Towards Object-Aware Development Tools

(Preprint *)

Andrei Chiş
University of Bern
andreichis.com

Abstract

Reasoning about object-oriented applications requires developers to answer contextual questions about their domain objects. Tailored development tools can support developers in this activity by providing relevant domain-specific information. Nonetheless, a high effort for extending development tools to handle domain-specific objects, together with diverging mechanisms for creating, sharing and discovering extensions, discourage developers to adapt their tools. To address this, we propose to enable contextual behavior in development tools by allowing domain objects to decide how they are handled in development tools. We show that combining this idea with mechanisms for specifying extensions using internal DSLs can significantly reduce the cost of tailoring development tools to specific domains.

Categories and Subject Descriptors D.2.6 [*Software Engineering*]: Programming Environments—integrated environments, interactive environments

Keywords Customization, Object-oriented Programming

1. Research Problem and Motivation

To take advantage of domain concepts during their day-to-day activities, developers need to navigate effectively between domain concepts and the code that implements those concepts [8]. Consider a graphical library for working with various graphical widgets modeled as objects. The graphical representation of a widget, the structure of graphical widgets within a composite widget, and the layout mechanism are several concepts that play an important role in working with a graphical library. Nonetheless, development tools like debuggers, code editors or object inspectors, the tools developers often use to interact with software systems, focus mainly on generic object-oriented idioms. This focus on generic idioms in development tools ignores the fact that developers use objects to capture high-level abstractions from their application domains, and would benefit from development tools aware of those application domains [6]. For example, an object inspector can incorporate a graphical representation of a widget and a search tool an extension for searching through a widget's structure of subwidgets.

A solution for making development tools domain-aware is to enable developers to adapt development tools to their own tasks and domains. To support this activity we propose an extension mechanism that attaches extensions to domain objects, supports inexpensive creation of domain-specific extensions through internal DSLs, and automatically selects extensions based on the domain model and the developer's interaction with the domain model (*i.e.*, development context). We designed three development tools incorporating this extension mechanism and integrated them into the Pharo IDE (`http://pharo.org`). Currently we created together with the developers of several frameworks and libraries several hundred extensions, and are studying the cost in terms of time and lines of code incurred by an extension.

2. Background

When interacting with objects, tools need to decide how to handle objects modeling different domain entities. Development tools employ a wide range of extension mechanisms to support this desideratum. For example, IDEs like *Eclipse*[1] and *IntelliJ*[2] are based on plug-in architectures that rely on extension points [10]. Live environments like *Self* model development tools as run-time objects that developers can directly edit [9]. *JGrasp* uses pattern matching to associate graphical views with objects based on their structure [5] (*e.g.*, an object having a tree-like structure is displayed using a tree view). These solutions support extension mechanisms that separate the business logic of domain objects from the logic

* In Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, October 30 – November 4, 2016, Amsterdam, Netherlands. DOI: `10.1145/2984043.2998546`

[1] `http://www.eclipse.org`

[2] `http://www.jetbrains.com/idea`

used to handle those objects in development tools. This separation, however, does not favor co-evolution of objects and tools: it increases the distance between objects and extensions, making it difficult for developers editing an object to be aware that an extension for that objects is present in a tool.

The alternative consists in making objects responsible for deciding how they are handled in development tools. This follows the idea of Pawson, which in the context of business systems, observed that many domain objects are *behaviorally-weak* (*i.e.*, their functionality is spread through out other *'controller'* objects) and proposed *naked objects* [7] as a way to move towards behaviorally-complete objects where all user actions are contained within the object. This approach is commonly used to visualize objects by making objects responsible to represent themselves in a textual way through dedicated methods (*e.g.*, toString in Java, __str__ and __repr__ in Python). Nevertheless, this approach of customizing development tools is only leveraged in a small number of tools.

3. Approach and Current Results

To create development tools that adapt to contextual situations, we propose to move towards behaviorally-complete objects that can decide how to be handled by development tools. Through our work on *moldable tools* [2] we support this idea by attaching extensions to domain-objects and allowing extension creators to specify together with their extensions an *activation predicate* capturing the development context in which those extensions are applicable. A moldable tool supports inexpensive extensions, maintains a development context and automatically selects at run-time object extensions applicable in the current development context.

