Implementing Babylonian/S by Putting Examples Into Contexts

Tracing Instrumentation for Example-based Live Programming as a Use Case for Context-oriented Programming

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Jens Lincke
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jens.lincke@hpi.uni-potsdam.de

Stefan Ramson
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
stefan.ramson@hpi.uni-potsdam.de

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Fabio Niephaus
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Based on context-oriented programming (COP), we implemented source code instrumentation for example-based live programming in a modular way. These tools provide programmers with feedback on the dynamic program behavior by showing traced values of example invocations of a program. For that, we have to trace intermediate, expression-level runtime states during the execution of an example. As the instrumentation is only intended to improve tool support, the default behavior of the system must not be altered. In this paper, we demonstrate how context-oriented programming can be used to keep the execution of examples separate from the default behavior even in the presence of expression-level behavior variations. We illustrate our approach by implementing Babylonian Programming using ContextS2 in Squeak/Smalltalk. Based on the implementation, we compare our COP-based Smalltalk implementation to the module-rewriting-based implementation for JavaScript.

CCS CONCEPTS

• **Software and its engineering** → *Object oriented languages*; *Integrated and visual development environments*.

KEYWORDS

context-oriented programming, example-based live programming, use case, Squeak/Smalltalk

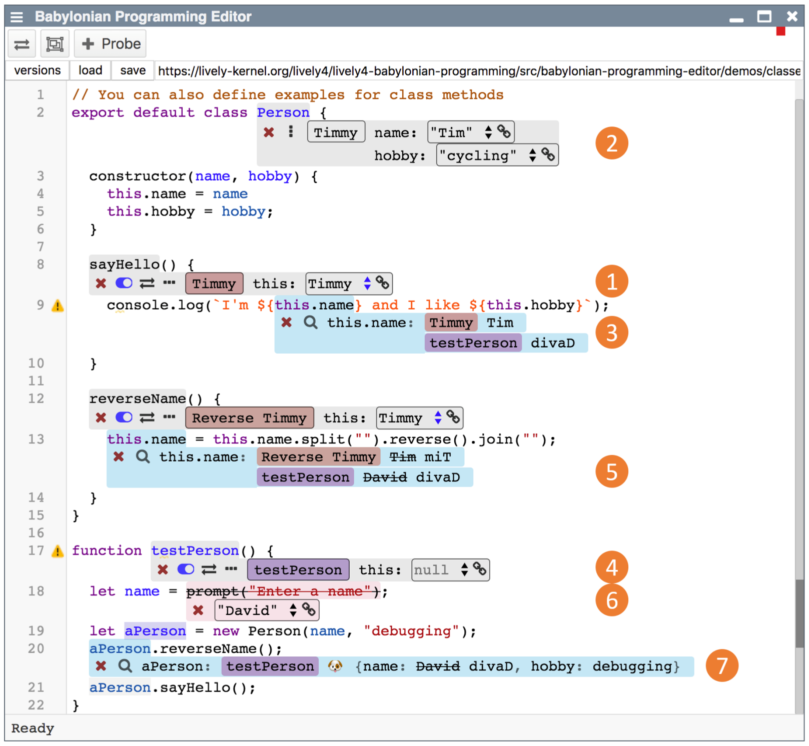
ACM Reference Format:
Patrick Rein, Jens Lincke, Stefan Ramson, Toni Mattis, Fabio Niephaus, and Robert Hirschfeld. 2019. Implementing Babylonian/S by Putting Examples Into Contexts: Tracing Instrumentation for Example-based Live Programming as a Use Case for Context-oriented Programming. In *Workshop on Context-oriented Programming (COP'19), July 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3340671.3343358

1 INTRODUCTION

Live programming promises to improve program comprehension and to make programming more accessible by providing immediate feedback on the dynamic behavior of a program [8]. This feedback on dynamic behavior often consists of intermediate run-time states of some execution of the program. In order to be able to execute the program, some input data is required. Example-based live programming environments try to make use of examples to be able to provide feedback on the dynamic behavior of different parts of systems [1, 7] (see Figure 1). Programmers can define explicit examples, which are then used by the system to execute the program and trace intermediate run-time states. Babylonian Programming is an approach for example-based live programming environments which enables programmers to use examples in larger systems which span multiple modules [7].

