Moldable Debugging

Andrei Chiş
feenk gmbh
Bern, Switzerland
andreichis.com

Tudor Gîrba
feenk gmbh
Bern, Switzerland
tudorgirba.com

***Abstract*—Programming languages evolved towards letting developers design software applications in terms of domain abstractions. What about development tools? While developers express software using programming languages, they craft software exclusively by interacting with development tools. Unfortunately, all too often developers rely on rigid development tools, focused on programming language constructs, unaware of application domains. On the one hand, we educate developers to shape custom and rich domain models. On the other hand, we still force them to use a one-size-fits-all-paradigm when it comes to development tools.**

One way to approach this abstraction gap is to empower developers to shape their development tools together with their domain models. In this extended abstract we explore how this can transform debugging and debuggers, and delve into what is needed to make this view a reality. We argue that to achieve this, debugging infrastructures need to support the straightforward and inexpensive creation of custom debuggers by leveraging the explicit representation of the underlying application domains.

I. Programming is modeling

Software applications capture abstract models of the real world as executable models (*i.e.*, programs) within the design space of a programming language. Model-driven engineering provides developers with different mechanisms for facilitating the creation of models, like domain-specific modelling languages [1]. Object-oriented programming in particular supports this desideratum by allowing developers to model their domains in terms of objects and message sends.

Debugging software applications requires developers to *(i)* navigate between domain abstractions and the code that addresses those abstractions, and *(ii)* understand domain abstractions together with their interactions. Traditional debuggers support this activity by focusing on generic stack-based operations, line breakpoints, and generic user interfaces. This impedes the ability of developers to take direct advantage of the domain, leading to a fragmentation of their domain-specific questions into low-level ones that can be answered with available tools [2]. To eliminate this abstraction gap developers should rely on debuggers that work at the level of abstraction of an application's domain and enable domain-specific views, queries and analyses [3]. While this goal is clear, it is not always straightforward to reach.

II. Developers as tool builders

When developers encounter domain-specific questions for which their debuggers or development tools do not offer a direct answer, they have the option to adapt those tools. Whittle *et al.* [4] observed that in the context of model-driven engineering many developers build their own tools or make heavy adaptations to off-the-shelf tools, even if this requires significant effort. Smith *et al.* [5] further noticed that developers take the initiative to build tools to solve the problems they face, especially when their organization's culture promotes this activity.

Hence, developers are willing to extend their tools. Supporting them in doing this implies not a focus on providing ready-made functionality, but a focus on offering rich programability [6]. Consider testing. With frameworks like SUnit, testing frameworks focused explicitly on significantly decreasing the cost of creating tests, encouraging the adoption of testing as an integral activity of the software development process. The same should happen for debugger extensions.

III. Enabling moldability

Extending debuggers to capture domain-specific aspects should be as obvious as writing unit tests. Attaining this goal is a challenging endeavour as it raises many questions: *What are the right extension mechanisms? How inexpensive can the creation of a domain-specific debugger really be?* Let us explore next the first of these questions in more details.

Understanding a domain model requires first and foremost reasoning about its individual domain objects. Traditional debuggers support this through the use of object inspectors that favor a generic view showing only the state of an object. While universally applicable, this solution does not take into account the varying needs of developers that could benefit from domain-specific views and exploration possibilities [7]. For example, we should display an object representing a parser using a view that shows its grammar productions, and a widget using a view that shows its actual graphical representation or its containment structure. Hence, a moldable debugging infrastructure should start by enabling developers to view model elements using multiple tailored views, and facilitate the creation and integration of new views.

Understanding domain objects in isolation is not enough. Depending on the application domain and their task, developers need to correlate information from multiple sources. For example, when debugging a parser both the grammar rules and the input being parsed are of interest; when debugging an event-driven system, the publisher, the subscriber, and the event are of interest. A moldable debugging infrastructure should support the creation of multiple domain-specific user

interfaces for debugging that extract and highlight relevant data from application domains. A tailored user interface consists in multiple widgets, each showing custom views for domain objects.

Once developers have custom user interfaces offering domain-specific views, they also need to navigate through the execution at the level of abstraction of those domains. If the domain is that of a parser, stepping through the execution at the level of grammar production or the input string is what is needed. If it is an event-driven system, the right level of abstraction is given by the propagation of events through the system. To enable this, a moldable debugging infrastructure needs to allow developers to create debugging operations that express and automate high-level abstractions from application domains. Debugging operations are then attached to the appropriate widgets.

