Moldable Tools

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Andrei Chiş

von Rumänien

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz

Institut für Informatik

source: https://doi.org/10.24442/boristheses.866 | downloaded: 4.7.2025

Moldable Tools

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Andrei Chiş
von Rumänien

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz

Institut für Informatik

Von der Philosophisch-naturwissenschaftlichen Fakultät
angenommen.

Bern, 19.09.2016

Der Dekan:

Prof. Dr. Gilberto Colangelo

This dissertation can be downloaded from `scg.unibe.ch`.

Copyright ©2016 by Andrei Chiş
`www.andreichis.com`

This work is licensed under the terms of the *Creative Commons Attribution — ShareAlike 2.5 Switzerland* license. The license is available at `http://creativecommons.org/licenses/by-sa/2.5/ch/`

Attribution–ShareAlike

ISBN: 978-1-326-74717-6
First edition, September 2016

Acknowledgments

This work builds on the shoulders of many. Its value goes beyond this writing, as it inadvertently shaped me. I warmly thank every person who directly, or indirectly, contributed to this work. What follows is for sure an incomplete enumeration, both quantitatively and qualitatively.

✻ ✻ ✻

First of all, I would like to express my gratitude to Oscar Nierstrasz for giving me the opportunity to work at the Software Composition Group. I thank him for his patience and dedication in guiding me throughout this parlous journey, on the stormy and previously unknown seas of PhD life.

I am in debt to Tudor Gîrba for believing that I can complete this work when I had many, many doubts. His enthusiasm, lovely chats and inspiring advises helped me overcome limitations that seemed insurmountable.

This academic journey would not have started without Radu Marinescu. He was my first contact with research and, unknowingly, motivated me to take this path. I am grateful for giving me the chance to work with him and for all the long talks about design, life and software visualizations.

I am grateful to Michele Lanza for promptly accepting to be on my PhD committee, and for reviewing this dissertation. I thank Gerhard Jäger for accepting to chair the examination.

I thank Andrea, for our numerous discussions and hikes through the Alps; Haidar, for listening to my crazy ideas and always being ready to go for a swim; Boris, for never saying no to a debate and a beer. Many thanks to Mircea for his support, impromptu coffee breaks and never-ending optimism. I thank Jan for all our chats and sparring sessions in the ring, Nevena for bringing a different perspective to our group discussion around the round table, and Leonel for our short-but-not-so-short talks and traveling advices for my Chilean detour.

I thank all other, current and past members of SCG. I thank Erwann, Fabrizio and Jorge for their advice at the beginning of my PhD, and Claudio, Mohammad, Yuri, Natalia and Olivier for their energy towards the end. I am in debt to Claudio for his help in proofreading and reviewing this dissertation.

I am grateful to my students, Max, Aliaksei, Roger and David, for working with me on fun and sometimes crazy ideas. I am grateful to my *Hilfsassistenten*, Joel, Mario and Mathias, for easing my life during the P2 course.

Thanks to the members of the GT Team: Aliaksei, Juraj, Stefan, and Doru. Without your input and help in implementing moldable tools, this dissertation would not have been possible.

Thanks to Stéphane Ducasse for challenging my ideas and facilitating many visits to Lille. Thanks to Alexandre Bergel for being an eager adopter of moldable tools and a great stay in Chile. Thanks also to all the people in the Pharo and Moose communities for using the tools developed in this dissertation, and generating many ideas and discussions that contributed to the improvement of this work.

Special thanks go to Iris, for her immeasurable help in finding a place to stay when I was a newbie in all things Swiss, dealing with administrative work during the years, and finally preparing the defense of this dissertation.

Thanks to the many friends that I made in Bern, and in particular to all the people that called the 3rd floor in Fellergut home. Thanks to my *old* friends for our never-ending trips and adventures.

Many, many thanks to my family for their love and support!

August 29, 2016
Andrei Chiş

Quand tu veux construire un bateau, ne commence pas par rassembler du bois, couper des planches et distribuer du travail, mais reveille au sein des hommes le desir de la mer grande et large.

(When you want to build a ship, do not begin by gathering wood, cutting boards, and distributing work, but awaken within the heart of man the desire for the vast and endless sea.)

Origin unknown. Possibly derived from a passage by *Antoine de Saint-Exupéry* in the *Citadelle*

Abstract

Development tools are a prerequisite for crafting software. They offer the lenses through which developers perceive and reason about their software systems. Generic development tools, while having a wide range of applicability, ignore the contextual nature of software systems and do not allow developers to directly reason in terms of domain abstractions. Domain-specific development tools, tailored to particular application domains, can address this problem. While it has clear advantages, incorporating domain abstractions into development tools is a challenging activity. The wide range of domains and contextual tasks that development tools need to support leads to costly or ad hoc mechanisms to incorporate and discover domain abstractions. Inherently, this limits developers from taking advantage of domain-specific information during the development and maintenance of their systems.

To overcome this problem, we propose to embed domain abstractions into development tools through the design of moldable tools that support the inexpensive creation of domain-specific extensions capturing domain abstractions, and that automatically select extensions based on the domain model and the developer's interaction with the domain model. This solution aims to reduce the cost of creating extensions. Towards this goal, it provides precise extension points together with internal DSLs for configuring common types of extensions. This solution facilitates automatic discovery by enabling extension creators to specify together with an extension an activation predicate that captures the context in which that extension is applicable.

We validate the moldable tools approach by applying it, in the context of object-oriented applications, to improve three development activities, namely: reasoning about run-time objects, searching for domain-specific artifacts, and reasoning about run-time behavior. For each activity we identify limitations of current tools, show how redesigning those tools following the moldable tools approach addresses the identified limitations, and discuss the cost for creating domain-specific extensions. We demonstrate that moldable tools address an existing need by analyzing the increase in domain-specific extensions after we integrated the moldable tools solving the aforementioned tasks with an IDE. We also show what kinds of custom environments developers can create by continuously adapting their development tools.

Contents

List of Figures

List of Tables

Listings

The conversion of an idea to an artifact, which engages both the designer and the maker, is a complex and subtle process that will always be far closer to art than to science.

Eugene S. Ferguson

1 Introduction

Software is expressed using programming languages and crafted with the aid of tools. Programming languages are frameworks of communication for transferring abstract models of the real world from the human mind to computers [Ingalls 1981]. They evolved to provide developers with a vocabulary that hides the details of the computer, and strive to enable developers to create and to express software applications in terms of domain abstractions. This occurred since expressing software in terms of domain abstractions rather than generic programming language constructs has a positive impact on program comprehension [Littman et al. 1987; Rajlich and Wilde 2002].

Nevertheless, software has no physical shape. Developers craft software exclusively by interacting with development tools. Development tools provide the means for transforming an abstract model of the real world from a human's mind into a program (*i.e.*, executable model), within the design space of a programming language. Hence, development tools have a direct impact on the thinking habits of developers, affecting how they perceive, craft and reason about software [Dijkstra 1972]. Even if developers can design a software application in terms of domain abstractions, to take advantage of domain abstractions when creating and evolving that application their development tools need to incorporate those domain abstractions.

1.1 Development Tools and Program Comprehension

Starting from basic code editors and compilers, software development tools evolved to support specific tasks from all phases of the software development cycle. This includes, but is not limited to, designing, writing, documenting, debugging and testing software. While addressing a specific task from the software development cycle, to increase their range of applicability, many development tools do not make any assumptions about the specific contexts in which developers use them. They handle software applications written in one or more programming languages in the same way, even if those applications model different domains. These types of tools are referred to as *generic*. A generic source code editor for a programming language for example handles all applications written in that language in an identical manner.

Developers, nonetheless, formulate detailed and domain-specific questions related to the applications they are developing and maintaining using abstractions from those application domains. Generic programming tools focusing on generic programming tasks and generic language constructs however are unaware of those domain abstractions. Hence, they force developers to refine their domain-specific questions into low-level generic ones and mentally piece together information from various sources [Sillito et al. 2008]. 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.

As opposed to a generic development tool, a domain-specific development tool tailored to a particular application domain is aware of the abstractions from that domain and can directly answer related questions. While it has clear advantages, adapting development tools to particular domains is a costly and challenging activity. A wide range of software development tools are not designed to support adaptations to specific domains, allow a limited set of adaptations or require a significant cost even for simple changes. This limits the ability of developers to take advantage of domain abstractions. By cost we refer to the effort in terms of time and lines of code, and the expertise required to incorporate domain abstractions into development tools.

Problem Statement:

> *By relying on costly mechanisms to incorporate and discover domain abstractions, software development tools limit the ability of developers to reason directly in terms of domain abstractions.*

The more applications we want to support, the more domains the tools should take into account. The more domains, the more abstractions the tool has to incorporate. The more abstractions the tool has, the higher the effort for developing that tool. The more the domains change, the higher the effort to keep the tool up-to-date.

Research Question:

> *How can we design development tools that reduce the cost for incorporating and managing domain concepts given the wide range of domains and contextual tasks that development tools need to support?*

1.2 Making Development Tools Domain-aware

Current research proposes two diverging directions for making development tools domain-aware: one focuses on adapting the language, the other on adapting the tool. Before going any further we need to discuss these two directions and motivate our choice, that of extending the tool rather than the language.

