Moldable Tools for Object-oriented Development
(Preprint*)

Andrei Chiş, Tudor Gîrba, Juraj Kubelka, Oscar Nierstrasz, Stefan Reichhart, and Aliaksei Syrel

Abstract Object-oriented programming aims to facilitate navigation between domain concepts and the code that addresses those domains by enabling developers to directly model those domain concepts in the code. To make informed decisions developers then formulate detailed and domain-specific questions about their systems in terms of domain concepts, and use tools to explore available information and answer those questions. Development tools however focus mainly on object-oriented idioms and do not expose or exploit domain concepts constructed on top of object-oriented programming idioms. Analysis tools are typically not tightly integrated with development tools. This has a negative effect on program comprehension, increasing the effort and the time for obtaining answers.

To improve program comprehension we propose to better integrate domain concepts and program comprehension tools into the development environment through *moldable tools*. Moldable tools are development tools that

* Chiş A., Gîrba T., Kubelka J., Nierstrasz O., Reichhart S., Syrel A. (2017) Moldable Tools for Object-Oriented Development. In: Mazzara M., Meyer B. (eds) Present and Ulterior Software Engineering. Springer, Cham DOI: 10.1007/978-3-319-67425-4_6

Andrei Chiş
Software Composition Group, University of Bern, Switzerland, scg.unibe.ch

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

are aware of the current development context and support inexpensive creation of domain-specific extensions. We elaborate on the idea of moldable tools and show how to apply moldable tools to support object-oriented programming. Through practical examples we show how developers can embed domain concepts into their development tools.

1 Introduction

Software applications express domain models from the real world as executable models (*i.e.*, programs) within the design space of a language. Program comprehension then requires developers to navigate between domain concepts and the code that addresses those domains. In the procedural paradigm domain concepts are spread throughout procedures and data structures, potentially making the recovery of domain concepts a tedious activity.

Object-oriented programming promised to addressed this problem by providing developers with better mechanisms to construct domain abstractions. Instead of using procedures and data structures developers can model a domain in terms of objects and object interactions (*i.e.*, message sends). Objects model domain entities and encapsulate the state of those entities together with their behaviour. Despite its early promise to solve the problem of navigating between domain concepts and their implementations developers still struggle to navigate between these worlds in object-oriented applications. Furthermore, a wide range of analyses and program comprehension tools have been proposed to aid developers in performing this navigation. Nevertheless, in spite of an ever-increasing number of program comprehension tools, these tools are still heavily underused [33]. Instead, developers mostly rely on code reading as their main technique 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: reading one hundred thousand lines of code takes more than one-man month of work.

In previous work we explored this aspect and argued that the lack of proper environments for developing object-oriented applications is an important factor that makes navigation between code and domain concepts an ever-present problem [26]. Two issues with current development environments that contribute to the problem are that they disconnect development and program comprehension tools, and they focus on generic rather than tailored development tools.

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 craft-

ing 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, current mainstream IDEs 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. This requires developers to manually bridge these two types of tools.

Generic tools. While addressing a specific task, many development tools do not make any assumptions about the actual *contexts* in which they are used. They handle in the same way different software systems, even if those systems model different domains. This means that they are unaware of application or technical domain concepts. For example, a generic object inspector handles all run-time objects in an identical manner. On the one hand, this increases its range of applicability; on the other hand, it makes it 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 [34]. 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. This has the potential to reduce code reading and improve program comprehension as tailored tools can directly provide developers with domain-specific information that they would otherwise need to find during development by reading and exploring source code or using external tools. 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, run-time inspectors, search tools) to take into account their actual application domains. To support this activity we introduced *moldable tools* [9]. A moldable tool is a development tool that is aware of the current development context, and supports inexpensive creation of domain-specific extensions using a plug-in approach.

Here we extend our previous work on this topic in which we introduced the idea of moldable development [8]. We expand on the idea of moldable development and explore its applicability to the development of object-oriented applications. Our overall contributions are:

- Discussing challenges for making moldable development practical and proposing an approach to achieve this based on moldable tools;

- Discussing mechanisms for designing moldable tools using object-oriented concepts;
- Exemplifying the creation of moldable tools for object-oriented programming using real world examples of tools and extensions.

2 Moldable development, moldable tools

The key idea behind *moldable development* is that developers adapt their development tools to be aware of the domain behind the applications under development. This scenario 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.* [45] 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, even if this requires significant effort. When studying homegrown tools in a large software company, Smith *et al.* [35] also observed that developers take the initiative to build tools to solve problems they face, especially when their organization's 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 such tools so that they:

- support inexpensive creation of domain-specific extensions, and
- 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.

