Coypu and Phausto: accessible live sound coding with Pharo

Domenico Cipriani
mspgate@gmail.com
Inria Lille
Villeneuve D'Ascq, France

Sebastian Jordan Montaño
sebastian.jordan@inria.fr
Univ. Lille - Inria - CNRS - Centrale Lille - UMR 9189 CRIStAL
Villeneuve D'Ascq, France

Stéphane Ducasse
stephane.ducasse@inria.fr
Univ. Lille - Inria - CNRS - Centrale Lille - UMR 9189 CRIStAL
Villeneuve D'Ascq, France

Abstract

On-the-fly music composition–live coding–is the art of making artistic performances by programming music and visualizations without stopping or restarting. The code aesthetics and the dynamic nature of the process, a.k.a. on-the-fly programming, are an important part of the performance. However, mainstream programming languages are not easily inspectable, are not dynamic in nature, and often act like a black box that users simply use.

Coypu and Phausto have been designed to turn an IDE into a live music workstation, enabling on-the-fly composition and DSP programming while focusing on being fully dynamic, inspectable, fun, and simple. They are implemented in the Pharo programming language, which is purely object-oriented and written in itself. Pharo syntax, rooted in the pedagogical tradition of Smalltalk, can be easily taught to introduce beginners to computer music practice. Pharo's reflective capabilities support a constructivist approach to discovering and learning the internal mechanisms of the system, as well as how to implement custom methods and classes. This differs significantly from traditional languages that act like black boxes and, in contrast to Pharo, do not easily allow inspection of the system code.

Coypu focuses on the creation, manipulation, and playback of musical patterns, with sound rendering that can be handled by different audio servers. Phausto provides an interface for programming synthesizers and native audio processing by leveraging an embedded Faust compiler. In this paper, we present the architecture and the core features of Coypu and Phausto. We examine critical aspects of the system, and outline future challenges and improvements. We have been using Coypu and Phausto for years for music composition, live coding performances, and teaching.

Keywords

Live coding, music, on-the-fly programming, Smalltalk, OOP, Pharo, DSP programming, Faust

1 Introduction

Over the last five years, we have been developing libraries [4, 7] aimed at transforming the Pharo IDE into an environment for music composition and the on-the-fly creation of sound synthesizers. [1] The first of these libraries, Coypu[2] was originally designed as a client for live coding sounds created with Symbolic Sound Kyma [30], with the explicit goal of working within a fully Smalltalk-based environment [6]. The second library, Phausto[3] was developed to provide an intuitive API for programming DSPs with the aim of making Pharo [11] autonomous for sound generation [4]. Phausto leverages the rich Faust ecosystem [21] which implements hundreds of functions for synthesis and native audio processing.[4]

Through our experience with Kyma as a compositional and sound design system, we came to appreciate how well Smalltalk supports lean, exploratory programming practices and how naturally its object-oriented model lends itself to the live creation of synthesis techniques and musical forms. Kyma is a powerful computer music system that includes hundreds of sound objects, tools for audio editing, spectral analysis and transformation, as well as support for MIDI and OSC. It also allows users to implement their own tools in Smalltalk. However, it does not permit users to create custom classes or define their own methods within the system. Moreover, as a proprietary and closed-source platform, access to reflection has been removed, limiting transparency and making the system difficult to inspect and extend. These limitations motivated us to adopt a free and open-source Smalltalk environment. We chose Pharo because it is a high-performance [22] implementation of Smalltalk that preserves its foundational ideas while extending them with modern features.

The work presented in this paper is driven by the conviction that Pharo's object model, lightweight syntax, ease of installation, and friendly development tools can facilitate the teaching and dissemination of live coding practices and audio programming. Pharo's lively and welcoming community, together with a wide range of learning resources (including two MOOCs[5] and several freely available books [8, 10, 11]) provides an accessible entry point for beginners and for people with little prior computer literacy. Our goal is to offer a gentle introduction to computer music and digital signal processing (DSP) programming, which can then serve as a foundation for approaching more advanced paradigms. Our vision builds on the tradition of using smalltalk to create interactive and creative software, following Alan Kay's original idea [13],inspired by Montessori's pedagogical approach and Piaget's constructivist learning theory [24]. Our live coding environment encourages the musician and the learner to take an active role, both playing with and modifying the tools they are using.

2 Background

The Dynabook, conceived by the Learning Research Group at Xerox Palo Alto Research Center as a dynamic medium for human beings of all ages [14], relied on Smalltalk as its programming language and environment. Already by 1977, the system could operate as a sound synthesizer and provided, alongside drawing and animation tools, a musical score editor named OPUS, capable

[1] Programming environments such as SuperCollider, Pure Data, and Csound provide sound synthesis, signal processing, and composition tools. However, none were integrated into a general-purpose IDE.

[2] Coypu is available at the MIT open source license under https://github.com/lucretiomsp/Coypu

[3] https://github.com/lucretiomsp/phausto

[4] https://faustlibraries.grame.fr/

[5] https://mooc.pharo.org/ and https://advanced-design-mooc.pharo.org/

This work is licensed under a Creative Commons Attribution 4.0 International License.
NIME '26, June 23–26, 2026, London, UK
© 2026 Copyright held by the owner/author(s).

of producing musical notation from data acquired through a keyboard connected to the computer.

