose track of the pages they were supposed to be working on, to the point of occasionally skipping part of the assigned content. This happened despite the brief introduction to working with GT where closing additional pages and getting the table of contents back was an explicit introductory task.

The overall impression of the lessons was that students had been working productively, mostly autonomously and at their own pace with the teaching materials provided.

6.1.3 Student Feedback

In order to verify our own observations, students were provided with a questionnaire for providing feedback. Of the 17 students, however only 11 returned feedback despite frequent reminders. The following yields

[2]The exact state of the materials is available at https://github.com/zeniko/processing-abstractions/tree/thesis in commit `71047704f7f70c13d3d01ac520618e15d569274f` of May 12th.

[3]See appendix D.1 on page 66 for the full questionnaire.

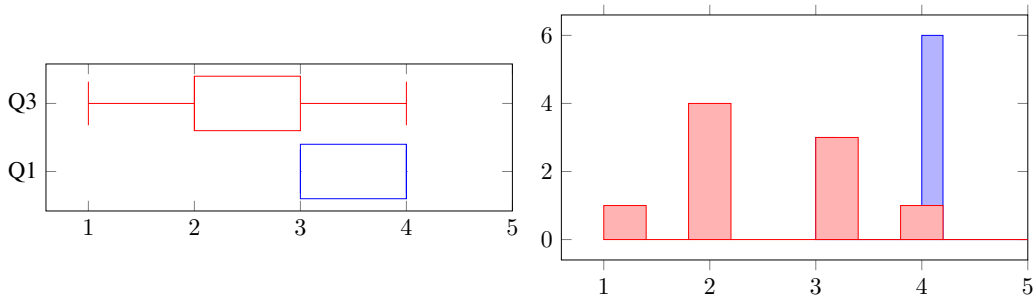

Figure 6.1: Answers to questionnaire questions 1 and 3: How well did you like working with this environment? How well do you think you could explain what you've learned today?

thus at best qualitative results.[4]

Additionally, three weeks later, the students have written another partial exam with individual tasks referring to the lessons with GT.

Student feedback shows the following: Students quite liked working with the provided environment (grading it mostly 4/5 with a bootstrapped 95%-confidence interval of between 3 and 4) and reported that it worked reasonably well but not yet perfectly (most students grading it either 3/5 or 4/5 with a bootstrapped 95%-confidence interval of between 2 and 4). This is consistent with our own observations.

Part of the reason for their liking working this way might be due to their noting programming (and one game programming project one year ago) as what they liked most about their computer science class with half the students naming this their "highlight".

When asked explicitly about the usefulness of the various abstraction views provided, students noted that they were very useful (mostly grading it either 4/5 or 5/5 with a bootstrapped 95%-confidence interval of between 4 and 5). Also, a majority of students indicated repeatedly interacting actively with the program samples.

What they liked the most was being able to work at their own speed (and optionally being able to decide for themselves whether to work together or alone). This is due to guidance from the environment, which allows teachers to introduce tasks for students to explore autonomously in order to build understanding. One student even explicitly noted that being able to see changes reflected instantly was gratifying.

Their main concern was two of the more engaged students noting that some explanations were not yet as clear as they could be, requiring them to ask instead of being able to work for themselves. Additionally, the quickest working student would have preferred fewer links within notebook pages, reducing the annoyance of losing the table of contents from web browser habits.

Students gave their feedback between one day and one week after the lesson. When asked to reword the learned content in their own words, most of them failed to describe what they'd learned entirely correctly. Would their answers have been graded, all but one student would have only been awarded at most half the points. Also, the desired connection with the previously taught content about transistors and logic gates remained unclear with this time all students getting at most half the points.

Finally, the partial exam resulted in students answering questions related to these lessons only about 46% correctly. The result does, however, weakly correlate with students' general commitment (measured by their final grade, yielding a rank correlation of 0.48).

6.1.4 Learnings

The desired effect of having students better understand abstraction layers seems to not yet have been achieved. While the environment was engaging and led students to explore on their own, explanations will

[4]Questionnaire data in anonymized for is available at https://github.com/zeniko/gyminf-thesis/blob/main/data/data_6_1.csv.

have to be expanded and made clearer for students to be able to fully understand what they are shown.

Part of the issue is, however, that time was too short and the implementation not fully fledged, so these findings might also be due to both of these. So further analysis will be required, and the environment will have to be tested at a larger scale with the other classes with better integration in the curriculum (as suggested by the author in 5.2).

At least some of the technical issues observed have since been remedied, although mostly by more explicitly telling students what to do when issues arise.

Also, since GT runs purely on the students' own computers, their progress can't be observed other than by monitoring their screens. For this, either a separate progress tracker (such as https://learningview.org/) would have to be used or a sequence of short tests not only checking for progress of reading but also of comprehension.

6.2 Second Round

6.2.1 Setting

The second evaluation round took place in a computer science class consisting of 23 ninth grade students. At that point, this class had passed about half the required content of the base curriculum to computer science [14, p. 145–146], including application usage, various encodings, algorithms and an introduction to programming using Processing eight weeks prior.[5]

Two lessons at the end of the school year could be set aside for an introduction to compilers as part of this thesis. These would usually have to be placed later in the curriculum, either together with the introduction to computer architecture or beyond.

The plan was to implement an excerpt of the course from 5.3. As a quick overview, "Human Resource Machine" was used for introducing students to the limitations of machine language and motivating the need for compiling programs before executing them on actual hardware. Afterwards, GT was introduced with additional stress on using the table of contents for navigation. Finally, students were asked to work through the provided content at their own speed.[6] Towards the end of the lessons, students were given time to fill out a questionnaire.[7]

One unplanned limitation of these lessons was summer being early with high temperatures. As a consequence, only 15 of the 23 students were present for the introduction and the introduction had to be moved to a different, cooler location.

6.2.2 Observations

In general, the students worked reliably with the content provided. In particular, this group seemed to quite naturally take notes within the GT notebook pages, making the content their own.

Working speed again was heterogenous, but some smaller groups formed, which supported each other. One student in particular volunteered repeatedly to help his peers.

Despite programming with Processing being rather fresh, fewer students seemed to interact with the sample programs provided, despite tasks asking them to do so explicitly. About half the students seemed content to observe views as static content.

This group had more problems getting GT even to run. Even though these students had already successfully downloaded and used apps on their own, somehow despite a separate GT launcher in the top-level folder being provided, many failed to start GT on their own.

[5] For timing reasons, unfortunately the programming project had to be postponed.

[6] The exact state of the materials is available at https://github.com/zeniko/processing-abstractions/tree/thesis in commit `6e22cddb176fdd46d410b9db40496bafa8a59c08` of June 30th.

[7] See appendix D.2 on page 68 for the full questionnaire.