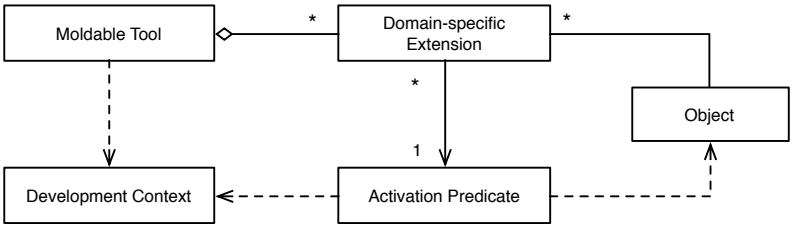


Fig. 1 Moldable tools for object-oriented programming: extensions are attached to objects; activation predicates detect applicable extensions.

2.2 Moldable Tools in a Nutshell

When looking at software development and evolution we observe that developers use a range of widely accepted tools like code editors, compilers, debuggers, profilers, search tools, version control tools, *etc.* Each tool embodies a design that addresses a certain activity in the software development cycle. To support inexpensive adaptations in these tools while preserving their intended design we proposed *moldable tools* [9]. Moldable tools enable domain-specific adaptations through development tools modeled as frameworks that support domain-specific plug-ins (*i.e.*, extensions). To facilitate discovery of extensions moldable tools allow extension creators to specify together with their extensions an *activation predicate* that captures the development contexts in which that extension is applicable.

By "development context" we intend both the current application domain and previous interactions with the domain. Related work on exploring how developers comprehend software showed that interactions with a domain play an important role in improving program comprehension [20, 25, 19]. For example interaction recording tools, like Mylar [17] and DFlow [23], provide support for automatically building a context as developers interact with development tools and filtering visible information based on that context [24].

Moldable tools enable developers to 'mold' domain abstractions into the tools by creating domain-specific extensions expressing 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.

3 Designing moldable tools

In this section we discuss how to design moldable tools using object-oriented concepts.

3.1 Enabling Domain-specific Extensions

The first step towards enabling moldable tools consists in choosing how to support domain-specific extensions. To propose a solution we start from the observation of Pawson that while objects should encapsulate all relevant behaviour, in the context of business systems, many domain objects are behaviorally-weak [32]: much of the functionality is contained in *'controller'* objects that sit on top of model objects, which in turn provide only basic functionality. To address this issue, Pawson proposed *naked objects* as a way to move towards behaviorally-complete objects where business actions are encapsulated in the actual domain objects [29].

The same situation arises when development tools need to provide custom behaviour for objects modeling different domain entities. One approach consists in designing development tools that encapsulate themselves the logic for how to handle domain-specific objects. On the one hand, this decouples the business logic from the logic used to handle objects in development tools. On the other hand, this decoupling can result in duplicated functionality between tools or the need to evolve objects and tools separately as requirements change.

Following the idea of Pawson a second approach consists in making objects responsible for deciding how they are handled in development tools. This allows different tools to reuse the same behaviour and enables a closer evolution of objects and tools. A common use for this approach is to visualize objects: in most object-oriented languages it is the responsibility of an object to represent itself in a textual way (*e.g.*, *toString* in Java, *printString* in Smalltalk). Development tools that need a textual representation of an object just ask that object for its representation.

Moldable tools for object-oriented programming build on the second approach and enable customization by asking objects to indicate the desired behaviour (Figure 1). Hence, objects become behaviorally-complete with regard to development tools, not only to the business domain. This reduces the distance between tools and objects and encourages application developers to evolve domain-specific tools together with their objects.

3.2 Specifying Domain-specific Extensions

The second step in applying moldable tools to object-oriented programming consists in choosing how to specify custom extensions. Researchers have explored multiple alternatives for tool building, including, but not limited to, internal DSLs, external DSLs, logical programming languages, and formal specification languages. Following the same line of reasoning as in the previous section, object-oriented programming already provides a modeling language in terms of objects. Therefore, we propose that moldable tools for

object-oriented programming enable the creation of domain-specific extensions using the underlying object-oriented language of the target application. Hence, developers do not have to learn a new programming language to be able to extend their tools. Moldable tools then model domain-specific extensions as objects. Developers specify an extension by creating and configuring an object using its API (*i.e.*, an internal DSL). This solution favors a design in which developers create custom extensions by using snippets of code to configure those extensions. Related work focusing on similar ideas indicates that this reduces the cost of creating extensions [27, 39].