In order to provide feedback on intermediate run-time states, the state execution of the example has to be traced. In addition, programmers might instrument the source code even further, for example with additional checks on the traced run-time state. At the same time, as Babylonian programming is intended as a programming tool, the instrumentation of the source code should not affect the behavior of the system under development. This poses a modularity challenge as the instrumentation is attached to the source code but may only be executed during the execution of an example.

The original implementation of Babylonian Programming used an approach based on the ES6 module system. The approach required an intricate rewriting of all modules containing any instrumentation. Further, it required a source code rewriting of the import statements of instrumented modules to load other instrumented


Babylonian Programming Editor
Probe
versions
load
save
https://lively-kernel.org/lively4/lively4-babylonian-programming/src/babylonian-programming-editor/demos/classe
Timmy name: "Tim" hobby: "cycling"
2
Timmy this: Timmy
1
this.name: Timmy Tim testPerson divaD
3
Reverse Timmy this: Timmy
this.name: Reverse Timmy ~~Tim~~ miT testPerson ~~David~~ divaD
5
testPerson this: null
4
"David"
6
aPerson: testPerson {name: ~~David~~ divaD, hobby: debugging}
7
Ready

```
// You can also define examples for class methods
export default class Person {

  constructor(name, hobby) {
    this.name = name
    this.hobby = hobby;
  }

  sayHello() {
    console.log(`I'm ${this.name} and I like ${this.hobby}`);
  }

  reverseName() {
    this.name = this.name.split("").reverse().join("");
  }
}

function testPerson() {
  let name = ~~prompt("Enter a name")~~;
  let aPerson = new Person(name, "debugging");
  aPerson.reverseName();
  aPerson.sayHello();
}
```

Figure 1: The editor implementing our tool design, showing a local example with the name "Timmy" (1), probes (3, 5) showing results from two different examples (1, 4), a general replacement that replaces the request for user action with a fixed value (6), and a probe showing changes on a complex object (7). [7]

modules instead of their unmodified counterparts. For every execution of a module, the system loaded these instrumented versions of modules in addition to the already present base version of the module. While the implementation is sufficient to enable the intended live programming experience, the underlying approach does not result in a clear architecture.

As the instrumentation of source code depending on the activated example is a system-wide behavioral variation, we propose to use context-oriented programming (COP) as an underlying architecture. COP provides a mechanism to structure behavioral variations for one concern as a layer. This layer can then be activated for the system and the modified behavior is active. Additionally, the activation can be limited, for example, to a dynamic scope.

In this paper, we illustrate how COP can capture this system-wide instrumentation in a modular way, resulting in an architecture describing only the essential parts of the system. Thereby, we describe the implementation of the instrumentation for an example-based live programming tool as a use case for COP in general. Further, the instrumentation of source code also provides a use case for means to express expression-level behavioral variations. We illustrate the fit between the requirements for implementing example-based live programming and the features of COP, by mapping the features to COP concepts. Further, to demonstrate the simplicity of the implementation resulting from our approach, we describe the details of our implementation in Squeak/Smalltalk [3] using ContextS2 (a successor of ContextS) [9].

Structure of this Paper

The following section 2 introduces the features of Babylonian Programming and the resulting needs for dynamic behavior variation. In section 3, we describe how we serve these needs using concepts from COP. We go on to explain our implementation in section 4. Based on the approach and our implementation, we discuss in section 5 differences to the previous implementation and potential future work for expression-level behavior instrumentation. Section 6 concludes the paper.

2 USE CASE AND RESULTING REQUIREMENTS FOR COP

In order to illustrate the need for COP in example-based programming tool support, we will describe the design of the Babylonian Programming editor [7]. In the course of this description we will point out whether and how each design element implies dynamic behavioral variations.

The purpose of the tool is to provide live feedback through displaying intermediate run-time state of the execution of an example (see Figure 1). The displayed run-time state is updated whenever the program or the example is changed. As the sole purpose of the tool and the corresponding additions to the source code is to provide better feedback to programmers, using the tool should not alter the behavior of the system under development.

The *examples* are explicitly specified by programmers. An example is an invocation of a callable element of the language, for example a function or a method. Consequently, an example provides all information necessary to invoke the function or method. In particular, these are the arguments and, in case of a method, a receiver. An example can have a name which describes the situation the example captures, for example "the ideal case" or "malformed search string". The example is associated with the function or method through a widget which is similar to an annotation. Finally, an example can be activated which allows users to see intermediate run-time state of the running example in the editor.

