

Towards moldable development tools

Andrei Chiş, Oscar Nierstrasz

Software Composition Group,
University of Bern
scg.unibe.ch

Tudor Gîrba

tudorgirba.com

Abstract

Integrated development environments (IDEs) form an essential category of tools for developing software that should support software engineering decision making. Developers commonly ask detailed and domain-specific questions about the software systems they are developing and maintaining. Unfortunately, rigid and generic IDEs that focus on low-level programming tasks that promote code rather than data, and that suppress customization offer limited support for informed decision making during software development. We propose to improve decision making within IDEs by moving from generic to context-aware IDEs through *moldable tools*. In this paper, we promote the idea of moldable tools, illustrate it with concrete examples, and discuss future research directions.

Categories and Subject Descriptors D.2.6 [Software engineering]: Programming Environments – Integrated environments, Interactive environments

Keywords IDEs, customization, adaptation, modeling

1. Research Problem and Motivation

When developers are faced with particular questions that require the customization of existing tools, if they cannot easily customize the relevant tool they will most likely not do so. Popular IDEs do not easily accommodate new functionality. An IDE like Eclipse, for example, allows users to customize the individual windows of the environment, and new tools can be downloaded and installed as “plug-ins”, but developing a new plug-in is highly non-trivial.

To improve decision making within IDEs we consider that an IDE should enable the customization of all its aspects to the current development context. Towards this goal

we propose to model the IDE as a set of interconnected moldable tools, where a moldable tool is *a development tool aware of the current development context that enables rapid customization to new development contexts*. Customization is needed as one cannot anticipate all relevant problems; awareness of the development context enables the tool to automatically detect relevant extensions. Moldable tools build on previous ideas related to extensible environments [3, 5].

In our vision developers adapt a moldable tool to a development context by (i) creating a context-specific extension for that tool and (ii) attaching to the extension an activation predicate (*i.e.*, coupling invariant) that captures the development contexts in which the extension is applicable. Then, at run time, a moldable tool automatically selects extensions appropriate for the current development context.

2. Moldable Development Tools

For this vision to be practical, the cost associated with creating context-specific extensions should be small. Nevertheless, the cost can vary significantly based on the tool and the type of extension that is required. In what follows we summarize three examples of moldable tools, motivate the type of extensions they support, and look at the effort required to extend them, in terms of lines of code.

2.1 Moldable Inspector

Object inspectors allow developers to comprehend the runtime of object-oriented systems. Nevertheless, traditional object inspectors favor generic approaches to display and explore the state of arbitrary objects. While universally applicable, these approaches do not take into account the varying needs of developers that could benefit from tailored views and exploration possibilities [2]. For example, when interacting with graphical objects, depending on the task at hand, a developer could need to inspect its visual representation, its structure in terms of graphical subcomponents, or its internal state.

To inspect objects using tailored views we propose *Moldable Inspector*,¹ an inspector model that supports multiple domain-specific views for each object and facilitates the cre-

[Copyright notice will appear here once 'preprint' option is removed.]

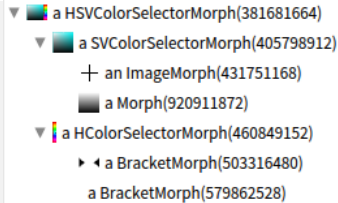
¹scg.unibe.ch/research/moldableinspector

ation and integration of new views [2]. In the current implementation views are created using an internal DSL and placed within methods that require predefined annotations. Until now we have created, together with the developers of several frameworks/libraries, 131 custom views for 84 objects belonging to 15 applications, requiring, on average, $9.2 \pm 6.6(M \pm SD)$ lines of code per view. An example illustrating how to instantiate a view that shows the structure of a graphical object is given below, together with the resulting view when applied on a graphical object.

```

1 aCanvas tree
2   title: 'Submorphs';
3   display: #yourself;
4   format: #printString;
5   icon: #scaledIcon;
6   children: #submorphs;
7   when: #hasSubmorphs

```



2.2 Moldable Spotter

Search tools enable developers to rapidly identify or locate entities of interest. Nevertheless, current software systems interleave many different types of highly interconnected data (e.g., source code, documentation, configuration files). Dealing with this reality requires developers to perform investigations that chain together multiple searches on different data types. Nevertheless, most mainstream IDEs address searching by means of many disconnected search tools, making it difficult for developers to reuse search results produced by one search tool as input for another search tool.

To reduce the time required to manually link disconnected search results we propose *Moldable Spotter*,² a model for expressing and combining search tools in a unified way. Moldable Spotter supports searches that build upon each other and makes it easy to express and integrate new types of searches on new data types. Using an internal DSL, we have created 100 custom searches for 30 different data types. On average, extending Moldable Spotter with a new type of search required 8 lines of code.

2.3 Moldable Debugger

Debuggers are essential for reasoning about the run-time behaviour of software systems as they give developers direct access to the running systems. Nevertheless, traditional debuggers rely on generic mechanisms to interact with the running systems, while developers reason about and formulate domain-specific questions using concepts and abstractions from their application domains. For example debugging event-based systems or parsers using generic stack-based debuggers can be problematic.

To enable developers to take advantage of domain concepts when debugging we propose the *Moldable Debugger*,³ a debugger model that enables developers to create specific

² scg.unibe.ch/research/moldablespotter

³ scg.unibe.ch/research/moldabledebugger

debuggers with custom debugging operations for stepping through the execution and custom user interfaces [1]. In the current implementation custom debuggers are created by subclassing a predefined class containing a template for a debugger, and overriding hook methods for constructing debugging actions and the user interface. We create, on top of a template of 1500 lines of code, six custom debuggers requiring between 60 and 600 lines of code. The cost is greater than in the previous two tools as the scope of an extension is larger (i.e., the entire debugger).

3. Discussion and Future Work

Moldable tools come with a price as they have to be extended by application or framework developers rather than by tool providers. Nevertheless, this can make considerable economical sense when working on a long-lived system. Furthermore, library developers can ship library-specific extensions together with their product.

Feedback from developers using and extending these tools is essential for their improvement. All tools presented here are part of the Moose (moosetechnology.org) platform and Moldable Inspector and Moldable Spotter are also integrated into the Pharo IDE (pharo.org) as part of the Glamorous Toolkit (gt.moosetechnology.org). We are currently working towards gathering and analysing usage data to learn how these tools are used and extended in practice [4]. Preliminary data indicate that developers use multiple existing extensions and create custom extensions for libraries and frameworks used in their projects.

This paper covers just a few tools and types of extensions. We are actively working on improving the customization support and applying the ideas of moldable tools to other tools, like editors and REPLs. Our long-term goal is to apply these ideas to the entire tool suite from an IDE.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Assessment” (SNSF project Nr. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015).

References

- [1] A. Chiş, T. Gîrba, and O. Nierstrasz. The Moldable Debugger: A framework for developing domain-specific debuggers. In *SLE*, volume 8706, pages 102–121. Springer International Publishing, 2014.
- [2] A. Chiş, T. Gîrba, O. Nierstrasz, and A. Syrel. The Moldable Inspector. In *Onward!* to appear, 2015.
- [3] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Trans. Softw. Eng.*, 12(12):1117–1127, 1986.
- [4] J. Kubelka, A. Bergel, A. Chiş, T. Gîrba, S. Reichhart, R. Robbes, and A. Syrel. On understanding how developers use the Spotter search tool. In *VISSOFT*. to appear, 2015.
- [5] A. Mørch. Three levels of end-user tailoring: Customization, integration, and extension. In *Computers and Design in Context*, pages 51–76. MIT Press, 1997.