Exemplifying Moldable Development

(Preprint*)

Andrei Chiş
Software Composition Group
University of Bern, Switzerland
andreichis.com

Tudor Gîrba
feenk.com, Switzerland

Juraj Kubelka
PLEIAD Laboratory
University of Chile, Chile
pleiad.cl

Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
scg.unibe.ch

Stefan Reichhart
stefan.reichhart@gmail.com, Switzerland

Aliaksei Syrel
Software Composition Group
University of Bern, Switzerland
scg.unibe.ch

ABSTRACT

Developing and evolving software requires developers to continuously make decisions about how to steer the design and implementation of their applications. To make informed decisions developers commonly formulate detailed and domain-specific questions about their software systems and use tools to explore available information and answer those questions. Development tools however focus on generic programming tasks while program comprehension and analysis tools typically are not tightly integrated with their development tools and environments. This has a negative effect on program comprehension as it increases the effort and the time needed to obtain an answer.

To improve program comprehension we propose that developers build software using development tools tailored to their specific application domains, as this can directly answer domain-specific questions. We introduce *moldable development* as an approach for developing software in which developers evolve development tools together with their applications. In this paper we sketch the idea of moldable development and give examples to illustrate how it can be applied in practice. Through these examples we show that given a low cost for extending development tools, developers can create relevant and useful customizations to help them evolve their own applications.

CCS Concepts

•Software and its engineering → Application specific development environments; Object oriented development; *Software design engineering;*

Keywords

Domain-specific tools, User interfaces, Programming environments, Program comprehension

1. INTRODUCTION

Researchers have estimated that program comprehension takes from 30% to 70% of the software development and maintenance time [34, 12, 17, 11, 20]. Given this and the sheer size and complexity of today's software systems, a wide range of analyses and program comprehension tools have been proposed to aid developers in answering their questions. Nevertheless, in spite of an ever-increasing number of program comprehension tools, these tools are still heavily underused [24]. Instead, developers mostly rely on code reading to understand and reason about their software systems. One the one hand, code reading is highly contextual: code indicates the exact behavior of an application. On the other hand, code reading does not scale: simply reading one hundred thousand lines of code takes more than one-man month of work.

We propose a novel approach to better integrate program comprehension tools into the development environment, and thus reduce reliance on code reading. To motivate our work,

*In Proceedings of the 1st Edition of the Programming Experience Workshop (PX/16), July 18, 2016, Rome, Italy.
DOI: 10.1145/2984380.2984385

we start by discussing two issues with current tool support that contribute to the problem.

Disconnected comprehension.

Separating program comprehension and development tools creates a gap between program comprehension and development, two activities that are deeply intertwined. For example, integrated development environments (IDEs) are an essential category of tools for crafting software. They aim to support software development and evolution by providing a uniform interface for all the tools needed by programmers during the software development process (*e.g.*, code editors, compilers, testing tools, debuggers). Nevertheless, a look at mainstream IDEs shows that they are centered around the code editor and promote code reading as a default way of reasoning about software. Developers can use additional program comprehension tools alongside the IDE or install them as 'plug-ins', however, with few exceptions these new tools do not integrate with existing development tools from the IDE. Apart from first finding what tools or extensions are applicable for their contexts, this also requires developers to manually bring data provided by these new tools into their current development tools.

Generic tools.

While addressing a specific task, many development and program comprehension tools do not make any assumptions about the specific *contexts* in which they are used. They handle in the same way software applications written in one or more programming languages even if those software applications model different domains. For example, a generic source code editor for a programming language handles all applications written in that language in an identical manner. On the one hand, this increases their range of applicability; on the other hand, it makes them less suited to handle detailed and domain-specific questions. Generic tools force developers to refine their domain-specific questions into low-level generic ones and mentally piece together information from various sources [25]. This offers limited support for informed decision making, leading to an inefficient and error-prone effort during software development and maintenance as developers cannot directly reason in terms of domain abstractions.

Both these issues can be improved by moving from building software using generic and disconnected tools for development and program comprehension to building software using development tools tailored to specific application domains, as this can directly answer domain-specific questions and does not require developers to look for relevant data elsewhere. For this vision to be possible we propose *moldable development*, an approach for crafting software in which developers continuously adapt and evolve their development tools (*e.g.*, code editors, debuggers, search tools, run-time inspectors) to take into account their actual application domains. This has the potential to reduce code reading and improve program comprehension as developers can incorporate into their development tools domain-specific information that they would otherwise need to find by reading and exploring source code or using external tools. Through this paper we aim to sketch, motivate and explore the feasibility of this idea. Towards these goals the main contributions of this paper are:

- Discussing challenges for making moldable development practical and proposing an approach to achieve this based on *moldable tools*;
- Illustrating *moldable development* through real world examples of how it can be applied to improve program comprehension.

2. MOLDABLE DEVELOPMENT, MOLDABLE TOOLS

The key idea behind *moldable development* is that developers extend their development tools as they evolve their applications. To be feasible it requires both that developers are willing to extend their tools, and that development tools are designed to capture domain abstractions.