Subsequently, a tool for manipulating and editing digitally sampled sound, implemented in Smalltalk-80, was developed at Apple Computer Inc. in 1985. The Sound Kit: A Sound Manipulator, designed by Mark Lentczner, provided facilities for interactive sound editing and processing [15]. A few years later, a more comprehensive Smalltalk-80-based system, the Music Toolkit [27] developed by Stephen Travis Pope, was presented at the International Computer Music Conference (ICMC) in 1987. The toolkit featured graphical editors and extensive functionality for note, score, and sound processing.

In 1991, one of the most widely adopted Smalltalk-based audio programming environments became commercially available: Kyma [29], designed by Carla Scaletti and developed by Symbolic Sound Corporation (SSC). Kyma has been widely used in professional sound design, including major film productions[6], and by artists such as Autechre, Cristian Vogel, Roland Kuit, and John Paul Jones. The system remains actively developed and is currently in its seventh major version. Kyma offers powerful sound synthesis and transformation capabilities and provides an accessible creative environment for composition and sound design. However, Kyma is a proprietary system with closed-source code that requires dedicated hardware, an Audio Processing Unit (APU); hence, accessibility is limited.

After several years of artistic practice, research, and performance using Kyma, we chose to adopt Pharo as a platform for extending the Smalltalk philosophy toward musicians, sound artists, and beginners interested in exploring computer music. Pharo is a modern implementation of Smalltalk, supported by extensive learning resources, an active community, and a high-performance virtual machine [19, 22, 23, 26]. This virtual machine is particularly well suited for the precise scheduling of timed musical events (as required by Coypu) and for efficient interaction with external audio engines through its Unified Foreign Function Interface (uFFI) framework [25] (as necessary for Phausto). We believe in open science [16], for this reason our libraries are fully open source as well of the whole Pharo ecosystem.

2.1 Openness and documentation practices

Our decision to release both libraries under the MIT license and to build upon the fully open-source Pharo ecosystem reflects a commitment to transparency, reproducibility, and community-driven innovation, values that Morreale et al. [20] have argued are central to the adoption of FLOSS in NIME research The question of documentation and knowledge sharing, as highlighted by Calegario et al. [3], is also a necessary step for the diffusion of our instruments; insufficient documentation can create barriers to adoption and limit the long-term sustainability of research tools. We address this concern through hands-on learning resources. Coypu provides a self-contained introductory script, "Coypu with EcoPhausto," available on the project wiki [7], which guides users through the complete workflow for a live performance, encouraging a constructivist learning approach. Phausto offers a separate tutorial package called MasterLu [8], consisting of 15 interactive lessons delivered directly within the Pharo environment, covering the creation and combination of Unit Generators, parameter control, UI widget design, parameter modulation, working with audio files, and examples of subtractive, additive, modal, and FM synthesis. An extensive FLOSS manual is currently in preparation, which will comprehensively document all techniques for creating patterns and performing live with Coypu, as well as how to build DSPs with Phausto; the manual will also include basic lesson on musical theory and audio programming.

[6]Some of the films are Star Wars I, Star Wars II, and Matrix Revolutions. An extensive list can be found at https://archive.symbolicsound.com/cgi-bin/bin/view/Products/KymaFilms

[7](https://github.com/lucretiomsp/Coypu/wiki/Coypu--with-EcoPhausto)

[8]https://github.com/lucretiomsp/MasterLu

3 Challenges

The quest to transform the Pharo IDE into an environment for on-the-fly music and DSP programming was divided into two main tasks. The first consisted of designing a Domain Specific language (DSL) for the intuitive creation of rhythms and melodies, together with a client capable of dispatching precisely timed events in the form of Open Sound Control (OSC) or MIDI messages to an external audio generator. The primary concern was whether Pharo's Delay scheduling mechanism could provide sufficient temporal precision to ensure minimal jitter during event playback.
The second task involved the development of a library enabling sound synthesis directly within Pharo; this required embedding an engine capable of computing compiled sounds at sample rate. When we began our exploration of the Pharo ecosystem in 2021, music and sound support within the system was limited. An existing package inherited from Squeak and early versions of Pharo, *Sound* [9], required significant modernisation. The package is sparsely documented and provides only limited support for synthesis and audio effects. Additionally, its verbose and cumbersome syntax made it unsuitable for expressive, real-time musical programming. Consequently, we chose to develop new libraries inspired by more recent approaches to live music programming, including ChucK [32], TidalCycles [18], and Mercury [12].

3.1 Comparison with existing tools

Several well-established environments exist for live coding music, including SuperCollider [17], TidalCycles, Sonic Pi, and Strudel. Each of these offers distinct strengths: SuperCollider provides a comprehensive sound synthesis engine and a mature programming language; TidalCycles excels at pattern-based composition with a concise domain-specific language built on Haskell; Sonic Pi offers an accessible entry point with an emphasis on education; Strudel brings TidalCycles-inspired patterns to the browser together with a Web Audio-based engine. Coypu and Phausto do not aim to replace these environments but rather to offer a complementary approach and an introductory set of tools that builds on the specific advantages of the Pharo ecosystem. There are three key differentiating factors. First, full reflectivity: unlike environments built on languages that act as black boxes, Pharo allows users to inspect, modify, and extend any part of the system at runtime, including the live coding tools themselves. This makes the learning process transparent and supports a constructivist pedagogical approach. Second, a rich IDE: Coypu and Phausto operate within Pharo's integrated development environment, which provides a debugger, object inspector, system browser, and version control (Iceberg). Musicians can therefore use the same tools employed by software engineers, facilitating understanding of programming concepts beyond the musical domain. Third, pure object-oriented design: the Pharo model encourages clean, modular code that can be extended through subclassing

