Scoping Changes in Self-supporting Development Environments using Context-oriented Programming

Jens Lincke Robert Hirschfeld
Hasso-Plattner-Institut
Universität Potsdam, Germany
{firstname.surname}@hpi.uni-potsdam.de

ABSTRACT

Interactive development in self-supporting systems like Smalltalk or the Lively Kernel allows for an explorative and direct development workflow. Because of the immediate and direct feedback loops, changes to core behavior can lead to accidentally breaking the programming tools themselves. By separating the tools from the objects they work on, this fatal self referentiality can be avoided, but at the expense of interactive development. In this paper we show how context-oriented programming (COP) can be used to separate tools from the objects under development. Instead of directly modifying meta-structures, changes should go into layers on top of these structures. Since layers can be scoped at run-time, changes do not affect the programming tools. We demonstrate this approach by showing examples of adapting core behavior in our self-supporting development environment Webwerkstatt with ContextJS, our COP extension for JavaScript.

1. INTRODUCTION

Creating and evolving tools while they are used is common practice among programmers: Writing shell scripts or extending text editors to optimize the programmers workflow are the most common examples. Self-supporting development environments such as Smalltalk [6], Self [18], Emacs [17], Squeak [9], and Lively Kernel [10] are systems where developers can evolve their environment while they are using it. All of these environments keep the software development tools such as editors, debuggers, or code browser in the same environment as the objects and meta-objects they are working on. This allows for a direct development style with short feedback loops. A good motivation for such interactive programming styles gives Bret Victor in his talk on `Inventing on Principle` [19], where changes to the source code can immediately be observed in the running program side by side.

Having such tight interaction between tools and meta-structures such as classes can make the development of core behavior difficult if tools are changing the classes or methods they are depending on. Errors or debug statements in some core part of the system may break the whole development environment and force a restart. Developers can get used to such behavior, create workarounds, or become overly careful when changing core parts of the system.

In this paper we propose to use context-oriented programming (COP) as a technique to isolate changes to the core system during development. Instead of modifying classes directly at run-time, changes should be applied to layers, that can then be scoped to specific parts of the system. With these layers new features can be tried out. When they meet the developers' needs, they can be applied to other parts of the system or even activated globally.

The remainder of the paper is structured as follows: Section 2 discusses different approaches of separating the tools from the objects they work on. Section 3 describes how COP layers can be used to scope changes during development. Section 4 illustrates this approach with examples. Section 5 describes tools that can support such a process. Section 6 discusses related work and section 7 concludes.

2. SELF-SUPPORTING DEVELOPMENT ENVIRONMENTS

This section presents three alternative workflows in self-supporting development environments. We show the problems of insufficient separation between tools and objects and discuss two approaches that address these.

2.1 No Separation

Figure 1 shows a very abstract overview of self-supporting development environments [8] such as Squeak [9] and Lively Kernel [10]. The environment has to be bootstrapped at some point. The developer can then start creating and modifying meta-objects such as classes. Because this happens at run-time, developers will get feedback from the objects while they are interactively programming. The changes to the meta-objects are typically also persisted outside of the development environment in some kind of code repository.

Since tools, meta-objects, and objects are in the same environment, there are no extra levels of indirection: the tools can be simple and powerful. But at the same time having no levels of indirection can be problematic and even fatal if the developers want to work on objects such as core classes that directly or indirectly affect the tools themselves. The tools depend on those core classes. Introducing an error or having an intermediate broken state is fatal for the whole

