Enhancing Liveness with Exemplars in the Newspeak IDE

Gilad Bracha
F5 Networks/Shape Security
USA
gilad@bracha.org

Abstract

Traditional program editing tools are separate from program evaluation tools.

Exceptions are debuggers as well as REPLs and similar tools such as Smalltalk workspaces and object inspectors, or computational notebooks. However, these are not designed for full scale program development. Most such tools do not support evaluation of code within an abstraction, such as a function or class instance. Debuggers do allow such evaluation, but require the user to navigate to a position in the code by manually initiating an execution path that leads to that position.

As a result of the separation of editing from evaluation, programmers writing or reading a program must mentally simulate its execution in order to understand it. To avoid the cognitive burden of such simulation, this paper argues that program editing should be tightly integrated with an evaluator, so that any expression or statement being edited can be evaluated at will. We describe the design of a development environment for the Newspeak programming language which fulfills this requirement.

1 Introduction

You have downloaded the code of a program from a source code repository, and you want to understand what the program does. You open it in an editor and start reading. As you read, you find yourself forced to simulate the execution in your mind and retain the state of the program in memory - your memory, not the computer's. This is obviously stressful and error prone.

Continuing the above scenario, suppose you realize how absurd this situation is. Executing programs is the computer's forte, its very raison d'etre. Why should humans attempt to emulate execution by hand? And so, you decide to execute the program. Easier said than done. In most cases, you have to build the program (often a non-trivial exercise requiring you to follow brittle and unclear instructions) and then you have to run it. You may not know what data to use, or may not have access to such data. Even if you do, you will want to observe the program internals at select points. And so you must run it in the debugger. You will have to decide where to break the execution to see what is going on, and when to proceed. Once you have proceeded, there is usually no way back, except restarting execution - unless you are very, very lucky and have a time traveling debugger. If you are interested in an execution path that was not executed, you'll need to run the program on alternate data (which?).

In short, none of this is convenient, or even feasible. Which is why the standard practice is to just load the code into a text editor and simulate the computation in one's head.

This paper describes a system that addresses the scenario above (and others). We have implemented an interactive development environment in which, by and large, code can always be evaluated, without the need to manually bring an executing program to a particular point in the code.

Our system is an IDE for the Newspeak programming language [10], but the ideas apply to a wide variety of languages. Newspeak is a language in the Smalltalk [13] tradition, but it differs significantly from Smalltalk, as it is a modular object-capability [16] language.

Smalltalk systems (and similar ones, like the advanced Lisps on Lisp machines) mitigate the scenario described above somewhat, but not in a fundamental way. We discuss this matter in detail in section 7.

2 Background: Newspeak and its Environment

Newspeak is a class-based language distinguished by two defining properties: all names are late-bound, and there is no global scope. It uses classes as its sole structuring construct; there are no packages, units, templates or module declarations in the traditional sense. Instead, top level classes define modules. These can have classes nested within them, recursively.

Newspeak code is typically developed in the context of an IDE in the tradition of Smalltalk and Self [24]. The latest incarnation of this IDE is web-based, and is the basis for this work. The IDE can be accessed online [20].

Though the Newspeak language does not have a global scope, the IDE maintains a namespace of top level classes that it has loaded, known as the *root namespace*. During development, this namespace is useful in circumventing many of the apparent difficulties that might otherwise arise from the absence of a top level scope.

3 Obtaining Exemplar Data

Our design is based on the idea that the writer of a program provides metatadata that describes how every method is invoked. In other languages, there may be other abstractions - functions, constructors, initializers, static "methods", rules, constraints and so on. Nevertheless, the same approach applies: any abstraction defined by the programmer abstracts over something, and the programmer must provide the system with sample data to instantiate the abstraction.

In our system, the abstractions are classes and methods. Classes are instantiated by method invocation, and so we only have to deal with methods as an abstraction mechanism. As an example, we show the header for a class BankAccount [18].

```
class BankAccount balance: b  <Integer>
(* :exemplar: BankAccount balance: 100 *) = (
|
        private balance_slot <Integer> ::= b.
|
)
```

The code above defines the class together with its *primary factory method*, which is named balance:. To instantiate the class, one has to invoke balance: on the class (which is itself an object), providing a suitable argument. An example instantiation is given above, by means of a *metadata comment*. Comments in Newspeak are delimited by (* and *), as in Pascal [14].