1.2.1 Generic Tools for Domain-specific Languages

High-level programming languages, such as C, Java or Python, referred to as general-purpose programming languages, lack language constructs specialized for a particular application domain. Developers combine multiple language constructs to model domain abstractions. Generic tools at the level of these languages are not aware of any domain abstractions. To avoid this issue, concepts from an application domain can be directly encoded in a language by creating external domain-specific languages or extending an existing language with new language constructs. Generic development tools at the level of an external domain-specific language or language extension can now directly work with domain abstractions encoded in the language. Voelter promotes this through the approach *Generic Tools, Specific Languages* that aims to *"shift the focus from building and adapting tools [...] to building and adapting languages to a domain"* [Voelter 2014]. Renggli proposed *HELVETIA* [Renggli et al. 2010b] an extensible system for embedding language extensions into an existing host language by extending the syntax of the host language in a way that does not break development tools. This direction embeds domain abstractions into development tools by first embedding those abstractions into the language and then building development tools that work at the level of abstraction of that language.

1.2.2 Domain-specific Tools for General-purpose Languages

Even if general-purpose programming languages do not provide language constructs that have a direct mapping to domain-specific abstractions, they provide developers with mechanisms to construct domain abstractions. For example, object-oriented programing enables developers to model a domain in terms of objects and their interactions; functional programming relies instead on immutable data structures and functional abstractions. Developers can also exploit the syntax of their general-purpose programming language to create readable APIs (*i.e.,* internal domain-specific languages) focusing on particular problems of specific domains [Fowler 2010]. Hence, domain abstractions do not have to always be encoded in a software application through direct language constructs; they can be expressed on top of general-purpose programming languages. Nevertheless, in this direction, tools at the level of general-purpose languages remain agnostic of the application domain. Developers need to explicitly incorporate their domain abstractions into their development tools.

Despite its pervasiveness, this way of modeling software applications is mainly pursued using generic development tools. Part of current development tools do allow developers to make domain-specific adaptations. However, when this is the case, these development tools can require significant effort for creating and discovering meaningful adaptations, or lack mechanisms for managing and evolving adaptations alongside applications. To explore how to address these issues and facilitate development using custom rather than generic tools, we selected the direction of adapting the tool rather than the language as the focus for our dissertation.

1.3 Tool Building

When developers encounter new questions and problems not addressed by their current tools they have the option to adapt their tools to directly handle those particular situations. An investigation by Whittle *et al.* [Whittle et al. 2013] in the context of model-driven engineering showed that many developers build their own tools or make heavy adaptations to off-the-shelf tools. When studying homegrown tools in a large software company, Smith *et al.* [Smith et al. 2015] also observed that developers take the initiative to build tools to solve the problems they face, especially when their organization's culture promotes this activity. Dawson and Straub [Dawson and Straub 2016] describe as a factor contributing to the success of GitHub[1] the fact that its developers had a direct focus on building their own development tools. This shows that developers do build tools to help themselves in their work. However, an important barrier towards creating new tools or adapting off-the-shelf tools is the cost.

1.3.1 Meta-tooling

Research into meta-tooling environments (*i.e.*, tools for building tools) has the potential to address the aforementioned problem by allowing developers to quickly and effectively customize development tools to their own application domains. One type of meta-tooling environment illustrating the benefits of meta-tooling is that of environments for software and data analysis. An example is Moose [Nierstrasz et al. 2005]. Instead of anticipating all types of possible analyses, Moose offers a number of meta-tools for building visualizations, parsers and data browsers that can be adapted to a variety of needs. Other examples include Rascal [Klint et al. 2009], SPOOL [Robitaille et al. 2000], GSEE [Favre 2001], as well as more generic data analysis environments like Wolfram Alpha[2].

Another well known example of an environment that we can consider a meta-tooling environment is the UNIX shell: it provides a set of default tools and focuses on enabling developers to easily create new tools by combining existing tools using an

[1] `http://github.com`

[2] `http://www.wolframalpha.com/`

approach based on data streams and stream operators like pipes and filters [Raymond 2003].

These meta-tooling environments favor the creation of new tools and analyses and are suitable in situations where it is not possible to anticipate what kind of tools and analyses will be needed. Furthermore, these tools and analyses are often built and need to be used outside of a developer's main development environment. However, when looking at software development and evolution developers use a range of widely accepted tools that cover the software lifecycle, like code editors, compilers, debuggers, profilers, search tools, version control tools, *etc.* While improving software development by introducing new types of tools is a valid option, a different approach consists in *(i)* allowing developers to customize existing development tools to their particular needs and *(ii)* moving towards customizations that happen directly in a developer's main development environment. For this to work, tool builders need to design extensible development tools.

1.3.2 Tool Extension

Extensible tools define points (*i.e.*, extension points) that developers can use to adapt the functionality of the tool. This approach provides a pattern that given a relevant set of extension points for extending both the functionality and user interface of a tool, can result in useful domain-specific development tools. The Eclipse Platform[3] is a widespread example of this approach: by providing customization through a flexible mechanism of extensions and extension points it can support the creation of a wide range of development tools [Yang and Jiang 2007], including full-fledged integrated development environments (IDEs) for generic, modeling[4] and domain-specific languages[5].

Extensible development tools enable developers to adapt them to their own domains through domain-specific extensions. However, they also come with a predicament as, to take advantage of extensible tools, developers need to create, discover and evolve extensions alongside their applications. Supporting these aspects through ad-hoc solutions requires an expensive effort for incorporating domain abstractions into tools, discouraging developers to pursue this activity.

Creating and evolving extensions

To benefit from an extensible development tool developers first need to create a domain-specific extension for that tool. A wide range of development tools do support custom extensions. Nevertheless, even if a development tool supports custom extensions, if developers cannot easily and effectively adapt the tool to their domains

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

[4] `http://eclipse.org/modeling/emf`

[5] `http://eclipse.org/Xtext`

they will most likely not do so, or only do it when the high effort can be justified. An integrated development environment like Eclipse, for example, allows developers to customize the environment and install new tools using *plug-ins*, but developing a new plug-in is highly non-trivial. Software applications are also in a continuous evolution to keep up with changes in requirements [Lehman 1980], leading to domain abstractions being added, modified or removed. As this happens, extensions addressing those abstractions must also be detected and updated. This introduces a new issue that an extension creator must take into account.

Discovering extensions

Creating and evolving extensions in an extensible tool is only one half of the problem. The other half consists in facilitating the discovery of domain-specific extensions. Developers cannot take advantage of existing extensions if they cannot easily determine what extensions are applicable for their current task and domain. IDEs can contain hundreds if not thousands of different tools and extensions. Finding the right one for a particular task can be a difficult problem. For example, recent studies pointed out that refactoring tools from IDEs [Murphy-Hill et al. 2009] as well as dedicated comprehension tools [Roehm et al. 2012] are heavily underused. Recommender systems approach this problem by suggesting tools and extensions to a developer based on what other developers used in similar situations. However, if other developers do not know about and use an extension, a recommender system would not be able to find it.

1.4 Our Approach: Tool Extension Through Moldable Tools

Thesis:

> *To reduce the cost for incorporating and managing domain abstractions, software development tools need to support inexpensive creation of domain-specific extensions, and automatically select extensions based on the domain model and the developer's interaction with the domain model.*

We propose to embed domain abstractions into development tools through extensible development tools that approach the creation and evolution of extensions from the perspective of meta-tooling: enable developers to quickly and effectively express domain-specific extensions for a development tool. To facilitate discovery we propose that extension creators specify together with their extensions an *activation predicate* that captures the context in which that extension is applicable.

We refer to software development tools that satisfy our thesis statement as *moldable tools*. Hence, a moldable tool is *a development tool aware of the current application domain and previous interactions with the domain (*i.e., *development context) that enables rapid customization to new development contexts*. Customization is needed as one cannot

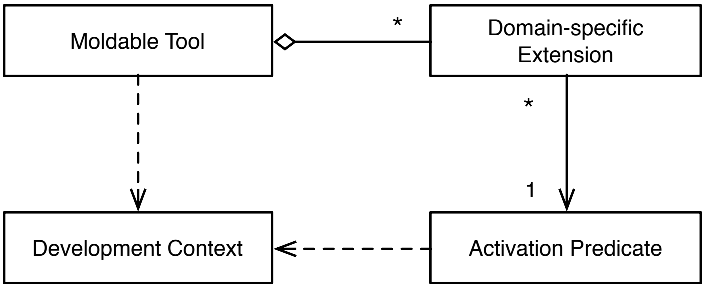


Figure 1.1: An overview of the Moldable Tools approach.

anticipate all relevant usage scenarios; awareness of the development context enables the tool to automatically detect relevant extensions.

In this vision, illustrated in Figure 1.1 (p.7), tool builders first need to design moldable development tools. We propose the following approach for the creation of moldable tools:

Identify common types of extensions Different tools require different types of extensions. We show how various research methods, like interviews, surveys and analyses of related works, can be employed to identify extension opportunities for development tools;

Optimize creation of common extension Moldable tools need to reduce the effort for creating and evolving common types of extensions. We show that one effective mechanism for achieving this goal is internal domain-specific languages (*i.e.*, libraries with dedicated APIs);

Do not limit possible extensions Supporting only common extensions limits the applicability of a moldable tool. We show that this can be avoided by allowing developers to create highly specialized extensions using general-purpose languages, at a higher cost.

Developers then adapt a moldable tool to a development context by *(a)* creating domain-specific extensions for that tool capturing relevant domain abstractions and *(b)* attaching to those extensions activation predicates that capture the development contexts in which those extensions are applicable. Then, at run time, a moldable tool automatically selects extensions applicable in the current development context.

This approach offers the following benefits:

Domain-specific comprehension Code reading is still a widely used solution for program comprehension. 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. By focusing on optimizing the creation of common types of extensions, moldable tools can enable developers to create extensions capturing meaningful data with a low effort. Systematic creation of extensions can improve program comprehension by reducing the time needed to locate domain-specific information in development tools. This can justify the effort for creating domain-specific extensions instead of relying on code reading as a way to obtain the same information.

