Applying Data-driven Tool Development to Context-oriented Languages

Marcel Taeumel, Tim Felgentreff, and Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam, Germany
{first.last}@hpi.uni-potsdam.de

ABSTRACT

There are numerous implementations of context-oriented programming on host languages that come with graphical programming environments. However, comprehensive tool support is often missing because building and integrating graphical tools is still laborious; many programmers cannot afford to be both tool user and tool builder. We present a novel, data-driven approach on programming tools to alleviate this problem. We implemented a framework in Squeak/Smalltalk and show how programmers can use it to create and adapt integrated tools for ContextS2.

1. INTRODUCTION

Building graphical programing tools is a challenging endeavor. Even simple tools require much code to be written because frameworks such as Eclipse/RCP[1] or Qt/UI[2] impose verbose patterns. Especially the code required for graphical, interactive widgets overshadows a tool's purpose, which is often primarily about accessing domain-specific data. Consequently, two different roles emerge: the role of a tool *builder* and the role of a *user*. This is problematic when deficiencies are discovered during tool usage; adaptation will be impeded if programmers are not versatile enough to take on both roles. We argue that this affects the practicability of new programming language extensions such as context-oriented programming (COP). For COP implementations, there is usually only basic tool support such as new code browsers. Programmers who want to use COP effectively in their projects do arguably not bother adapting tools but primarily writing COP code.

There is not only one strategy to design and implement COP for a programming language along with tool support. Indeed, for many languages where COP is available, at least two different implementations exist [1]. Chances are that even basic tools, once created with an effort, cannot be reused across different COP implementations. Even for shared host languages, the extension's meta-model may differ and hence existing tools have to be adapted. For an example, ContextS uses classes [9] to represent layers, and its successor ContextS2 uses symbols (see Appendix A). Here, the programmer who implements COP is not necessarily a tool builder, but only a user of similar COP tools. To provide basic tool support for the new implementation, the programmer has to learn the tool framework and become a tool builder, which may involve a steep learning curve.

When, eventually, one or more tools for a specific COP implementation exist, integration with the underlying programming environment and its other tools is important. COP is a complementary extension for object-oriented languages and thus COP tools should be complementary, too. For example, having a code browser that shows layers and partial methods often does not affect other tools such as the debugger or the test runner. Without proper tool support, programmers have to fall-back on tools that are not adapted for COP. In such scenarios, the language extension can make rather simple tasks such as debugging more time-consuming and error-prone.

Thus, we see a need for better tool building support, especially for diverse language extensions such as COP. We address the following research question:

> How can we support programmers to build integrated graphical programming tools for language extensions such as COP?

In this paper, we present a novel, data-driven perspective on graphical programming tools and a corresponding framework in Squeak/Smalltalk[3] that supports an iterative tool building process with low effort, that is, few lines of code and immediate feedback. We apply our framework to create integrated tools for ContextS2, a COP implementation for Squeak/Smalltalk. Our framework also works with non-COP applications; our data-driven approach is, arguably, applicable to other object-oriented language environments.

In the following Section 2, we describe the challenges that arise when building and integrating tools for COP. In Section 3, we present our approach and the resulting tools for ContextS2. Finally, we conclude in Section 4.

2. GOALS AND CHALLENGES

Programming tools support *exploration* and *modification* of software artifacts such as classes, tests, traces, or domain-specific objects. In traditional Smalltalk environments, such

[1]Eclipse Rich Client Platform, `http://eclipse.org`
[2]Qt User Interfaces, `http://qt-project.org`
[3]`http://www.squeak.org`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
COP'14, July 28 - August 01, Uppsala, Sweden
Copyright is held by the owner/authors(s). Publication rights licensed to ACM.
ACM 978-1-4503-2861-6/14/07 ...$15.00.
http://dx.doi.org/10.1145/2637066.2637067

tools include class browsers, test runners, debuggers, and object inspectors. Exploration means continuously posing and answering questions about code structure and run-time behavior. For example, "Which code has class C?" or "Which state has object O?". Modification means altering particular data, that is, source code or run-time state.