The comment begins with a colon-delimited identifier :exemplar:. This is a convention that identifies it as a metadata comment. The identifier is used to determine how to interpret the rest of the comment. Newspeak's reflection system provides access to metadata, and the Newspeak IDE interprets metadata comments whose identifier begins with exemplar as data indicating how to generate exemplar data for a give abstraction.

The remainder of the comment in our example is an expression that creates an instance of BankAccount. The IDE can use this information to create an exemplar instance of BankAccount. However, there is a question as to what scope should this expression be evaluated in.

In the case of top level classes, we evaluate the instantiation expression with respect to the IDE's root namespace. The same is true for class methods of top level classes.

A similar mechanism is used for methods. BankAccount has a method withdraw:, shown below:

```
public withdraw: amount <Integer>
(* :exemplar: withdraw: 100 *) = (
  amount > balance ifTrue: [
   Error signal: 'Overdraft not allowed. Withdrawal amount ',
                 amount printString,
                 ' exceeds balance ',
                 balance printString
  ].
  balance_slot:: balance - amount
)
```

Here we see that the exemplar provides a sample invocation of withdraw:. The method invocation is evaluated in the scope of the enclosing instance - that is, the instance of BankAccount we derived from the metadata for the classes' factory above.

Nested classes are properties of instances, just like methods. As with methods, the exemplar of a nested class is computed in the scope of the surrounding classes' exemplar.

In another language, different rules would have to used to determine the scope in which to evaluate exemplar-generating code. Nevertheless, the principle of using metadata to specify how to instantiate an abstraction remains applicable.

Some languages have non-executable constructs such as type declarations. Even in these cases, it should be possible to specify how to produce sample data to provide concrete exemplars of these declarations (e.g., sample records or tuples for record or tuple types, or example functions for function type signatures).

3.1 Alternative Mechanisms for Generating Exemplar Data

There are various refinements and alternatives to the mechanism described above. Methods that take no parameters do not really require exemplar data. A common case is a class which uses the default factory, new. We simply invoke new on the class to obtain an exemplar. In Newspeak, there are several conventions for method names which give a strong indication of what parameters are expected, and we intend to utilize these as well, to reduce the burden of writing metadata (see discussion in section 6 below).

Tests are a natural source for exemplar data. Type information can be helpful as well - if we know the type of a variable, and we know how to instantiate the type, we can populate the variable. However, types may be given as names of abstract classes (and in other languages, interfaces) in which case we may not know how to instantiate the type.

Finding senders of methods can help alleviate the burden of metadata in cases where the senders (or their senders, recursively) already have exemplars. This would be somewhat similar to what is done in Margin Notes [15].

Conceivably, one might perform sophisticated static analysis to determine how to create exemplars.

In all these cases, one eventually has to extract source code that will be used as metadata so that the exemplars can be recreated when loading code from source.

Another option is to locate existing instances in the heap, which may have been produced by tests, debugging and so on. In fact, we experimented with this strategy in the past [4–7]. It was insufficient by itself. It also suffers from the fact that one cannot produce exemplars for newly created or loaded code. Our metadata based stategy allows us to read code from source and produce exemplars for it immediately. In contrast,

to produce reusable metadata from existing instances, we would need to know the provenance of these instances.

4 Using Exemplar Data

As indicated in the introduction, our primary goal is to use exemplars to provide bindings for names in the program so users can evaluate expressions, in order to ease their cognitive load. Exemplars may be leveraged in additional ways: they can help create tests or infer types for example.

4.1 Easing Cognitive Load

Figure 1 shows the class BankAccount as displayed by the IDE.

The class is automatically displayed in the context of an object inspector on an instance. The instance is the result of evaluating the exemplar code, and is used as the value of self throughout the class instance methods.

The instance variables are listed at the bottom of the inspector. In our case there is only one such variable, balance_slot whose value is shown to be 100 - the value passed in to the class' factory method in the exemplar code. Immediately above the instance variables is an *evaluator* - a text pane in which free form code may be typed and evaluated.

All of the above is very standard. What is unusual is that the methods are displayed in the same context, and that their source can be evaluated. Furthermore, the programmer did not take any action in order to create the object being inspected. It is not the programmer's responsibility to ascertain suitable arguments with which to instantiate the class - this happens automatically when the class is viewed.