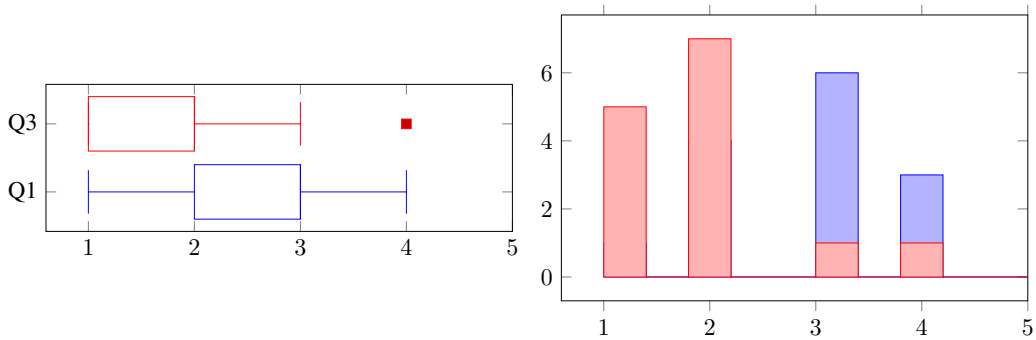

Figure 6.2: Answers to questionnaire questions 1 and 3: How well did you like working with this environment? How well do you think you could explain what you've learned today?

Part of these issues might, however, relate to high temperatures making it more difficult for students to focus.

6.2.3 Student Feedback

Of the 15 students present, 14 managed to hand in the questionnaire (with the last student's computer previously running out of battery power).[8] With this small number of answers, again no reasonable quantitative evaluation is possible.

The students' answers show that many of them (12/15) have worked slower than expected, only learning about lexer and parser in the hour provided. Also, disappointingly only two of the 15 were at least somewhat confident that they would be able to explain the learned content to their peers.

This is consistent with students failing to answer basic questions about the need for compilation (half of them not answering or answering entirely wrong) but slightly better about the roles of lexer and parser (one third answering correctly and one third getting at least half of their answer correct).

It is difficult to judge how much of this is due to the environment not meeting expectations and how much is due to summer, as when asked explicitly about their thoughts of the provided environment, students wrongly referred to the general learning setting instead of the "Processing Abstractions" environment.

The only general remark that students agreed on was that they had enjoyed programming (11/15), which in contrast to the other class (see 6.1), however, still did not result in as much tinkering (with only 4/15 students reportedly playing around and exploring).

6.2.4 Learnings

Unfortunately, not much was to be learned from this test, mostly due to environmental factors. A repetition of this setting will thus be required at a later time. At least the content that students actually got to seems to have been sufficiently clear for them to somewhat understand.

[8] Questionnaire data in anonymized for is available at https://github.com/zeniko/gyminf-thesis/blob/main/data/data_6_2.csv.

7 Conclusion

In order to better understand complex content – at least with regards to computer science –, didactic literature recommends that students perform a *Sichtenwechsel*, *i.e.* observe the same entity at different abstraction levels, in order to better understand the subject at hand. As part of our teaching computer science to high school students, we've been particularly interested in being able to do so with regards to popular programming and less popular computer architecture.

While various programming IDEs provide some possibilities to get relevant insights into a computer's inner workings, none of them offer an all-in-one solution as a building block at a complexity level manageable for high school students. As part of this thesis, the new teaching environment "Processing Abstractions" has thus been introduced, implementing a compiler and runtime for the Processing language inside of Glamorous Toolkit (GT), which allows teachers to flexibly combine various forms of content in a unified environment. On the basis of the working Processing system, a wide variety of views of various aspects of it has been added, allowing students to inspect a program from source code to machine bytecode and through its execution.

This enabled the creation of interactive teaching material that engages students through its liveness and by allowing teachers to lead them at their own pace and depth through the material, having students explore and experience what steps computers have to take for transforming their idea of a program into something sufficiently concrete that can actually be executed in a (virtual) machine.

Working with students showed that the environment mostly worked and managed to get students involved. While students seem to have learned through their own investigation into lower abstraction levels, significant effects could not (yet) be measured. This was in large part due to both a small sample size and too brief an observation window.

Further usage and studies will be required for verifying our initial assumptions. This remains planned for the school years to come. Additionally, the foundational idea of abstraction levels has to be introduced alongside explicitly, if students should be able to get a better understanding of when abstractions might leak and how that could be relevant.

7.1 Future Work

Continuing this path further, there are multiple ways to proceed. On the one hand, there are several aspects still missing from the environment itself. On the other hand, the same concept of *Sichtenwechsel* might be applicable in other domains as well. Finally, a larger study confirming that this kind of approach is empirically sound is required.

Within the environment itself, there are several kinds of views that we feel are still missing. In particular, at this point most views show a single state along abstraction deconstruction. Visualizations of how to get from one to the next are still missing and would have to happen mainly in students' minds for now. Even just animating the Runsteps view instead of students having to click through, with differences from one to the other step being highlighted, might help. More interesting would, however, be animating the process of parsing tokens into an AST or more ambitiously the process of taking Processing source code, parsing it and then translating it.

Also, most lists provide access to native GT objects, which have views not optimized for students. Instead of *e.g.* giving direct access to a bytecode object, which only contains a pointer back to the Smalltalk source, these could be wrapped or extended, so that their default view continues to be informative such as showing a short explanation of what the selected command does.

Then the machine code shown to students is for a stack machine. Common microprocessors, such as those of the x64 architecture at the heart of our students' computers, are, however, register based. Compiling code to a matching machine language by *e.g.* transpiling it to C and then having it compiled to x64 machine code would be a possible approach. Another approach would consist in directly translating it to Intel Assembly or corresponding bytecode. In order to then run such code, a matching virtual machine or a better way to show intermediary processor states would be required, if the execution steps are again to be observed.

Two other steps that are missing for inspecting the process of compilation are type inference and optimizations. Processing and Smalltalk are both dynamically typed languages, which circumvents type checking considerations during the compilation to bytecode, as type differences are handled through inheritance and only optimized by a JIT. Exposing type information and visualizing a type inference algorithm such as Hindley-Milner could be added. Python's syntax would even allow type hints, which could also be used to allow students to experiment with types.

Optimization on the other hand happens at various abstraction levels. With regards to choosing the right algorithm for a problem, a program's runtime behavior could be timed and shown. Alternatively, most programs written by high schoolers should be understandable and analyzable by current LLMs, so that adding a LLM-enabled view could give students feedback at the highest level. At lower levels, GT offers a `GtMethodAdvice` system for analyzing source code at the method or expression level, which could be exposed and/or expanded. Similarly, GT's IR could be optimized further than `IrMethod>>optimize` does and its optimizations transformations exposed to students.

As for the implementation of Processing, many bits are missing. Mainly support for object-oriented programming should be achievable, since Python's object model should again map sufficiently well onto Smalltalk's. What will, however, not be realistically possible is to get a sufficiently full Python inside GT, which would allow importing (arbitrary) further modules. This should, however, not be as much of an issue, since similar to the Processing IDE, hitting the limitations of the environment should be taken as a hint that the environment has been outgrown and to move to more capable tools, continuing the investigations using professional debuggers, memory viewers, *etc.*

For students coming from a different programming language, adapting the environment to their language of choice might also be doable. Since GT already contains support for parsing dozens of languages (among other languages JavaScript, C(++), Rust or even Visual Basic), matching a subset of that language from its AST to Smalltalk should be doable in the same way as `ProcessingTranspiler` was implemented. At least when sticking to the Processing API, the rest of the environment could be reused

with minimal adjustments.

Going beyond programming, the same principle might also be applicable to other domains: We have already seen the Filius environment, which allows students to deconstruct networks to various depths. In natural sciences, processes can be investigated and deconstructed in a similar fashion in a simulated environment. Based on a framework like NetLogo,[1] which already presents multiple views of the same phenomenon, further views for deconstructing and understanding the observed behavior could be added. Having the simulation inside a moldable environment would certainly help.