Many COP implementations reuse constructs of the host language such as classes and methods. Therefore, tools to manipulate these artifacts can easily be reused to create COP applications. However, challenges arise when new kinds of questions require adapted *exploration tools* with additional views into the system. Such additional relationships between artifacts are often not supported out of the box. For COP, tools have to understand the concept of layers to relate them to other artifacts, even if layers are classes themselves. Debuggers, for example, can look very different, even cryptic, if they reveal details about COP-specific method dispatch in the call stack.

2.1 Questions for Exploration Tools

We think that questions about the static code structure, which should be answered with COP tools, include:

- Q1 Which layers refine class C (or method M)?
- Q2 Which classes (or methods) are refined by layer L?
- Q3 In which methods can layer L be activated?
- Q4 In which methods can layer L be deactivated?

In addition, the concrete run-time behavior as influenced by layers should be observable in COP tools. We think that tools should answer the following questions, too:

- Q5 Which layers are currently active in process P?
- Q6 What is the current interface for object O considering active layers?

2.2 Building Exploration Tools

In an object-oriented and class-based programming language, building exploration tools basically means to create *adapters* that connect interfaces of *software artifacts* with interfaces of *graphical widgets*. Tools being interface adapters accomplish the following tasks: 1) choose and setup widgets, 2) choose and access artifacts, 3) react to artifact changes, 4) react to user input through widgets.

Tool architectures range from monolithic to composite designs, often influenced by a tool framework. Monolithic tools, on the one end, may coalesce all features in very few classes, which decreases cohesion per class and impedes code comprehension. For example, Squeak 4.5 employs only five classes to implement code browser, debugger, and object inspector with about 4500 lines of Smalltalk code.[4] Composite designs, on the other end, may specify custom classes even for single artifacts, which increases coupling per class and impedes code comprehension, too. For example, the OmniBrowser [4] framework wraps concrete artifacts in abstract nodes and the Glamour [6] framework wraps concrete widgets in abstract presentations.

Tools are built iteratively and adaptations should take place whenever deficiencies are discovered. That is, however, of low interest for the tool user if the corresponding source code cannot be localized and modified with low effort. Runtime components, especially the graphical ones, often have no direct connection to the underlying, data-specific code. One approach to mitigate the problem of complex tool designs is to support declarative initialization scripts. Glamour, for example, supports describing widget layout, artifact access, and basic browsing interactions in one script.

We argue that the most volatile part of source code in exploration tools deals with accessing software artifacts. Often, such tools consist of several adjacent list-based widgets, which propagate user selections to exploit relationships between artifacts. Programmers should not bother reading or skimming code that (re-)implements these recurring concepts. Therefore, we propose a *data-driven* approach where programmers can focus on the underlying software artifacts. They can easily localize and modify code that deals with reading and preparing artifacts such as classes, methods, layers, and partial methods.

2.3 Reusing Exploration Tools

One key decision of a COP implementation is its meta-model: The internal representation of layers and partial methods have an impact on the applicability of existing tool support. This means that tools can often not be reused across different implementations. Thus even if the host language is shared, existing exploration tools will not be functional for the new artifacts. There are many languages with multiple COP implementations:

Java	ContextJ [2], ContextJ* [10], JCop [3]
JavaScript	ContextJS [13], COP.js[5]
Lisp	ContextL [7], Ambiance [8], Lambic [17]
Python	ContextPy [11], PyContext [18]
Ruby	ContextR [16], Phenomenal Gem [15]
Smalltalk	ContextS [9], ContextS2 (see Appendix A)

This illustrates that it is not obvious how to best implement COP for a programming language. To demonstrate the differences in the meta-model, we consider question Q1: "Which layers refine class C?". In ContextS, we can search all subclasses `CsComposition` for partial methods whose selectors conform to advice class `C` such as `#advicePrintOn`:

```
CsComposition allSubclasses select: [:class |
  class methodDict keys anySatisfy: [:selector |
    selector beginsWith: #advice, C name]].
```

In contrast, ContextS2 stores partial methods in method dictionaries of layered classes sharing the base method's selector. The same question has to be answered by searching directly through the method dictionary of the class and filtering by method type:

```
(C methodDict values
  select: [:method | method isContextSMethod])
  gather: [:method | method partialMethods
    gather: [:partial | partial layers].
```