[9]https://github.com/pharo-contributions/pharo-sound

and composition, and the use of design patterns. We recognise that environments such as SuperCollider offer a more mature and extensive sound synthesis ecosystem, and that TidalCycles' mini-notation is more expressive for complex pattern transformations. Our contribution is not in surpassing these tools on their established strengths, but in providing an environment for a new kind of musical expression on-the-fly.

4 Pharo in a nutshell

Pharo is an open-source object-oriented language based on Smalltalk-80, dynamically typed and reflective. Pharo includes an immersive integrated development environment, which provides a rich set of engineering tools. The most relevant to our live coding practice are:

1. **the Playground:** a workspace for experimenting with scripts where code can be evaluated interactively. Here users can create and modify musical performances and DSPs created with Coypu and Phausto.
2. **the System Browser:** a tool for navigating, creating and modifying classes and methods. It supports reflective inspection of the entire system and allow live modification of the code.
3. **the Inspector:** enables runtime exploration of objects by inspecting their internal state and allowing to evaluate code.
4. **the Debugger:** a live debugger. It supports stack inspection, dynamic evaluation fo code, live code compiling and immediate re-execution.
5. **Iceberg:** Pharo's Git client.

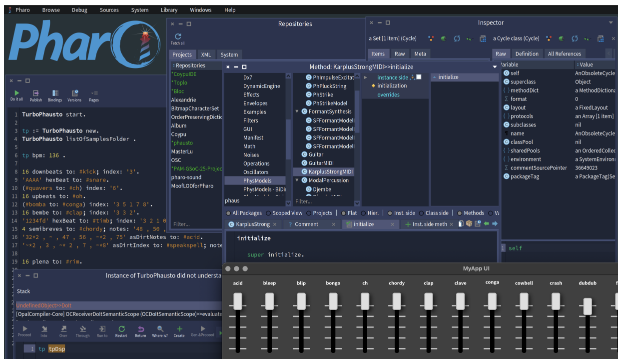

Figure 1: Pharo14 IDE with the most relevant tool for live coding music, including a mixer UI made with Bloc.

4.1 Pharo's Object Model and Syntax Essentials

The language model and syntax of Pharo is very close to that of its ancestor, Smalltalk, and can be read as a kind of pidgin English [11]. Everything is an object and every object is an instance of a class. Every class is also an object and method lookup follows the inheritance chain. A consequence of Pharo's message-sending model is that objects tend to have very small methods that delegate tasks to other objects, rather than relying on large, procedural methods that take on too much responsibility. Furthermore, Pharo's object-oriented programming encourages the intentional use of design patterns.

There are only 6 reserved keywords and Objects communicate only by sending messages to each other. There are three types of messages:

1. **Unary messages** consists of a single word like `asRhythm` sent to a receiver (like *#rumba*). In `#rumba asRhythm`, *#rumba* is the receiver and `asRhythm` is the message selctor.
2. **Binary messages.** are sent to a receiver with a single argument. Their selector looks like a mathematical symbol(for example, >). In `99 > 8`, the receiver is *99*, the selector is > and the argument is *7*.
3. **Keyword messages.** are messages whose selectors consist of one or more keywords (like `freq:`). In the expression `performance freq: 96 bpm.`. the message selector is `freq:` the receiver is *performance*, and it has the argument: *96 bpm*.

Unary messages have the highest precedence, followed by binary messages, and then keyword messages. Parentheses can be used to change this order of evaluation.
Pharo is a reflective programming language; the system can query and modify various aspect of its structure and execution. This form of reflection was strongly promoted by Smalltalk-80, the ancestor of Pharo. Through tools such as the Inspector, developers can examine objects, modify the values of their instance variables, and even send messages to them at runtime.

5 Solution overview

The primary objective was to design a mechanism for storing musical sequences and dispatching events to an audio server. This was accomplished by using an existing library[10]. for handling OSC communication and implementing a lightweight scheduler capable of advancing in time through a playhead-like mechanism; This scheduler was achieved thanks to Pharo's concurrency features [9]. Subsequently, MIDI support was added via FFI bindings[11] to the PortMidi library, enabling the system to communicate with external and internal MIDI devices.

The second task was achieved by embedding a Faust compiler within Pharo[12] allowing the native sound generation within Pharo. Phausto does not transpile Pharo code into Faust code; instead, it relies on the Faust Box API[13] to generate box expressions that are then compiled into a DSP [4].

6 Coypu on a business card

Coypu [5] is a Pharo package[14] for programming music *on-the-fly*[15] acting as a client for an external audio generator server (OSC or MIDI compatible) or for an internal DSP instantiated with Phausto (see Section 6). Coypu was designed to provide a natural and intuitive environment for live coding music, thanks to Pharo's minimal syntax and native support for interactive programming. If Pharo's syntax is compact enough to fit on a postcard, Coypu's syntax fits on a business card. The design of Coypu's API is guided by three linguistic principles [6]:

- **Iconicity** Code should be structured in a way that mirrors the musical structure it produces.