COP'12, June 11, 2012, Beijing, China

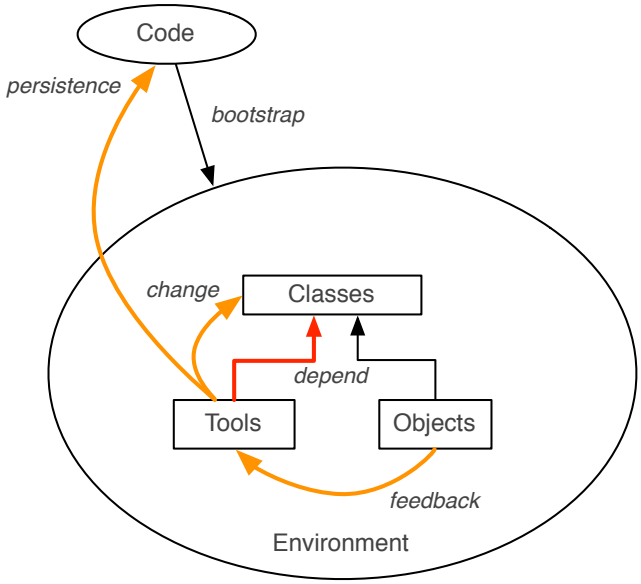

Figure 1: Abstract flow of changes and feedback in a self-supporting development environment

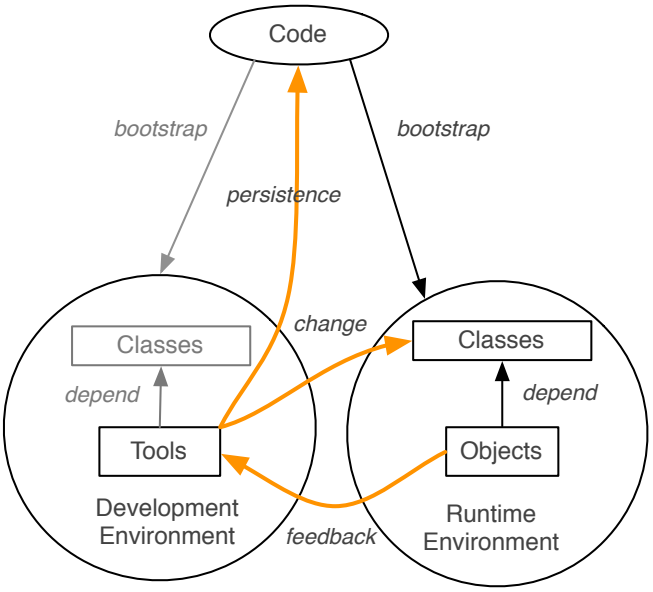

Figure 2: Separated tool environment

environment. That is why developers have to be careful with what they write, when they are modifying those shared objects.

2.2 Separate Runtime Environment

A general approach to separate the tools from the objects they work on is to separate the runtime environment of the tools from the runtime environment of the objects. Development Environments such as Eclipse [5] fit into such a scenario as depicted in Figure 2.

The development tools run in a separate environment and mainly work on static code. To interact with objects, the system under development has to be bootstrapped by external code. The tools can still interact with the objects under development, but only through interprocess communication, which is more complicated to implement than direct access to objects.

Having such a separation makes it hard to modify the tools on the go. That is how tools become products which are hard to shape during their usage.

2.3 Separate Meta-Objects

An other approach to allow tools and objects in the same environment is to use different sets of meta-objects as shown in Figure 3. The tools can have direct access to the objects and change even core parts of the system. Since they modify a different version of the meta-objects and not their own, the tools cannot be used to break themselves anymore just by accident. Systems like ChangeBoxes [4] can be used to develop in such a way. One limitation of it is that objects are coupled tightly to one version of their meta-class. This means that the scope of the behavioral variation cannot be extended to preexisting objects. New behavior is only available for objects instantiated afterwards.