Note, that the different interfaces reflect the different meta-models of the implementations, even though in this case, about the same amount of code is required. The ContextS2 implementers could port these changes if they were able to localize the appropriate source code locations in the ContextS tools. Such tools, however, are often just replicas of regular tools with some COP-specific features. For example, the COP browser is a regular code browser with an additional view to distinguish layers. Such modified replicas will

[4] The reader can complement an understanding of Smalltalk's expressiveness by studying comparative discussions such as `http://c2.com/cgi/wiki?JavaVsSmalltalk`

[5] `http://colmarius.net/cop`

get outdated if not maintained with the base system; so they will arguably get outdated. Consequently, for a new COP implementation, programmers are likely to start *again* from the current version of the base system's tools and figure out extension points *again* instead of reusing and adapting existing COP tools.

2.4 Integrating Exploration Tools

Today's programming environments do not facilitate seamless integration of both data and graphics for new tools. Although visual integration is common, there is often no integration of new software artifacts and relationships. Here, visual integration means Eclipse perspectives, Squeak system windows, or Textmate buffer views. However, integration of artifacts and relationships requires using shared communication channels between tools. For example, Eclipse is built around the JDT Java Model, which is not easily extensible; new tools cannot simply introduce new artifacts to existing tools. Consequently, graphical tools are often self-contained and integrate only visually. We think that there has to be an equivalent of Unix' pipes-and-filters or Emacs' buffers-and-processes for graphical environments.

As an effect, programmers tend to spend much time switching between different tools [12]. They cannot directly work with their artifacts but have to manually transfer data and input between various intermediate tool front-ends. There is an interaction style that mitigates this switching problem: context menus. They pop up over graphical representations of software artifacts and support invoking other tools to further work with those artifacts. But to reduce the number of switches, tools tend to be feature-rich and standalone. This impedes integration between tools meaning they cannot be combined in an easy fashion to complete unforeseen programming tasks.

The research project CodeBubbles [5] shows that there can be a better metaphor for visual programming environments. The project focuses on the software artifacts and provides small, self-contained tools, called *bubbles*. Having very specific features, bubbles can be combined to work on different aspects of the same artifact. In such an environment, a new COP implementation would integrate with the programming environment by providing additional bubbles that represent layers, partial methods, and other artifacts.

3. DATA-DRIVEN TOOL BUILDING

In this section, we present our new approach to build all kinds of graphical exploration tools. We implemented a framework for Squeak/Smalltalk and apply our approach to build tools for ContextS2.

3.1 Our Data-driven Perspective

We see graphical programming tools as data processing pipelines whose intermediate results are displayed on screen. That is, software artifacts are repeatedly transformed when users navigate relationships to other artifacts. On screen, characteristic information is arranged in list-based widgets—or other visualizations—to reveal an appropriate degree of insight. The basic idea is illustrated in Figure 1.

By projecting this data-driven perspective onto programming tools, we support programmers to focus on their domain-specific software artifacts. In contrast to the state-of-the-art, our data-driven programming tools are not about learning and using distracting intermediate interfaces, but rather about working with the underlying data itself. For example, a user may reason about the rules of practice in an arbitrarily complex user interface in the following way:

> If I click *the name of my project* on the left-hand side, that tool magically shows me my project's involved layers on the right-hand side.

In our data-driven perspective, we anticipate thoughts that focus on artifacts like this:

> If I choose *my project* in the left-hand list, this very artifact will flow to the right-hand widget and there it is *transformed* to its layers, which will then be displayed.

Already, programmers are in control of choosing the artifacts of interest in list-based widgets to be processed by tools. Having our perspective, we can also make the rules of transforming software artifacts explicit and customizable because there are clear boundaries in the user interface. For example, Figure 2 applies our data-driven perspective to the programming environments Eclipse and Smalltalk. The whole environment consists of rectangular boxes that exchange software artifacts and transform them before displaying them. So, we can establish simple rules of modularity in the graphical interface language.