Each method has an associated Debug button. Clicking on this button opens a debugger at the beginning of the method, using the exemplar receiver and arguments. In addition, one may select and evaluate code snippets within the method editor itself, and the results are displayed below the method body. In Figure 1, we see the value false as the result of evaluating amount > balance.

It is often useful to provide multiple exemplars. A method may be called with different kinds of arguments; distinct argument values may provoke different control paths within a method. In the latter case, having multiple exemplars is necessary to produce complete code coverage, so that code in every part of the method can be evaluated at will.

It is perfectly acceptable to provide multiple metadata comments each producing a distinct exemplar. The system will then provide a menu of exemplars in place of the Debug button, with each menu entry invoking the debugger on the appropriate exemplar. To distinguish among the entries, each exemplar should be named, ergo, :exemplar-1:, exemplar-false: and the like.

It is useful to have an additional evaluator pane where arbitrary expressions can be entered and evaluated in the context of the method. Such an evaluator can be opened via

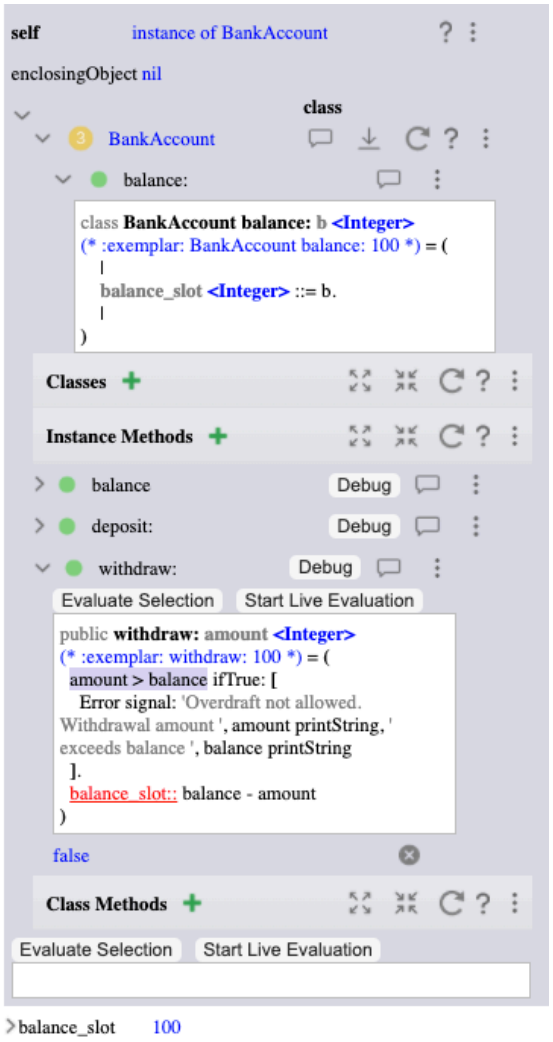

Figure 1. Class BankAccount

a menu option, but is not open by default. One needs to be careful managing screen real estate. In previous attempts [6, 7], we exposed a full debugger UI for every method. This occupied too much space.

Another attractive option is to evaluate code as one types, so one can see the results as one codes. The Start Live Evaluation button is intended to toggle this functionality, which can also be very distracting. Currently this feature only works in free form evaluators. We hope to enable it in method editors in the near future.

Exemplars can also be used for name completion, though we have not yet explored this option. See the discussion of types in the following subsection.

4.2 Additional Uses

Exemplars have an intriguing relationship with tests. Tests can be used to generate exemplars, but the inverse is almost true as well: exemplars can help generate tests; what's missing is a specification of the desired test result. Nevertheless, and interactive tool can use exemplars to produce part of the test boilerplate, leaving it to the user to complete.

Types too have a bidirectional relationship with exemplars. Type information would help produce exemplars, but, as noted in section 3.1, some information might be missing. Conversely, exemplars can be used to produce type information. This idea is closely related to live typing [26]. An obvious concern is that the derived type information may be too specific, but this can be mitigated by using several exemplars, carefully chosen.

5 An Advanced Example

The previous example is, by design, very simple. The only parameters are integer literals. Often, the parameters are objects that need to be instantiated themselves, and these in turn may have arguments and so on. Below is an example illustrating how such situations are handled.

The code shows the header of class MetadataParsing, a utility class used by the system itself to parse metadata. The class factory method takes two parameters: a platform object and a scanner class.