2.1 Towards Moldable Development

In the context of model-driven engineering Whittle *et al.* observed that to improve the way they evolve their applications many developers build their own tools or introduce major adaptations to off-the-shelf tools [33], even if this requires significant effort. When studying homegrown tools in a large software company Smith *et al.* [26] also observed that developers take the initiative to build tools to solve problems they face, especially when their organization culture promotes this activity. This shows that developers do build tools to help themselves in their work. Nevertheless, adapting development tools to specific domains is not a widespread activity.

To increase its adoption we argue that moldable development has to have as its foundation development tools designed so they can inexpensively accommodate domain abstractions. As an analogy, in the past testing was perceived as difficult since writing tests was a costly activity. With the introduction of SUnit [3] and other testing frameworks the cost of creating and managing tests decreased significantly, thus encouraging the adoption of testing as an integral activity of the software development process.

We propose to accommodate domain abstractions in development tools by designing development tools that:

- support inexpensive creation of domain-specific extensions;
- enable developers to easily organize and locate suitable extensions.

Both aspects are needed: even if extensions are easy to build, difficulty in finding and deciding when an extension is applicable discourages developers from embracing the activity of adapting their development tools.

2.2 Moldable Tools

In previous work we proposed the idea of *moldable tools* [8] as a way to reduce the cost of incorporating domain abstractions into development tools. We define a moldable tool as *a development tool aware of the current development context that enables rapid customization to new development contexts.* Moldable tools enable developers to *'mold'* domain abstractions into the tools by creating domain-specific extensions capturing those abstractions, and attaching to those extensions *activation predicates* that capture the development contexts in which extensions are applicable. Then, at run time, a moldable tool automatically selects extensions appropriate for the current development context. Hence, developers do not have to manually infer when an extension is applicable.

To investigate the applicability of moldable tools we applied it to three challenging activities performed by developers during software development: *(i)* reasoning about run-time objects, *(ii)* searching for domain-specific artifacts, and *(iii)* reasoning about run-time behavior. For each one we showed that relevant problems can be addressed if developers are able to adapt their development tools to their own contextual needs. For completeness we give next a short description of the tools that we built to address these challenges.[1]

Moldable Inspector.

Traditional object inspectors favor generic approaches to display and explore the state of arbitrary objects, while developers could benefit from tailored views and exploration possibilities. To address this gap we propose the *Moldable Inspector*, an object inspector that makes the inspection context explicit, supports multiple domain-specific views for each object, and facilitates the creation and integration of new views [9]. Until now we have created, together with the developers of several frameworks and libraries, 131 custom views for 84 types of objects belonging to 15 applications, requiring, on average, 9.2 lines of code per view.

Moldable Spotter.

Generic and disconnected search tools impede search tasks over domain-specific entities. To address this problem we propose the *Moldable Spotter*, a search tool that directly enables developers to discover and search through domain concepts by creating custom searches for their domain objects and automatically presents searches for domain objects, as developers are interacting with those objects [27]. We also show that by taking into account generic searches through code, *Moldable Spotter* can provide a single entry point for embedding search support within IDEs. Based on 124 search extensions currently present in the Pharo IDE the average cost of creating an extension in lines of code is 9.2.

[1]These tools are developed as part of the GToolkit project. More information about the tools can be found at http://gtoolkit.org. All these tools are also part of the Pharo IDE (http://pharo.org).

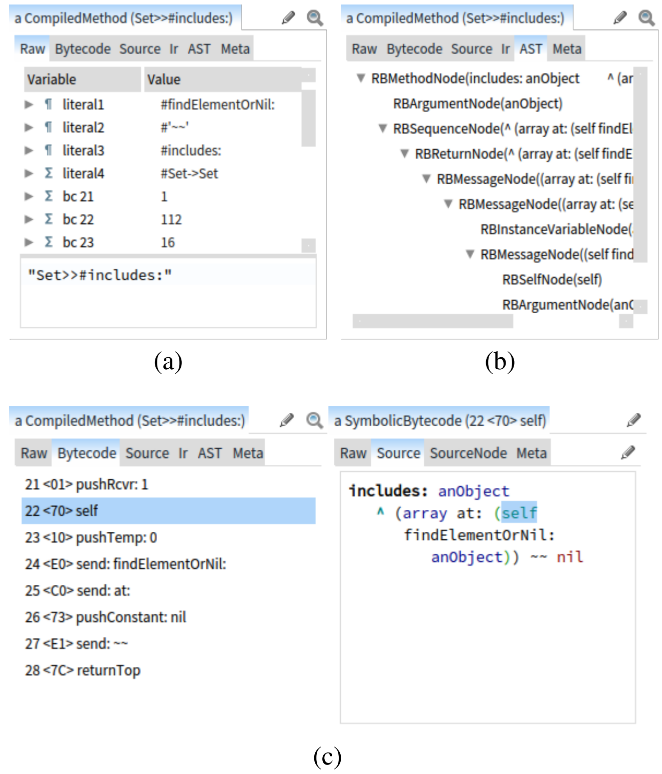


Figure 1: Using the Moldable Inspector to visualize compiled code: *(a)* The *Raw* view shows the implementation of the object; *(b)* The AST view shows the AST from which the code was compiled; *(c)* The mapping between bytecode and source code can be explored by selecting bytecodes in the *Bytecode* view.