We implemented a framework that employs our data-driven perspective in Squeak/Smalltalk. We call the rectangular boxes in tools *panes*, which manage *queries* and represent

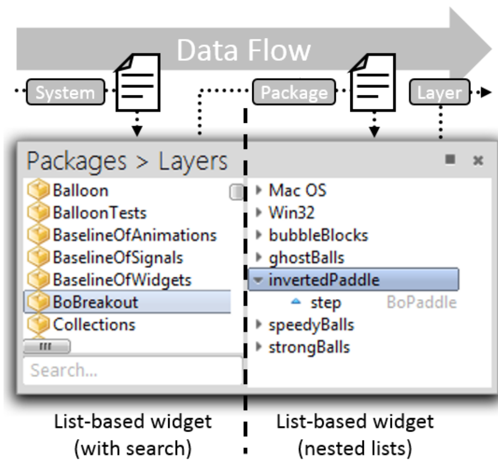


Figure 1: Our data-driven perspective where programming tools process scripts on software artifacts to exploit relationships.

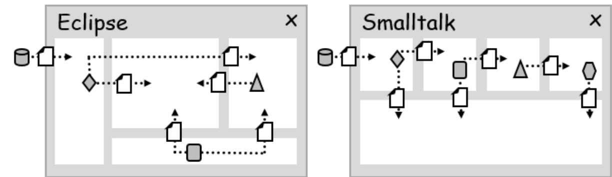


Figure 2: Our perspective applied to traditional programming environments. Software artifacts include projects, files, classes, and methods.

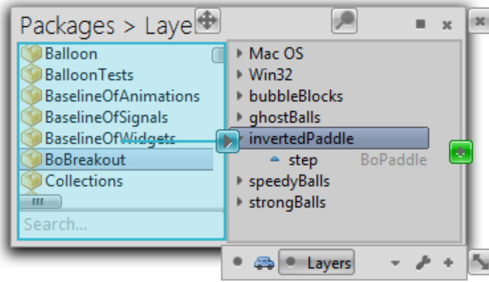


Figure 3: A halo above panes provide access to queries (bottom), incoming connections (left, blue), and outgoing connections (right, green).

complete tool definitions. In combination with an interactive halo[6] (Figure 3), panes provide a direct link from a tool's graphical appearance to the underlying artifact transformation code. Our framework has the following components:

Pane A rectangular placeholder in the user interface. Dispatches between a configurable set of *queries* when new objects arrive using type information. Exchanges objects with other *panes* via user-controlled selections in *views*.

Query Encapsulates source code to (1) transform objects and (2) extract properties such as labels or icons to be rendered by *views*. Depending on the view, properties can range from primitive types such as strings or bitmaps to more elaborate ones such as Morphs [14].

View Represents a widget or visualization of arbitrary complexity. It is embedded into one *pane* and can be dismissed by that pane on data-driven updates. Propagates user input such as selections back to the pane.

Interpreter Processes a query with some objects to produce an intermediate model structure, which should be understood by views.

Queries are stored in a dedicated database to persist modifications and support managing several versions across tool boundaries. Duplicating tools does not involve copying many lines of Smalltalk code but only information about pane geometry, inter-pane connections, and references to queries. Having this, a tool's footprint is quite small.

3.2 Tool Support for ContextS2

In our implementation, queries are lists of Smalltalk *block closures*. The interpreter processes such lists sequentially by evaluating blocks in order, and passing Smalltalk objects between them. There are two processing modes for each block: one-by-one or all-at-once. Having this, the interpreter provides no stream semantics but offers ways to process the particular object buffer, which resides between blocks. The one-by-one mode saves the corresponding amount of Smalltalk code for processing collections.