We applied the moldable tools idea to improve the way developers visualize objects through the *Moldable Inspector* [3], an inspector that allows each object to represent itself in multiple ways. This inspector works by enabling developers to attach to each object multiple custom views and selecting at run time appropriate views. Views are attached to an object by defining in its class methods having a predefined annotation with a parameter used to order views (line 2).

```
1 Morph>>#submorphsIn: aCanvas inContext: aContext
2 <gtInspectorPresentationOrder: 80>
3 aCanvas tree
4   title: #Submorphs;
5   display: [ self ];
6   format: #printString;
7   icon: #scaledIcon;
8   children: #submorphs;
9   when: [:aMorph | aMorp hasSubmorphs ]
```

These methods construct views using an internal DSL that optimizes for the creation of several types of common views (*e.g.*, tree, list, text, code, graph). Lines 3–9 show the code for creating a tree view for a widget. Line 9 specifies the activation predicate that indicates that the extension is available only when the widget contains other widgets. An activation predicate can further access previously inspected objects using the parameter aContext. The Moldable Inspector implementations (`http://gtoolkit.org`) is part of the Pharo IDE that currently contains 165 inspector extensions for 105 types of objects, requiring on average 9 lines of code for an extension. We applied a similar approach to creating a moldable search interface with similar results, and further observed that creating their very first extension took six developers, on average, 16 minutes and 9 lines of code [4].

Through the *Moldable Debugger* [1] we further investigated how to apply the moldable tools idea to support domain-specific debugging. Towards this goal the Moldable Debugger models the run-time stack of a process as an object, and attaches domain-specific debuggers to stack objects. Activation predicates now check the stack object to determine if a domain-specific debugger is applicable. For example, a domain-specific debugger for a graphical library is only applicable if the run-time stack contains an invocation of that library. In this case, too, by using a DSL, we could create custom debuggers in under 500 lines of code

To make development tools domain-aware we propose an extension mechanism that attaches extensions to domain-objects and automatically selects appropriate extensions using activation predicates. To reduce the cost of creating extensions we investigate the design of dedicated internal DSLs. We validated the extension mechanism and the proposed DSLs by applying them to three development tools. Currently we are performing a larger user study to investigate the process of how developers extend moldable tools. Further work is needed to fully apply this idea to other types of development tools and to better understand how developers use domain-specific extensions during their activities.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Analysis” (SNSF project No. 200020-162352, Jan 1, 2016 - Dec. 30, 2018).

References

[1] A. Chiş, M. Denker, T. Gîrba, and O. Nierstrasz. Practical domain-specific debuggers using the Moldable Debugger framework. *Computer Languages, Systems & Structures*, 44, Part A:89–113, 2015.

[2] A. Chiş, T. Gîrba, and O. Nierstrasz. Towards moldable development tools. In *PLATEAU '15*, pages 25–26, 2015.

[3] A. Chiş, T. Gîrba, O. Nierstrasz, and A. Syrel. The Moldable Inspector. In *Onward! 2015*, pages 44–60, 2015.

[4] A. Chiş, T. Gîrba, J. Kubelka, O. Nierstrasz, S. Reichhart, and A. Syrel. Moldable, context-aware searching with Spotter. In *Onward!*, 2016.

[5] J. H. Cross, II, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, and L. N. Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *Trans. Comput. Educ.*, 9(2):13:1–13:32, June 2009.

[6] O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening Traits. *Journal of Object Technology*, 5(4):129–148, May 2006.

[7] R. Pawson. *Naked Objects*. Ph.D. thesis, Trinity College, 2004.

[8] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC '02*, pages 271–278

[9] R. B. Smith, J. Maloney, and D. Ungar. The Self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. *SIGPLAN Not.*, 30(10):47–60, Oct. 1995.

[10] Z. Yang and M. Jiang. Using Eclipse as a tool-integration platform for software development. *IEEE Software*, 24(2):87–89, Mar. 2007.