Using a platform object is a common idiom in Newspeak. The platform object provides access to standard functionality. In the case of MetadataParsing it is used to access collection classes such as List and Map. The workspace scope used for top level class exemplars provides access to the platform directly. [1]

The scanner class we use in the exemplar is nested in class NewspeakPredictiveParsing. Nested classes in Newpeak are properties of instances of their enclosing class, and so we must instantiate NewspeakPredictiveParsing in order obtain a scanner class. The factory of NewspeakPredictiveParsing again takes a platform, but also an instance of an AST class, produced in turn by instantiating NewspeakASTs with platform.

```
class MetadataParsing usingPlatform: platform scannerClass: S < NewspeakPredictiveParsing Scanner class >
(* :exemplar:
[
|
asts = NewspeakASTs usingLib: platform.
Parser = NewspeakPredictiveParsing usingPlatform: platform asts: asts.
|
MetadataParsing usingPlatform: platform scannerClass: Parser Scanner
] value
*)
= (
(*
Module for parsing Newspeak language metadata.
*)
|
  List = platform collections List.
  Map = platform collections Map.
  Scanner = S.
|
)
```

The exemplar is computed by a closure (delimited by square brackets). The closure is then executed by invoking the method value. This idiom allows us to define local variables that name subtrees of the overall exemplar expression. In the above example, the use of the closure is not essential; we could have written the exemplar expression as

```
MetadataParsing
  usingPlatform: platform
  scannerClass: (
    NewspeakPredictiveParsing
      usingPlatform: platform
      asts: (NewspeakASTs usingLib: platform)
  ) Scanner
```

However, in cases where intermediate expressions must be shared, using local variables is necessary. Using closures allows us to instantiate objects in a completely general way using only expressions. In some other languages exemplars may have to be specified via constructs other than expressions

6 Limitations

Here are some downsides/difficulties with our approach:

The burden of specifying metadata. Adding the metadata is not too burdensome if it is done on an ongoing basis during development. However, converting a substantial preexisting code base can be challenging.

Erroneous exemplars. What happens if the exemplar code raises an exception? In that case, we fall back to the regular, "dead" display. Thus, we can never guarantee the live presentation of every last piece of code. If a method or class displays as "dead" one can paste its exemplar code into an evaluator for the surrounding scope (i.e., the enclosing object 'presenter, or, for top-level classes, an IDE workspace). The code may then be evaluated and debugged in the normal fashion. We are considering UI affordances to streamline this process.

Dependency Management. We cannot produce exemplars of a method without access to classes whose instances

[1] The use of platform is so common that if a class uses a factory named usingPlatform:, the IDE automatically provides an exemplar without requiring explicit metadata.

are used as its parameters. If we load a module in complete isolation, we may not be able to instantiate the metadata if it relies on classes that are not already loaded in the IDE.

Exemplars interacting with the running system. Care must be taken when producing and using exemplars of classes that are part of the running system - the existing platform libraries or the IDE itself. It may be too easy to tamper with the underlying system in this situation. There are well known techniques against "shooting one self in the foot": running a distinct copy of the system in a separate process or actor and manipulating it via mirrors for example [9, 27].

Security. Loading code into the IDE is now a security issue, since viewing the code may cause execution in the IDE scope, which has full access to the platform, including writing files etc. Some restriction to a safer sandbox, or preventing exemplar metadata from running without some review/permission seems desirable.

Discouraging Abstraction. Some may complain that by providing concrete exemplar data, the system discourages programmers from thinking abstractly and considering all possible values.

Documentation Rot. As code evolves, comments often become obsolete through lack of maintenance. However, compared to traditional comments, developers have a stronger incentive to maintain exemplar metadata. The situation is more directly analogous to maintaining tests. Should the metadata evaluation fail, developers lose the considerable benefits of exemplars. Furthermore, because metadata is accessible via reflection, developers can access it easily and reliably. This makes it straightforward to automatically verify that exemplars are operational.

7 Related Work

The concept of unifying editing and evaluation tools based on examples in the text originates in [12]. That work extends Java and Eclipse and uses a two pane tool which integrates an editor in one pane, and trace view in another. It uses traces of complete exemplar execution to provide variable bindings, whereas we use exemplars to invoke methods, but rely on either classic fix-and-continue debugging or explicit user evaluation within methods. The paper discusses extensions to allow recursively inspection of objects, but this was not implemented.