In order to specify which part of the run-time state should be displayed at which point in time, programmers can add *probes* to expressions in code [5]. A probe traces values of the selected expression during the execution of an example. The corresponding UI widget displays the value directly within the editor. As the example-based tooling should not affect the actual system behavior, probes should only be present during the execution of examples.

Programs being edited with the example-based editing tool might contain side effects, calls of expensive functions, or functions which require user interactions. In order to be able to still use examples in such a program, programmers can *replace* these calls with other expressions. This is similar to the way mock objects are used in unit testing [6]. The original Babylonian Editor design only supported global replacements, which applied to all examples. However, the replacement expression might differ between examples. Thus, in the implementation described in this paper, the example-based editor will allow programmers to specify general and example-specific replacements. These example-specific replacements require that requirements can be scoped to the execution of an example.

Another feature of example-based programming tools is to allow programmers to keep track of assumptions. This allows programmers to quickly check whether their hypotheses about the program behavior holds or where it is violated. *Assertions* are probes with an additional expression which is used to check the traced values. When an assertion is violated, the program is not interrupted but the violation is recorded and the violation is displayed to the user. Again, the original Babylonian Editor did not support this, but the implementation described in this paper will support them. As programmers might want to make very specific assertions, such as the actual return value of a method call, assertions can be made example-specific. This in turn means, as it did for replacements, that assertions should be scoped to the example they refer to.

3 APPROACH: COP FOR BABYLONIAN PROGRAMMING

Based on the described programming tool, we will now describe how the implementation of each design element can generally be mapped to COP concepts given an exploratory self-sustaining programming environment based on a class-based object-oriented programming language [2].

The instrumentation of source code for tracing the example execution introduces *behavioral variations* on two levels into the system (see Figure 2): method-level, expression-level.

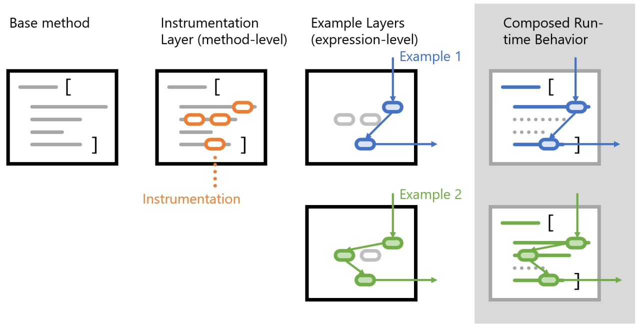

Base method
Instrumentation Layer (method-level)
Example Layers (expression-level)
Composed Run-time Behavior
Example 1
Instrumentation
Example 2

Figure 2: An overview of our mapping of the required behavior variations for instrumenting the source code onto layers.

At the method-level, each method that contains a probe, a replacement, or an assertion has to behave differently whenever it is activated during the execution of an example. As this affects a complete method, this variation can be expressed through a method which replaces the complete source code of the original method with the instrumented source code.

At the expression-level, instrumented parts of a method, for example an expression wrapped in an example-specific replacement, has to behave differently depending on the particular example being executed right now. We specify the behavioral variations on the expression-level through annotations embedded into the code

Analogous to the two granularities of behavioral variations, there are also two kinds of layers: one layer for all general instrumentations and one layer for each example. First of all, there is one layer containing all probes added anywhere in the system and all general replacements and assertions. These instrumentations form a consistent behavioral variation, as, for example, a method without any examples or probes might still contain a general replacement which has to be activated for all examples. Additionally, there is one layer for each example. This layer contains the replacements and assertions related to that example.

When executing an example, the fact that we are executing an example, and the specific example make up the context of this execution. We compose the general instrumentation layer and the layer for the example based on this context information. The example is executed in its own green thread. As we plan to implement this in the context of an exploratory live programming environment, the system under development might be running in parallel. In order to keep the behavior of the system under development unaffected from the activation of the instrumented version of the code base, we use dynamic scopes to limit the activation of the layer composition to the execution of the example.