3.3 Context-aware Extensions

Domain objects provide one axis for selecting domain-specific extensions: a moldable tool selects extensions for those domain objects currently investigated in that tool. Each extension object has an activation predicate specified when the extension is created. Activation predicates can determine if the extension is applicable or not based on the state of its associated object, the state of other accessible objects from the domain model and previous developer interactions with the domain.

For example, in many graphical frameworks, graphical widgets are organized in a tree structure. Extensions for navigating or searching through the structure of sub-widgets can have an activation predicate checking if the widget has sub-widgets. Also when navigating from a parent widget to one of its sub-widgets an inspection tool can show a view highlighting how the sub-widget is positioned within the parent widget. This view can only be made available when navigating from a parent widget to a sub-widget, and not when inspecting a sub-widget in isolation.

4 Addressing Domain-specific Problems

To investigate the usefulness and practical applicability of moldable tools in the context of object-oriented programming we focus on three activities performed by developers during software development and maintenance, namely: *(i)* reasoning about run-time objects, *(ii)* searching for domain-specific artifacts, and *(iii)* reasoning about run-time behavior. We selected them as they are pervasive, challenging and time-consuming activities during software development and maintenance. For each one we explore how relevant problems

can be addressed if developers are able to adapt their development tools to their own contextual needs.[2]

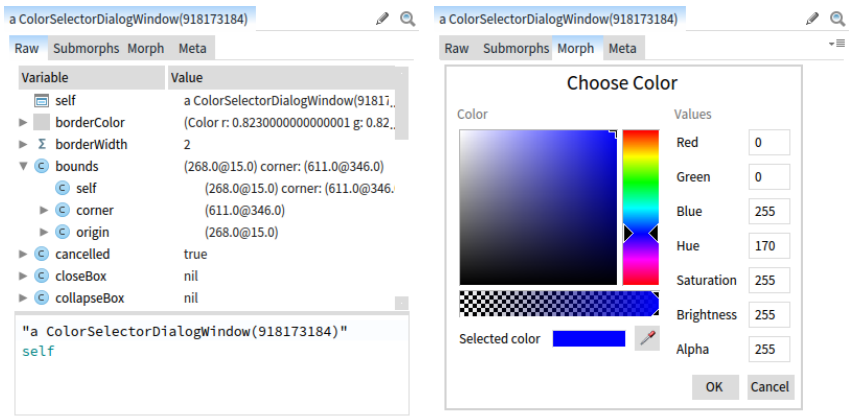
4.1 Reasoning about run-time objects

Since objects only exist at run time, understanding object-oriented applications entails the comprehension of run-time objects. Object inspectors are an essential category of tools that allow developers to perform this task. An object inspector provides a simple interface that allows a user to inspect all the fields of an arbitrary object, and recursively dive into those fields. To better understand what software developers expect from an object inspector we performed an exploratory investigation. We identified the need for object inspectors that support different high-level ways to visualize and explore objects, depending on both the object and the current developer need [10]. Traditional object inspectors however favor a generic view that focuses on the low-level details of the state of single 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.

To address this issue this we introduced the *Moldable Inspector*. The essence of the Moldable Inspector is that it enables developers to answer high-level, domain-specific questions by allowing them to adapt (*i.e.*, mold) the whole inspection process to suit their immediate needs. To make this possible, instead of a single generic view for an object, the Moldable Inspector allows each object to represent itself using multiple domain-specific views, facilitates the creation and integration of new views and uses the current development context to automatically find, at run-time, appropriate views [10]. To validate this approach we created until now, together with the developers of several frameworks and libraries, more than 131 custom views for 84 types of objects belonging to 15 applications, requiring, on average, 9.2 lines of code per view.

For example, depending on her task, a developer working with a widget object may need to explore both the internal structure of that object (*e.g.*, state) and its visual representation. We can address this by attaching to a widget object two views that directly show its state (Figure 2a) and its visual representation (Figure 2b). Now the Moldable Inspector can show both these views when a developer interacts with a widget object (Figure 2).

[2] The three covered 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`).



(a) (b)

Fig. 2 Using the Moldable Inspector to explore a widget object: *(a)* The *Raw* view shows the implementation of the widget; *(b)* The *Morph* view shows the visual representation of the widget.