Integrated comprehension Separating tools for program comprehension and development creates a gap between program comprehension and development, two activities that are deeply intertwined. Moldable tools propose a novel approach to better integrate program comprehension tools into the development environment. Providing domain-specific information directly in development tools can reduce the reliance on external program comprehension tools. Hence, developers do not have to manually integrate data provided by external program comprehension tools into their current development tools.

Context awareness Extension users do not have to manually decide when an extension is appropriate. Extension creators help them by explicitly indicating when an extension is appropriate. This makes it possible for a moldable tool to suggest appropriate extensions for the current development context and minimize manual work for locating applicable extensions.

1.5 Validating the Moldable Tools Approach

To validate the *Moldable Tools* approach and show that domain-abstractions can be embedded into development tools with low costs we apply it in this dissertation to tools for developing object-oriented applications. Object-oriented programming is a good candidate for validating *Moldable Tools*: on the one hand, object-oriented programming uses objects to capture and express a model of the application domain; on the other hand, development tools and environments for object-oriented programming focus only on object-oriented idioms and do not capture domain abstractions constructed on top of object-oriented programming idioms.

To investigate the usefulness and practical applicability of *Moldable Tools* in the context of object-oriented programming in this dissertation 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. Next we discuss limitations of current tools in these areas and present research questions that need to be addressed to overcome these limitations.

Reasoning about run-time objects 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. 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. Traditional object inspectors however favor a generic view that focuses on the low-level details of the state of single objects. While universally applicable, this approach does not take into account the varying needs of developers that could benefit from tailored views and exploration possibilities. Addressing this problem raises the following research question:

> *How can an object inspector provide developers with a unified workflow for exploring multiple objects using views tailored to their own contextual needs?*

Searching for domain-specific abstractions Software systems involve 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 and disconnected search tools. This impedes search tasks over domain-specific entities, as considerable effort is wasted by developers recovering and linking 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. Improving the way searching is integrated in an IDE raises the following research question:

> *How can an IDE enable direct searches through domain abstractions instead of requiring developers to focus on manually recovering those abstractions?*

Reasoning about run-time behavior Debuggers are essential for reasoning about the run-time behavior of software systems. Nevertheless, 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, since developers need to recover concrete domain concepts using generic mechanisms. The following research question arises:

> *How can a debugging infrastructure support developers in debugging their applications at the level of abstraction of the underlying domain model?*

Contributions

The novelty of this dissertation resides in showing that relevant problems encountered by developers during software development and maintenance can be addressed by designing development tools following the *moldable tools* approach. Towards this goal the main contributions of this dissertation are:

1. The concept and specification of *moldable tools*, an approach for improving decision making within development tools by incorporating domain abstractions into development tools though inexpensive domain-aware customizations [Chiş et al. 2015b; 2016a;c].
2. The application of *moldable tools* to address the research questions identified in Section 1.5 (p.8). This shows the benefits of enabling developers to adapt development tools to their own domains [Chiş et al. 2015a;c; 2016b] and validates the usefulness of the *Moldable Tools* approach.
3. The implementation and extension to several domains of three development tools (*i.e.*, debugger, object inspector, search tool) following the *moldable tools* approach [Chiş et al. 2015a;c; 2016b]. This demonstrates that the proposed approach has practical applicability.

The following list details on the contributions of (2) and (3) addressing the research questions identified in Section 1.5 (p.8), which serve as a validation for the *moldable tools* approach:

Reasoning about run-time objects using the 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 that object inspectors enable developers to adapt the inspection workflow to suit their immediate needs. We show that this can be achieved if object inspectors make the inspection context explicit, support multiple domain-specific views for each object and facilitate the creation and integration of new views. Through the *Moldable Inspector* we show how an inspector model providing these feature looks like.

We published this approach in a workshop [Chiş et al. 2014b], a tool demo [Chiş et al. 2015d] and a conference publication [Chiş et al. 2015c]. We implemented the Moldable Inspector model in Pharo[6] as part of the Glamorous Toolkit project[7]. Moldable Inspector is the default object inspector in the Pharo 4 (May 2015) and Pharo 5 (May 2016) releases.

Moldable, context-aware searching with Moldable Spotter. Generic search tools disconnected from each other impede search tasks over domain-specific entities. To address this problem we propose that search tools directly enable developers

[6] http://pharo.org

[7] http://gtoolkit.org

to discover and search through domain concepts by making it possible for developers to easily create custom searches for their domain objects and automatically discover searches for domain objects 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. Through Moldable Spotter we show how to design a search framework following these ideas.

We published this approach in a poster paper [Syrel et al. 2015] and a conference paper [Chiş et al. 2016b]. We implemented Moldable Spotter in Pharo as part of the Glamorous Toolkit project. Moldable Spotter is the default search tool in the Pharo 4 and Pharo 5 releases.

Practical domain-specific debuggers using the Moldable Debugger. We propose to address the abstraction gap between debugging needs and debugging support by allowing developers to create domain-specific debuggers with little effort. We show how this can be supported through a *moldable debugger* that 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 published this approach in a workshop [Chiş et al. 2013], a conference [Chiş et al. 2014a] and a journal publication [Chiş et al. 2015a]. We implemented the Moldable Debugger model in Pharo as part of the Glamorous Toolkit project. Moldable Debugger is the default debugger in the Pharo 5 release.

1.6 Outline

This dissertation is structured as follows:

Chapter 2 – We discuss and compare previous research related to both extending tools and languages, and demonstrate a gap related to approaches for improving development tools that focus on both enabling inexpensive adaptations and behavioral variation dependent on the current context.

Chapter 3 – Starting from the aforementioned analysis of related work we identify requirements for improving tools supporting domain-specific extensions and introduce the *moldable tools* approach for tool extension.

Chapter 4 – We investigate limitations of current object inspectors through interviews and an online survey, identify a need for object inspectors that focus on more than the state of single objects and propose a model for object inspectors that can adapt themselves to both the inspected objects and the immediate developer needs.

Chapter 5 – We discuss requirements for making search tools domain-aware, explore how they are supported in several exploration tools and propose a solution that allows developers to inexpensively incorporate domain concepts in the search process as well as to discover searches applicable for their own contexts.

Chapter 6 – We identify and motivate a set of requirements for enabling domain-specific debugging, and propose a debugging framework that enables developers to create domain-specific debuggers and switch between them at run-time.

Chapter 7 – Allowing developers to create and discover extensions for individual tools creates a basis for improving program comprehension. However this raises further issues related to integrating and evolving extensions for individual tools. In this chapter we discuss how to integrate moldable tools into a coherent environment and how to evolve them alongside applications.

Chapter 8 – We discuss further research directions and conclude this dissertation.

[...] one of the most important aspects of any computing tool is its influence on the thinking habits of those that try to use it, and [...] I have reasons to believe that that influence is many times stronger than is commonly assumed. [Dijkstra 1972]

Edsger W. Dijkstra

2 Enabling Customization

Tools play a prominent role in software development. Since their inception, researchers explored directions for improving software development by improving the tools developers use to craft software. One important direction, which is also the focus of this dissertation, consists in enabling developers to adapt development tools to their own problems and domains. To form a general image of related works, in this chapter we survey and analyze the state of the art. First, we discuss three dimensions that form the foundation of the discussion. Second, we review approaches for generating tools from language specifications that, while not the direct focus of this dissertation, play an important role in this field of research. Third, we review approaches that allow developers to adapt their development tools instead of the language.

2.1 Discussion Dimensions

Researchers have proposed a wide and diverse set of approaches for enabling developers to incorporate domain concepts into their development tools. These approaches range from simple ones targeting adaptations for precise tasks and domains (*e.g.*, profiling object-oriented applications), to heterogeneous solutions that spawn across tasks and domains (*e.g.*, building custom analysis tools). Nevertheless, they all share the same goal of allowing developers to create and work with more than a single custom adaptation. Hence, we start by discussing different solutions for integrating adaptations, and group related works starting from the mechanism used to enable communication between domain-specific adaptations. Numerous dimensions can be further used to evaluate and compare approaches within and between groups. For the purpose of this dissertation we scope the discussion to three dimensions: *(a)* the architectural solution for incorporating new adaptations, *(b)* the cost for creating a domain-specific adaptation (Section 2.1.2 (p.16)) and *(c)* a tool's ability to adapt its behavior based on the development context (Section 2.1.3 (p.18)). We selected them as they are paramount to incorporating and using domain abstractions in development tools.

2.1.1 Types of Tool Integration

The first design decision a tool builder needs to take is what architectural solution to select for incorporating new adaptations into the tool. This is commonly understood as designing the tool as a framework and selecting a plug-in architecture; a plug-in, or a combination of plug-ins, forms a domain-specific adaptation. Regardless of the actual plug-in architecture a tool builder needs to ensure that plug-ins work together to fully support the developer. Hence, plug-in integration becomes an issue. In the more generic context of tool integration Wasserman [Wasserman 1990] and Thomas & Nejmeh [Thomas and Nejmeh 1992] describe several types of tool integration: platform integration, presentation integration, data integration, control integration and process integration. Presentation integration, data integration and control integration are directly relevant also for development tools supporting customization through plug-ins.