Moldable Debugger.

Traditional debuggers rely on generic mechanisms to introspect and interact with the running systems (*i.e.*, stack-based operations, line breakpoints), while developers reason about and formulate domain-specific questions using concepts and abstractions from their application domains. To address this abstraction gap we propose the *Moldable Debugger*, a framework for developing domain-specific debuggers [7]. The moldable debugger is adapted to a domain by creating and combining domain-specific debugging operations with domain-specific debugging views, and adapts itself to a domain by selecting, at run time, appropriate debugging operations and views. We have created, 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. EXEMPLIFYING MOLDABLE DEVELOPMENT

To illustrate how moldable development can be applied during the development of an application to improve comprehension we present two concrete use cases. In each one the developers/maintainers of an application created together with the first two authors custom extensions for the aforementioned moldable tools, as they evolved their application.

3.1 Opal Compiler

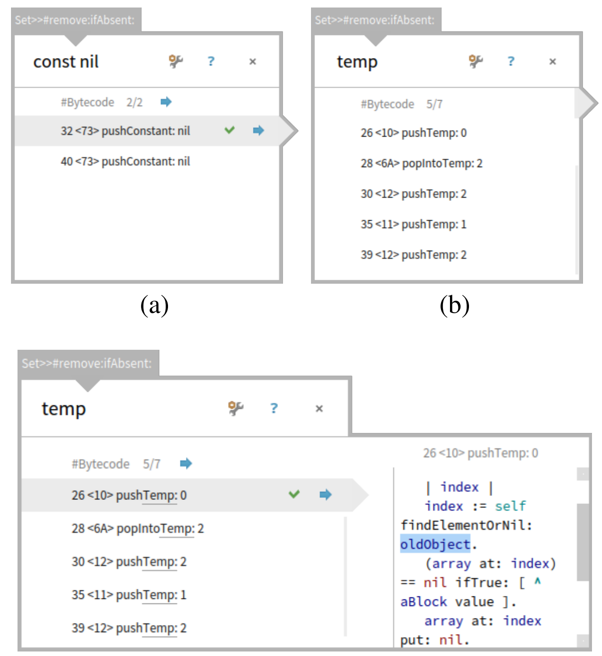
Opal[2] is a new compiler infrastructure for Pharo[3] focusing on customizability. It has been part of Pharo since the Pharo 3 release (May 2014).[4] Initially Opal was developed using the standard development tools of Pharo.

Developing a new compiler is a challenging activity involving multiple steps: parsing the source code into an abstract syntax tree (AST), translating the AST into an intermediate representation (IR), translating the IR into bytecode, and optimizing at the level of the AST, IR and bytecode. Types of bugs specific to compilers and encountered during development were those related to incorrect generation of bytecode from IR, and wrong mappings between source code or AST nodes and bytecode caused by compiler optimizations.[5] Debugging these types of bugs just by reading code or using generic debuggers and inspectors is a difficult endeavor as the information needed (*i.e.*, the mapping between source code, AST nodes, IR and bytecode) is highly domain-specific and not present in these tools by default. To make this information explicit we extended several development tools together with the Opal team while Opal itself was being developed.

Moldable Inspector extensions.

In Pharo methods are represented as instances of the CompiledMethod class and they hold the corresponding bytecode. Inspecting the attributes of a CompiledMethod object in a generic object inspector only gives details about the format in which bytecode is represented (header, literals, trailer) and shows the numeric code of the bytecode. For example in Figure 1a we can see that the inspected method has 4 literals, and the second bytecode stored at index 22 has the code 112. This provides no insight into what the actual bytecode does, the source code of the method, the AST, the IR or the mapping between these representations. To address these issues we gradually extended the object inspector with several custom views that are applicable when inspecting CompiledMethod objects: a human-friendly representation of the bytecode (left side Figure 1c), the source code of the method, the AST (Figure 1b), and the IR. Using the bytecode view the developer can see that the bytecode at index 22 corresponds to pushing self[6] to the top of the stack. To show the mapping between bytecode and source code, whenever a bytecode is selected a new view is opened to the right showing the source code of the method and highlighting the

[2] http://www.smalltalkhub.com/#!/~Pharo/Opal
[3] http://pharo.org
[4] http://pharo.org/news/pharo-3.0-released
[5] pharo.fogbugz.com/f/cases/14606, pharo.fogbugz.com/f/cases/12887, pharo.fogbugz.com/f/cases/13260, pharo.fogbugz.com/f/cases/15174
[6] self represents the object that received the current message; this in Java.



(a) (b) (c)

Figure 2: Searching through bytecode using Moldable Spotter: *(a)* searching for accesses to nil; *(b)* searching for instructions accessing temporary variables; *(c)* When selecting a bytecode the mapping with the source code is shown.