4.2 Searching for domain-specific abstractions

Software systems consist of many different kinds of domain-specific and interrelated software entities. Search tools aim to support developers in rapidly identifying or locating those entities of interest. Nevertheless, our analysis of mainstream IDEs and current exploration approaches shows that they support searching through generic search tools that are not well-integrated into the IDE. In particular current IDEs decouple tools for searching through code, external data and run-time objects. This impedes search tasks over domain-specific entities as considerable effort is wasted by developers to recover and link data and concepts relevant to their application domains. Furthermore, this limits discoverability as one has to be aware of a domain abstraction in order to know what to look for.

For example, a web server can rely on XML files to model descriptors for web services and store security roles (*e.g.*, admin, employer, manager). In an IDE providing just generic searches, locating a server's descriptor requires a developers to use a file search. Finding what web services use a given security role requires then multiple textual searches through XML files. A domain-specific search tool can enable developers to directly locate descriptor files and search through the web services that use a given security role.

To address this problem we propose that search tools directly enable developers to discover and search through domain concepts. We have proposed the *Moldable Spotter*, a search tool that allows objects to express multiple custom searches through the data that they encapsulate [38]. Moldable Spotter further enables developers to easily create custom searches for their domain

objects and automatically discover searches for domain object as they are interacting with those objects. We also show that by taking into account generic searches through code we can provide a single entry point for embedding search support within IDEs. Based on 124 search extensions for 41 types of objects currently present in the Pharo IDE the average cost of creating a custom search extension is, just like in the case of the Moldable Inspector, 9.2 lines of code.

4.3 Moldable Debugger

Debuggers are essential for reasoning about the run-time behaviour of software systems. 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. This mismatch creates an abstraction gap between the debugging needs and the debugging support leading to an inefficient and error-prone debugging effort, as developers need to recover concrete domain concepts using generic mechanisms.

To address this abstraction gap we have proposed 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. A domain-specific debugger is attached to an object modeling the run-time stack. Hence, stack objects are allowed to express how developers interact with them. We 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.*, it affects the entire debugger).

5 Adapting tools to domain objects

To exemplify how moldable development can be applied during the development of an application to adapt moldable tools to specific domains we present two concrete use cases. We selected two frameworks, the Opal compiler (Section 5.1) and PetitParser (Section 5.2), given that their developers or maintainers created custom extensions for moldable tools as they evolved them. These are also two mature frameworks that cover two different domains, each with its own challenges. Opal requires users to reason about and navigate between multiple representations of source code (*e.g.*, source code,

AST nodes, IR, bytecode). If an error occurs during parsing, PetitParser users need to understand how the parsing consumed the input and locate that part of the input (or the grammar) that caused the error; often the method where the error is raised is different from the method where the parser actually made an erroneous decision.

5.1 Opal Compiler

Opal[3] is a compiler infrastructure focusing on customizability that has been part of Pharo[4] since the Pharo 3 release (May 2014).[5] 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.[6] Debugging such 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 3a 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 attached several custom views to CompiledMethod and SymbolicBytecode ob-

[3] `http://www.smalltalkhub.com/#!/~Pharo/Opal`

[4] `http://pharo.org`

[5] `http://pharo.org/news/pharo-3.0-released`

[6] `pharo.fogbugz.com/f/cases/14606`, `pharo.fogbugz.com/f/cases/12887`, `pharo.fogbugz.com/f/cases/13260`, `pharo.fogbugz.com/f/cases/15174`