[10]Originally developed by Markus Gaelli for Squeak smalltalk
[11]Implemented by Antoine Delaby and Santiago Bragagnolo
[12]We use the Faut architecture named *Dynamic Engine* which was fine-tuned for this project by Stéphane Sletz.
[13]https://faustdoc.grame.fr/tutorials/box-api/
[14]A Pharo package is a modular unit of software organisation that groups together a set of classes, methods, traits and extensions
[15]On-the-fly programming is a programming style in which the programer/performer modifies a program without stopping or restarting it [32].

- **Economy.** The syntax has been designed to minimize user input, facilitating rapid interaction during live performances.
- **Semantic equivalence.** The same musical intent can be expressed in multiple syntactic forms, allowing users to balance economy or iconicity, with semantically equivalent alternatives.

Coypu assumes the role of the *score*, while the audio server functions as the *orchestra*, following the traditional Csound paradigm [2]. Coypu's orchestra is represented by the *Performance*, a singleton object of which only one instance exists. A Performance contains a virtually unlimited number of *Sequencers*, each of which may have a different length and stores the information required to play a specific instrument. This information can be as minimal as a *trigger* and a *note number*, or extended to include any synthesis parameter implemented by the instrument. Parameter values can also be organized in arrays of varying lengths: this can result in polymetric behaviour during the Sequencer's playback.

During playback, the Performance iterates over all Sequencers using a shared metrical subdivision. At each step, Sequencers are queried in a loop, and parameter indices advance only when a *gate* is encountered, at which point the current parameter values are dispatched to the instrument. This mechanism allows rhythmic and parametric variation to emerge from the structure and length of the sequences rather than from multiple clocks. Currently, Coypu uses a fixed metrical subdivision of 1/16th notes for all Sequencers; future work will introduce independent metrical subdivisions per Sequencer.

6.1 Architecture

The Performance needs to be assigned a Performer. When the message play is received by the Performance, it delegates to its performer. The Performer implements the *play* message, which contains a hook to the method *playEventAt:in:*. This hook is implemented differently in each Performer subclass. There are 5 Performer's subclasses.

Figure 2 illustrates Coypu's architecture. The performance needs to be assigned a Performer; when the message *play* is received by the Performance, it delegates to its performers. The Performer is an abstract class and its subclasses implements this *play* selector. There are 5 Performer's subclasses.

- The **PerformerLocal** sends OSC messages to an audio server running on the same computer, listening on port 8000.
- The **PerformerSuperDirt** sends OSC messages to a SuperCollider program running SuperDirt.
- The **PerformerKyma** sends OSC messages to an external Symbolic Sound APU running Kyma.
- The **PerformerMIDI** sends MIDI messages to an external or internal MIDI device.
- The **PerformerFaust** interacts with a Faust DSP embedded in the same Pharo image via FFI.

Instruments implemented on the local audio server or in Kyma are required to handle at least two OSC messages:

- `/<instrumentName>Gate`, used to trigger sound events;
- `/<instrumentName>Note`, which specifies the MIDI note number[16].

[16]Here, `<instrumentName>` and `<parameterName>` denote placeholders that are replaced by the actual instrument and parameter identifiers.

Additional synthesis parameters, such as *cutoff*, can be controlled by enabling the instrument to receive messages of the form: `/<instrumentName><parameterName>`.

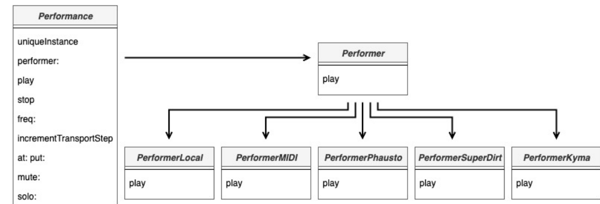

Figure 2: Coypu delegates event dispatch to the Performer assigned to a given Performance. The specific Performer subclass determines the audio server.

6.2 Usage and API

The code in Listing 1 illustrates a minimal *Performance* played using sounds generated by a *local* audio server[17], which communicates with Pharo via OSC.

```
| p notes|

p := Performance uniqueInstance performer: PerformerLocal.

#(1 0 0 1 0 0 0 0) asSeq to: #kick.
'020F' hexBeat to: #snare.
#rumba asRhythm to: #conga.
11 randomTrigs to: #ch; level: '0.8 0.4 0.3'.
#(5 16) euclidean index: ' 1 3 5 2'; to:  #bongo.

p play.
p freq: 96 bpm.
p mute: #(#snare #bleep).
p solo: #(#psg #cowbell).

'38 , 46/7 , -*4' asDirtNotes to: #sub; detune: '8  3'.

#shiko asRhythm arpeggiate: '62-sus4 48-min7'; to: #fm2Op.

n := 32 randomWalksOn: Scale sakura octaves: 2 root: 48.
64 quavers notes: n to: #acid.

p stop.
```

Listing 1: Coypu usage example

Line 1 assigns to the variable *p* a *Performance* whose *Performer* handles sending OSC messages to the audio server.
Lines 5–9 illustrate how different *Sequencers* are created and assigned to a *key* in the *Performance* using the *to:* message. This key corresponds to the `/<instrumentName>` prefix used in the OSC messages sent to the audio server.
Sequencers can be created using several mechanisms:

- sending the message *asSeq* to an array of 1s and 0s;
- sending *hexBeat* to a string containing hexadecimal symbols;
- sending *randomTrigs* to an integer;
- sending *asRhythm* to one of the world rhythms available in Coypu[18];

[17]For example Pure Data, ChucK, SuperCollider, or Max/MSP.
[18]Numerous world rhythms described in [31] are included with detailed commentary on the geographical, historical, and social origins of the rhythms, to fulfil an educational purpose

- sending *euclidean* to an array of two integers, where the first specifies the number of triggers and the second the total number of pulses.