Data Integration

Integrating plug-ins requires that they share data and maintain relations between data managed by individual plug-ins. One solution consists in relying on a shared repository: any plug-in can access and modify data produced by any other plug-in. This simplifies the problem of exchanging data between plug-ins, however, it forces all plug-ins to conform to the same data representation. A second solution consists in allowing plug-ins to exchange data through a common interface. This allows for flexibility and enables each plug-in to have its own data representation. The communication interfaces can range from simple character streams to complex data structures. For example, in UNIX, tools take as input a character stream and produce another character stream. While this puts no restriction on the exchanged data, each tool needs to convert its internal data structures to streams of characters. To reduce this effort, other approaches rely on interface description languages for describing the data structures through which tools communicate at a higher level of abstraction [Lamb 1987; Snodgrass and Shannon 1986]. These approaches support plug-ins that use internal representations tailored to their needs and can exchange structured data with other plug-ins, instead of communicating through character streams or sharing the same data structures.

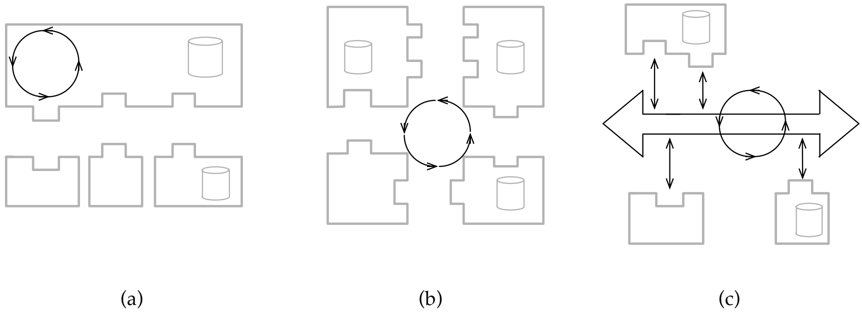


Figure 2.1: Tool architectures for control integration of plug-ins: (a) Standardized Backbone; (b) Components and Connectors; (c) Message Bus.

Control Integration

Tools that support multiple plug-ins need to enable plug-ins to communicate so they can notify each other of events, or request services. As they communicate plug-ins also need to share data. Hence, control integration also requires data integration: data integration takes care of properly representing and storing data while control integration is concerned with how plug-ins interact and request services from each other. Plug-ins can use many mechanisms to communicate with other plug-ins. One approach consists in having plug-ins that expose their services through explicit interfaces; instead of using interfaces plug-ins can also communicate just by making changes to shared data structures. Plug-ins can also only produce events and rely on message buses to deliver those events to other plug-ins.

Presentation Integration

This direction states that plug-ins should share the same "look and feel" from a user's perspective and support similar interaction paradigms. Hence, presentation integration should minimize learning and usage interference so a developer that knows how to use a plug-in can learn to use another plug-in without significant overhead. This is usually addressed in development tools by having plug-ins running in the same system window and using the same UI framework. For development tools, presentation integration also implies control and data integration, unless completely different tools run in the same window.

2.1.2 Plug-in Architectures for Development Tools

The term "plug-in" is highly overloaded in today's systems and there are numerous solutions for instantiating frameworks using plug-ins [Fayad et al. 1999]. We build a classification, illustrated in Figure 2.1 (p.15), based on how the lifecycle of an adaptation and the interaction between adaptations are handled within an environment (*i.e.*, control integration). The items in this classification are not mutually exclusive; the tools that we will discuss in the remainder of this chapter combine them to various degrees.

Standardized Backbone

One solution for enabling adaptations in development tools is to design plug-ins that fit against a standardized tool-specific interface (*i.e.*, standardized backbone). In this solution the tool controls the entire lifecycle of a plug-in through dedicated interfaces for initializing, running and stopping the plug-in. This mechanism is typically implemented by allowing plug-ins to provide callbacks that are then executed when events of interest are reified by the encompassing tool (*e.g.*, a file is opened or deleted, code is changed, an error is raised). The tool can further provide a set of common data structures (*e.g.*, data structures for representing code elements like classes, methods, annotations, abstract syntax trees), as well as standard interfaces for manipulating data or adding visual components to the user interface of the tool (*e.g.*, buttons, menu entries).

This approach favors central control and can reduce data duplication by allowing adaptations to share information; if needed plug-ins can still maintain their own data. Nevertheless, this approach limits the scope of a plug-in only to a particular tool making it difficult to reuse plug-ins between tools or environments. Constraints imposed by the tool can also limit the types of possible adaptations, or make their implementation difficult. Furthermore, a plug-in that acts incorrectly (*e.g.*, fails, requires too many resources) can leave the entire environment in an invalid state.

An example of a tool that uses this approach is *MetaSpy* [Ressia et al. 2012b], a framework for creating domain-specific profilers. In MetaSpy developers create custom profilers by subclassing a predefined class and overriding methods called when certain events, like method calls or attribute access, occur in the profiled code. In general, for object-oriented systems, inheritance and method overriding are common mechanisms to implement a standardized backbone.

Components and connectors

Tools designed using a standardized backbone require plug-ins to conform to a predefined life cycle and limit plug-in reuse in other tools. Instead of supporting domain-specific adaptations by plugging in custom functionality into predefined interfaces, a different approach consists in enabling developers to create new custom tools capturing domain-concepts by composing individual tools (*i.e.*, components). An individual tool defines interfaces for obtaining and providing services (*i.e.*, input and output ports; connectors). These tools follow ideas from component-based software engineering in which software is built by gluing together prefabricated components [Szyperski 2002]. This is also referred to as the toolkit approach [Snodgrass and Shannon 1986]. Here there is no single major component that controls the lifecycle of all adaptations. Each adaptation can have its own lifecycle depending on how the underlying components are connected; components can be connected in more than one way.

An example of this approach, discussed in Section 2.3.4 (p.27), is the *UNIX Shell*: developers create new commands by composing existing commands using pipes and filters. Composing the same commands in a different order can result in different tools.

Message bus

Creating custom adaptations by implementing hooks or by directly combining components through connectors means that tools and plug-ins need to be aware of each other. For example, a code editor can define an interface for plugging in a compiler and a syntax highlighter. Even if the compiler and syntax highlighter can be changed as long as the new ones implement the required interface, the code editor is still aware that it needs these extensions. A different approach consists in using a middleware that handles communication between plug-ins through message passing and keeps individual plug-ins and tools unaware of each other.

This approach relies on a message bus that forwards events between plug-ins. When added to the event bus, plug-ins can register interests in particular events. At run time, plug-ins generate events that are passed to the subscribed plug-ins by the message bus; depending on the message bus different types of messages like responses or acknowledgements can be used. Developers customize a tool following these approach by creating custom plug-ins and registering them with the message bus. A message bus reduces coupling between plug-ins favoring reuse. Depending on its design, it can also integrate plug-ins written in different languages. Nevertheless, the message bus needs to provide the right level of granularity for messages and avoid communication bottlenecks. Developers further have to take care of properly orchestrating the communication between plug-ins.

Monto [Sloane et al. 2014] is an example of a code editor that uses a message bus. Every time the developer edits the code, Monto sends a code-changed event. The compiler is just a listener that responds to the event and sends back to the message bus a response message containing the status of the compilation together with possible compilation errors. The editor then updates the UI based on that answer. Hence, the code editor has no dependency to the compiler.

2.1.3 The Cost of Custom Adaptations

A dimension in our discussion is the cost for creating a domain-specific adaptation. The term *cost* has many overloaded meanings when applied to software development. To disambiguate our comparison, we define cost starting from the definition used by De Volder [De Volder 2006] as *"the effort in terms of time and lines of code, and the expertise required to incorporate a domain abstraction into a development tool."* By expertise we refer to the knowledge required to understand and use the adaptation features of a tool and not the domain knowledge necessary to extract abstractions from a domain. We assume developers adapting a tool know their domain model.

Evaluating the precise cost of domain-specific extensions for all the tools that we will discuss next is a challenging task. Hence, we rank existing tools and techniques by looking at their focus on reducing the cost for creating domain-specific adaptations, using a three-point scale (*i.e.*, high, medium and low). Tools with the rank *high* allow developers to create domain-specific adaptations, however, do not have as main goal to reduce the effort and expertise required from a developer to create an actual adaptation. Tools with the rank *low* are designed to reduce the cost of domain-specific adaptations. Tools with the rank *medium*, while also designed to reduce the cost of an adaptation, require a high expertise.

Our definition of cost includes lines of code (LOC). In general, lines of code must be considered with caution when measuring complexity and development effort. The metric does not necessarily indicate the time needed to write those lines and it can vary significantly depending on the expressiveness of the programming language used for creating adaptations. For example, we give a medium rank to tools that support the creation of adaptations through logical programming languages or formal approaches (*e.g.*, attribute grammars, BNF). While supporting powerful adaptations with few lines of code, writing those lines requires a high expertise. Overall, despite its disadvantages, LOC gives a good indication of the size of a domain-specific adaptation. We consider that a small size makes the construction cost affordable when the expertise required to write those lines is also small.

2.1.4 Context-awareness in Development Tools

The second dimension in our comparison builds on context-oriented programming (COP), a paradigm for the creation of context-aware systems that can dynamically adapt behavior to their context of use and in reaction to changes in context [Hirschfeld et al. 2008; Salvaneschi et al. 2012]. COP enables systems to take advantage of contextual information to provide deeply personalized functionality. This dimension plays an important role in extensible development tools as depending on the development context, different information and adaptations can be of interest. Requiring a developer to be aware of all existing adaptations and to manually select relevant ones as its context changes introduces a significant overhead in the comprehension process.

A central concept in the design of such a solution is the actual context. COP defines a context in generic terms as *"any information that can be used to characterize the situation of an entity"* [Dey 2001] or *"any information which is computationally accessible"* [Hirschfeld et al. 2008]. For the purpose of our comparison we restrict the context to *the actual application domain and a developer's previous interactions with the domain*; we refer to this information as forming a development context. Previous work on exploring how developers comprehend software showed that interactions with a domain play an important role in improving program comprehension [Ko et al. 2005; Murphy et al. 2005; Ko et al. 2006]. For example interaction recording tools, like Mylar [Kersten and Murphy 2005] and DFlow [Minelli et al. 2015], provide support for automatically building a context as developers interact with development tools and filtering visible information based on that context [Minelli et al. 2016].