The Babylonian-style editor [17] for Javascript is quite similar to our work in many ways. It supports the definition of multiple, named exemplars by the user much as we do. It too is based on the use of traces.

Margin Notes [15] produces exemplar data from prior executions of Ruby code. Margin Notes does not support full evaluation of code based on the gathered exemplars differs and the data gathered is not integrated into the source code.

Debuggers provide a live evaluation environment for code inside abstractions, but one has to navigate to the code via an execution path: one has to call it. The code browsing experience is constrained by the execution path; for example, if one wants to examine all the callers of a function, one cannot observe valid bindings in any of these except the actual caller currently on the stack.

Self [24], being a prototype based language with a live IDE, naturally provides a context where the instance is always available for evaluation, and instance variables are always populated. However, Self does not provide bindings for parameters. And when a language does not rely on prototype objects, the problem of dealing abstractions and their parameters becomes even more acute. This is the principle issue we address.

REPLs allow for code to be incrementally written and evaluated. They are not typically geared toward reading and evaluating existing code. Often, they are not suited to the creation of long lived code. Consider how awkward it is to define a function in the Python REPL for example.

Languages often require declarations such as modules or packages which do not denote values, and support for these in many traditional REPLs is either lacking or poor. In any event, in order to see values inside an abstraction such a function, one has to manually make a call. See [25] for an extensive discussion of REPLs.

Classic BASIC implementations allowed for the creation and editing of code in an interactive REPL. These relied in systemic use of line numbers, and allowed new code to be entered by adding lines with intermediate numbers. As such, they are closer in spirit to the approach we advocate than many later REPLs. However, they did not scale well beyond a single function. Similar, more sophisticated facilities, called workspaces, were available in APL systems since the 1960s. However, in all these systems, the only way to populate variables inside an abstraction is to make a call and navigate though an execution path.

Emacs is an editor with an built in scripting language, elisp, and it can evaluate elisp expressions. The difficulty arises when one has to evaluate code within an abstraction, like a function.

The most sophisticated environments are those of Lisp, especially Lisp machines, and of Smalltalk, but they still have the same basic limitations.

Smalltalk workspaces allow for a much more fluid experience than REPLs. Code snippets can be selected and evaluated, but they are not intended for the creation of complete programs. Methods and classes must be created in class browsers. These allow for evaluation of expressions that have meaning in static scopes (either the global scope, or that of the class) but not of expressions whose value relies on the instance or the parameters of a method.

Even closer to our model are Smalltalk object inspectors, but one has to have an object in hand to use them, Also, as

in Self, they do not offer support for examining the values of parameters or local variables.

As noted above, exemplars have a close relationship to unit testing.

The Pyret programming language [2] allows function declarations to optionally incorporate unit tests as part of their syntax. This information could be used in place of the metadata we use, however the existing tooling, which includes an online editor and REPL, does not do so. The idea of extending Newspeak syntax to allow, or even require, exemplars is intriguing however.

Python's doctest module [3] supports execution of tests based on example code provided in comments. There is however no live editor which would tie the data produced by the examples to the code being edited.

Computational notebooks combine sample data with code unlike the systems above. Unfortunately, they provide scant support for navigating codebases. Even the constrained navigation model typical of debugging is not supported. One can see the bindings of expressions to data in the notebook, but one cannot usually navigate into the bodies of functions being called.

Last but not least, the Glamorous Toolkit [11, 23] is a programming environment where object inspectors play a central role. Exemplars are closely related; if everything is an object in the language, than everything is an object inspector in the tooling. [2]

8 Status

Exemplars work in all code views: in class presenters, when searching for methods or classes by name, when listing senders or implementers of a message, in the unit test suite presenter, and of course in the debugger.

Nevertheless, exemplar support in the Newspeak IDE is a work in progress, as is the web based IDE itself. It is definitely not production ready software by any stretch of the imagination. As noted above, we expect this functionality to evolve, especially in terms of the UI.

In the existing Newspeak codebase, most classes and methods do not have exemplar metadata, and so these classes are displayed "dead" as is the norm. Once the exemplar implementation is more stable, we plan on adding metadata throughout the system. At that point we will have achieved the goal of having live exemplar data everywhere we view or edit code.

Source code is available at [19]. The system is available for download as an Electron [1] app on the web [21], and can also be run online in a web browser [20]. Further information is available at [22]. A brief video demonstrating the system described in this paper is available at [8].