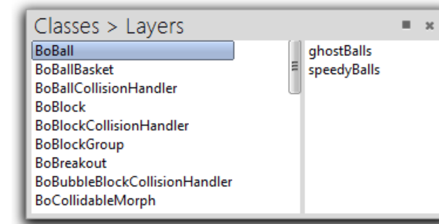
There are two kinds of blocks in a query: object transformation and property extraction. A query typically consists of some transformation blocks followed by some extraction blocks. Here is an example for transforming classes one-by-one[7] into their methods and extracting the selector of each method into the property `#text`. An `#icon` is provided, too:

```
[:class | class methodDict values].
[:method | {#icon -> (IconService for: method).
            #text -> method selector}].
```

Blocks can extract multiple named properties of an object but views have to know about those names to take advantage of it. We implemented several standard views such as lists, tables, and trees that can render, for example, `#text`, `#icon`, `#tooltip`, or `#color`. Additionally, our framework stores an object reference in the `#object` property.

We will now show *all the code* needed to build tools that answer the questions from the beginning. For each tool, a small screenshot should provide a first visual impression. Due to space constraints, we avoid listing code blocks for sorting objects. For each tool, there are two panes that exchange selected artifacts from the left to the right. The domain of our running example is a small Breakout game (see Appendix B), which was implemented with COP layers to control the scope of various in-game items.

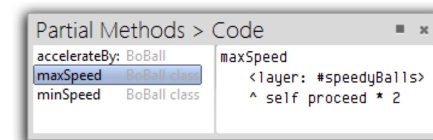
Q1: Which layers refine class C?



This tool can be opened for one or more *packages*. It lists all classes in this package and provides access to all involved layers. Each of the two panes has one query. Note the only slightly bigger footprint compared to the listing in Section 2.3 due to block syntax:

```
"Query for left classes list."
[:package | package classes].
[:class | {#text -> class name}].
"Query for right layers list."
[:class | (class methodDict values
  select: [:method | method isContextSMethod])
  gather: [:method | method partialMethods
    gather: [:partial | partial layers]]].
[:layer | {#text -> layer}].
```

Q2: Which methods are refined by layer L?



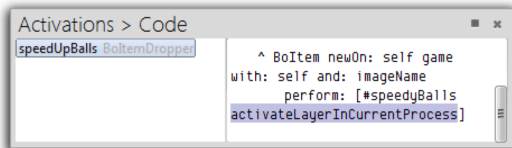
This tool can be opened for one or more *layers*. It lists all partial methods that affect those layers and provides access to the source code. Each of the two panes has one query:

[6]Halos provide overlay buttons and are invoked by a dedicated user interaction on graphical elements. The halo concept origins from the Morphic framework [14].

[7]There are several meta information stored with each query block such as processing mode, expected input type, view class, and whether it is transformation or extraction. Due to space constraints, we only show the blocks' sources.

```
"Query for left method list."
[:layer | Array streamContents: [:methods |
  SystemNavigation default
    allSelectorsAndMethodsDo: [:b :s :method |
      method isContextSMethod ifTrue: [
        method partialMethods do: [:pm |
          (pm layers includes: layer)
            ifTrue: [methods nextPut: pm]]]]]].
[:partial | {#text -> partial selector}].
[:partial | {#text -> partial methodClass}].
"Query for right code view."
[:partial | {#text -> partial method getSource}].
```

Q3: In which methods can layer L be activated?



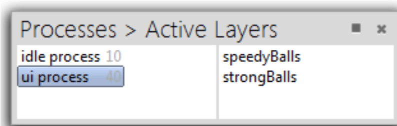
This tool can be opened for one or more *layers*. It lists all methods that directly activate layers by sending the appropriate messages. It also shows the code of the containing method. Each of the two panes has one query:

```
"Query for left method list."
[:layer | (#(withLayerDo: withLayersDo:
             activateLayersInCurrentProcess
             activateLayerInCurrentProcess)
  gather: [:message | SystemNavigation default
    allCallsOn: message])
  select: [:ref | ref compiledMethod
    literalStrings includes: layer]].
[:ref | {#text -> ref selector}].
[:ref | {#text -> ref actualClass}].
"Query for right code view."
[:ref | {#text -> ref sourceCode}].
```

Q4: In which methods can layer L be deactivated?

This tool is similar to the one for layer activations above but it looks for calls to `#deactivateLayerInCurrentProcess` and `#deactivateLayersInCurrentProcess`.

Q5: Which layers are currently active in process P?