In the same vein, in a psychology course, students could be exposed to views of the subconscious, neurology down to biology and maybe even physics, when discussing behavior – for a better understanding of the various influences on what might be perceived as purely mental; or in a music class, students could get harmonics decomposed into oscillation and ratios – for a better understanding of what causes harmony; *etc*. Applying this principle in other domains is, however, left to specialists of those. What is nonetheless desired in all cases is an interdisciplinary approach, once lower abstraction levels go beyond one's own domain.[2]

However first, we have to proceed with a further investigation into whether our students from different classes are indeed profiting in the way we intended them to.

[1] *Cf. e.g.* https://www.netlogoweb.org/launch#http://www.netlogoweb.org/assets/modelslib/Sample%20Models/Biology/Ants.nlogo.

[2] Similarly, how chemistry and physics will have to be involved when discussing the innards of a modern microprocessor.

A

Installing and Using “Processing Abstractions”

In order to set up “Processing Abstractions”, first download GT 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 (access it through figure A.1):

```
Metacello new
   repository: 'github://zeniko/\ac{GT}-exploration:thesis/src';
   baseline: 'GtExploration';
   load.
Metacello new
   repository: 'github://zeniko/processing-abstractions:thesis/src';
   baseline: 'ProcessingAbstractions';
   load.

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

"Make indenting keyboard shortcuts available to non-US-English keyboard layouts
(cf. https://github.com/feenkcom/gtoolkit/issues/3002)."
LeSnippetElement keyboardShortcuts
   at: #IndentSnippet
      put: BlKeyCombinationBuilder new alt shift arrowRight build;
   at: #UnindentSnippet
      put: BlKeyCombinationBuilder new alt shift arrowLeft build.

"Make the zoom in keyboard shortcut available to de-CH keyboard layouts
(cf. https://github.com/feenkcom/gtoolkit/issues/4624)."
TLeWithFontSize compile:
   ((TLeWithFontSize methodNamed: #initializeFontSizeShortcuts) sourceCode
      copyReplaceAll: 'equal' with: 'shift minus').

"Patch unneeded addressbar out of YouTube snippet
(cf. https://github.com/feenkcom/gtoolkit/issues/4560)."
LeYoutubeReferenceElement compile:
   ((LeYoutubeReferenceElement methodNamed: #updatePicture) sourceCode
      copyReplaceAll: '</iframe>'' ' with: '</iframe>''; removeChildAt: 1 ').
```

Figure A.1: QR-link to the code for `startup.st` for convenience

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

Verify that everything works as desired and then close and save the changes to the image.

Warning: If you've already used GT before, the contents of your local knowledge database will also be included in GT's image. Therefore, rename that folder (usually `lepiter/default` in your documents folder) before starting the GT meant for distribution, and undo the renaming before again starting your own GT instance.

Also, since the executable `GlamorousToolkit.exe` is located in a subdirectory under Windows and Linux, adding a top-level link can help students. See https://github.com/zeniko/gtRunner for a ready-to-use drop-in.

B
GT Processing API

This appendix lists the available API calls implemented in Processing Abstraction's partial implementation of Processing.[1] This has been autogenerated from the GT page "Processing API":

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

`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)`.

[1] For comparison, the full API or Processing's Python mode is available at https://py.processing.org/reference/.

`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)`. If an optional fifth argument is given, corners are rounded by that many pixels.

`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),color(gray)`
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.

`max(a, b)`
Returns the larger of the two values (`max(a)` returns the largest value contained in the list `a`).

`min(a, b)`
Returns the smaller of the two values (`min(a)` returns the smallest value contained in the list `a`).

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

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

`randomSeed(seed)`
Reinitializes the random generator with the given `seed` number. Using the same number will result in the exact same sequence of pseudo-randomly generated numbers.

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

`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 Lists

Lists are objects which provide their own methods. Note that for the following commands, `list` is a variable referencing a `list`.

`len(list)`
Returns the number of items in this list.

`list.append(value)`
Appends the value to the end of the list. (`list + [value]` instead produces a *new* list. `list + otherList` produces a *new* list out of two lists.)

`list.pop()`
Removes the last item from the list and returns the removed argument. (`list[:-1]` instead produces a *new* list without the last item.)

`list.reverse()`
Reverses this list's items. (`list[::-1]` instead produces a *new* list with its items reversed.)

`list.sort()`
Sorts this list's items. (`sorted(list)` instead produces a *new* sorted list.)

B.5 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
Technical Implementation of "Processing Abstractions"

C.1 Classes

C.2 Views

The following views are available as arguments for `ProcessingSource>>>renderLiveView:` (see page 22). See figure C.2 for an overview of all view implementors.

`gtAbstractionsFor:`
(implemented by `ProcessingProgram`) combines `gtSourceCodeFor:`, `gtTreeFor:`, `gtBytecodeFor:` and `gtOutputFor:`, linking the source to the other three views through a common `Announcer` reacting to selection changes (shown in figure C.3).

`gtBytecodeFor:`
uses the `CompiledMethod` instances of each method in a compiled class' `class methodDict` to access its `symbolicBytecodes` and show the resulting `SymbolicBytecode`s' bytes and mnemonic (shown on the right in figure C.9). `gtBytecodePlusSourceFor:` combines this view with `gtSourceCodeFor:`.

`gtBytecodePlusIRFor:`
combines the `gtBytecodeFor:` with `gtIntermediaryRepresentationFor:` (shown in figure C.9).

`gtHexDumpFor:`
shows type individual bytes for all compiled methods.

`gtIntermediaryRepresentationFor:`
uses the `OpalCompiler` for translating the transpiled Smalltalk code to `IRInstruction`s (shown on the left in figure C.9). `gtIntermediaryRepresentationPlusSourceFor:` combines this view with `gtSourceCodeFor:`.

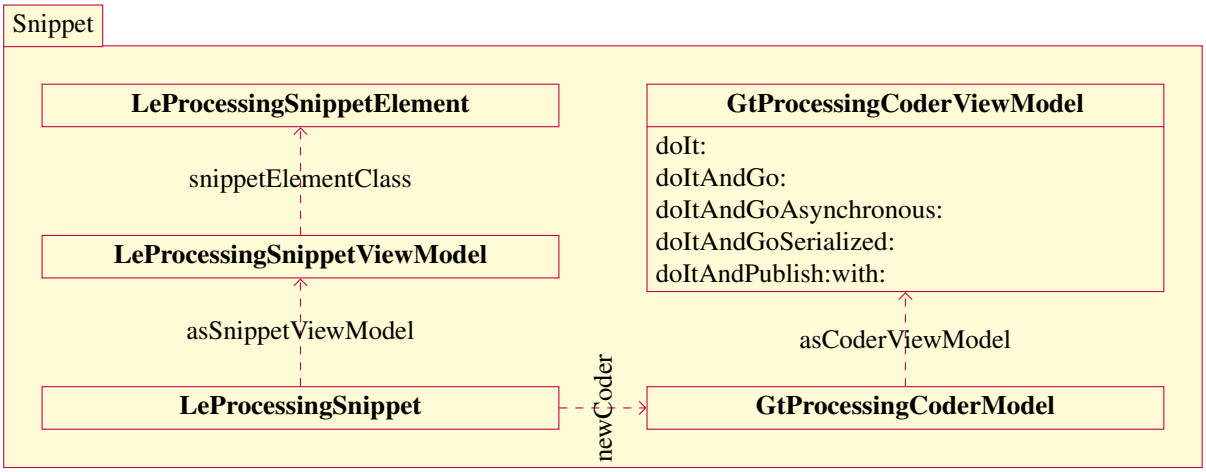

Figure C.1: Diagram of classes involved in `LeProcessingSnippet`

`gtOutputFor:`
displays a newly created `ProcessingCanvasElement` with attached event listeners for interactivity (shown on the bottom right in figure C.3). `gtOutputPlusSourceFor:` combines this view with `gtSourceCodeFor:`.