4 IMPLEMENTATION

We have implemented the described approach in Squeak/Smalltalk [3] using ContextS2 [9]. The implementation consists of three parts: an extended compiler to enable the description of expression-level behavioral variations, the mechanisms to activate and scope the behavioral variations, and the tracing infrastructure. We will start by explaining the way the examples and annotations are integrated into the Smalltalk source code.

```
Number>>addAndLog: aNumber
  <exampleNamed: 'small number' self: #one given: 20>
  <exampleNamed: 'large number' self: #one given: 94280292019202>
  | result |
  result := "<bpProbe>"self + aNumber"</bpProbe>".
  "<bpReplace with: [MockTranscript value showln: result] for: 'Number>>#incrementAndLog>>#withMockLogging'>"
  "<bpReplace with: nil>"
    Transcript showln: result
  "</bpReplace>"
  "</bpReplace>".
  ↑"<bpAssert for: 'small number' that: [result = 21]>"result"</bpAssert>"

Number class>>#one
  ↑1
```

Listing 1: An annotated example method. The method contains two examples defined in pragmas (which are static annotations): "small number" and "large number". The method calculates the sum of the receiver and the argument, prints the result on the Transcript (a kind of log), and returns the calculated sum. The method contains a probe, a general replacement, a replacement for an example from another method, and an example-specific assertion.

```
BPSmalltalk <: OhmSmalltalk {
  Expression := AnnotatedExpression | NormalExpression
  NormalExpression = Operand (MessageChain CascadedMessage*)?
  AnnotatedExpression = expressionAnnotationStart Expression expressionAnnotationEnd
  AnnotationInformation = Operand KeywordMessage?
  comment := ~"\"<" "\"" (~"\"" any)* "\""
  expressionAnnotationStart = "\"<" space* ~"/" space* AnnotationInformation space* ">\""
  expressionAnnotationEnd = "\"</" space* Operand space* ">\""
}
```

Listing 2: The modified Smalltalk grammar in Ohm syntax [10]. The BPSmalltalk grammar inherits the OhmSmalltalk grammar and overrides the Expression rule of the OhmSmalltalk grammar. The NormalExpression rule is copied from the OhmSmalltalk grammar and the ANnotatedExpression rule accepts the new annotations.

```
Number>>addAndLog: aNumber
  <layer: #bpInstrumented>
  <exampleNamed: 'small number' self: #one given: 20 >
  <exampleNamed: 'large number' self: #one given: 94280292019202 >
  | result |
  result := (self bpTrace: (self + aNumber) forProbe: 1 inContext: thisContext).
  (self
    bpReplace: [(self
      bpReplace: [Transcript showln: result]
      with: nil)]
    with: [MockTranscript value showln: result] for: 'Number>>#incrementAndLog>>#withMockLogging').
  ↑(self bpAssertAfter: [result] that: [result = 21]
      forAssertion: 2 given: 'small number' inContext: thisContext)
```

Listing 3: The method source code after the rewriting added the instrumentation calls (square brackets denote block closures). The pragma in the first line after the method selector denotes that this is the partial method for `addAndLog:` in the Babylonian programming instrumentation layer.

4.1 Method Format

In the final editor, programmers should be able to interact with examples and code instrumentations through graphical widgets embedded in source code (see Figure 1). As these are not relevant for the implementation using COP, we will focus on the transport representation of the annotated source code (for an example method see Listing 1).

Examples are expressed in Smalltalk pragmas (given in angle brackets in Listing 1). The symbol passed as the value to the keyword `self:` is the selector of the class method which creates the receiver for the example execution. The subsequent keywords specify the arguments for the parameters of the method.

To enable programmers to express expression-level behavior variation, we had to provide a mechanism to annotate sections of source code. We decided for an embedded markup approach instead of a stand-off markup approach, as this would allow us to keep on using the existing Squeak/Smalltalk version control tool set. In order to keep the source code downwards compatible, so it can be loaded in a system lacking our extensions, we decided to describe the annotations in comments. We currently provide three annotations: `bpProbe`, `bpReplace`, `bpAssert` (see Listing 1).

4.2 Compiler Extension

Before we are able to provide programmers with feedback on the examples, we have to translate the annotated source code to an instrumented executable form. To introduce as little modifications to the Squeak/Smalltalk base system as possible, we have implemented our approach through rewriting the source code. The rewritten source code is then sent to the standard Squeak/Smalltalk compiler. Compiling an instrumented method results in two methods being compiled: a base method which does not contain any trace of the instrumentation in the resulting byte-code, and a partial method which contains the instrumented behavior.

We have implemented the parsing of the annotated source code through a specialized Smalltalk grammar in Ohm/S (see Listing 2). To keep the syntax of the annotations familiar to Smalltalk programmers, we decided to allow programmers to specify them as Smalltalk keyword messages. The grammar allows for nested annotations but not for overlapping ones.