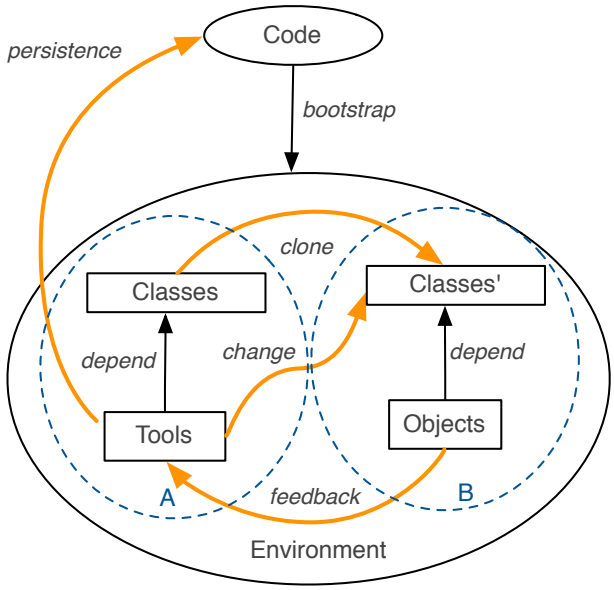

Figure 3: Separated Meta-structures and object sets

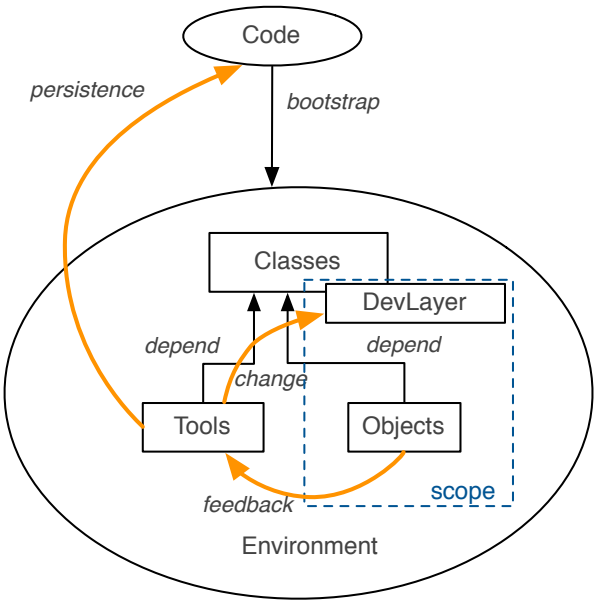

Figure 4: Development with Scoped Behavioral Adaptations

3. USING SCOPED BEHAVIORAL ADAPTATIONS FOR EVOLVING SELF-SUPPORTING DEVELOPMENT ENVIRONMENTS

In this section we show how to employ COP in the process of evolving self-supporting development environments as shown in Figure 4. The programmers don't modify the core classes and methods directly, but use COP layers, that can be scoped to only affect the behavior of the objects under construction.

3.1 ContextJS

In the following examples, we use ContextJS [12, 13, 11], our library-based COP extension to JavaScript. ContextJS is a JavaScript library and implemented with method wrappers [2]. It allows defining behavioral variations to objects and scoping them in various ways.

3.2 Separating Changes in Layers

In the following example, an anonymous object with the property `n` and a function `count` is created and assigned to the global variable `EventCounter`:

```
EventCounter = {
  n: 0,
  count: function(evt) {
    this.n = this.n + 1;
  }
}
```

If this `EventCounter` is used by tools in a self-supporting development environment, changing it could be dangerous. Objects and behavior can be changed at any time in JavaScript. If developers are interested which events are counted, they could add an `alert` statement into the count method. The alert statement can be used to display values to the user:

```
EventCounter.count = function(evt) {
  alert("evt: " + evt);
  this.n = this.n + 1;
}
```

But this might instantly bring the system to a halt, if it is used by the tools because of the potentially many alerts.

By using COP and defining the new behavior in a layer, the problem can be circumvented as follows:

```
cop.create("DevLayer").refineObject(EventCounter, {
  count: function(evt) {
    alert("evt: " + evt);
    this.n = this.n + 1;
  }
})
```

The layer hides the original behavior and without proceeding to the base behavior. Since we do not add a new crosscutting concern here, but work on the base method source, there is not need to proceed to other partial methods of the layers or the base system.

3.3 Scoping Changes - Layer Activation

To actually affect the system, the `DevLayer`[1] can then be scoped in various ways as needed. The standard mechanism in COP is the dynamical scoping:

```
cop.withLayers([DevLayer], function() {
  // a call of EventCounter.count() in this scope
  // will execute the new behavior in DevLayer
})
```

In interactive environments, most control flows are triggered by events. Other scoping mechanisms such object specific and structural scoping [12] are better suited to activate the new behavior. Figures 6 and 7 demonstrate such layer activations.