`gtOutputShapesFor:`
shows a list of all `ProcessingCanvasShape`s in the order they were drawn (shown in figure C.12). Clearing the canvas with `background(...)` also clears this list.

`gtSlicesFor:`
shows a list of `ProcessingTranspilationSlice`s with the corresponding expressions in Processing source code and Smalltalk transpilation highlighted (shown in figure C.11).

`gtSourceBytesFor:`
shows a list of `SmallInteger`s corresponding to the bytes of the source code after UTF-8 encoding (shown on the right in figure C.4) `gtSourceBytesPlusSourceFor:` combines this view with `gtSourceCodeFor:`.

`gtSourceBytesPlusCharsFor:`
combines `gtSourceCharsFor:` and `gtSourceBytesFor:` (shown in figure C.4).

`gtSourceCharsFor:`
shows a list of `Character`s corresponding to each of the Processing source code's characters (shown on the left in figure C.4). `gtSourceCharsPlusSourceFor:` combines this view with `gtSourceCodeFor:`.

`gtSourceCodeFor:`
displays a fresh read-only editor instance from `SmaCCParseNode>>gtSourceEditorWithHightlight:` with a custom `BrTextEditorReadonlyWithNavigationMode` mode (shown on the top left in figure C.3, on the left in C.7, *etc.*).

`gtStepsFor:`
shows a list of `ProcessingRunSteps` and their own `gtAbstractionFor:` view, consisting of their `gtSourceCodeFor:`, `gtBytecodeFor:`, `gtVariablesFor:`, `gtStackFor:` and `gtOutputFor:` (shown

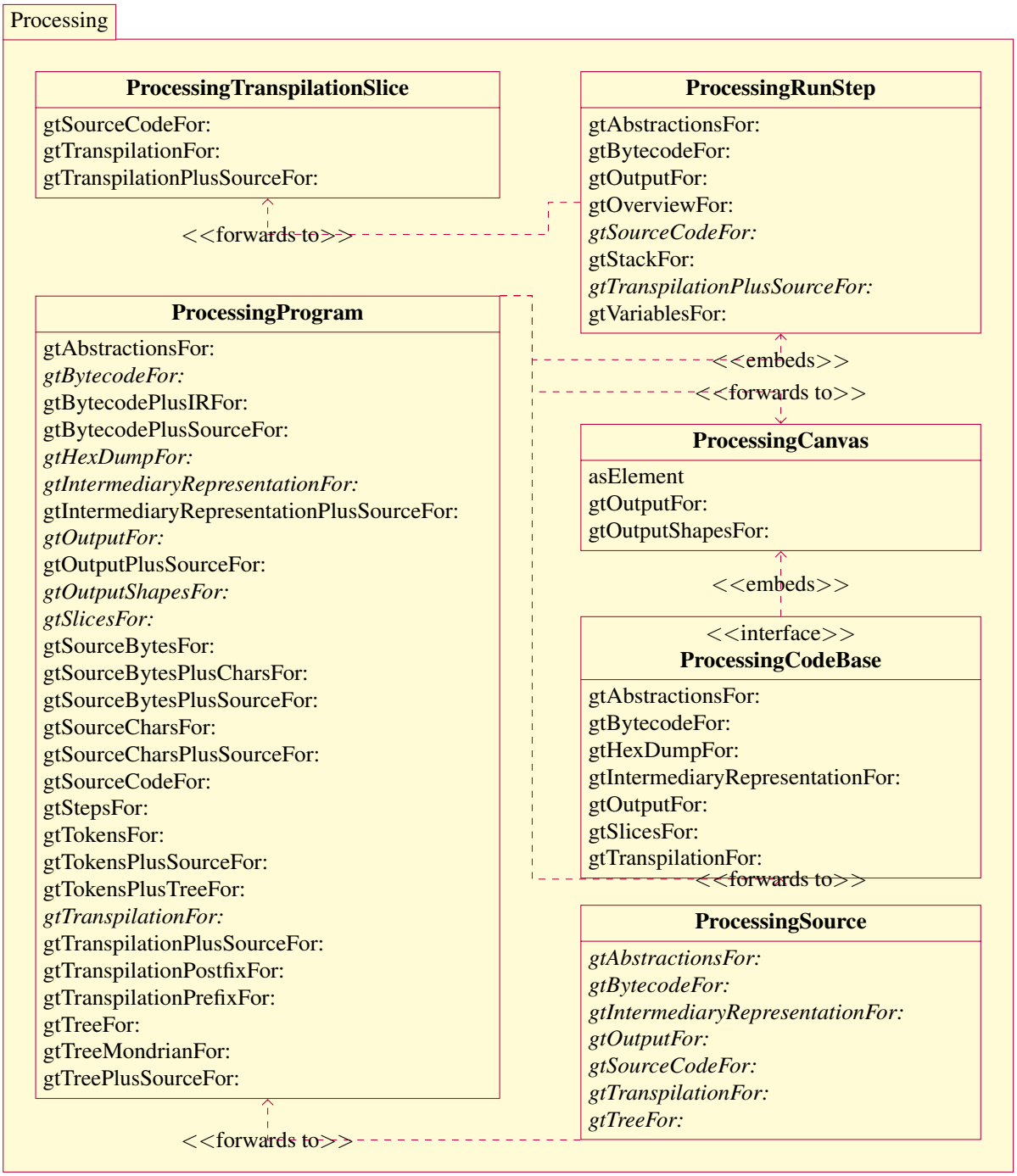

Figure C.2: Diagram of all views provided and their implementors (methods set in *italics* are forwarded)

in figure C.10). `ProcessingRunStep>>gtOverviewFor:` is a simplified variant of this shown from the Processing/Python snippet.

`gtTokensFor:`
shows a list of `SmaCCToken`s that were produced by `ProcessingParser` (shown on the left in figure C.5). `gtTokensPlusSourceFor:` combines this view with `gtSourceCodeFor:`.

`gtTokensPlusTreeFor:`
combines `gtTokensFor:` with `gtTreeFor:` (shown in figure C.5).

`gtTranspilationFor:`
shows the transpiled Smalltalk code in `GT`'s code viewer, which separates methods and adds syntax highlighting. `gtTranspilationPlusSourceFor:` combines this view with `gtSourceCodeFor:` (shown in figure C.7).

`gtTranspilationPostfixFor:` and `gtTranspilationPrefixFor:`
show a `String` produced by `ProcessingTranspilerVariant` in either its `prefix` or `postfix` modes (shown at the bottom in figure C.8 respectively).

`gtTreeFor:`
shows a treelist of `PyRootNode`s for expression roots and `SmaCCToken`s for structural tokens (shown on the right in figure C.5).

`gtTreeMondrianFor:`
shows a horizontal `GtMondrian` tree produced by `ProcessingTreeMondrianCreator` (shown in figure C.6). `gtTreePlusSourceFor:` combines this view with `gtSourceCodeFor:`.