The *Performance* is started with the *play* message(line 11); individual *Sequencers*.can be muted (line 13) or soloed(line 14), and the playback speed of the Performance can be modified (line 12)

Line 16 shows an example of creating *Sequencers*, using a syntax that is a simplified version of Tidal Cycles' mini-notation[19].

Line 29–32 shows how to arpeggiate chords, while the next line creates a random walker on a Sakura scale spanning two octaves, starting from MIDI note 48.

7 Phausto in a lap

Phausto is a lightweight library and API for sound generation and DSP programming in the Pharo environment. It embeds a Faust compiler to deliver advanced audio capabilities, using Foreign Function Interface calls to communicate with it[4]. Phausto is developed with three primary objectives:

(1) To allow sound artists and musicians to program synthesizers and effects, and compose music using Pharo.
(2) To teach DSP programming to beginners and offer a fast prototyping platform for audio developers.
(3) To enrich Pharo applications with sounds and enhance the system's accessibility through auditory cues.

Phausto makes DSP programming fast and intuitive, combining the modular's synthesiser approach to Pharo's Object Oriented principles for clean and understandable code.
The functions defined in Faust standard libraries[20] have been implemented in Pharo as subclasses of the Unit Generator class. These Unit Generators can be patched together and turned into Faust Boxes through FFI calls to *libfaust* library that provide an access point to the Box API. Once a combination of Unit Generator has become a Box, it can be converted into a DSP ready to compute samples and to be connected to the underlying audio driver[21]. The semantics of Phausto and the subdivision of Faust's standard library functions into Phausto Unit Generators subclasses is deeply inspired by the ChucK programming language, from which we borrowed the ChucK operator (=>) to connect Unit Generators.
In Phausto, a DSP is created by sending the `asDsp` message to a Unit Generator or a combination of Unit Generators. The `stereo` message converts the resulting DSP into a stereo configuration. The DSP must then initialised and started to produce sound. Parameters can be controlled by sending the keyword message `setValue:parameter:` to the DSP, and sound generation can be stopped programmatically when required.

```
oscillator := PulseOsc new.
envelope := ADSREnv new label: 'PulseOsc'.
filter :=  OberheimBPF new normFreq: (LFOTriPos new).
reverb := ZitaLight new.

synth := oscillator => envelope => filter => reverb .
dsp := synth stereo asDsp.
dsp init.
dsp start.

dsp traceAllParams
dsp setValue: 0.2 parameter: 'PulseOscDuty'.
dsp playNote: 51 prefix: 'Pulse'  dur: 0.3.

dsp displayUI.
dsp stop.
```

Listing 2: This codes shows a simple single oscillator synth processed by a bandpass filter and reverb.

Listing 2 is an example of a simple synthesiser with controllable parameters. Lines 1–4 show how a new instance of a Unit Generator is created and stored in a variable. The oscillator is first connected to an envelope; their combination then becomes the input of a filter, which is subsequently patched into a reverb. The normalized cutoff frequency of the filter is modulated by a positive triangular low-frequency oscillator. All connections are expressed using the ChucK operator (=>), whose implementation in Phausto delegates the choice of action to the argument. For example, when the argument is an envelope, the receiving Unit Generator is multiplied by the envelope, whereas when the argument is a filter or an effect, the receiver becomes the input of the Unit Generator passed as the argument. On line 6, the resulting signal chain is converted into a Dsp and stored in a variable. The following lines initialize the DSP[22]. A custom label is assigned to the envelope trigger and to the oscillator frequency. In Phausto, all Unit Generator parameters are, by default, associated with UI controls that can be modified at runtime. These controls can be accessed either through a dedicated widget, displayed by sending the *displayUI* message to a Dsp instance (line 13), or programmatically from the Playground by sending the *setValue:* message to the DSP[23]. By exposing parameters as UI primitives by default, Phausto enables more concise code than its Faust counterpart, where parameters must be explicitly declared and UI primitives require more verbose definitions. When a DSP parameter list includes a pair of parameters sharing the same root name and suffixed with `Gate` and `Note`[24], a note number can be scheduled for a specified duration, as shown on line 12. This mechanism enables algorithmic composition using Pharo's native *Delays* and *Processes* , as illustrated in the following example.

```
pattern := [
dsp
    playNote: (Random new nextIntegerBetween: 28 and: 76)
    prefix: 'PulseOsc'
    dur: 0.11.
(Delay   forMilliseconds: 123)  wait ].

[ 256 timesRepeat: pattern ] fork.
```

Listing 3: A short algorithmic composition

The syntax and workflow of Phausto can be learned through the 15 interactive lessons provided in the *MasterLu* additional package[25]. This tutorial covers the creation and combination of Unit Generators, controlling their parameters, and designing or modifying UI widgets for parameter manipulation. It also introduces parameter modulation, basic algorithmic patterns, creating lists and sums of Unit Generators, working with audio files, and examples of subtractive, additive, modal, and FM synthesis.