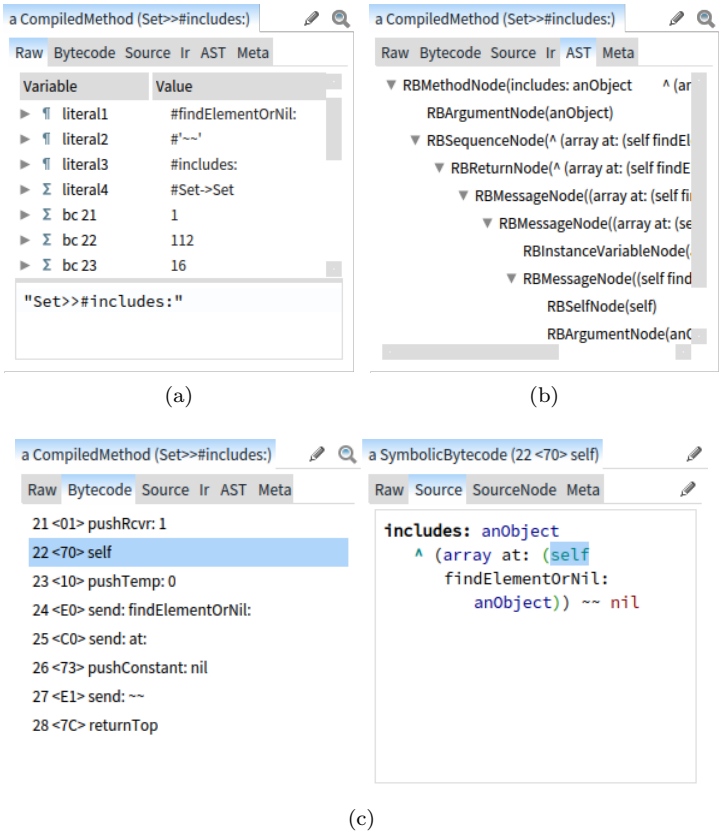


Fig. 3 Using the Moldable Inspector to visualize compiled code: *(a)* The *Raw* view shows the implementation of the object; *(b)* The *AST* view shows the AST from which the code was compiled; *(c)* The mapping between bytecode and source code can be explored by selecting bytecodes in the *Bytecode* view.

jects: a human-friendly representation of the bytecode (Figure 3c, left side), the source code of the method, the AST (Figure 3b), and the IR. Using the bytecode view, the developer can see that the bytecode at index 22 corresponds to pushing self[7] onto 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 code corresponding to the selected bytecode (Figure 3c); 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

[7] self represents the object that received the current message; this in Java.

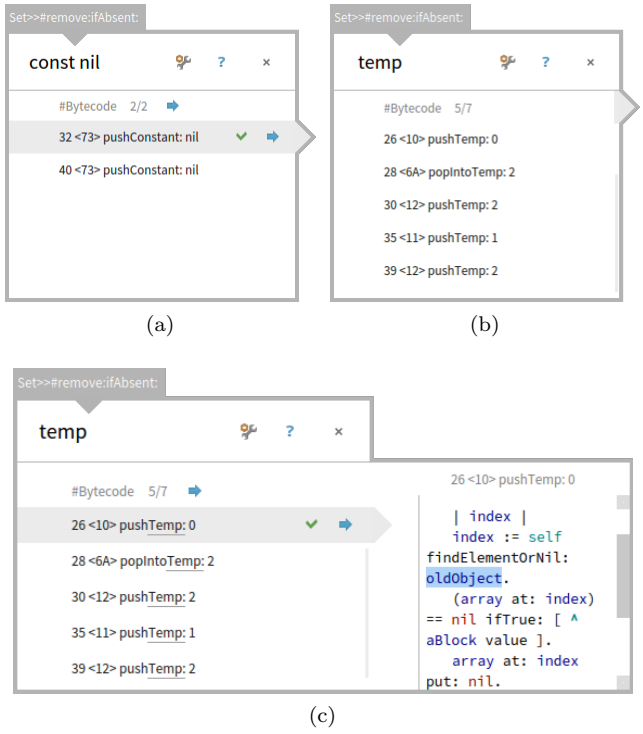


Fig. 4 Searching through bytecode using Moldable Spotter: *(a)* searching for accesses to nil; *(b)* searching for instructions accessing temporary variables; *(c)* When selecting a bytecode the mapping with the source code is shown.