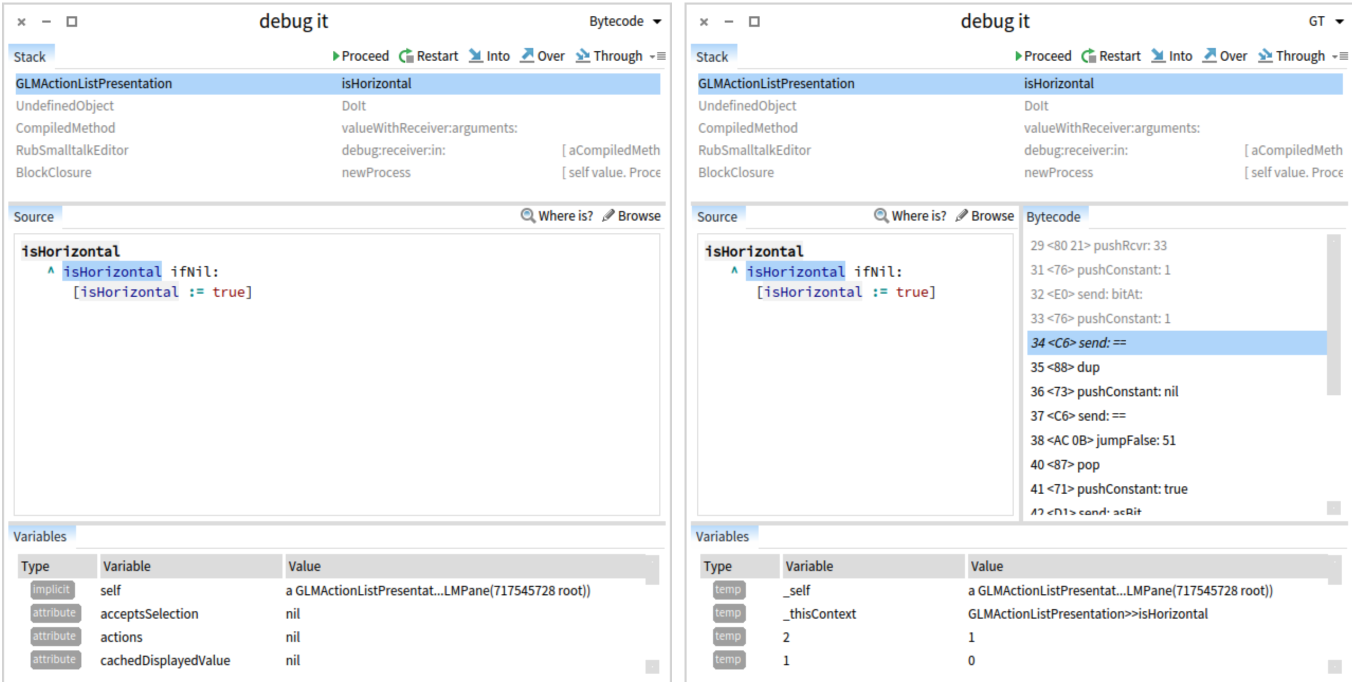
code corresponding to the selected bytecode (Figure 1c); this relies on the ability of the inspector to display two or more objects at once.

Extensions to the Moldable Inspector are constructed using code snippets that return graphical objects. We provide an internal domain-specific language (*i.e.*, a fluent API) that can be used to directly instantiate several types of basic graphical objects such as list, tree, table, text and code; any other graphical object from Pharo can also be used in an extension. Extensions are then attached to objects by defining within their classes methods that construct those extensions, and marking those methods with a predefined annotation. For example, lines 1–8 show the code for creating the AST view displayed in Figure 1b.

```
1  CompiledMethod>>#gtInspectorASTIn: aComposite
2    <gtInspectorPresentationOrder: 35>
3    aComposite tree
4      rootsExpanded;
5      title: 'AST';
6      display: [ self ast ];
7      children: [ :aNode | aNode children ];
8      format: [ :aNode | aNode gtPrintString ]
```

Moldable Spotter extensions.

Apart from inspecting compiled code, especially when compiling long methods, common tasks consist in locating certain types of bytecode instructions (*e.g.*, pop, return),



(a) (b)

Figure 3: Debugging a boolean slot: *(a)* When using a generic debugger developers cannot access the bytecode generated by the boolean slot; *(b)* An extension to the debugger that shows the bytecode of the current method and supports stepping at the bytecode level; this gives direct access to bytecode generated by the boolean slot.

message sends (*e.g.*,, send: printString) or accesses of literal values (pushLit: Object). A generic tool to search through source code or object state does not provide this type of functionality. To support these tasks we extended the search framework from Pharo with a custom search through the human-friendly representation of bytecode previously introduced (lines 9–15). Creating extensions for Moldable Spotter follows the same principle as in the case of the Moldable Inspector; only a different API and annotation are used.

```
9   CompiledMethod>>#spotterForBytecodesFor:
        aStep
10    <spotterOrder: 15>
11    aStep listProcessor
12        title: 'Bytecode';
13        allCandidates: [ self symbolicBytecodes ];
14        itemName: #printString;
15        filter: GTFilterSubstrings
```

This extension supports all the aforementioned searches as well as others, such as looking for when a constant is pushed to the stack (Figure 2a) or finding all instructions that access temporary variables (local variables and method parameters; Figure 2b). After finding a bytecode the developer can open it in the inspector or directly spawn the view showing the mapping to source code in the search tool (Figure 2c).

Moldable Debugger extensions.