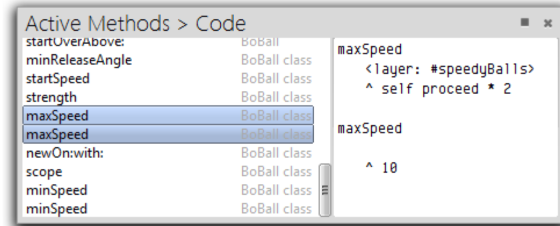


This tool can be opened for the *environment*. It lists all processes that can be scheduled and provides access to currently active layers. Meta programming has to be employed to access the waiting processes in the scheduler. Each of the two panes has one query:

```
"Query for left process list."
[((Processor instVarNamed: #quiescentProcessLists)
  gather: [:ll | ll asOrderedCollection])
  copyWith: Processor activeProcess].
[:process | {#text -> (process caseOf: {
  [Project uiProcess]
    -> ['ui␣process'].
  [Processor backgroundProcess]
    -> ['idle␣process']}
  otherwise: [process name "cryptic id"])}].
[:process | {#text -> process priority}].
```

```
"Query for right active layers list."
[:process | process csInfo].
[:csInfo | csInfo activeLayers].
[:layer | {#text -> layer}].
```

Q6: What is the current interface for object O considering active layers?



This tool can be opened for one or more *objects*. It lists all methods of the objects' class and filters them by active layers. It also shows the code of the method or partial method. Each of the two panes has one query:

```
"Query for the left method list."
[:object | object class].
[:class | class methodDict values,
           class class methodDict values].
[:method | method isContextSMethod
  ifFalse: [method]
  ifTrue: [method methodChainFor: Processor
    activeProcess csInfo activeLayers]].
[:method | {#text -> method selector}].
[:method | {#text -> method methodClass}].
"Query for the right code view."
[:method | {#text -> (method getSource,
                       String cr)}].
```

3.3 Data-driven Tool Integration

We illustrated how our framework supports to build tools for ContextS2 in a data-driven way. Each tool's data transformation and property extraction code, called query, is prominent, compact, and accessible via the graphical interface. This supports to create a toolset for various, if not all, programming tasks. But how do programmers integrate those tools, that is, switch from one tool to another?

A first possibility forms inter-tool pane connections. In our framework, programmers can establish dataflow visually between panes in the *same* window (Figure 3). Such dataflow can also be established between panes in *different* tool windows. For example, programmers can open a browser to list all layers for a selected class (Q1), and another one to list all partial methods for a layer (Q2). Dataflow between both tools can then be used to, effectively, browse all partial methods in a class via its layers. The concept of context menus is not needed.

A second possibility forms emergent tool construction via artifact collections. We provide a special view called *artifact list*. In this view, programmers can use drag-and-drop to collect software artifacts, not just their visual representations, from other views to evaluate queries with them as illustrated in Figure 4. The compact editors for each artifact resemble Code Bubbles' bubbles [5]. Once collected, programmers use the pane's halo to switch to another query or write a new one (Figure 3). Together with changing the view to an appropriate visualization, programmers can explore those collected artifacts. Having this, tools as such—being just

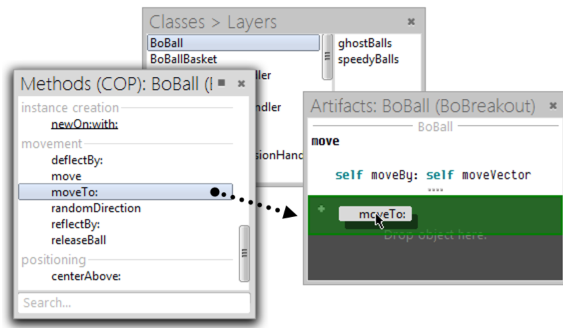


Figure 4: Programmers can collect artifacts across tools via drag-and-drop in a special artifacts view to evaluate queries on them.

windows with single panes—would have a shorter lifetime and queries would be more prominent. Even the concept of panes and views would fade into the background; it would all be about software artifacts and their transformations.