If the developers are satisfied with their new behavior, they might want to apply it to the whole environment. They can do so by activating the layer globally. If they discover an error they can still deactivate the layer and fall back to the old behavior.

```
DevLayer.beGlobal();
// -> activate layer for all objects, even the tools

DevLayer.beNotGlobal();
// -> deactivate layer if necessary
```

3.4 Merging Changes back into the Base System

When the developers have tested their new behavior with the whole environment, they might decide to merge their changes back into the base system. We have added some filtering to our example, that should be integrated into the base system:

[1]cop.create creates the layer and assigns it to a global variable

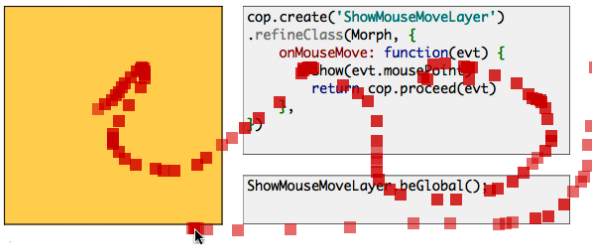

Figure 5: A layer used during the development, that visualizes move events of the mouse. The layer is globally activated. When the mouse is moved over any object including the workspace with the code, the red rectangles are shown.

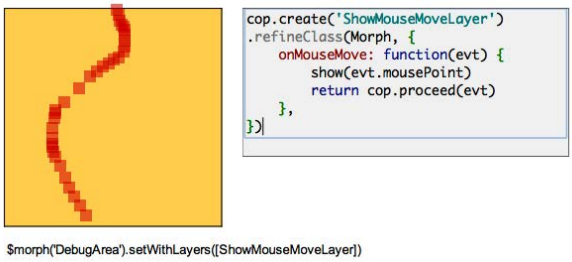

Figure 6: The ShowMouseMoveLayer which is defined in the right workspace is only activated for the yellow rectangle on the left. The layer is structurally scoped by activating the layer for the DebugArea.

```
cop.create("DevLayer").refineObject(EventCounter, {
  count: function(evt) {
    if (evt.type === 'mousedown') {
      this.n = this.n + 1;
    };
  }
})
```

This merging step has still to be executed manually, but since the layer can be directly mapped on the original structures, this process is straight forward:

```
EventCounter = {
  n: 0,
  count: function(evt) {
    if (evt.type === 'mousedown') {
      this.n = this.n + 1;
    };
  }
}
```

If the partial layer definition contains a proceed statement, merging will become more complex, but automatically combining several partial layers and the base system is still possible. We already used a similar technique in a more efficient ContextJS implementation [11].

4. EXAMPLES IN WEBWERKSTATT

We have gained experience using development layers when evolving Lively Kernel in our self-supporting development environment Webwerkstatt [14]. This section shows three small examples that illustrate such usages.

4.1 Example 1: Visualizing Events

The first example shows how layers are used during development to scope the effects of debugging code.

Figure 5 shows how code was added to the base system to see where mouse move events are fired. Adding such debugging code to core behavior can be problematic, as it also affects the tools such as the workspace. Figure 6 shows how by using object specific and structural scoping mechanisms such adaptations of the base system can be scoped, so that the debugging behavior is only active for some specific objects in the system (in that case the yellow rectangle called `DebugArea`). The layer is structurally scoped, by using `setWithLayers`. This scoping mechanism is not general,

Figure 7: A workspace with the text coloring source and an example instance.

but a domain-specific scoping mechanism introduced by the Morphic [15] framework used for Lively Kernel.

4.2 Example 2: Text Coloring

An example of a behavioral adaptation in the development environment is the feature of text coloring via keyboard shortcuts. Figure 7 shows the complete source in a workspace and the text object, on which the new feature was interactively tried out. The layer is refining the keyboard processing method and will color the current selection if specific shortcuts are pressed.

This is a good example where shaping the tools happens directly while they are used. Besides its simplicity, the example also includes modifying the keyboard behavior of all text objects, which can break the whole environment. Using layers allows to directly and interactively develop the features locally and immediately use it in the whole system afterwards.