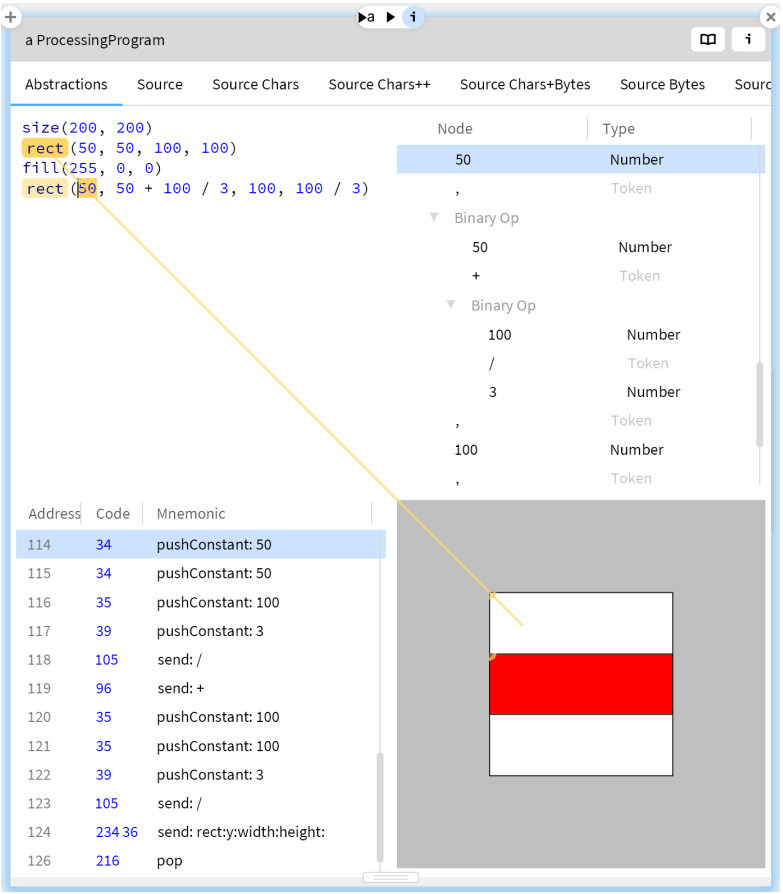

Figure C.3: The Abstractions view with source code, AST, bytecode and output showing

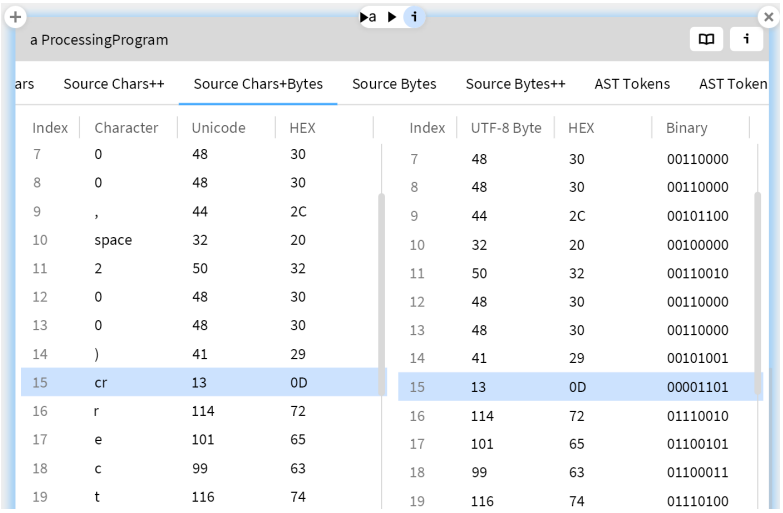

Figure C.4: The Characters and Bytes views, connected

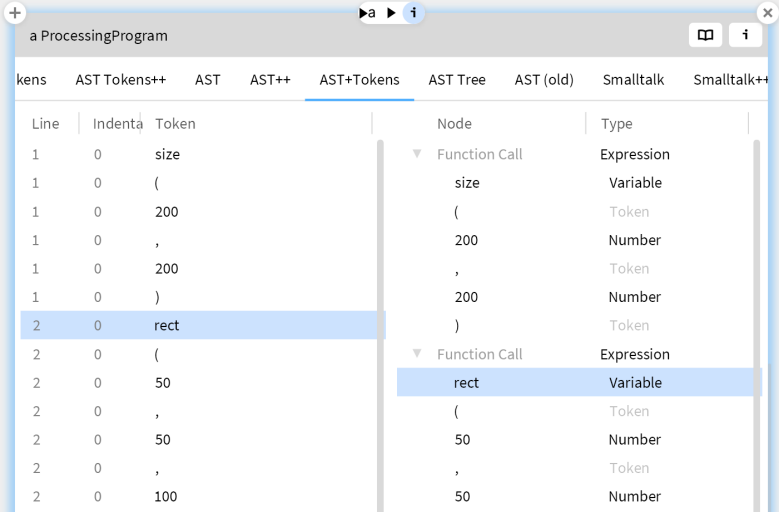

Figure C.5: The Tokens and AST views, connected

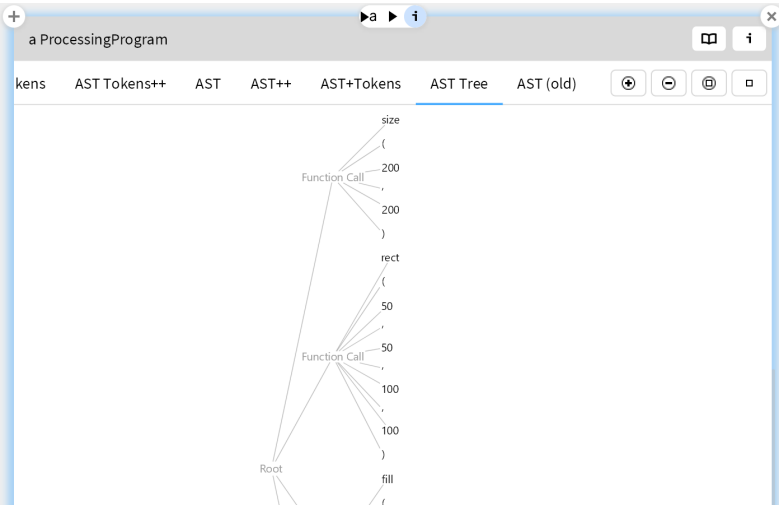

Figure C.6: The AST as a pannable and zoomable tree

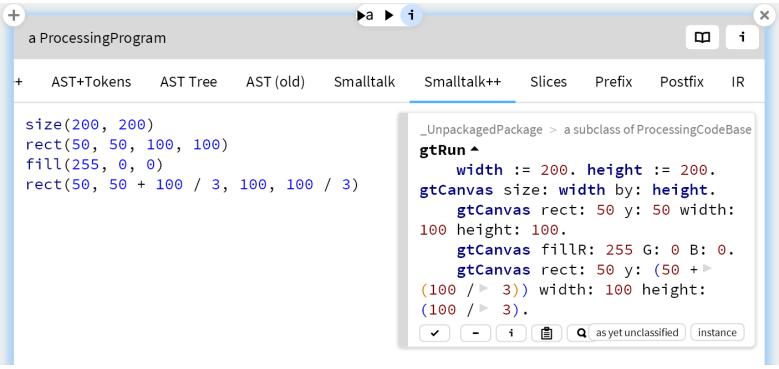

Figure C.7: The Transpilation view, showing Processing and Smalltalk code side by side

```
size(200, 200)
# Variable für Farbe und Durchmesser
x = 200
while x > 0:
    # Zeichne konzentrische Kreise in ändernder
Farbe
    fill(200 - x, 128, x)
    ellipse(100, 100, x, x)
    x = x - 20
```