[19]For a detailed description of this syntax see [5]
[20]These functions implements oscillators, filters, effects, physical models, compressors and low-level utilities for math, timing and control.
[21]CoreAudio for MacOs, ALSA for Linux, RtAudio for WIndows.
[22]Initialization selects the audio backend, sample rate, and buffer size; sending the *start* message then begins audio rendering.
[23]The *traceAllParams* message sent to a Dsp displays the name of all the modifiable parameters
[24]For example, *PulseOscTrigger* and *PulseOscNote* in our example
[25]https://github.com/lucretiomsp/MasterLu

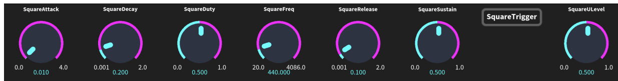

Figure 3: Default UI created with Bloc when the Coypu IDE package is installed.

7.1 TurboPhausto and EcoPhausto

TurboPhausto is a set of synthesizers and effects designed for programming music on-the-fly with Coypu. It is inspired by the SuperDirt[26] audio engine for SuperCollider, it extends the Pharo IDE to fully support live performance and algorave practices. TurboPhausto includes approximately 50 MB of high-quality royalty free audio samples recorded by Legowelt, The Analogue Cops and Lucretio. The samples include bass and snare drums, hihats, cymbals, percussions, bleeps, effects, as well as a complete English alphabet produced with a linear predictive coding (LPC) speech synthesiser. EcoPhausto is a lighter version that loads a limited set of audio samples and synthesizers, with no effects, designed for educational purposes and to run on slower machines.

By sending the message *start* to these classes, a DSP is constructed from the loaded sample players, synthesizers, and effects. The DSP is initialized, started, and assigned to a Performance, whose Performer is a PerformerPhausto. When a TurboPhausto or EcoPhausto instance is stored in a variable, the corresponding Performance and DSP are saved as instance variables, allowing them to be manipulated and interacted with. Users can create their own subclasses of TurboPhausto to suit their needs. They can include fewer or more instruments and chain different types of effects.

8 Future work

Currently, Coypu uses a fixed metrical subdivision of 1/16th notes for all Sequencers; future work will introduce independent metrical subdivisions per Sequencer. Coypu's event scheduling has been refined to reduce jitter, and a wide range of methods for generating rhythms and melodies has been implemented, including a Markov-chain-based jazz generator that follows a set of harmony constraints. Also, our current focus is on ensuring the stability of the system, refining the user interface, and implementing a few additional features, such as support for polyphony and cyclical structures in Coypu.

The coverage of the Faust libraries currently stands at 70%, with plans to achieve full support by the end of the year. Once all the functions of Faust's *Physical Models* library have been ported to Phausto, we plan to provide an extensive tutorial on building instruments based on this type of synthesis.

We are also working on a more flexible and comprehensive MIDI implementation for controlling instruments created with Phausto; at present, multitimbrality and Control Change messages are not supported. In addition, we plan to implement offline audio analysis tools in Pharo, including FFT analysis and RMS detection, as well as the computation of spectral features such as the Centroid and Chroma from the magnitude spectrum. An internal oscilloscope and spectrum analyzer for the real-time output of Phausto are also under development. Finally, we plan to provide a graphical data-flow environment, inspired by Pure Data, for designing DSP by connecting Phausto's Unit Generators. This will offer an alternative paradigm for users who prefer visualizing audio signal flow and will also support teaching and learning purposes.

9 Preliminary Users Evaluation

While a formal empirical evaluation has not yet been conducted, we report preliminary findings from an informal user study carried out during a workshop at the Festival della Robotica in Pisa (May 9–10, 2025). The workshop was attended by 40 participants ranging in age from 11 to 60 years, none of whom had prior experience with live coding. During the workshop, participants were introduced to four live coding environments: Coypu (with EcoPhausto), Strudel [28], Mercury [12], and Sonic Pi [1]. Each environment was presented through a guided hands-on session of comparable duration, after which participants were asked to complete an informal questionnaire regarding their experience. The results indicated that participants favoured Coypu over the other three environments in terms of ease of use, intuitiveness of the syntax, and overall enjoyment. Although these findings are encouraging, we acknowledge that this preliminary assessment has significant methodological limitations: the questionnaire was informal, the sample was drawn from a single event, and the evaluation did not employ standardised instruments or controlled experimental conditions. Moreover, the way in which environments were presented may have introduced bias. To address these limitations and provide more robust evidence for our accessibility claims, we are planning to ask the Faculty of Psychology at the University of Padova for a collaboration for a structured qualitative study. This study will aim to employ validated assessment instruments and controlled conditions, and a larger sample, in order to rigorously evaluate the accessibility, usability, and learning outcomes associated with Coypu and Phausto compared to established live coding environments

10 Conclusion

This paper presents the tools we have been developing to turn the Pharo environment into an instrument for on-the fly music programming. Since its first presentation at Pharo Days 2022 in Lille[27], Coypu has been used for live performances in online streams, including *Kyma Toe and Shell* (2022) and *Toplap Solstice Stream* (2023), as well as at algoraves such as Ohm Berlin (2023). It has also been featured at conference concerts including *ESUG*[28] (2023, 2024, 2025) and *ICLC*[29] (2024, 2025). Workshops have been held at the *ICLC Satellite Event* in Düsseldorf (2023), the *Festival della Robotica* in Pisa, *Robot Festival* Bologna, and the *Audio Developer Conference* in Bristol (2025).