4.3 Example 3: Developing Autocompletion

A more complex example is the development of source code completion for the development environment as shown in Figure 8. This feature is still under development and not yet part of the core development system. It adapts over 10 methods in two classes, but the figure shows only the area where the new behavior can be tried out and the workspace with the code under development. By using layers, the feature that could interfere with the workspace itself can be developed locally and tried out in other places by changing the scope of the layer.

Figure 8: A snapshot of developing autocompletion of source code, showing a workspace and an area where the new code is activated.

5. PROPOSED TOOL SUPPORT

The workflow in the previous sections is purely based on standard ContextJS syntax and developers have to consciously follow it. They have to actively write their new behavior as layers and scope the new behavior to see the effects but thereby exclude their own tools. If some point of the manual process is not followed, the developers will risk losing parts of their work by accidentally breaking their tools. We propose to move this process into tools to lift mental burdens from the developers.

5.1 Capturing Changes

An enhanced tool support should provide developers with the default view of the system. The view should allow them to directly change classes and methods as they are used to. The system captures these changes but does not apply them directly to the meta-structures, such as editing a method of a class, but adds the change implicitly to a layer. Such a layer can then be scoped, so that its effect cannot break the tools by accident. Figure 9 shows three levels of tools support:

- (A) No tool support, layers have to be defined explicitly.
- (B) The development layer is defined by the workspace, the new behavior is defined in terms of normal class definition syntax. This way developers need not to learn about a new syntax but can stick to their normal ways of modifying classes.
- (C) The development layer is defined by a Smalltalk like, code browser: The user interface and not the syntax is responsible for where the belongs to.

5.2 Interactive Layer Composition

Besides tools that support developers in creating layers, there is a need for tools that allow interactively (de-)activating and scoping of layers. A tool like the one sketched in Figure 10 could allow for faster experimentation with different compositions of development layers.

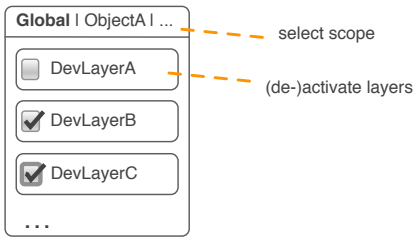

Figure 10: Tool support that allows to directly (de-)activate development layers for a specific scope.

6. RELATED WORK

Any system that allows for dynamically scoped behavioral variations should be able to employ these mechanism during development in self-supporting environments. We used context-oriented programming [3, 7] as an approach for scoping behavioral adaptations at run-time. For an overview of existing COP implementation in various languages see [1].

The problem of breaking tools during the development of interface code was explicitly mentioned as an application of perspectives in the *subjective programming* language *Us* [16]. Us changes message passing of *Self* [18] to incorporate perspectives on layers. In one of their examples they describe how perspectives are used to try out and combine different changes. The authors describe a fallback approach for a debugger to a safer perspective, but the approach seems to be not implemented or evaluated.

ChangeBoxes [4] are a mechanism that allows to manage multiple development branches of an application within a single, running application. This mechanism could basically be used to work with tools of one branch on tools in another branch, so that the tools cannot break themselves as discussed in section 2.3. But the authors do not apply their approach to this domain.

7. CONCLUSION AND FUTURE WORK

Evolving tools in self-supporting development environments can on the one hand be more direct and more interactive than in more separated environments. On the other hand it can be more dangerous, because changing core parts of the system can also lead to accidentally breaking it. In our approach to developing in self-supporting development environments, instead of changing classes directly, we applied context-oriented programming to mitigate this problem by encapsulating changes into layers, which allow to try them out safely.

To illustrate our approach, we showed examples from our work on evolving the Lively Kernel in our self-supporting development environment Webwerkstatt.

Since this workflow relies on developers actively employing layers, as part of future work we want to support an automatic and implicit creation of layered methods. Merging layers with the base system is manual work, which should be automated by tool support in the future.