One cannot easily use source-level debuggers to reason about bytecode. Debugging actions in these debuggers normally skip over multiple bytecode instructions. For example, stepping into a message send with multiple parameters skips over all *push* instructions that place the required parameters on the stack. The same issue arises when developing compiler plugins that alter the default bytecode generated for a given instructions (*e.g.*, slots can generate custom bytecode for reading and writing object attributes [30]).

As a use case illustrating this problem consider a class that uses a boolean slot. A boolean slot occupies a single bit of a (hidden) integer slot that is shared by all classes of a single hierarchy. If multiple classes within the same hierarchy introduce boolean slots, they will be efficiently mapped to this shared integer slot. This is however transparent to users who can use the attribute normally (in the method from Figure 3a isHorizontal is defined as a boolean slot). Transparency is useful when using slots, however, when debugging the actual slot objects one needs access to the actual bytecode generated by the slot. This is not available in a generic debugger.

To facilitate bytecode debugging, in this and other situations, we developed an extension to the Pharo debugger that gives direct access to the bytecode and supports stepping through the execution of a program one bytecode instruction at a time. Creating a custom debugger from the Moldable Debugger is not as straightforward as in the case of the previous two tools. In the current implementation extensions are created by subclassing predefined classes for customizing the user interface and logic of the debugger. For this extension we need to create a custom user interface by subclassing DebuggingView and a custom debugging action by creating a subclass of DebugAction (Figure 4). The total cost of this extension is 200 lines of code. The same debugging scenario is shown in Figure 3b using this extension. Now the developer can see and interact with the actual bytecode generated by the boolean slot.

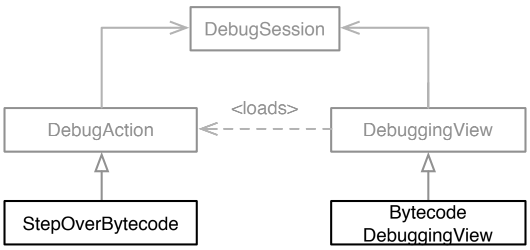


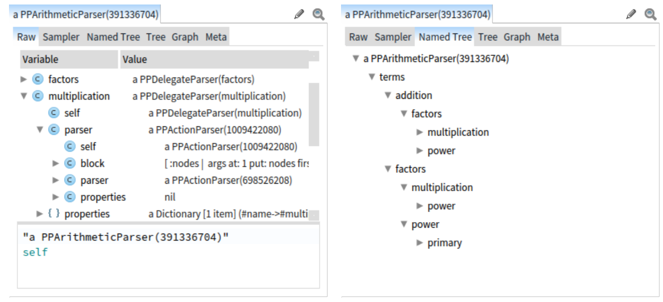
Figure 4: The structure of a custom extension for debugging bytecode. The extension provides a custom user interface and a debugging action for stepping over bytecode instructions. DebugSession provide basic functionality for supporting debugging and does not need to be extended.

3.2 PetitParser

PetitParser is a framework for creating parsers, written in Pharo, that makes it easy to dynamically reuse, compose, transform and extend grammars [22]. A parser is created by specifying a set of grammar productions in one or more dedicated classes. To specify a grammar production a developer needs to: (i) create a method that constructs and returns a parser object for that part of the grammar; (ii) define in the same class an attribute having the same name as the method.

PetitParser is a framework meant to be used by many developers, other than just its creators, to specify parsers. As the specification of a parser consists of classes and methods, parsers can be developed only using generic development tools, like code editors and debuggers. This raises some problems: the specification of the parser is used to instantiate at run-time a *tree* of primitive parsers (*e.g.*, choice, sequence, negation); this tree is then used to parse the input. Developers debugging a parser need to manually link primitive parsers with the grammar production that generated them. Also adding, renaming and removing productions requires working with both a method and an attribute having the same name.

To ease the creation of parsers the PetitParser developers initially built a custom code editor that allowed the creators of a parser to only work in terms of grammar productions instead of attributes and methods. This only covers part of the problem. To further improve the development and debugging of parsers we created together with the current maintainers of PetitParser extensions for several other development tools. These development tools were built after the release of PetitParser, during its maintenance cycles.



(a) (b)

Figure 5: Using Moldable Inspector to visualize a parser object: *(a)* The *Raw* view shows how the parser is implemented; *(b)* The *Named tree* view shows only the structure of the grammar using a tree view.

Moldable Inspector extensions.

As previously mentioned actual parsers are instantiated as objects. Viewing these objects using a generic object inspector only shows how they are implemented and gives no immediate insight into the structure of the parser. For example in Figure 5a, showing the attributes of a parser for arithmetic expressions, it is not obvious how to see the grammar structure. To provide this information directly in the inspector we extended the inspector with several views that show the tree structure of the grammar using different representations. Figure 5b contains a view showing the structure of the grammar using a tree.