After parsing the source code, we re-write the code to a version including instrumentation hooks which can be activated depending on the context (see Listing 3). The major part of the rewriting happens by generating new code for the nodes corresponding to annotated expressions as defined in the grammar (see Listing 4). During the rewriting, we also add a pragma which denotes that the method belongs to the ContextS2 layer named `bpInstrumented`.

4.3 Activation and Scoping Mechanism

Paralleling the ContextS2 scoping protocol, we implemented a method on the class `BlockClosure` which scopes the activation of the instrumented method behavior to the execution of that block (see Listing 5). The method takes an example as an argument and creates a `Tracer` instance for that example. The method then activates the ContextS2 layer `bpInstrumented` and sets the dynamic variable[1] `BPActiveTracer` to the newly created tracer during the execution of the block to be traced. The scoping of example-specific instrumentations is implemented in the tracer by checking for each instrumentation call whether it applies to the current example (as illustrated for replacements in Listing 6). We deactivate the `bpInstrumented` layer for the execution of the `BPActiveTracer` methods, as this enables us to use the instrumentation on its own implementation.

5 DISCUSSION

Based on our experience with using COP for implementing the instrumentation, we will discuss whether COP is a fit for the behavior we wanted to implement, how our approach compares to an approach using global re-translation, and how expression-level behavioral variations imply tool support.

5.1 COP for Example-based Live Programming

In general, our implementation shows that COP fits the requirements for an implementation of the tracing infrastructure of an example-based live programming tool. By using COP, we reuse a well-understood mechanism for implementing the behavior adaptation. Consequently, we separate the infrastructure for managing the behavior adaptation from the instrumentation implementation, at least at the method level.

At the expression-level, we can not reuse the existing ContextS2 infrastructure, as it only allows the replacement of full methods in layers. However, by implementing the expression-level behavior adaptation using dynamic variables, the implementation remains compact (see Listing 5 and Listing 6).

5.2 Comparison to Global Re-translation Approach

Another way of implementing the instrumentation of source code would be to do a global re-translation. This approach is used by the original Babylonian Programming implementation in Javascript [7]. All modules including any instrumentation or examples are re-translated and stored under a prefixed name. In a second step, the import statements of the translated modules are re-written to import the adapted version of modules if they are available. The example invocations then activate the function or methods from the adapted modules.

One major drawback of this approach is that indirect references to instrumented modules can not be found. If there is an import sequence from some adapted module A to some adapted module C via an unmodified module B, the approach would not load the adapted module C. The reason for this is that the algorithm does not rewrite in module A the import for module B, as there is no adapted version. This could be mitigated by a full re-translation of the system which is not feasible, as this would not allow for short feedback loops. Alternatively, this would require a static analysis of the dependency chains which would then be used to determine all modules whose module imports would need to be adapted. Besides the increased complexity of the mechanism, this would also require adapting and re-loading even more modules.

In comparison, our COP-based approach is less complex and does not require additional logic to work for any module in the system. However, as we introduce additional dispatch logic to all instrumented methods, we introduce a runtime penalty in the base system. The re-translation approach avoids this by not modifying the base system behavior at all.

5.3 Tool Support for Expressing Expression-Level Behavior Variation

The implementation of our approach illustrates that expression-level behavioral variations require additional tool support to make them usable. COP implementations working at the method-level can separate the source code of a partial method from the source code of the base method by defining the partial method as a new method. The source code of the base method is thus not affected by the related partial methods. In contrast, expression-level behavioral variations have to be associated with expressions within the source code in one of two ways: as stand-off markup or as embedded markup. Both approaches require some degree of tool support.

When using stand-off markup, the annotations are not stored as part of the source code itself but in some additional data structure.