Node	Type
▼ Function Call	Expression
size	Variable
(	Token
200	Number
,	Token

```
(size 200 200)
(= x 200)
(while (> x 0) (do
  (fill (- 200 x) 128 x)
  (ellipse 100 100 x x)
  (= x (- x 20))
  (delay 50)
))
```

```
200 200 size
$x 200 =
{ x 0 > } {
  200 x - 128 x fill
  100 100 x x ellipse
  $x x 20 - =
  50 delay
} while
```

Figure C.8: Source code, AST and a transpilation to two pseudo languages (with pure prefix and postfix notation)

a ProcessingProgram

ces | Prefix | Postfix | IR | IR++ | Bytecode | Bytecode++ | Bytecode+IR | Output | Output++

Label	Instruction
----	FUNCTION gtRun
0	goto: 1
1	pushLiteral: 200
	storeLiteralVariable: width
	popTop
	pushLiteral: 200
	storeLiteralVariable: height
	popTop
	pushLiteralVariable: gtCanvas
	pushLiteralVariable: width
	pushLiteralVariable: height
	send: #size:by:
	popTop

Address	Code	Mnemonic
----	----	FUNCTION gtRun
89	32	pushConstant: 200
90	203	popIntoRcvr: 3
91	32	pushConstant: 200
92	202	popIntoRcvr: 2
93	00	pushRcvr: 0
94	03	pushRcvr: 3
95	02	pushRcvr: 2
96	161	send: size:by:
97	216	pop
98	00	pushRcvr: 0
99	34	pushConstant: 50
100	34	pushConstant: 50
101	35	pushConstant: 100

Figure C.9: The IR and Bytecode views, connected

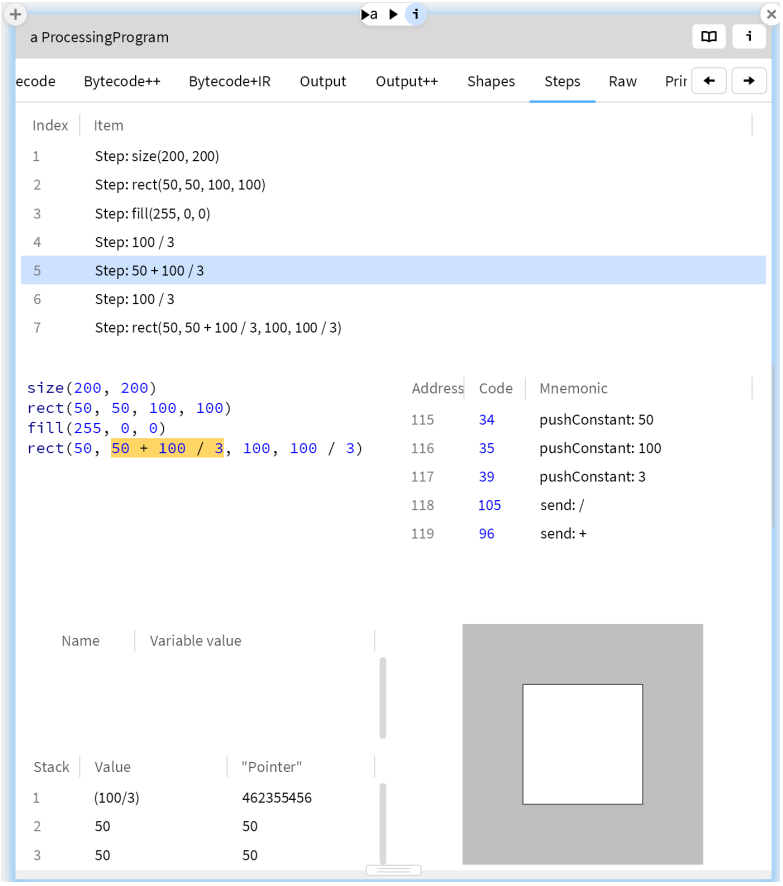

Figure C.10: The Runsteps view allows users to step through execution and inspect variables and stack values

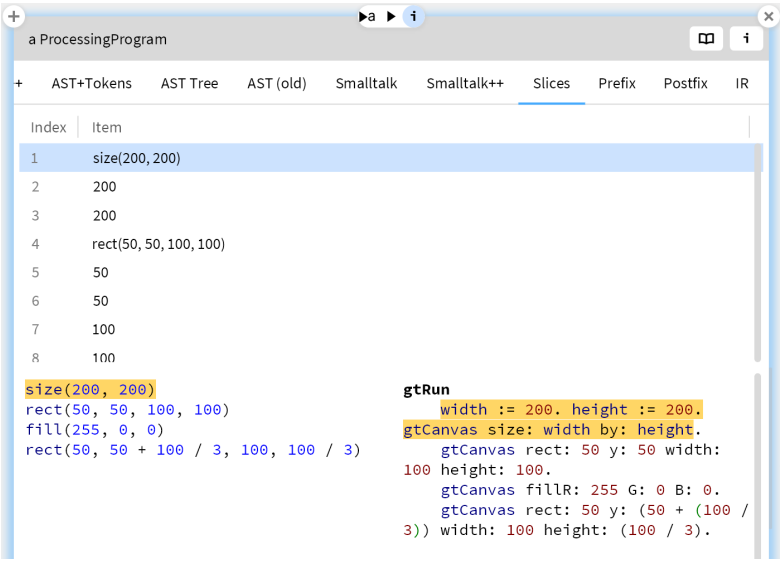

Figure C.11: The Slices view shows the list of objects connecting Processing and Smalltalk code

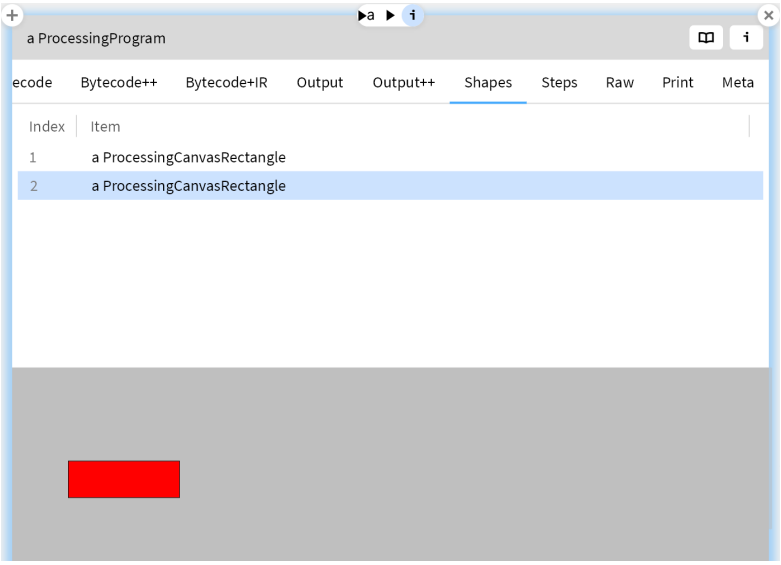

Figure C.12: The Shapes displays all output shapes individually

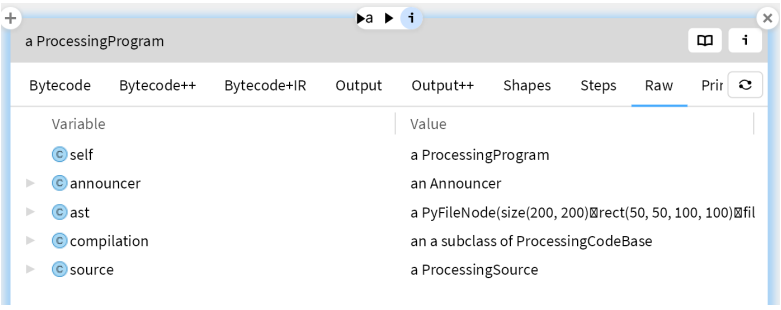

Figure C.13: The Raw view is provided by GT and shows all variables of an instantiated object