[2]This does not imply that the approach is limited to object-oriented programming; exemplars can represent live values, objects or otherwise.

9 Conclusions

Computers should relieve programmers of the burden of simulating execution when reading code by providing live data for all expressions in the program. This is can be achieved by integrating sample expressions for instantiating abstractions into the source code, and using editors integrated with live evaluation engines to navigate that code.

References

[1] [n.d.]. *Electron JS web site.* https://www.electronjs.org/

[2] [n.d.]. Pyret programming language web site. Retrieved August 16, 2021 from https://www.pyret.org/

[3] [n.d.]. *Python doctest module online documentation.* Retrieved August 16, 2021 from https://docs.python.org/3/library/doctest.html

[4] Gilad Bracha. 2012. *Debug Mode is the Only Mode.* https://gbracha.blogspot.com/2012/11/debug-mode-is-only-mode.html

[5] Gilad Bracha. 2013. *Live 2013 Newspeak Demo.* https://youtu.be/74WqdS_58uY

[6] Gilad Bracha. 2013. *Making Methods Live.* https://gbracha.blogspot.com/2013/04/making-methods-live.html

[7] Gilad Bracha. 2017. *Newspeak exemplar-mode demo from Live Literate Programming talk at Programming 17.* https://youtu.be/Yv7yX27Tx4U

[8] Gilad Bracha. 2021. *Exemplars in the Newspeak Web IDE.* https://youtu.be/qKWPSvcF0zA

[9] Gilad Bracha and David Ungar. 2004. Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications.*

[10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. 2010. Modules as Objects in Newspeak. In *European Conference on Object-Oriented Programming.*

[11] Andrei Chis. 2016. *Moldable Tools.* Ph.D. Dissertation. University of Bern, Institue of Computer Science.

[12] Jonathan Edwards. 2004. Example Centric Programming. *ACM SIGPLAN Notices* 39 (December 2004), 84 – 91. Color version at http://www.subtext-lang.org/OOPSLA04.pdf.

[13] A. Goldberg and D. Robson. 1983. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley.

[14] K. Jensen and Niklaus Wirth. 1978. *Pascal User Manual and Report* (second ed.). Springer-Verlag.

[15] Geoffrey Litt. 2018. *Margin Notes.* https://www.geoffreylitt.com/margin-notes/

[16] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control.* Ph.D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA.

[17] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *Journal on The Art, Science, and Engineering of Programming* 3 (2019).

[18] Newspeak Team. 2021. *Class BankAccount Source Code.* Ministry of Truth. https://github.com/newspeaklanguage/newspeak/blob/master/BankAccount.ns

[19] Newspeak Team. 2021. *Newspeak github Repository.* Ministry of Truth. https://github.com/newspeaklanguage/newspeak

[20] Newspeak Team. 2021. *Newspeak IDE on WASM.* Ministry of Truth. https://newspeaklanguage.org/samples/primordialsoup.html?snapshot=HopscotchWebIDE.vfuel

[21] Newspeak Team. 2021. *Newspeak Programming Language downloads page.* Ministry of Truth. https://newspeaklanguage.org/downloads.html

[22] Newspeak Team. 2021. *The Newspeak Programming Language web site.* Ministry of Truth. https://newspeaklanguage.org/

[23] The Feenk Team. [n.d.]. *Glamorous Toolkit.* Retrieved August 19, 2021 from https://gtoolkit.com/ See also feenk.com and moldabledevelopment.com.

[24] David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices.* 227–242. Published as Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, number 12.

[25] Thomas van Binsbergen, Mauricio Verano Merino, Pierre Jeanjean, Tijs van der Storm, Benoit Combemale, and Olivier Barais. 2020. A principled approach to REPL interpreters. In *Onward! 2020 - Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Co-located with SPLASH 2020.* 84–100. https://doi.org/10.1145/3426428.3426917

[26] Hernán Wilkinson. 2019. VM support for live typing: automatic type annotation for dynamically typed languages. In *Programming '19: Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming* (Genova, Italy). Association for Computing Machinery, New York, NY, USA.

[27] Allen Wirfs-Brock, Juanita Ewing, Harold Williams, and Brian Wilkerson. 1996. A Declarative Model for Defining Smalltalk Programs. invited talk at OOPSLA 96; available at http://www.smaltalksystems.com/publications/_awss97/index.htm.