We distinguish between context-agnostic and context-aware tools:

Context-agnostic tools support no behavioral variation dependent on context. Their behavior is fixed and changes in the context trigger no changes in behavior.

Context-aware tools enable new, modified, or removed behavior during use, depending on how the contexts evolves. Hence, they can self-adapt their behavior in reaction to changes in the context. They can also take only limited parts of the context into account.

2.1.5 Summary

Figure 2.2 (p.20) summarizes these dimensions using a three-dimensional space. On the x and y axes (*Cost* and *Behavioral variations*) tools are assigned a category. On the z axis, denoting the *tool architecture*, tools can spawn over multiple categories. We used these dimensions to discuss approaches where developers incorporate domain concepts into their development tools by adapting the language or the tool.

2.2 Generating Tools from Language Specifications

One direction for embedding domain abstractions into development tools is to specify domain-specific languages capturing those abstractions and generate tools from language specifications. In this case the main effort goes into designing the language. Many different types of approaches have been proposed to simplify the creation of custom languages including modular compilers, modeling languages, language workbenches, *etc.* Voelter [Voelter 2014] and Renggli [Renggli et al. 2010b] provide comprehensive reviews. In this section we give a brief overview by focusing on the types of generated tools and on how language specifications are developed.

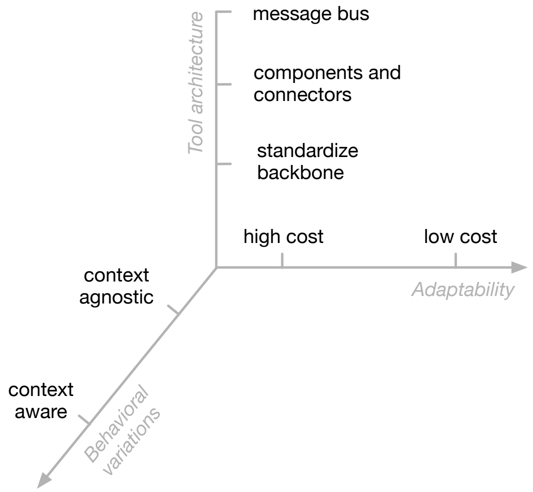


Figure 2.2: Dimensions for discussing related works.

2.2.1 Projectional Editors

Early examples of development tools generated from language specifications consisted in projectional editors (*i.e.*, syntax-directed editors). The ALOEGEN [Medina-Mora 1982] system for generating ALOE editors is one example. For each language ALOEGEN takes as input a description of the abstract syntax (the language constructs and their relations), concrete syntax (representations for each language construct) and a set of action routines. A different solution is followed by the Synthesizer Generator [Reps and Teitelbaum 1984], which uses attribute grammars as specifications for the syntax and semantics of a language. The Cornell Program Synthesizer [Teitelbaum and Reps 1981] and Poe [Fischer et al. 1984] also rely on attribute grammars as a specification mechanism.

Other tools use custom solutions for describing the language: MENTOR [Donzeau-Gouge et al. 1980] provides an editor for structured data expressed as abstract syntax trees using a custom formalism named METAL [Kahn et al. 1983]. Tenma *et al.* propose a system for generating language-oriented editors where features of the target language are represented using object-oriented concepts [Tenma et al. 1988]. PSG supports a hybrid editor which allows structure-oriented editing as well as text editing; the language is specified using a formal definition language, covering the whole spectrum of a language's syntax, context conditions, and dynamic semantics [Bahlke and Snelting 1986].

These tools follow what can be called a monolithic approach where the developer gives the language formalism to a black-box system, which in turn generates the editor. There is no direct focus on making the system extensible or supporting interoperability between editors for different languages.

2.2.2 Development Environments and Analysis Tools

As the generation of language editors spread, the focus moved towards generating more complete environments or other types of tools. The Gandalf project, for example, extends ALOEGEN with version control with the goal of *permitting environment designers to generate families of software development environments semiautomatically without excessive cost* [Habermann and Notkin 1986]. CENTAUR starts from a formal specification of a particular programming language (including syntax specified using METAL and semantics specified using ASF or TYPOL) and generates a structured editor, an interpreter, a debugger and other tools, all of which have graphic interfaces [Borras et al. 1988]. PECAN complements editors with multiple graphical views including expression trees, data type diagrams and flow graphs [Reiss 1985]. Pan targets not only editors for source code but also languages like design languages, specification languages or structured documentation languages for creating other software documents [Ballance et al. 1992]. LISA, apart from editors and debuggers, generates a wide range of tools for visualizing program structures and animating algorithms [Henriques et al. 2005].

Not only editing, debugging and visualizations tools can be generated. Devanbu proposed GENOA, a system for generating code analyzers [Devanbu 1992]. An analyzer is expressed by reasoning over abstract syntax trees using a specification language independent of the actual programming language under test. A companion system generates interfaces between GENOA and existing languages. Aria is a system built using GENOA that can generate testing and analysis tools based on an abstract semantics graph representation for C and C++ programs [Devanbu et al. 1996]. This reduces the cost of creating testing and analysis tools as developers can just focus on application-specific aspects without having to take into account the generation and traversal of graphs representing their applications.

These approaches, while still favoring black-box systems for generating tools, moved towards a central repository for storing an intermediate representation of the program together with uniform interfaces for accessing data. This change made it possible to easily integrate generated tools into a cohesive environment. It also enabled language builders to extend these systems with new tools that conformed to their interfaces (*i.e.*, standardized backbone).

2.2.3 Language Workbenches

With more and more systems supporting the generation of development tools, the attention shifted towards improving the development process of language specification and their interoperability. Klint proposed Meta-Environment, a system for developing formal language definitions based on the formalism ASF + SDF [Klint 1993]. Using these definitions Meta-Environment generates several tools like editors and debuggers [van den Brand et al. 2005]. The system also generates tools for working with the language specification. The current version of Meta-Environment follows a component-based approach: all tools from the environment are modeled as components that communicate using a tool bus, separating coordination from computation [Brand et al. 2001]. As long as components adhere to the communication protocol of the tool bus they may be written in any language.

The Xtext[1] project from Eclipse can generate complex text editors starting from an EBNF-like language specification. While two different languages cannot be combined developers can extend an existing language. MPS[2] provides dedicated projectional editors for languages created using a dedicated modeling language. MPS supports languages which are fully integrated with each other. IMP is a meta-tooling platform intended to reduce the cost for creating and customizing language-specific IDEs in Eclipse [Charles et al. 2007]. To reduce the cost needed for customization IMP provides a combination of frameworks, templates, and generators.

These approaches shifted the focus towards extensibility and composition of languages. If in the past the main goal was to generate a comprehensive tool suite for a language, current language workbenches have the added requirement of interoperability between tools across different languages.

2.3 Adapting Tools to Specific Domains

As discussed in Section 1.2 (p.3) developers can directly express domain abstractions using constructs provided by their general-purpose programming language, without having to design dedicated languages. In this situation however, to take advantage of those concepts within their development tools, developers need to explicitly adapt their tools. In this section we analyze several tools that allow domain-specific adaptation based on the three dimensions discussed in Section 2.1 (p.13).

2.3.1 Overview

We group related tools based on the three types of plug-in architectures presented in Section 2.1.2 (p.16). For each tool we highlight the plug-in architecture on top of which it is built, classify the tool based on the cost for creating an adaptation and indicate if the tool supports a mechanism for selecting adaptations based on a developer's context. Table 2.1 (p.24) gives an overview of this comparison. In the remainder of this section we discuss the tools from each group and motivate our comparison. Given the large number of development tools currently available, the presented set of tools

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

[2] https://www.jetbrains.com/mps/

is not exhaustive; instead it focuses on a set of representative tools covering a wide range of approaches and techniques.

2.3.2 Live Environments

The software development cycle requires developers to perform multiple activities like writing code, reading code, testing, debugging, deployment, *etc.* Developers benefit from short feedback loops when performing these activities. Short feedback loops are, however, not provided by all development tools. Some introduce significant overhead, forcing developers to make many context switches between different development activities. A classical example is the long feedback loop between writing and compiling C++ code. Live environments aim to provide shorter feedback loops and give a sense of immediate feedback by enabling developers to permanently interact with a running system.

Smalltalk-80 is an early example of a live environment [Goldberg 1984]. Developers evolve a Smalltalk application by editing and manipulating objects in a running system. Furthermore, within a Smalltalk 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. Because it is a running system any change in the code of the IDE, for example, is immediately visible to the developer. Modern environments based on Smalltalk-80, like Pharo and Squeak[3], follow these ideas.

Another example of a live environment is *Self* [Ungar and Smith 1987]. Unlike Smalltalk, which uses classes to create objects, Self is a prototype-based system: developers evolve an application by only interacting with objects. Like Smalltalk, Self can be described as both a language and an environment that has no system-level distinction between using an application and changing or programming it [Smith et al. 1995]. Both actions are done by manipulating the state and behavior of objects. For example, the entire user interface of Self is composed of uniform graphical objects [Maloney and Smith 1995]. Hence, the entire environment is available for direct modification just by interacting with objects in a uniform way.