We aim for expressing *all* kinds of programming tools with queries. Regular code browsers are similar to the tools presented above and hence no challenge. Even more elaborate tools such as debuggers map to our data-driven perspective: the input object for a debugger is a suspended *process* object; the first query can extract the *stack* and put it into a list widget; the second query can show *code* for the current selection in the stack; the third query can reveal more context information such as the message *receiver*.

Being not restricted to the COP domain, we are still exploring the possibilities of our data-driven approach by rebuilding all kinds of programming tools.

4. CONCLUSION

Programming systems like Squeak/Smalltalk provide short edit-compile-run cycles, which support iterative application building. Programmers can modify pieces of source code and immediately observe changed behavior in running applications. Graphic frameworks such as Morphic leverage this idea for visual applications. Programmers can interactively explore and adapt graphical widgets and hence mold the desired user experience. With our data-driven perspective, we found an additional abstraction to further facilitate the idea of modifying applications in use and applied it to build tools for COP implementations and applications.

5. REFERENCES

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A Comparison of Context-oriented Programming Languages. In *Proceedings of the 1st International Workshop on Context-Oriented Programming*, page 6. ACM, 2009.

[2] M. Appeltauer, R. Hirschfeld, and H. Masuhara. Improving the Development of Context-dependent Java Applications with ContextJ. In *Proceedings of the 1st International Workshop on Context-Oriented Programming*, page 5. ACM, 2009.

[3] M. Appeltauer, R. Hirschfeld, H. Masuhara, M. Haupt, and K. Kawauchi. Event-specific Software Composition in Context-oriented Programming. In *Software Composition*, pages 50–65. Springer, 2010.

[4] A. Bergel, S. Ducasse, C. Putney, and R. Wuyts. Creating Sophisticated Development Tools with OmniBrowser. *Computer Languages, Systems & Structures*, 34(2):109–129, 2008.

[5] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola Jr. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 455–464. ACM, 2010.

[6] P. Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, 2009.

[7] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *Proceedings of the 2005 Symposium on Dynamic Languages*, pages 1–10. ACM, 2005.

[8] S. González, K. Mens, and A. Cádiz. Context-Oriented Programming with the Ambient Object System. *Journal of Universal Computer Science*, 14(20):3307–3332, 2008.

[9] R. Hirschfeld, P. Costanza, and M. Haupt. An Introduction to Context-oriented Programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, pages 396–407. Springer, 2008.

[10] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[11] R. Hirschfeld, M. Perscheid, C. Schubert, and M. Appeltauer. Dynamic Contract Layers. In *Proceedings of the 2010 Symposium on Applied Computing*, pages 2169–2175. ACM, 2010.

[12] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.

[13] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming*, 76(12):1194–1209, 2011.

[14] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *Proceedings of the 8th Symposium on User Interface and Software Technology*, pages 21–28. ACM, 1995.

[15] T. Poncelet. The Phenomenal Gem. Master's thesis, Louvain School of Engineering, 2012.

[16] G. Schmidt. ContextR & ContextWiki. Master's thesis, Hasso-Plattner-Institut, Potsdam, 2008.

[17] J. Vallejos, P. Costanza, T. Van Cutsem, W. De Meuter, and T. D'Hondt. Reconciling Generic Functions with Actors. In *ACM SIGPLAN International Lisp Conference, Cambridge, Massachusetts*, 2009.

[18] M. Von Löwis, M. Denker, and O. Nierstrasz. Context-oriented Programming: Beyond Layers. In *Proceedings of the 2007 International Conference on Dynamic languages*, pages 143–156. ACM, 2007.