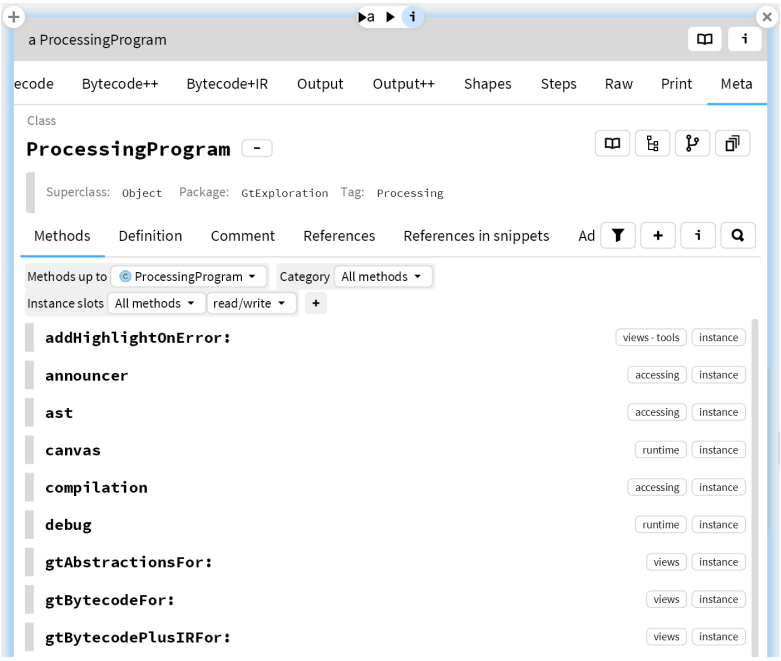

Figure C.14: The Meta view is provided by GT and shows all methods defined for the instantiated object

D
Questionnaires

The questionnaires were implemented in https://forms.office.com/ in the students' native language.

D.1 Questionnaire for 6.1

Feedback zur heutigen Unterrichtssequenz

1. **Wie hat Ihnen die heutige Unterrichtssequenz gefallen?**
   gar nicht □—□—□—□—□ sehr gut

2. **Welche Themen haben Sie heute alle bearbeiten können?**

   - □ Arbeiten mit Glamorous Toolkit
   - □ Maschinensprache und Prozessor
   - □ Funktionen eines Compilers
   - □ Anhang

3. **Wie sehr trauen Sie sich zu, die heutigen Inhalte jemand anderem zu erklären?**
   gar nicht □—□—□—□—□ *easy-peasy*

4. **Wie viele der Python-Prograämmchen haben Sie selbst verändert?**

   - ○ Keines
   - ○ Eines
   - ○ Zwei bis drei
   - ○ Vier oder mehr

5. **Was ist ein Stack?**

   _____

6. **Was machen Lexer und Parser?**

   _____

7. **Wie gut hat die Lernumgebung für Sie funktioniert?**
   gar nicht □—□—□—□—□ problemlos

8. **Was hat Ihnen an der Lernumgebung gefallen?**

   _____

   _____

9. **Welche Änderungen an der Lernumgebung wünschen Sie sich für die nächste Klasse?**

   _____

   _____

10. **Wie hilfreich fanden Sie die Nebeneinanderstellungen der unterschiedlichen Schritte beim Ausführen/Übersetzen eines Programms?**
    weglassen □—□—□—□—□ bitte mehr davon

11. **Beschreiben Sie in eigenen Worten: Wie wird ein Programm auf einem Prozessor ausgeführt?**

    _____

    _____

12. **Beschreiben Sie in eigenen Worten: Wie wird ein Programm in einer Hochsprache wie Processing für den Prozessor aufbereitet?**

    _____

    _____

13. **Was hatte das heutige Thema mit Silizium, Transistoren, Gattern und Schaltungen zu tun?**

    _____

    _____

Feedback zum Informatikunterricht der letzten zwei Jahre

14. **Was war für Sie das Highlight vom Informatik-Unterricht? (Was hat Ihnen am meisten Eindruck gemacht?)**

    _____

15. **Was ist Ihnen vom Informatik-Unterricht alles geblieben (einige Stichworte zum Stoff)?**

    _____

    _____

16. **Was hat Ihnen am Informatik-Unterricht gefallen?**

    _____

    _____

17. **Wenn Sie mir vor zwei Jahren einen Hinweis geben könnten: Was hätten Sie sich für den Unterricht anders gewünscht?**

    _____

    _____

D.2 Questionnaire for 6.2

Feedback zur heutigen Unterrichtssequenz

1. **Wie hat Ihnen die heutige Unterrichtssequenz gefallen?**
   gar nicht □—□—□—□—□ sehr gut
2. **Welche Themen haben Sie heute alle bearbeiten können?**
   - □ Arbeiten mit Glamorous Toolkit
   - □ Human Resource Machine (aus Maschinensprache und Prozessor)
   - □ Lexer und Parser
   - □ Transpiler und Compiler
   - □ Optimierer
3. **Wie sehr trauen Sie sich zu, die heutigen Inhalte jemand anderem zu erklären?**
   gar nicht □—□—□—□—□ *easy-peasy*
4. **Wie viele der Python-Prograämmchen haben Sie selbst verändert?**
   - ○ Keines
   - ○ Eines
   - ○ Zwei bis drei
   - ○ Vier oder mehr
5. **Weshalb kann ein Prozessor ein Processing-Programm nicht ohne Übersetzung ausführen**

   _____
6. **Was machen Lexer und Parser?**

   _____
7. **Wie gut hat die Lernumgebung für Sie funktioniert?**
   gar nicht □—□—□—□—□ problemlos
8. **Was hat Ihnen an der Lernumgebung gefallen?**

   _____

   _____
9. **Welche Änderungen an der Lernumgebung wünschen Sie sich für die nächste Klasse?**

   _____

   _____
10. **Wie hilfreich fanden Sie die Nebeneinanderstellungen der unterschiedlichen Schritte beim Ausführen/Übersetzen eines Programms?**
    weglassen □—□—□—□—□ bitte mehr davon
11. **Beschreiben Sie in eigenen Worten: Wie wird ein Programm in einer Hochsprache wie Processing für den Prozessor aufbereitet?**

    _____

    _____

12. **Was hatte das heutige Thema mit Codierung und was mit Programmieren zu tun?**