The Lively Kernel is a collaborative Wiki-like development environment applying ideas from Smalltalk to the web [Ingalls et al. 2008]. The Lively Kernel runs in a web browser and enables multiple users to share content by constructing and interacting with objects. Users can also modify the system itself including the development tools. To avoid situations in which a user makes changes to a tool that break the whole system, the Lively Kernel models tools as scriptable objects that can easily be cloned [Lincke and Hirschfeld 2013]. Developers can then change scripts and objects directly in a cloned tool and avoid breaking the original tool. Lively Kernel supports users in sharing and discovering scripts for creating custom tools.

[3] http://www.squeak.org

		Standardized backbone	**Components and connectors**	**Message bus**		
Type	**Tool**	**Plug-in architecture**			**Cost**	**Context**
Section 2.3.2 Live Environments	Smalltalk	●	●	○	▮▮▮	—
	Self	●	●	○	▮▮▮	—
	Lively Kernel	●	●	○	▮▮▮	—
Section 2.3.3 Pluggable Frameworks	Eclipse	●	●	○	▮▮▮	perspectives
	EMACS	●	●	○	▮▮	modes
	MetaSpy	●	○	○	▮	—
	GSEE	●	●	○	▮	—
Section 2.3.4 Composable Tools	UNIX	○	●	○	▮	—
	Vi	○	●	○	▮	—
	Monto	○	○	●	▮▮	—
	FIELD	○	○	●	▮▮▮	—
	Code Bubbles	●	○	●	▮▮▮	—
Section 2.3.5 Query-Based Tools	JQuery	●	○	○	▮▮	domain model
	(Wuyts)	●	●	○	▮▮	—
	SVT	●	●	○	▮▮	—
Section 2.3.6 Rule-Based Transformations	ConTraCT	●	●	○	▮▮	preconditions
	(Wrangler)	●	●	○	▮▮	preconditions
Section 2.3.7 Meta-Models	Glamour	○	●	○	▮	—
	VIVIDE	○	●	○	▮	domain model
	OmniBrowser	●	●	○	▮	—

Table 2.1: Comparison between tools using the dimensions presented in Figure 2.2 (p.20): the plug-in architecture, the cost for creating an extension and their context awareness. ● relies on that architecture; ○ does not use that architecture. ▮▮▮ high cost for creating adaptations; ▮▮ medium cost; ▮ low cost. — if the tool does not provide a solution for selecting adaptations based on the context; the name of the solution if this is not the case.

Live environments while not directly supporting cheap and context-aware tooling, provide a solid foundation for enabling tool building. They rely on plug-in architectures where developers can reuse available components and leverage predefined interfaces; data is usually stored in a central location. For example, Smalltalk has a prominent focus on development tools and on adapting those tools [Johnson et al. 1989] and uses the notion of an *image* as repository for storing objects. This focus on tool building in live environments promotes a culture in which developers are encouraged to adapt their tools. Many approaches discussed in the remainder of this chapter build on this foundation to improve tool building, including *moldable tools*.

2.3.3 Pluggable Frameworks

Many development tools rely on an abstract design that embodies how the tool should work and provide hooks (*i.e.,* predefined interfaces) developers can use to inject their functionality through plug-ins; individual plug-ins can be created by combining components provided by the tool. Data representing program elements is maintained by the tool; plug-ins can also maintain their own data.

The *Eclipse Platform* is a previously mentioned example following these ideas. The Eclipse Platform consists of a set of frameworks and common services that build a basic runtime on which functionality is added by loading new plug-ins. Development tools are designed as plug-ins that offer developers various customization possibilities through dedicated extension points [Yang and Jiang 2007]. For example there exist extension points for adding new context menu entries or shortcuts to the code editor, or actions to the debugger. Creating a plugin requires a developer to provide several files: a manifest file describing the plugin, an XML file expressing its dependencies, a class providing callbacks called during the lifecycle of the plug-in, *etc.*

To group views and commands related to a particular development task (*e.g.,* coding, debugging), Eclipse provides *perspectives* [Clayberg and Rubel 2008]. A perspective is a mechanism that determines the visible actions, data and views (*i.e.,* graphical widgets) within a the tools, and provides a layout for arranging views. Plug-ins can extend existing perspectives or can provide their own. Developers can manually change between perspectives, or tools can suggest developers perspectives when they detect certain events (*e.g.,* the debugger suggests to developers switching to the debug perspective when a breakpoint is reached at run-time). Other IDEs, like Visual Studio[4] and IntelliJ[5], provide similar extension and grouping mechanisms [Nayyeri 2008; IntelliJSDK 2016].

A highly influential plug-in based system is *Emacs*, an extensible and customizable text editor [Stallman 1981]. In Emacs the functionality of the editor is defined in

[4] `https://www.visualstudio.com`

[5] `https://www.jetbrains.com/idea`

a library of commands. Each command has a name and is bound to a particular keystroke. Users can freely change the key bindings of commands or add new commands as the editor is running. Unlike in Eclipse, commands do not require a file structure; a command can be expressed as a function bound to a keystroke. To facilitate management of commands and key bindings, both can be grouped using editing modes. An editing mode groups specialized features for working with a particular type of file or features that can be toggled during editing. Modes can change the meanings of certain key bindings as well as add or remove key bindings and commands. This provides a simple and flexible customization mechanism used in other text editors like Sublime Text[6] and Atom[7].

Eclipse and Emacs provide an explicit approach for extending tools. This approach however is generic and does not focus on reducing the cost of creating an extension. In both cases developers extend a tool by writing a plugin in the language in which the tool is implemented using a generic API. This has the advantage that developers to not have to learn a new language to extend the tool. These tools further support grouping of commands and enable developers to find specific commands using build-in textual search support. While search support is a step towards helping developers locate commands, it does not enable tools to automatically detect commands applicable for the current development context.

To reduce the cost of creating a plug-in, other approaches reduce their generality and focus on addressing specific problems and tasks. For example, *MetaSpy* is a framework for quickly prototyping new domain-specific profilers [Ressia et al. 2012b]. In MetaSpy a new profiler is created by subclassing an abstract `Profiler` class and indicating how to extract domain-specific events from the model using a set of predefined or custom instrumentation strategies. Another example is *GSEE* [Favre 2001], an environment for exploring and visualizing software, in which developers specify how to extract data from a system by implementing a set of predefined interfaces (*e.g.*, `Successor` for modeling a one-to-many relation). The framework uses these interfaces to extract and display a model using customizable visualizations. In both these frameworks developers need to decide when to use a profiler or a visualization.

Tools for analyzing source code, like *Checkstyle*[8], *Findbugs* [Ayewah et al. 2008] and *Pmd*[9], also follow these ideas and focus on reducing the cost of creating custom analyses. Custom analyses are implemented as AST visitors using dedicated APIs. Hence, developers do not have to deal with the issues of extracting and managing ASTs; they can focus only on the implementation of their own analysis. Furthermore, these tools can adapt their behavior based on the content of an XML configuration file. The focus of these tools is not to automatically detect what analyses to apply in a given context. It is rather on enabling developers to create and use custom analyses for their projects and domains, instead of relying on generic analyses.

[6] https://www.sublimetext.com

[7] https://atom.io

[8] http://checkstyle.sourceforge.net

[9] http://pmd.github.io

2.3.4 Composable Tools

Instead of providing hooks a plug-in can implement, other development tools and environments support the creation of adaptations by connecting, using various mechanism, already available pieces of functionality. We distinguish between tools that use components and connectors and tools that rely on a message bus.

By design these approaches support tool creation at two level of abstractions. On the one hand, developers can create tools within the design space of the available components solely by combining and customizing those components. Given a set of components that covers the problem domain, this enables the creation of useful tools with very low effort. Many tools optimize for this use case by focusing on solutions to discover and share components. On the other hand, when the available component set is not enough, developers can create new ones. This increases the scope of the tool, however, it requires more effort than just combining components.

Directly Connecting Components

The UNIX shell is a highly influential environment that enables developers to create programs by combining a set of existing commands using a pipes and filters pattern [Raymond 2003]. For example, listing all files from a directory modified on *February 24* can be done by combining the *ls* command for displaying the the list of files from a directory with the *grep* command searching for lines matching a regular expression: *ls -l | grep "Feb 24"*. Pipes link commands only through a stream of characters; while commands could also share data through the file system the vast majority of UNIX commands communicate only through their input and output streams. To find basic commands the UNIX shell relies on autocompletion and textual searches. This simple architecture supports the creation of powerful tools as small and concise snippets of code.

A different mechanism to compose commands following similar ideas is used in *Vi*, an advanced text editor. Vi is built on the idea of command composability [Joy 1980] and proposes a set of general-purpose commands that can be combined to form larger commands. For example, to delete text, users need to combine the delete command *d* with a command that moves the cursor over a portion of the text; that can be *$* (move cursor to the end of current line), *w* (move cursor over one word), or any other movement command. Emacs, on the other hand, provides specialized commands for each of these actions (*kill-word* and *kill-line*) and in general favors monolithic commands. Newer editors based on Vi, like Vim (Vi iMproved) [Oualline 2001], follow the same ideas regarding composability of commands. Unlike UNIX, where commands share data through streams, in Vi commands operate on a common data structure modeling the current text.

Vi and UNIX significantly reduce the cost of crafting custom tools by combining commands. For example to combine commands in Vi developers only need to switch to the command mode and type the letter representing the individual commands. This low cost has the side effect that developers do not need to focus on saving and managing custom tools if they can easily create those tools when needed. This solution is however highly domain-specific and works well only for simple and rapid ways to compose tools.