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 them with a predefined annotation. For example, lines 1–8 show the code for creating the AST view displayed in Figure 3b. Using this approach development tools can ask objects for their graphical representations.

```
1  gtInspectorASTIn: aComposite
2    <gtInspectorPresentationOrder: 35>
3    aComposite tree
4        rootsExpanded;
5        title: 'AST';
6        display: [ :aMethod | aMethod 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), message sends (*e.g.*, send: printString), or accesses to 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 attached to CompiledMethod objects a custom search (lines 9–15) through the human-friendly representation of bytecode previously introduced. 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. As a result, development tools can also ask objects which domain-specific searches they support.

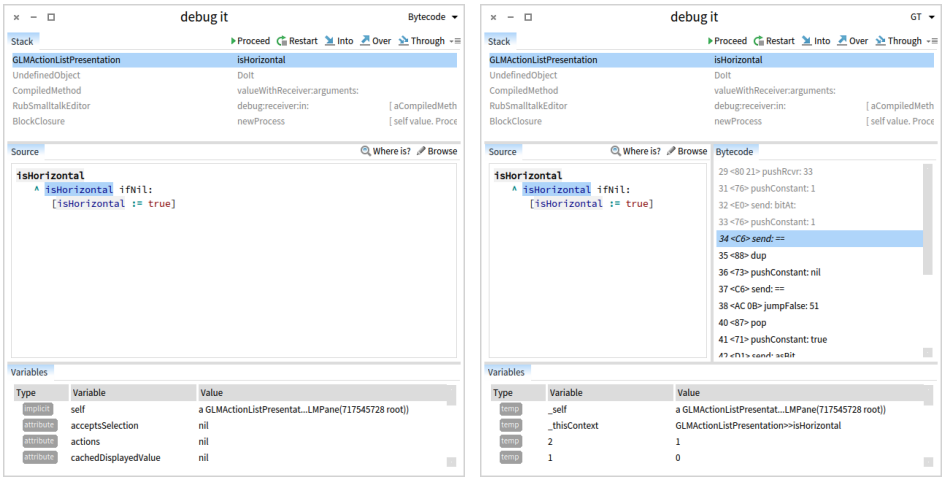
```
9   spotterForBytecodesFor: aStep
10    <spotterOrder: 15>
11    aStep listProcessor
12        title: 'Bytecode';
13        allCandidates: [ self symbolicBytecodes ];
14        itemName: #printString;
15        filter: GTFilterSubstrings
```

This extension supports all the aforementioned searches as well as others, such as looking for when a constant is pushed to the stack (Figure 4a) or finding all instructions that access temporary variables (local variables and method parameters; Figure 4b). 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 4c).

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 [42]).

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 5a isHorizontal is defined as a boolean slot). Although transparency is useful when using slots, when de-



(a) (b)

Fig. 5 Debugging a boolean slot: *(a)* Developers cannot use a generic debugger to access the bytecode generated by the boolean slot; *(b)* An extension to the debugger 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.

bugging the actual slot objects, one needs access to the 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 needed to create a custom user interface by subclassing DebuggingView, and a custom debugging action by creating a subclass of DebugAction (Figure 6). The total cost of this extension is 200 lines of code. The same debugging scenario is shown in Figure 5b using this extension. Now the developer can see and interact with the actual bytecode generated by the boolean slot.