While addressing the basic information needs in a debugger, the aforementioned mechanisms are not enough. During debugging, developers cannot know in advance in what situations they will find themselves in [8]. Hence, they might begin with the wrong user interface and set of actions. If they do not know that certain views and actions are applicable for their current debugging context, they will not use them. A moldable debugging infrastructure can address this by attaching to every view, widget and action an *activation predicate*, *i.e.*, a boolean condition over the state of the program capturing those states in which the extension is applicable. Then, only extensions applicable in the current debugging context are made available; if two or more user interfaces are applicable a developer should be able to switch at run time between them.

IV. On the cost of tool building

For developers to extend their debuggers, the cost associated with creating extensions should be small. How we define cost, and the way to reduce it, depends on the mechanism for defining extensions. If we automatically generate debugging extensions from a model's specification [9], [10], the cost goes into creating well-formed specifications for models. If we provide developers with a toolset for constructing debugger extensions on top of their models, cost is related to the effort needed to develop and maintain these extra extensions.

Through the Moldable Debugger[1] we explored the second direction, in the context of object-oriented programming [11], as object-oriented programming provides developers with a direct way of expressing domain models using objects. Debuggers however rarely take those models into account.

We observed that the cost of creating custom views for domain objects can be significantly reduced. Moose,[2] a platform for software and data analysis comes with more than 230 extensions for visualising objects, ranging from parsers and compiled code to graphical widgets. Custom views are expressed using an internal DSL that supports different kinds of views. On average creating a view requires 9 lines of code. Combining these views to form custom user interfaces and adding debugging actions increases the cost. Moose also ships with six custom debuggers. By providing internal DSLs for constructing user interfaces and specifying debugging actions, a debugger can be created in under 500 LOC of code, on top of a base implementation of 1500 LOC. Certainly, the LOC metric must be taken with care as it does not indicate the time and expertise needed to write the lines. Nevertheless, it does provide a good indication of the small size of these domain-specific debuggers. By supporting each step of the customisation through an internal DSL, developers do not have to learn new syntaxes, only dedicated APIs.

Given the difficulty of debugging, improving how developers view and navigate their models is needed, even if the cost of creating custom debuggers is high. A low cost can make this activity even more appealing. In today's world, we rarely develop an application without tests, or depend on external components without tests. In tomorrow's world, we should be as demanding when it comes to debugger extensions.

[1]The Moldable Debugger is a model of an extensible debugger. *GTDebugger* is an implementation of this model in Pharo (pharo.org) as part of the *GToolkit* (gtoolkit.org).

[2]moosetechnology.org

References

[1] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray, "Globalizing modeling languages," *Computer*, vol. 47, no. 6, pp. 68–71, 2014.

[2] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Trans. Softw. Eng.*, vol. 34, pp. 434–451, Jul. 2008.

[3] O. Nierstrasz, "The death of object-oriented programming," in *FASE 2016*, ser. LNCS, P. Stevens and A. Wasowski, Eds., vol. 9633. Springer-Verlag, 2016, pp. 3–10.

[4] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, *Model-Driven Engineering Languages and Systems: 16th International Conference, (MODELS 2013).* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ch. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?, pp. 1–17.

[5] E. K. Smith, C. Bird, and T. Zimmermann, "Build it yourself! Homegrown tools in a large software company," in *Proceedings of the 37th International Conference on Software Engineering*. IEEE – Institute of Electrical and Electronics Engineers, May 2015.

[6] T. Gîrba and A. Chiş, "Pervasive Software Visualizations," in *Proceedings of 3rd IEEE Working Conference on Software Visualization*, ser. VISSOFT'15. IEEE, Sep. 2015, pp. 1–5.

[7] A. Chiş, T. Gîrba, O. Nierstrasz, and A. Syrel, "The Moldable Inspector," in *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, pp. 44–60.

[8] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss, "Debugger canvas: industrial experience with the code bubbles paradigm," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1064–1073.

[9] P. R. Henriques, M. J. V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu, "Automatic generation of language-based tools using the LISA system," *Software, IEE Proceedings* -, vol. 152, no. 2, pp. 54–69, 2005.

[10] E. Bousse, J. Corley, B. Combemale, J. Gray, and B. Baudry, "Supporting Efficient and Advanced Omniscient Debugging for xDSMLs," in *8th International Conference on Software Language Engineering (SLE)* , Pittsburg, United States, Oct. 2015.

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