Middleware-connected Components

FIELD is an example of programming environment where all communication between tools is achieved using message sends to a common message server [Reiss 1990]. Each tool runs in a separate process and registers patterns describing the messages it is interested in with the message server. The message server forwards a message to all tools that registered a matching pattern. Messages can also have associated replies, returned to the original sender by the message server. *CodeBubbles* [Bragdon et al. 2010a], a user interface based on collections of lightweight editable fragments, called bubbles follows a similar idea. CodeBubbles was designed to be embedded in an IDE, however, it does not directly depend on the IDE. Instead it uses a message-based interface to communicate with the underlying IDE [Reiss 2012]. These solutions ensure separation between individual tools and favor reuse of individual tools. Nevertheless, while enabling reuse and configurable tools, they do not focus on reducing the cost of creating or configuring custom tools.

An environment that has as goal to reduce the effort for incorporating components into a development tool is *Monto* [Sloane et al. 2014]. To achieve this Monto proposes an architecture that distinguishes between sources that publish notifications when changes to user-edited text occur, servers that provide functionality, and sinks that consume products from servers. Components communicate via text messages across an off-the-shelf messaging layer. Monto was used to create a basic editor for writing and compiling Java code by linking the Sublime editor with a Java compiler in less than 500 lines of Python and Scala code. While it reduces the effort for incorporating tools, like the previously discussed tools, Monto has no support for modeling and using a development context to select tools.

2.3.5 Query-based Tools

Developers need to repeatedly find information relevant for their tasks and domains [LaToza and Myers 2010]. A wide range of tools that query a system based on various criteria have been proposed, especially in the field of feature location [Dit et al. 2012], to help them find relevant data. Part of these tools enable developers to formulate their own custom queries using logical programming languages. Developers create an adaptation by combining a custom query extracting data from a system with widgets graphically displaying the query results. These tools rely on a common repository (*i.e.*, knowledge base) that is being queried by all adaptations.

A tool for creating code browsers following this idea was proposed by Wuyts [Wuyts 1996]. Developers build an adaptation by formulating a query using a logical programming language detecting structural information and constructing a user interface using a reflective UI builder. The UI builder provides reusable graphical components for which part of the domain knowledge has to be specified when the component is used in an application. *SVT* applies a similar approach to a different type of task: building software visualizations [Grant 1999]. Developers use queries written in Prolog to extract data from both the code and the runtime. A set of mappings between data and views are then used to select one or more suitable views for displaying a query result. Additional Prolog code can be dynamically loaded that contains more view specifications and mappings. These and other tools focus on solutions for enabling expressive customizations. They, however, do not provide support for detecting relevant adaptations once the developer has finished building them.

JQuery is a tool that has a direct focus on reducing the cost of building a code browser [De Volder 2006]. Towards this goal JQuery limits the type of code browsers that it supports to hierarchical code browsers having as an interface a tree-view widget with collapsible and expandable nodes. Developers create a custom code browsers by specifying a query executed over a source model (*e.g.*, a database containing facts about a program's structure), and a list of attributes for organizing the query results into a tree [Janzen and de Volder 2003]. To enable developers to discover browsers JQuery organizes them using context menus. Browsers displayed in the context menu of a code entity are determined dynamically: each browser can have in a configuration file a predicate that checks if the browser can be applied on the selected code entity. However, this does not take into account other information present in a development context like previous interactions with the environment.

The advantage of these tools is that they allow a developer familiar with the query language to quickly build adaptations that only need to use available graphical widgets. Tools are inherently extensible within the boundaries set by the underlying language. Furthermore the query language can ease creation of certain types of queries. For example the declarative logic programming language used by JQuery is extended with predicates providing access to data about a program's structure (*e.g.*, inheritance hierarchy, location and targets of method calls, field accesses). Developers can further extend the query language with new predicates, an activity that requires significantly more effort than just creating a browser using available predicates. Nevertheless, the extensibility power of these tools also comes with a predicament: developers unfamiliar with logical programming languages have to leave their comfort zone and learn a new programming paradigm [Janzen and de Volder 2003; Lozano et al. 2015; Caracciolo 2016].

We observe that tools in this category, like composable tools, distinguish between two modes of creating adaptations. The first aims to reduce the effort for building an adaptation as long as developers remain within the confines of the provided logical language and UI widgets. The second enables greater flexibility by allowing developers to extend the language and add new types of widgets at a greater cost.

2.3.6 Rule-based Transformations

Another category of development tools support adaptation by enabling developers to compose transformations over the structure of a program, with the goal of increasing programmer productivity by automating programming tasks. A basic transformation rule in this case expresses a one-step transformation on a fragment of a program [Visser 2005]. More complex tools are then created by composing basic transformations following a component-based approach.

As it is difficult to cover all possible domain-specific refactorings, refactoring tools often follow this approach. They provide developers with a set of basic refactorings and allow them to compose larger refactorings from existing ones. Composition is usually supported through dedicated languages built on formal foundations. This enables highly expressive and short programs. For example, ContTraCT uses a dedicated refactoring editor for a language providing two basic composition operations: `and` and `or` [Kniesel and Koch 2004]. Li and Thompson on the other hand, discuss how to compose refactorings using the concrete syntax of Erlang and a dedicated API [Li and Thompson 2012]. RubyTL proposes an external domain-specific language embedded in Ruby [Cuadrado et al. 2006].

One important aspect of rule-based transformations is that they should only be applied on precise program fragments. Hence, apart from a transformation, a rule also needs to recognize the specific program fragment that it should transform. Recognition can be achieved by associating preconditions to rules that match the structure of the program and possibly verify some semantic conditions [Visser 2005]. In the case of refactorings, preconditions check if the required program elements are present in the code and if the semantics of the refactored code can be preserved. This mechanism provides an alternative to using a development context for determining what rules can be applied on a given fragment of code.

2.3.7 Meta-models

A model is a simplification of a system built with an intended goal in mind [Bézivin and Gerbé 2001]. Meta-models specify how to construct models for a class of systems [Seidewitz 2003]. Approaches applying this idea enable developers to leverage a meta-model to specify a model of the tool and generate the concrete tool from the model.

OmniBrowser [Bergel et al. 2008] is a tool builder exemplifying this approach. OmniBrowser supports the creations of browsers (*i.e.*, applications with a GUI that are used to navigate a graph of domain elements) based on an explicit meta-model. A domain model is described in a graph and the navigation in this graph is specified in a metagraph. Nodes in the metagraph describe states the browser is in, while edges express navigation possibilities between those states. OmniBrowser then dynamically composes widgets such as list menus and text panes to build an interactive browser that follows the navigation described in the metagraph. OmniBrowser uses the programming language in which it is implemented (*i.e.*, Pharo) as a modeling language. OmniBrowser does not tackle the problem of when to use custom code browsers. It leaves this decision solely to the developer.

Glamour is another tool for building data and code browsers where developers build a model of their tool using a components and connectors architecture [Bunge 2009]. The model expresses the behavior of the tool and it is independent of any GUI framework. Developers specify the model of a browser using an internal domain-specific language. A *renderer* is responsible for generating from the model a tool for a concrete platform. Glamour offers renderers for the Morphic GUI framework [Maloney and Smith 1995] and Seaside web development framework [Ducasse et al. 2004]. Developers can create complex code browsers in a few hundred lines of code. However, like OmniBrowser, Glamour cannot select code browsers based on a development context.

Following ideas from UNIX, Taeumel *et al.* propose VIVIDE [Taeumel et al. 2012], a tool following a data-driven approach to rapidly build graphical tools by modeling graphical tools as multiple pipelines that transform and display data. The configuration of a tool is based on scripts: developers use concise scripts as glue between data and graphical views. Scripts are written using Squeak, the language in which the tool is implemented. This approach is shown to reduce the cost of creating development tools like debuggers and object inspectors [Taeumel et al. 2014]. VIVIDE has a large focus on presentation integration. All tools are displayed in a single horizontal unbounded tape that is embedded into a scrollable area. Different tools are made available to developers depending on the types of data (*i.e.*, objects) they are interacting with (*e.g.*, source code, run-time information). Hence the domain model is used to determine appropriate domain-specific tools.

2.3.8 Summary

There exist a wide range of solutions for constructing development tools that support domain-specific adaptations. This section grouped these approaches based on the mechanism they use to support adaptations into the following categories:

Live Environments Allow developers to change and evolve both their applications and the development environment in a uniform way.

Pluggable Frameworks Provide an abstract design that embodies how a tool should work and give developers hooks they can use to customize the tool.

Composable Tools Support the creation of custom tools by combining, using various mechanisms, already available pieces of functionality. Enable individual components to be reused in different contexts and between tools.

Query-based tools A tool is created by combining a custom query extracting data from a system with widgets graphically displaying the query results.

Rule-based transformations Developers create custom tools by composing basic transformations over the structure of a program.

Meta-models Allow developers to use a meta-model to specify a model of their tool and generate the concrete tool from the model.

We further discussed tools in these groups based on two dimensions: their ability to adapt to a development context and the cost of creating a domain-specific adaptation. Table 2.1 (p.24) presented an overview of this analysis.

2.4 Conclusions

We observe that tools employ diverse mechanisms for reducing the cost of an adaptation including internal and external DSLs, logical programming languages, formal specifications, component-based architectures, *etc.* Nevertheless, in spite of enabling developers to create custom adaptations, many tools do not focus on supporting behavior changes based on a development context, or only take parts of the context into account. This indicates a lack of approaches for improving development tools that focus on both enabling inexpensive adaptations and behavioral variation dependent on the current context. In the next chapter we discuss moldable tools, our solution for addressing this gap.