5.2 PetitParser

PetitParser is a framework for creating parsers, written in Pharo, that makes it easy to dynamically reuse, compose, transform and extend grammars [30]. 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)

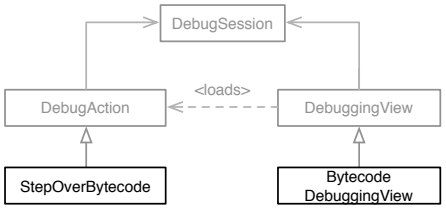
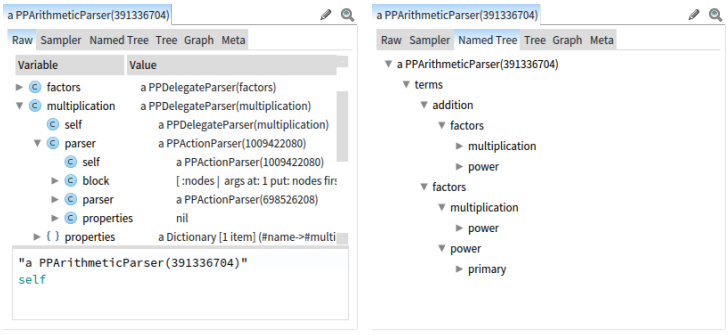


Fig. 6 The structure of a custom extension for debugging bytecode. The extension provides a custom user interface and a debugging action for stepping over bytecode instructions. DebugSession provides basic functionality for supporting debugging and does not need to be extended.



(a) (b)

Fig. 7 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.

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 to the grammar production that generated them. 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 just 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

Fig. 8 Using Moldable Spotter to search for productions containing the word 'hex' in a parser for Java code.

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.

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 7a, showing the attributes of a parser for arithmetic expressions, the structure of the underlying grammar is not clearly evident from the inspected objects. To provide this information directly in the inspector we attached to parser objects views that show the tree structure of the grammar using other representations. Figure 7b contains a view showing the structure of the grammar using a tree list.

Moldable Spotter extensions

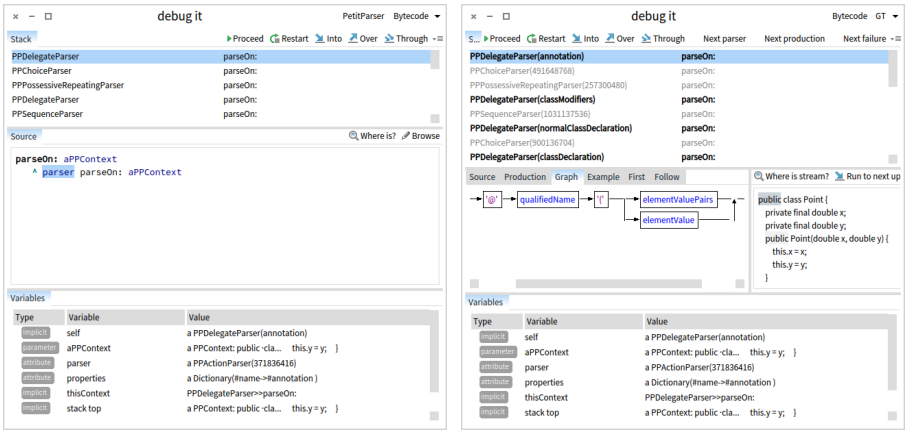
Parser classes can contain also other methods and attributes apart from those used to model grammar productions. When using a method or attribute search to look for a production these extra methods and attributes can return unrelated results. To avoid this we extended the search infrastructure from Pharo by attaching to classes representing parsers, searches that work at the level of grammar productions (lines 16–27). As this search should only be applicable to classes modeling parsers, an activation predicate checks this explicitly (lines 26–27). Attaching this search to an object requires an environment to also model code entities (*e.g.*, methods, classes, annotations) as objects. Figure 8 illustrates a scenario in which a developer searches in a parser for Java code for productions that contain the word 'hex'.