[1] The dynamic variable can have different values depending on the dynamic scope. [4]

```
BPSourceRewriter>>AnnotatedExpression: aNode startTag: startTag actualExpression: expression endTag: endTag
  | annotation |
  annotation := self value: startTag.
  ↑self
      perform: ('{1}:with:' format: {annotation tag}) asSymbol
      withArguments: {annotation . self value: expression}

BPSourceRewriter>>bpReplace: annotation with: originalExpressionSource
  ↑(annotation includesKey: #for)
      ifTrue: [
        '(self bpReplace: [{1}] with: {2} for: {3})' format: {
          originalExpressionSource . annotation at: #with . annotation at: #for}]
      ifFalse: [
        '(self bpReplace: [{1}] with: {2})' format: {
          originalExpressionSource . annotation at: #with}]
```

Listing 4: The methods implementing the rewriting of the source code. The upper method dispatches to the individual rewriting methods for each kind of annotation. The lower method describes the rewriting for replacements in particular.

```
BlockClosure>>bpTraceForExample: anExample
  | tracer |
  tracer := BPTracer forExample: anExample.
  #bpInstrumented withLayerDo: [
    BPActiveTracer value: tracer during: self].
  ↑tracer trace
```

Listing 5: The method setting the scope for the tracing instrumentation. The call `withLayerDo:` activates the ContextS2 layer and the call `value:during:` sets the tracer during the execution of the block closure.

```
Object>>bpReplace: aBlock with: anObject for: exampleName
    ↑#bpInstrumented withoutLayerDo: [
        BPActiveTracer value replace: aBlock with: anObject for: exampleName]

BPTracer>>replace: originalCode with: replacementCode for: exampleName
    ↑self example exampleName = exampleName
        ifTrue: replacementCode
        ifFalse: originalCode
```

Listing 6: The methods implementing the activation of example-specific replacements.

The code editing tools have to be aware of the annotations in order to take care of displaying them to the programmers.

When using embedded markup, the annotations of expressions interrupt the original source code. Contrary to the original goal of modularity, this re-introduces concerns different from the main concern of the enclosing module. In the end, the embedded annotations might worsen the comprehensibility of the original source code. To mitigate this, tool support is needed to display the annotations in an unobtrusive way, or remove them altogether on demand.

6 CONCLUSION

Babylonian programming requires the instrumentation of source code at expression-level to provide immediate feedback on intermediate run-time state. The instrumentation is only relevant for the execution of examples, and furthermore some instrumentations are only relevant for certain examples. We used context-oriented programming as a mechanism to implement the instrumentation for Babylonian Programming without affecting the base system. The implementation expresses general instrumentation through layered methods and example-specific instrumentations through layered expressions. The resulting architecture remains concise and might serve as a blueprint for other example-based live programming implementations.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of HPI's Research School[2] and the Hasso Plattner Design Thinking Research Program.[3]

REFERENCES

[1] Jonathan Edwards. 2004. Example Centric Programming. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA) 2004.* ACM, New York, NY, USA, 124–124. https://doi.org/10.1145/1028664.1028713

[2] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *The Journal of Object Technology* 7, 3 (2008), 125–125. https://doi.org/10.5381/jot.2008.7.3.a4

[3] Daniel Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of OOPSLA 1997*, Vol. 32. ACM, 318–326. https://doi.org/10.1145/263698.263754

[4] Martin Löwis, Marcus Denker, and Oscar Nierstrasz. 2007. Context-Oriented Programming: Beyond Layers. *Proceedings of the 2007 international conference*

[2]www.hpi.uni-potsdam.de/research_school
[3]www.hpi.de/en/research/design-thinking-research-program

on Dynamic languages in conjunction with the 15th International Smalltalk Joint Conference (ICDL 2007) (2007), 143–156. https://doi.org/10.1145/1352678.1352688

[5] Sean McDirmid. 2013. Usable Live Programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!) 2013*. ACM, New York, NY, USA, 53–62. https://doi.org/10.1145/2509578.2509585

[6] Gerard Meszaros. 2007. *xUnit test patterns: Refactoring test code.* Pearson Education.

[7] 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. *The Art, Science, and Engineering of Programming* 3, 3 (2019). https://doi.org/10.22152/programming-journal.org/2019/3/9

[8] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (2018). https://doi.org/10.22152/programming-journal.org/2019/3/1

[9] Marcel Taeumel, Tim Felgentreff, and Robert Hirschfeld. 2014. Applying Data-Driven Tool Development to Context-Oriented Languages. *Proceedings of 6th International Workshop on Context-Oriented Programming (COP 2014)* (2014). https://doi.org/10.1145/2637066.2637067

[10] Alessandro Warth, Patrick Dubroy, and Tony Garnock-Jones. 2016. Modular Semantic Actions. *Proceedings of the 12th Symposium on Dynamic Languages - DLS 2016* (2016). https://doi.org/10.1145/2989225.2989231