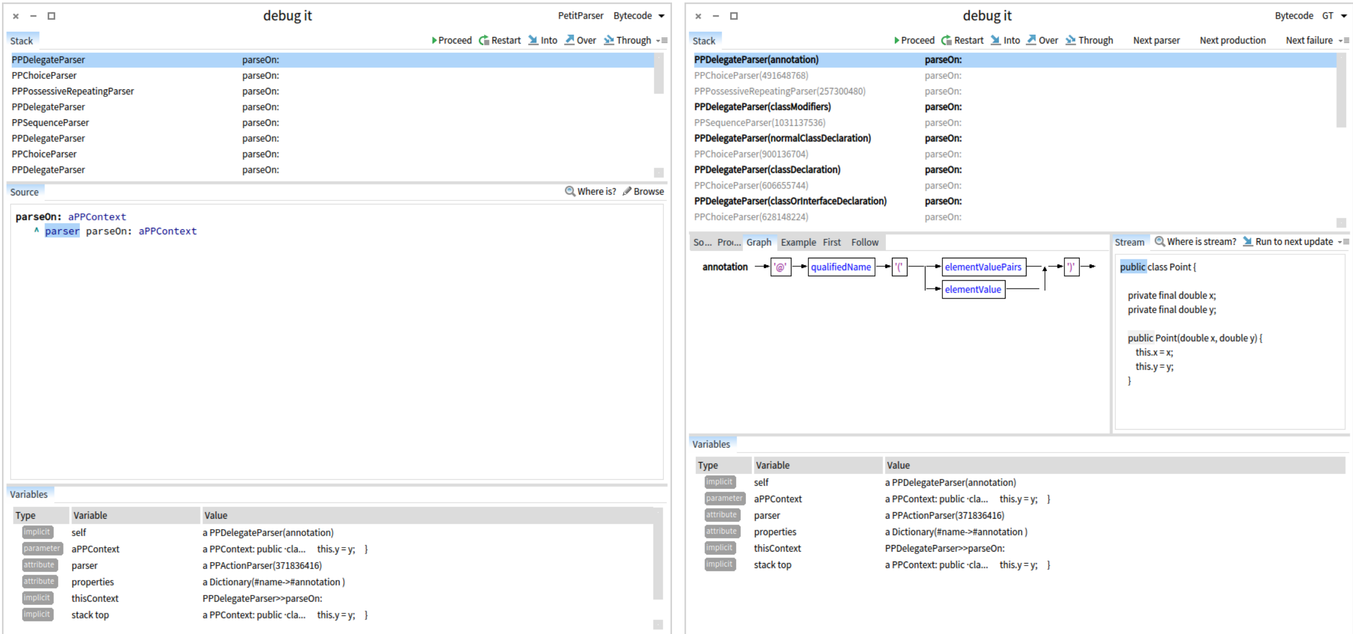
Figure 6: Using Moldable Spotter to search for productions containing the word ‘hex’ in a parser for Java code.

Moldable Spotter extensions.

Parser classes can contain also other methods and attributes apart from those used to model grammar productions. When using method or attribute search to look for a production these extra methods and attributes can return unrelated results. To avoid this we extended the search infrastructure from Pharo with searches that work at the level of grammar productions. Figure 6 illustrates a scenario in which a developer searches in a parser for Java code for productions that contain the word ‘hex’.

Moldable Debugger extensions.

Debugging a parser using a generic debugger also poses



(a) (b)

Figure 7: Debugging a parser: *(a)* A generic debugger has no knowledge about parsing and cannot provide information and debugging actions related to the parser grammar or the input stream; *(b)* An extension for the Moldable Debugger showing parsing related information and providing debugging actions at the level of the grammar and the input stream.

challenges. One the one hand generic debuggers only provide debugging actions and breakpoints at the level of source-code instructions (*e.g.*, step over instruction). On the other hand, they neither display the source code of grammar productions nor do they provide easy access to the input being parsed. This is evident in Figure 7a, which shows a debugger opened on a parser object for Java code: the stack trace shows only parseOn: methods belonging to primitive parsers; to determine how much parsing has advanced one needs to use the inspector to locate the input stream and the current position in the stream and manually determine the character corresponding to that position in the stream.

To overcome these issues, other tools for working with parser generators provide dedicated domain-specific debuggers (*e.g.*, ANTLR Studio, an IDE for the ANTLR [1]). In the case of PetitParser we developed a custom extension for the debugger (Figure 7b). First, this extension offers debugging operations at the level of the input (*e.g.*, setting a breakpoint when a certain part of the input is parsed) and of the grammar (*e.g.*, setting a breakpoint when a grammar production is exercised). Second, it provides a dedicated user interface for the debugger that highlights the grammar productions in the stack, shows information about a selected grammar production (*e.g.*, source code, visual representation), shows the input being parsed, and highlights how much parsing has advanced in the input stream. Creating these extensions followed the same approach as for Opal. We further reused several custom extensions already provided by PetitParser, like a graph view showing the structure of a grammar production (Figure 7b), and functions for computing the first and follow sets for a grammar production. In the end this debugger required 600 lines of code (excluding the aforementioned extensions). It required more code than for the Opal debugger mainly because it provides more custom debugging actions.

4. DISCUSSION

4.1 Applicability

Section 3 presented two examples showing how to improve reasoning about applications by extending development tools. The chosen examples cover a compiler and a parser. Apart from them we also applied moldable development to other types of applications from various domains: Glamour [6] (a framework for creating data browsers based on ideas from reactive programming), FileSystem [5] (a library for interacting with file systems), MessageTally (a library for profiling code), Metacello [5] (a package management system), *etc.* Developers of other libraries related to Pharo also started to create and provide extensions, especially for the Moldable Inspector and Moldable Spotter, as part of their releases. Examples include Zinc[7], a framework to deal with the HTTP networking protocol, and Roassal [2], an engine for scripting visualizations.