Performances in 2025 relied on Phausto as the audio engine, and can therefore be classified as full Pharo-based performances. This highlights that these two libraries have, *in practice*, transformed Pharo into a musical instrument in its own right. Additionally, a new package, CoypuIDE is under development to offer a set of ready-to-use interface widgets (knobs, faders, buttons, visualisers) and to implement custom ones using Bloc[30], which is a UI framework that will allows us to have native windows.

In the latest two years of its development, Phausto has arrived to a stable state, operating reliably on all platforms and reducing significantly unexpected instabilities. The coverage of the Faust libraries currently stands at 70%, with plans to achieve

[26] https://codeberg.org/musikinformatik/SuperDirt

[27] https://days.pharo.org/index.html

[28] https://esug.org/

[29] https://iclc.toplap.org/

[30] https://github.com/pharo-graphics/Bloc

full support by the end of the year. A framework for exporting DSP created in Phausto to Cmajor[31] patches is mature and fully supports UI sketching with Bloc.

At the same time, we are working to expand Pharo's adoption as a musical instrument among musicians, sound artists, and music schools through workshops, conference participation, journal publications, and educational partnerships.

Figure 4: Lucretio performing with Pharo at LAUT Barcelona during *Algo: Noise - Audiovisual Live Coding* (ICLC 2025).

11 Ethical Standards

All research conducted in this work adhered to accepted principles of ethical and professional conduct. The development and testing of Coypu and Phausto involved only software and musical performance with voluntary participation of human musicians. No experiments caused harm or risk to participants. All contributions were based on informed consent, and no data requiring anonymity or privacy concerns were collected.

Funding for this work was initially provided by the Pharo Association, and the authors declare no financial or non-financial conflicts of interest. This research did not involve animal subjects.

We strived to ensure transparency, reproducibility, and ethical engagement in all aspects of developing and evaluating our musical programming environment.

Acknowledgments

We would like to thank the Pharo Association for the initial funding that supported the early development of Phausto, and the Evref team at Inria for their invaluable support over the past years in the process of turning Pharo into a musical instrument. We also thank the Faust team at GRAME, Lyon (in particular Stéphane Sletz) for their guidance and support regarding the Faust libraries and architectures. We would alike to thank Ge Wang for friendly allowing us to borrow the ChucK operator. In addition, we are grateful to Cesare Ferrari of Cmajor Software Ltd. for imlpementing the Phausto-to-Cmajor *polyphonic wrapper*. Finally, we thank the ESUG organizers for providing us with the opportunity to share our project and sound art with the international Smalltalk community.

References

[1] S. Aaron. 2016. Sonic Pi – Performance in Education, Technology and Art. *International Journal of Performance Arts and Digital Media* 12, 2 (2016), 171–178. https://doi.org/10.1080/14794713.2016.1227593

[2] Richard Boulanger (Ed.). 2000. *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming.* The MIT Press.

[3] F. Calegario, J. Tragtenberg, C. Frisson, E. Meneses, J. Malloch, V. Cusson, and M. M. Wanderley. 2021. Documentation and Replicability in the NIME Community. In *Proceedings of the International Conference on New Interfaces for Musical Expression.* Shanghai, China.

[4] Domenico Cipriani, Alessandro Anatrini, and Sebastian Jordan Montaño. 2024. Phausto: Embedding the Faust Compiler in the Pharo World. In *Proceedings of the International Faust Conference (IFC-24), Soundmit, Turin, Italy, November 21-22, 2024.* Turin, Italy.

[5] Domenico Cipriani, Sebastian Jordan Montaño, and Nahuel Palumbo. 2025. Coypu: Gnawing Music On-The-Fly With Pharo. In *ICLC 2025 - International Conference on Live Coding.* Barcelona, Spain. https://hal.science/hal-05341056

[6] Domenico Cipriani, Sebastian Jordan Montaño, Nahuel Palumbo, and Stéphane Ducasse. 2025. Composing and Performing Electronic Music on-the-Fly with Pharo and Coypu. In *International Workshop on Smalltalk Technologies (IWST 2025).* CEUR-WS.org. https://hal.science/hal-05306163 https://ceur-ws.org/Vol-4139/Paper03.pdf.