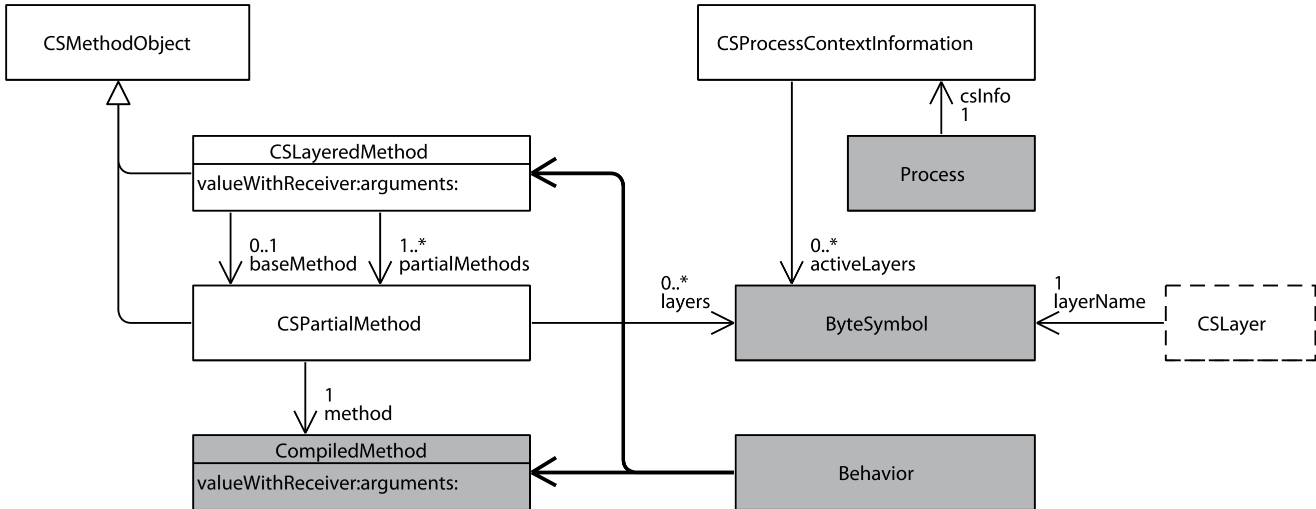


Figure 5: The architecture of ContextS2 (white) embedded into the Squeak environment (gray).

APPENDIX

In this appendix, we provide more information about ContextS2, a COP implementation for Squeak/Smalltalk. It was implemented for Squeak 4.5 and can be downloaded here: `www.hpi.de/hirschfeld/squeaksource/ContextSTrunk`

A. CONTEXTS2 ARCHITECTURE

Figure 5 shows the architecture of ContextS2. To illustrate integration into the base system, the classes of ContextS2 are in white and classes in the Squeak core are in gray.

The static core of ContextS2 uses a facility of the Squeak virtual machine (VM) where any object in the method dictionary of a `Behavior`, which is a supertype of `Class`, that is not a `CompiledMethod` is activated with the message `valueWithReceiver:arguments:`. In contrast to ContextS [9], where partial methods are encapsulated in a layer class, this allows ContextS2 to attach partial methods directly to the class' layers. When a layered version for a method is created, a `CSLayeredMethod` is inserted into the method dictionary. This object references both the original `baseMethod` and holds a list of `CSPartialMethod` objects.

At run-time, each `Process`, which is Squeak's notion of a thread, is created with `CSProcessContextInformation`. This object holds a reference to a list of symbols that name the currently active layers, and provides meta-level facilities to compute the order of layered method activations. When a `CSLayeredMethod` is activated, it uses the context information object in the current `Process` to calculate the order in which its `partialMethods` need to be activated. The outer-most `CSPartialMethod` is then activated through the `valueWithReceiver:arguments:` method. When a `proceed` call is made in a `CSPartialMethod`, the next partial method or the base method is activated. If no base method exists, that is a method only defined in layers, and no further partial method is active, a *cannot proceed* exception is raised.

Note, that the `CSLayer` class merely provides a meta-level interface for ContextS2 and that it does not affect the implementation of context-oriented programming itself. Its instances can be used to find all methods and partial methods belonging to a layer.

B. BREAKOUT

We used a Breakout game for our queries as shown in Figure 6. The player controls the *paddle* via the left and right keyboard buttons, and tries to guide the *ball* to hit and remove the *blocks*. When blocks are removed, they randomly drop *buffs*, which the player can collect with the paddle. These buffs have different effects such as enlarging the paddle, making the ball faster, or adding additional balls. These effects are implemented as COP layers that are activated for a given time frame, and later deactivated after the timeout. Additionally, layers are used to implement platform-specific keyboard handlers.

Figure 6: A Breakout game with relevant game items highlighted