Section 3 also shows that by adapting development tools comprehension can be improve from multiple perspectives: in the case of Opal, improving tools helps the developers of

[7] http://zn.stfx.eu

Opal to better reason about their system and fix bugs faster; in the case of PetitParser, improving tools does not directly help the developers of PetitParser itself but rather developers that use it to create and evolve parsers.

For the two examples presented in this paper, as well as in other situations, as discussed in Section 2.1, developers do build tools to help them in their activities. Nevertheless, in many cases these tools are being built and used outside of the main developing environment. Through moldable development we aim to encourage developers to bring tool development into their environment.

Currently we are using Pharo as an environment for exploring tool building, however, there is no conceptual limitation that ties moldable development and moldable tools to Pharo. Indeed, Pharo offers expressive introspection capabilities that simplify the creation of tools like debuggers and inspectors. Nevertheless, we anticipate no technical limitations that would make it difficult to provide moldable tools for other programming languages and IDEs. Mainstream IDEs, like Eclipse or IntelliJ, provide multiple customization possibilities (*e.g.*, plug-ins, extensions and extension points, perspectives) that can be leveraged to support moldable tools.

4.2 Future Challenges

Until now we explored moldable development by investigating how to incorporate domain concepts into several development tools. Nevertheless, these development tools do not live in isolation, but are integrated in an IDE. IDEs contain many other tools that need to interact and work together. As more tools offer the possibility to create extensions synchronization is needed between extension from multiple tools. This raises the need for a *moldable environment* that can adapt tools to domains in a uniform and consistent way.

Moldable development is based on developers evolving tools during the software development process. Hence, as the application evolves, changes in the application can lead to changes in the created tools. This requires a more thorough methodology to keep domain-specific extensions synchronized with the actual applications.

5. RELATED WORK

There exists a large body of research that investigates tool building with the aim of improving program comprehension. In this section we restrict our focus to approaches that target development tools. We classify and discuss related work in this area based on how development tools are created.

5.1 Automatic Generation of Tools

Early examples of adapting development tools consisted in generating projectional editors based on a language specifications (*e.g.*, ALOE [19], The Synthesizer Generator [23]). They were followed by more complex development environments targeting a wider range of language specifications. The Gandalf project, for example, extends ALOE with support for version control with the goal of *"permitting environment designers to generate families of software development environments semiautomatically without excessive cost"* [14]. Meta Environment [16] generates editors and TIDE [29] debuggers for languages defined using ASF+SDF. LISA [15] generates a wide range of tools for visualizing program structures and animating algorithms for languages defined using attribute grammars. The Xtext[8] project from Eclipse can generate complex text editors, while MPS[9] provides dedicated projectional editors. These examples cover only a small part of solutions that generate development tools from a formal language specification. Unlike them moldable development focuses on the creation of tools where a language specification is missing. An example is object-oriented programming where applications can be expressed in terms of an object model that does not require a formal specification.

Based on these solutions for building development tools, several approached were proposed that focus on improving program comprehension by improving the language. An example is *Generic Tools, Specific Languages* [31]. This approach focuses on first creating domain-specific languages for an application and then adapting development tools to those languages. *mbeddr*[10], an extensible set of integrated languages for embedded software development, is an instantiation of this approach that supports tools like projectional editors [32] and debuggers [21]. Another example is *Helvetia*, an extensible system for embedding language extensions into an existing host language. Helvetia enables extensions of the syntax of the host language in a way that does not break development tools, like debuggers and compilers. These approaches aim to improve program comprehension by first improving the programming language and then adapting development tools to those languages. Moldable development focuses on improving the tool rather than the language.

5.2 Manual Creation of Tools

Apart from automatic generation of development tools, a different direction consists in enabling developers to manually adapt development tools or create new ones. This direction can be found in modern IDEs, like Eclipse, IntelliJ or VisualStudio. They enable developers to customize their functionality using plug-ins, however, developing a new plugin is not a straightforward activity. Through moldable tools we aim to significantly reduce this effort.

To support the creation of sophisticated development tools, OmniBrowser [4] relies on an explicit meta-model. To create a new development tool, developers need to specify the domain model of the tool as a graph and indicate the navigation paths through the graph. JQuery [10] supports the creation of various code browsers through a declarative language that extracts and groups code related data. A visual approach to building development tools is proposed by Taeumel *et al.* [28]: developers create new tools by visually combining concise scripts that extract, transform and display data. They show that their solution supports the creation of tools like

[8] http://www.eclipse.org/Xtext/
[9] https://www.jetbrains.com/mps/
[10] http://mbeddr.com/

code editors and debuggers with a low effort. Like moldable tools, these approaches also promote the creation of custom development tools to improve comprehension.

6. CONCLUSIONS

In this paper we argued that one solution for reducing the reliance on code reading during program comprehension is to enable developers to evolve their development tools alongside their applications. We proposed *moldable development* as an approach for achieving this goal. Through two use cases we showed that program comprehension can be improved if developers persevere in customizing their tools.