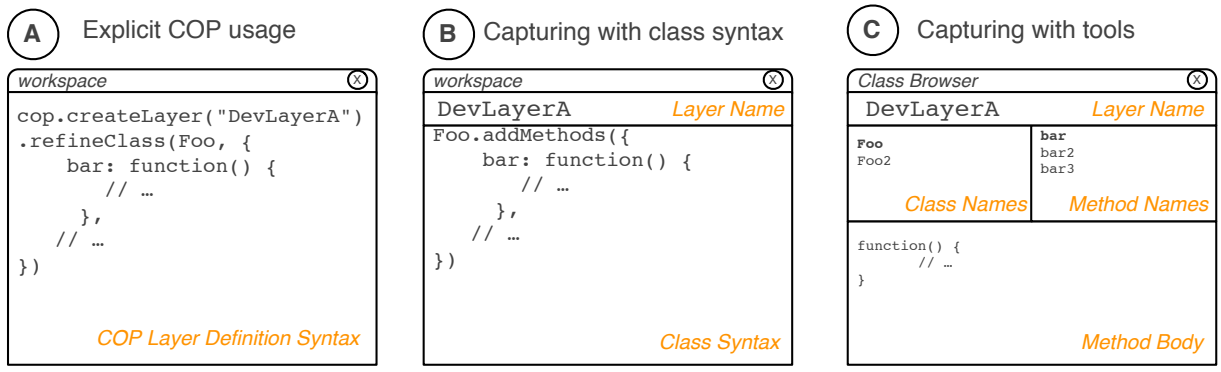

Figure 9: Tool support for development layers

Acknowledgements

This work has been supported by the HPI-Stanford Design Thinking Research Program.

8. REFERENCES

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A Comparison of Context-oriented Programming Languages. In *Proceedings of the Workshop on Context-oriented Programming (COP), co-located with ECOOP 2009, Genoa, Italy*. ACM, 2009.

[2] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In E. Jul, editor, *ECOOP'98 — Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 396–417. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0054101.

[3] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM.

[4] M. Denker, T. Gîrba, A. Lienhard, O. Nierstrasz, L. Renggli, and P. Zumkehr. Encapsulating and Exploiting Change with Changeboxes. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 25–49, New York, NY, USA, 2007. ACM.

[5] E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.

[6] A. Goldberg. *SMALLTALK-80: The Interactive Programming Environment*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

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

[8] R. Hirschfeld and K. Rose, editors. *Self-Sustaining Systems, First Workshop, S3 2008, Potsdam, Germany, May 15-16, 2008, Revised Selected Papers*, volume 5146 of *Lecture Notes in Computer Science*. Springer, 2008.

[9] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. *ACM SIGPLAN Notices*, 32(10):318–326, 1997.

[10] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel A Self-Supporting System on a Web Page. In *S3 2008*, LNCS 5146. Springer-Verlag Berlin Heidelberg, 2008.

[11] R. Krahn, J. Lincke, and R. Hirschfeld. Efficient Layer Activation in ContextJS. In *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing (C5)*. IEEE, 2012.

[12] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming*, 2011.

[13] J. Lincke, R. Krahn, and R. Hirschfeld. Implementing Scoped Method Tracing with ContextJS. In *Workshop on Context-oriented Programming (COP) 2011, co-located with ECOOP 2011, Lancaster, UK*, COP '11, pages 6:1–6:6. ACM, 2011.

[14] J. Lincke, R. Krahn, D. Ingalls, M. Roder, and R. Hirschfeld. The Lively PartsBin–A Cloud-Based Repository for Collaborative Development of Active Web Content. volume 0, pages 693–701, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[15] J. H. Maloney and R. B. Smith. Directness and Liveness in the Morphic User Interface Construction Environment. In *UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 21–28. ACM, 1995.

[16] R. B. Smith and D. Ungar. A Simple and Unifying Approach to Subjective Objects. *Theory and Practice of Object Systems*, 2(3):161–178, 1996.

[17] R. Stallman. *EMACS the Extensible, Customizable Self-documenting Display Editor*, volume 2. ACM, 1981.

[18] D. Ungar and R. B. Smith. Self: The Power of Simplicity. *Lisp and symbolic computation*, 4(3):187–205, 1991.

[19] B. Victor. Inventing on Principle. Invited Talk at Canadian University Software Engineering Conference (CUSEC), January 2012.