```
16    spotterForProductionsFor: aStep
17    <spotterOrder: 10>
18    aStep listProcessor
19        title: 'Productions';
20        allCandidates: [ :aParserClass |
21            aParserClass productionMethods ];
22        candidatesLimit: 5;
23        itemName: [:aProduction | aProduction selector];
24        filter: GTFilterRegex;
25        itemFilterName: [:aProduction| aProduction selector];
26        when: [ :aParserClass |
27            aParserClass inheritsFrom: PPCompositeParser ];
```

Moldable Debugger extensions

Debugging a parser using a generic debugger also poses 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 9a, which shows a debugger opened on a PetitParser parser 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 then 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 9b). 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. Third, this extension is applicable only when the run-time stack contains a call to a parser object. Creating this extension 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 9b), 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.



(a) (b)

Fig. 9 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.

6 Discussion

In this section we discuss the applicability of moldable tools for other application domains and IDEs, and emphasize future challenges.

6.1 Applicability

Section 5 presented two examples showing how to improve reasoning about applications by extending development tools. The chosen examples cover a compiler and a parser. 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), and 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 of such libraries include Zinc[8], a framework to deal with the HTTP networking protocol, and Roassal [2], an engine for scripting visualizations.

[8] `http://zn.stfx.eu`

Section 5 also shows that by adapting development tools comprehension can be improved from multiple perspectives: in the case of Opal, improving tools helps the developers of 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 here, as well as in other situations, as discussed in Section 3.1, 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 development environment. Through moldable development we aim to encourage developers to adapt development tools to their application domains.

6.2 Moldable Tools in Other Languages

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.

One key requirement for moldable tools is that the application entities that the tools are to become aware of must be modeled as run-time entities. Such entities include not only domain objects, but also other software entities such as packages, classes, methods, annotations, files, source code, bug reports, documentation, examples, repositories, configurations, *etc.* Custom extensions can then be uniformly attached to domain objects, source code entities and external resources. Pharo provides direct support for this requirement as all code entities are modeled as objects. Modern IDEs also provide an object-oriented model for representing code and project related data (e.g., JDT in Eclipse). Hence, we do not view this aspect as a limitation in porting moldable tools to other IDEs.

6.3 Future challenges

Until now we have explored moldable development and moldable tools 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, these extensions will need to be synchronized. 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 moldable 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.

7 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.

7.1 Automatic generation of tools

Early examples of adapting development tools consisted in generating projectional editors based on a language specifications (*e.g.*, ALOE [22], The Synthesizer Generator [31]). 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 [18] generates editors and TIDE [41] generates 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[9] project from Eclipse can generate complex text editors, while MPS[10] 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 through moldable tools focuses on the creation of tools where a language specification is missing. The example discussed here is object-oriented programming where applications can be expressed in terms of an object model that do not require a formal specification.

[9] `http://www.eclipse.org/Xtext/`

[10] `https://www.jetbrains.com/mps/`

Based on these solutions for building development tools, several approaches were proposed that focus on improving program comprehension by improving the language. An example is *Generic Tools, Specific Languages* [43]. This approach focuses on first creating domain-specific languages for an application and then adapting development tools to those languages. An instance of this approach is *mbeddr*[11], an extensible set of integrated languages for embedded software development that supports tools like projectional editors [44] and debuggers [28]. 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.

7.2 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 is at the core of several live environments that blur the line between applications and IDEs. For example, within a Smalltalk-80 [13] system there is no distinction between the application code and the code of the system (*e.g.*, compiler, parser, IDE, development tools). Developers can access and modify the code of the IDE in the same way as they do their own application code. In Self [40], a prototype-based system, developers evolve an application by only interacting with objects. Furthermore, Self draws no system-level distinction between using an application and changing or programming it [36]. Both actions are performed by manipulating the state and behavior of objects. Nevertheless, while these environments do not directly support cheap and context-aware tooling, they provide a solid foundation for tool building.

Modern IDEs, like Eclipse, IntelliJ or VisualStudio, 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. Several text editors, like Emacs and Vi, also have a direct focus on extensibility. Vi is built on the idea of command composability: it provides a set of minimalist commands that can be composed together [16]. Emacs allows developers to cleanly add new commands [37]; unlike Vi it targets more monolithic commands for special-purpose activities.

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

[11] `http://mbeddr.com/`

the navigation paths through the graph. JQuery [11] 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.* [39]: 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 code editors and debuggers with a low effort. Like moldable tools, these approaches also promote the creation of custom development tools to improve comprehension.

8 Conclusions

We have investigated the issue of navigating between domain concepts and their implementation in the context of object-oriented applications. We argued that one solution for improving this navigation and also 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 navigation 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. To reduce the cost of creating domain-specific extensions we proposed *moldable tools*. Moldable tools enable customization through fine-grained plug-ins, and they support developers in locating applicable plug-ins. We showed that by attaching extensions to objects and providing internal APIs for the creation of extensions, 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 them 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 [21] and exploring how to better incorporate visualizations into tools [12].

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.

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, J. Kubelka, O. Nierstrasz, S. Reichhart, and A. Syrel. Exemplifying moldable development. In *Proceedings of the 1st Edition of the Programming Experience Workshop (PX 2016)*, page to appear, 2016.
9. 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.
10. 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.
11. 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.
12. 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.
13. A. Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.
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. W. Joy. An introduction to display editing with Vi. In *In UNIX User's Manual Supplementary Documents,*. USENIX Association, 1980.
17. M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

18. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):176–201, 1993.
19. 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.
20. A. J. Ko, H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 125–135, 2005.
21. 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.
22. R. I. Medina-Mora. *Syntax-directed Editing: Towards Integrated Programming Environments*. Ph.D. thesis, Carnegie Mellon University, 1982. AAI8215892.
23. 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.
24. R. Minelli, A. Mocci, R. Robbes, and M. Lanza. Taming the IDE with fine-grained interaction data. In *Proceedings of ICPC 2016 (*24^{th} *International Conference on Program Comprehension)*, page to appear, 2016.
25. G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 33–48, Berlin, Heidelberg, 2005. Springer-Verlag.
26. O. Nierstrasz. The death of object-oriented programming. In P. Stevens and A. Wasowski, editors, *FASE 2016*, volume 9633 of *LNCS*, pages 3–10. Springer-Verlag, 2016.
27. J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, Mar. 1998.
28. 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.
29. R. Pawson. *Naked Objects*. Ph.D. thesis, Trinity College, Dublin, 2004.
30. 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.
31. T. Reps and T. Teitelbaum. The synthesizer generator. *SIGSOFT Softw. Eng. Notes*, 9(3):42–48, Apr. 1984.
32. A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.
33. 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.
34. 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.
35. 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.
36. 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.

37. R. M. Stallman. Emacs the extensible, customizable self-documenting display editor. *ACM SIGOA Newsletter*, 2(1-2):147–156, Apr. 1981.
38. 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.
39. 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.
40. D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, Dec. 1987.
41. 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.
42. 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.
43. M. Voelter. *Generic tools, specific languages*. Ph.D. thesis, Delft University of Technology, 2014.
44. 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.
45. 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.