Moldable development poses a predicament as developers have to invest time and effort in customizing development tools. Nevertheless, this can make considerable economical sense, if the cost of adapting tools outweighs the effort required to reason about applications using generic and disconnected tools. Through moldable tools we showed that the cost can be low even when adapting complex tools.

Moldable development also raises challenges related to how to enable meaningful customizations in development tools, how to better incorporate it in the software development cycle and how to design complete integrated development environments that support this approach, rather than just individual tools. We are actively pursuing these challenges by analyzing how developers use and extend moldable tools [18] and exploring how to better incorporate visualizations into tools [13].

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). We also thank Claudio Corrodi for his corrections and improvements to the final draft.

7. REFERENCES

[1] ANTLR – debugging ANTLR grammars using ANTLR Studio, accessed June 3, 2016. http://www.placidsystems.com/articles/article-debugging/usingdebugger.htm.

[2] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, Sept. 2013.

[3] K. Beck. Simple Smalltalk testing: With patterns. www.xprogramming.com/testfram.htm.

[4] A. Bergel, S. Ducasse, C. Putney, and R. Wuyts. Creating sophisticated development tools with OmniBrowser. *Journal of Computer Languages, Systems and Structures*, 34(2-3):109–129, 2008.

[5] N. Bouraqadi, L. Fabresse, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. Sockets. In *Deep Into Pharo*, page 21. Square Bracket Associates, Sept. 2013.

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

[7] 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. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[8] A. Chiş, T. Gîrba, and O. Nierstrasz. Towards moldable development tools. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '15, pages 25–26, New York, NY, USA, 2015. ACM.

[9] 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*, Onward! 2015, pages 44–60, New York, NY, USA, 2015. ACM.

[10] K. De Volder. JQuery: A generic code browser with a declarative configuration language. In *Proceedings of the 8th International Conference on Practical Aspects of Declarative Languages*, PADL'06, pages 88–102, Berlin, Heidelberg, 2006. Springer-Verlag.

[11] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[12] R. K. Fjeldstad and W. T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. In *Proceedings GUIDE 48*, Apr. 1983.

[13] T. Gîrba and A. Chiş. Pervasive Software Visualizations. In *Proceedings of 3rd IEEE Working Conference on Software Visualization*, VISSOFT'15, pages 1–5. IEEE, Sept. 2015.

[14] A. N. Habermann and D. Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[15] 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* -, 152(2):54–69, 2005.

[16] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, 1993.

[17] A. Ko, B. Myers, M. Coblenz, and H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971 –987, Dec. 2006.

[18] 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 *Proceedings of 3rd IEEE Working Conference on Software Visualization - New Ideas and Emerging Results*, VISSOFT-NIER'15, pages 145–149. IEEE, Sept. 2015.

[19] R. I. Medina-Mora. *Syntax-directed Editing: Towards Integrated Programming Environments*. Ph.D. thesis, Carnegie Mellon University, 1982. AAI8215892.

[20] R. Minelli, A. M. and, and M. Lanza. I know what you did last summer: An investigation of how developers spend their time. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 25–35, Piscataway, NJ, USA, 2015. IEEE Press.

[21] D. Pavletic, M. Voelter, S. Raza, B. Kolb, and T. Kehrer. Extensible debugger framework for extensible languages. In J. A. de la Puente and T. Vardanega, editors, *Reliable Software Technologies – Ada–Europe 2015*, volume 9111 of *Lecture Notes in Computer Science*, pages 33–49. Springer International Publishing, 2015.

[22] L. Renggli, S. Ducasse, T. Gîrba, and O. Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, pages 1–4, Malaga, Spain, June 2010.

[23] T. Reps and T. Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, Apr. 1984.

[24] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.

[25] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34:434–451, July 2008.

[26] 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.

[27] A. Syrel, A. Chiş, T. Gîrba, J. Kubelka, O. Nierstrasz, and S. Reichhart. Spotter: towards a unified search interface in IDEs. In *Proceedings of the Companion Publication of the 2015 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH Companion 2015, pages 54–55, New York, NY, USA, 2015. ACM.

[28] M. Taeumel, M. Perscheid, B. Steinert, J. Lincke, and R. Hirschfeld. Interleaving of modification and use in data-driven tool development. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 185–200, New York, NY, USA, 2014. ACM.

[29] M. van den Brand, B. Cornelissen, P. Olivier, and J. Vinju. TIDE: A generic debugging framework — tool demonstration —. *Electronic Notes in Theoretical Computer Science*, 141(4):161 – 165, 2005. Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005) Language Descriptions, Tools, and Applications 2005.

[30] T. Verwaest, C. Bruni, M. Lungu, and O. Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 959–972, New York, NY, USA, 2011. ACM.

[31] M. Voelter. *Generic tools, specific languages*. Ph.D. thesis, Delft University of Technology, 2014.

[32] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb. Mbeddr: An extensible C-based programming language and IDE for embedded systems. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 121–140, New York, NY, USA, 2012. ACM.

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

[34] M. Zelkowitz, A. Shaw, and J. Gannon. *Principles of Software Engineering and Design*. Prentice Hall, 1979.