_____

_____

Feedback zum Informatikunterricht

13. **Was war für Sie das Highlight vom Informatik-Unterricht? (Was hat Ihnen am meisten Eindruck gemacht?)**

_____

14. **Was ist Ihnen vom Informatik-Unterricht alles geblieben (einige Stichworte zum Stoff)?**

_____

_____

15. **Was hat Ihnen am Informatik-Unterricht gefallen?**

_____

_____

16. **Welche Änderungen wünschen Sie sich fürs kommende Schuljahr in Informatik?**

_____

_____

Abbreviations

API Application Programming Interface 7
ALU Arithmetic Logic Unit 41
AST Abstract Syntax Tree 9
CPU Central Processing Unit 5
GT Glamorous Toolkit 10
IDE Integrated Development Environment 1
IR Intermediary Representation 9
JIT Just-In Time Compiler 16
JVM Java Virtual Machine 12
LLM Large Language Model 5
REPL Read-Evaluate-Print-Loop 4
VM Virtual Machine 15

Bibliography

[1] Computersysteme. https://oinf.ch/kurs/vernetzung-und-systeme/computersysteme/. Accessed: 2025-07-23.

[2] Debug your Python code with PyCharm. https://www.jetbrains.com/pycharm/features/debugger.html. Accessed: 2025-06-17.

[3] feenk - about. https://feenk.com/about/. Accessed: 2025-07-16.

[4] Human Resource Machine: Hour of code edition. https://tomorrowcorporation.com/human-resource-machine-hour-of-code-edition. Accessed: 2025-06-19.

[5] Little Man Computer. https://oinf.ch/interactive/little-man-computer/. Accessed: 2025-06-19.

[6] Online compiler, AI tutor, and visual debugger for Python, Java, C, C++, and JavaScript. https://pythontutor.com/. Accessed: 2025-06-17.

[7] PyDev - Python IDE for Eclipse. https://marketplace.eclipse.org/content/pydev-python-ide-eclipse. Accessed: 2025-06-17.

[8] Python debugging in VS Code. https://code.visualstudio.com/docs/python/debugging. Accessed: 2025-06-17.

[9] Python mode for Processing. https://py.processing.org/. Accessed: 2025-07-31.

[10] What games use Lua? A quick overview. https://luascripts.com/what-games-use-lua. Accessed: 2025-07-14.

[11] What is Jython? https://www.jython.org/. Accessed: 2025-07-31.

[12] Why your website should be under 14kB in size. https://endtimes.dev/why-your-website-should-be-under-14kb-in-size/. Accessed: 2025-07-21.

[13] What is DBN? https://dbn.media.mit.edu/whatisdbn.html, 2001. Accessed: 2025-07-31.

[14] Lehrplan 17 für den gymnasialen Bildungsgang. https://www.bkd.be.ch/content/dam/bkd/dokumente/de/themen/bildung/mittelschulen/gymnasium/ams-gym-lehrplan-17-neu-ab-sj19-20-gesamtdokument.pdf, 2016. Accessed: 2025-06-19.

[15] 10 best programming languages for kids of any age. https://codeweek.eu/blog/10-best-programming-languages-for-kids-of-any-age/, 2020. Accessed: 2025-07-01.

[16] Overview of tools combining Processing and Python. https://tabreturn.github.io/code/processing/python/2022/08/02/overview_of_tools_combining_python_and_processing.html, 2022. Accessed: 2025-07-21.

[17] How does Glamorous Toolkit's PythonBridge work? https://www.fractolog.com/2024/08/how-does-glamorous-toolkits-pythonbridge-work/, 2024. Accessed: 2025-07-31.

[18] 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.

[19] Joel Becker, Nate Rush, Beth Barnes, and David Rein. Measuring the impact of early-2025 AI on experienced open-source developer productivity. https://metr.org/Early_2025_AI_Experienced_OS_Devs_Study.pdf, 2025. Accessed: 2025-07-15.

[20] Clément Béra. *Sista: a Metacircular Architecture for Runtime Optimisation Persistence*. PhD thesis, 09 2017.

[21] Clément Béra and Eliot Miranda. A bytecode set for adaptive optimizations. In *International Workshop on Smalltalk Technologies*, 2014.

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

[23] 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.

[24] Andrei Chiş. *Moldable Tools*. PhD thesis, University of Bern, September 2016.

[25] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. Towards moldable development tools. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '15, pages 25–26, New York, NY, USA, 2015. ACM.

[26] 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.

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

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

[29] Michael Egger. The law of leaky abstractions: A guide for the pragmatic programmer. https://medium.com/@mesw1/ the-law-of-leaky-abstractions-a-guide-for-the-pragmatic-programmer-9bf80545c43f, 2024. Accessed: 2025-06-16.

[30] Francisco Buitrago Flórez, Rubby Casallas, Marcela Hernández, Alejandro Reyes, Silvia Restrepo, and Giovanna Danies. Changing a generation's way of thinking: Teaching computational thinking through programming. *Review of Educational Research*, 87(4):834–860, 2017.

[31] Lorenzo Ganni. Von Neumann machine simulator. https://vnsim.lehrerlezius.de/, 2023. Accessed: 2025-06-19.

[32] Daniel Garmann. Introduction to the world of FILIUS. https://www.lernsoftware-filius.de/downloads/Introduction_Filius.pdf, 2015. Accessed: 2025-07-30.

[33] Tudor Gîrba. WTF is moldable development? https://blog.container-solutions.com/wtf-is-moldable-development, 2022. Accessed: 2025-07-16.

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

[35] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., USA, 1983.

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

[37] 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.

[38] 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.

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

[40] Satwik Kansal. What the f*ck Python. https://colab.research.google.com/github/satwikkansal/wtfpython/blob/master/irrelevant/wtf.ipynb. Accessed: 2025-07-21.

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

[42] 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.

[43] Michael Kölling, Neil C. C. Brown, and Amjad Altadmri. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education*, WiPSCE '15, pages 29–38, New York, NY, USA, 2015. Association for Computing Machinery.

[44] Charalampos Kyfonidis, Pierre Weill-Tessier, and Neil Brown. Strype: Frame-based editing tool for programming the micro:bit through Python. In *Proceedings of the 16th Workshop in Primary and Secondary Computing Education*, WiPSCE '21, New York, NY, USA, 2021. Association for Computing Machinery.

[45] 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.

[46] 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.

[47] Michael Meyer. Scripting interactive visualizations. Master's thesis, University of Bern, November 2006.

[48] Eliot Miranda et al. The Cog VM source tree. https://github.com/OpenSmalltalk/opensmalltalk-vm, 2025. Accessed: 2025-07-31.

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

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

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

[52] 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.

[53] Thomas Pornin. Constant-time crypto. https://www.bearssl.org/constanttime.html, 2018. Accessed: 2025-06-20.

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

[55] 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.

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

[57] Joel Spolsky. The law of leaky abstractions. https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/, 2002. Accessed: 2025-06-16.

[58] 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.

[59] Nicholas H. Tollervey. The visual debugger. https://codewith.mu/en/tutorials/1.2/debugger, 2023. Accessed: 2025-06-17.

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

[61] Florian Wörister and Maria Knobelsdorf. A block-based programming environment for teaching low-level computing (discussion paper). In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*, Koli Calling '23, New York, NY, USA, 2024. Association for Computing Machinery.

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