[7] Domenico Cipriani, Nahuel Palumbo, Sebastian Jordan Montaño, and Stéphane Ducasse. 2024. Phausto: fast and accessible DSP programming for sound and music creation in Pharo. In *IWST 2024: International Workshop on Smalltalk Technologies.* Lille, France. https://hal.science/hal-04826894

[8] Koen De Hondt, Stéphane Ducasse, Sebastian Jordan Montano, and Esteban Lorenzano. 2024. *Application Building with Spec 2.0.* Book on Demand – Keepers of the lighthouse. 240 pages. http://books.pharo.org

[9] Stéphane Ducasse and Guillermo Polito. 2020. Concurrent Programming in Pharo. http://books.pharo.org/booklet-ConcurrentProgramming/

[10] Stéphane Ducasse, Guillermo Polito, and Juan Pablo Sandoval. 2023. *Testing in Pharo.* Book on Demand – Keepers of the lighthouse. 80 pages. http://books.pharo.org

[11] Stéphane Ducasse, Gordana Rakic, Sebastijan Kaplar, and Quentin Ducasse. 2022. *Pharo 9 by Example.* Book on Demand – Keepers of the lighthouse. http://books.pharo.org Originally written by A. Black and S. Ducasse and O. Nierstrasz and D. Pollet with D. Cassou and M. Denker.

[12] T. Hoogland. 2019. Mercury: a live coding environment focussed on quick expression for composing, performing and communicating. In *Proceedings of the International Conference on Live Coding (ICLC).* Madrid, Spain.

[13] Alan C. Kay. 1993. The Early History of Smalltalk. In *ACM SIGPLAN Notices*, Vol. 28. ACM Press, 69–95. https://doi.org/10.1145/155360.155364

[14] Alan C. Kay and Adele Goldberg. 1977. Personal Dynamic Media. *Computer* 10, 3 (1977), 31–41. https://doi.org/10.1109/C-M.1977.217672

[15] Mark Lentczner. 1985. Sound Kit: A Sound Manipulator. In *Proceedings of the International Computer Music Conference (ICMC).* International Computer Music Association, Burnaby, BC, Canada.

[16] Sabina Leonelli. 2023. *Philosophy of open science.* Cambridge University Press.

[17] J. McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal* 26, 4 (2002), 61–68. https://doi.org/10.1162/014892602320991383

[18] A. McLean. 2014. Making Programming Languages to Dance to: Live Coding with Tidal. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design.* Madrid, Spain.

[19] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two decades of Smalltalk VM development: live VM development through simulation tools. In *Proceedings of International Workshop on Virtual Machines and Intermediate Languages (VMIL'18).* ACM, 57–66. https://doi.org/10.1145/3281287.3281295

[20] F. Morreale, S. A. Bin, A. P. McPherson, P. Stapleton, and M. Wanderley. 2020. A NIME of the Times: Developing an Outward-Looking Political Agenda for This Community. In *Proceedings of the International Conference on New Interfaces for Musical Expression.* 160–165. https://doi.org/10.5281/zenodo.4813294

[21] Yann Orlarey, Dominique Fober, and Stéphane Letz. 2009. Faust: an efficient functional approach to dsp programming. *New computational paradigms for computer music* (2009), 65–96.

[22] Nahuel Palumbo, Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. 2025. Meta-compilation of Baseline JIT Compilers with Druid. *The Art, Science, and Engineering of Programming* 10, 1 (feb 2025). https://doi.org/10.22152/programming-journal.org/2025/10/9

[23] Nahuel Palumbo, Guillermo Polito, Stéphane Ducasse, and Pablo Tesone. 2026. Are Abstract-interpreter Baseline JITs Worth it? An Empirical Evaluation through Metacompilation. In *International Symposium on Code Generation and Optimization (CGO 2026).* Sydney, Australia. https://inria.hal.science/hal-05407834

[24] Jean Piaget. 1952. *The Origins of Intelligence in Children.* W.W. Norton & Co., New York, NY.

[25] Guillermo Polito, Stéphane Ducasse, Pablo Tesone, and Ted Brunzie. 2020. Unified FFI - Calling Foreign Functions from Pharo. http://books.pharo.org/booklet-uffi/

[26] Guillermo Polito, Pablo Tesone, Jean Privat, Nahuel Palumbo, and Stéphane Ducasse. 2023. Heap Fuzzing: Automatic Garbage Collection Testing with Expert-Guided Random Events. In *International Conference on Software Testing.*

[27] Stephen Travis Pope. 1987. A Smalltalk-80-based Music Toolkit. In *Proceedings of the International Computer Music Conference (ICMC).* International

[31] These Cmajor patches can be used as audio plugins in VST3- and AU-compatible DAWs via the Cmajor patch loader. (https://cmajor.dev/)

Computer Music Association, Champaign–Urbana, IL, USA.
[28] F. Roos and A. McLean. 2023. Strudel: Live Coding Patterns on the Web. In *Proceedings of the 7th International Conference on Live Coding (ICLC)*. Utrecht, Netherlands. https://doi.org/10.5281/zenodo.7842142
[29] C. Scaletti. 1989. The Kyma/Platypus Computer Music Workstation. *Computer Music Journal* 13, 2 (1989), 23–28.
[30] Carla Scaletti. 2002. Computer Music Languages, Kyma, and the Future. *Computer Music Journal* 26, 4 (2002), 69–82.
[31] Godfried T. Toussaint. 2019. *The Geometry of Musical Rhythm: What Makes a "Good" Rhythm Good?* (2nd ed.). Chapman and Hall/CRC, New York. 370 pages.
[32] Ge Wang, Perry R Cook, et al. 2003. ChucK: A concurrent, on-the-fly, audio programming language. In *ICMC*.