We become what we behold.
We shape our tools, and thereafter our tools shape us.
John Culkin talking about Marshall McLuhan

3 Moldable Tools

In this chapter we introduce and discuss the *moldable tools* approach for adapting development tools to specific domains through inexpensive and domain-aware extensions. Moldable tools provide hooks for precise customizations; extensions encapsulate the contexts in which they are applicable. We start by identifying a set of design principles for moldable tools based on the analysis of related works from Chapter 2. Then we present the moldable tools approach and show how to instantiate moldable tools for supporting the development of object-oriented applications.

3.1 Design Principles for Moldable Tools

The first dimension discussed in Section 2.1 (p.13) is concerned with how plug-ins communicate within a tool. The remaining two dimensions explore two complementary issues: the cost of a plug-in and context-awareness in development tools. These two dimensions have a direct influence on how developers adapt development tools to specific domains. Hence, in this section we discuss for each of these two dimensions, a series of design principles for improving adaptation of development tools to specific domains.

3.1.1 Enabling Inexpensive Creation of Custom Adaptations

Despite the diverse set of mechanisms used to support custom adaptations we have observed a repetitive set of goals: tool builders want to reduce the cost of creating certain types of adaptation without limiting the space of allowed adaptations. Component-based tools, query languages and rule-based tools directly follow this desideratum. They enable cheap creation of adaptations that use only available components and language constructs, while supporting the creation of new components and language constructs at a higher cost. Pluggable frameworks address this aspect by providing fine- and coarse-grained extension points together with default extensions; this way developers can only customize the needed functionality.

Based on the aforementioned observation we identified the following as high-level principles for designing development tools focusing on inexpensive adaptations:

DP 1. Identify common types of tool-specific adaptations: Different development tools target different activities of the software development cycle. Hence, to simplify the adaptation process tool builders should identify common types of tool-specific adaptations developers need across multiple domains. For example, debuggers should provide breakpoints at the level of the domain: breakpoints in terms of events when working with event-based systems; breakpoints at the level of a grammar when working with a parser. Code editors can embed alternative ways to edit domain-specific entities: state diagrams for state machines or mathematical notations for complex formulas.

DP 2. Simplify the creation of common adaptations: Simplifying the creation of all possible adaptations for a tool is not feasible due to the wide range of domains and development tasks. For a given tool, tool builders can instead significantly reduce the cost for creating common types of adaptations. For example, in a debugger many domain-specific breakpoints, and other actions like logging and invariant checking, can be created by using an instrumentation mechanism to insert a code snippet at a given location; encapsulating how a code snippet is inserted in the code allows developers to only focus on the domain-specific aspect. A tool for visualizing domain entities can allow developers to easily try out multiple common types of views (*e.g.*, tree, lists, graphs, charts).

DP 3. Do not limit the types of possible adaptations only to common ones: While tool builders can simplify the creation of a wide range of common types of adaptations, supporting only those adaptations limits the scope of a development tool. Hence, tool builders should not hardcode possible types of adaptations and allow unanticipated ones, even if they entail a higher cost: the extension mechanism should be itself extensible. For example, JQuery, the tool discussed in Section 2.3.5 (p.28), allows developers to both create browsers using a logical programming language and extend the language with new predicates. Tools for visualizing domain entities should not hardcode a predefined set of visualizations, but allow developers to use any kind of graphical widget.

3.1.2 Context-aware Adaptations

Several frameworks and composable tools support mechanisms to group together relevant adaptations, nevertheless, require developers to manually select the appropriate group for a task. IDEs can automatically suggest appropriate groups (*e.g.*, debug perspective in Eclipse); text editors offer different commands based on the type of file being edited (*e.g.*, modes in Emacs). These solutions however use ad hoc mechanisms rather than modeling a development context, and work on groups rather than individual adaptations. JQuery and VIVIDE can further filter relevant queries based on properties of selected software entities.

Tools supporting adaptations through rule-based transformations rely on a different approach: they come with preconditions that only enable developers to apply a transformation if its associated preconditions hold. For example, refactorings should only be applied if they preserve the semantics of the transformed code. This requires precise formulation of preconditions based on AST nodes frequently done using pattern matching. Nevertheless, this is a flexible and versatile idea that can be applied to select adaptations if one replaces preconditions over code structure with preconditions over the development context. Based on this observation we propose the following as design principles for enabling development tools that provide behaviour adaptations based on a development context:

DP 4. Attach activation predicates to adaptations: The adaptation mechanism should enable developers creating an adaptation to specify together with their adaptation an activation predicate (*i.e.*, precondition) indicating when the adaptation is appropriate (*e.g.*, a domain-specific breakpoint should decide if it should be displayed in the debugger or not). Activation predicates should be applied on the current development context.

DP 5. Update adaptations based on a development context: Applicable adaptation can change as a developer is using a tool. Development tools should react to changes in a development context by detecting new suitable adaptations or invalidating current ones that are no longer applicable. For example, a debugger should update the set of applicable breakpoints as developers navigate through the execution of their applications.

3.1.3 The Moldable Tools Approach

Development tools like code editors, compilers, debuggers, profilers, search tools, *etc.*, embody a design that addresses a certain activity from the software development cycle. To support inexpensive adaptations in these tools while preserving their intended design, moldable tools enable adaptations through precise extension points. Developers create extensions for an extension point by configuring predefined components or by creating new components that conform to an extension point's interface. This combines the standardized backbone and the components and connectors approaches discussed in Section 2.1.1 (p.14). A moldable tool integrates all extension into the same window (*i.e.*, presentation integration). Furthermore, extensions share the same data structures (*i.e.*, data integration).

To enable behavioral variations based on the context, each extension has an attached activation predicate, capturing the development contexts in which that extension is applicable. A moldable tool is aware of the current development context and uses activation predicates to select extensions applicable in that context.

3.2 Moldable Tools for Object-oriented Programming

Moldable tools propose a high-level approach to improve program comprehension through domain-specific adaptations. To validate the moldable tools approach and show that it improves program comprehension we apply it in this dissertation to tools for developing object-oriented applications and show that it can address the research questions presented in Section 1.5 (p.8). In this section we discuss how to instantiate moldable tools for supporting the development of object-oriented applications by using object-oriented concepts.

3.2.1 Modeling Development Tools

Object-oriented programming expresses applications in terms of objects and object interactions. Objects model domain entities and encapsulate the state of those entities together with their behavior [Dahl and Nygaard 1966]. While objects should encapsulate all relevant behavior, in the context of business systems, Pawson observed that many domain objects are *behaviorally-weak* [Riel 1996]: much of the functionality is implemented in *'controller'* objects that sit on top of model objects, which in turn provide only basic functionality. To address this Pawson proposed *naked objects* as a way to move towards behaviorally-complete objects where user actions consisting of viewing objects, and invoking behaviors are encapsulated in the actual objects [Pawson 2004].

When developing object-oriented applications, developers interact with objects using development tools. Development tools need to decide how to handle objects modeling different domain entities. One approach is to specify the logic for how to handle an object in the development tools themselves. 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. Another approach consists in making objects responsible for deciding how they are handled in development tools. This allows different tools to reuse the same behavior and enables a closer evolution of objects and tools.

A common usage of the second 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 ask that object for its textual representation. Objects can provide multiple representations supporting different goals. For example the textual representation returned by `printString` in Smalltalk (`__str__` in

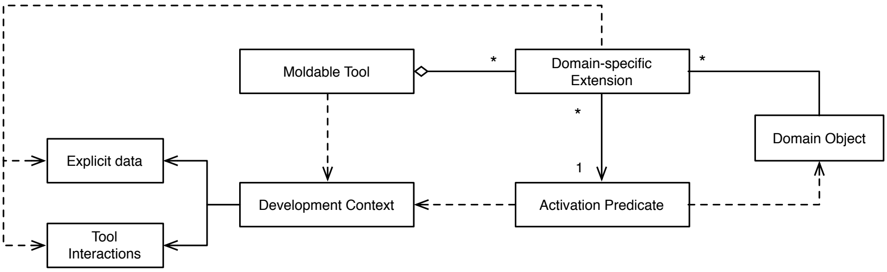


Figure 3.1: Domain-specific extensions are attached to objects. Activation predicates filter extensions also based on an object's state.

Python) summarizes an object, while `storeString` (`__repr__` in Python) provides a representation from which the receiver object can be reconstructed. Hirschfeld *et al.* show how to explicitly change the representation of an object depending on the context [Hirschfeld et al. 2008].

To reduce the distance between the code of an object and the code of a tool for working with that object and to favor co-evolution of tools and objects, moldable tools for object-oriented programming follow the second approach: they enable objects to decide how to be handled in development tools. Hence, moldable tools *provide extension points that ask domain objects to indicate the desired behavior* (Figure 3.1 (p.37)). This way objects become behaviorally-complete with regard to development tools, not only to the business domain.

A side-effect of this decision is that all software entities based on which one wants to customize a tool need to be modeled as objects. In the case of run-time objects this is straightforward. Nevertheless, this also includes software entities like packages, classes, methods, annotations, files, source code, bug reports, documentation, examples, repositories, configurations, etc. The live environments discussed in Section 2.3.2 (p.23) already show that is is possible as they model all entities available in the environment as objects. Many other IDEs also provide object-oriented models for representing code and project related data (*e.g.*, JDT in Eclipse). Hence, we do not consider this as a limitation that prevents the practical implementation of moldable tools. Modeling all software entities as objects makes is possible to uniformly attach extensions to run-time objects, source code entities and external resources.

3.2.2 Modeling Domain-specific Extensions

The second aspect related to applying moldable tools to object-oriented programming consists in how to specify custom extensions. As explored in Section 2.3 (p.22), developers can create extensions using internal DSLs, external DSLs, logical programming languages or formal specifications. Following the same line of reasoning as in the previous section, object-oriented programming already provides a modeling

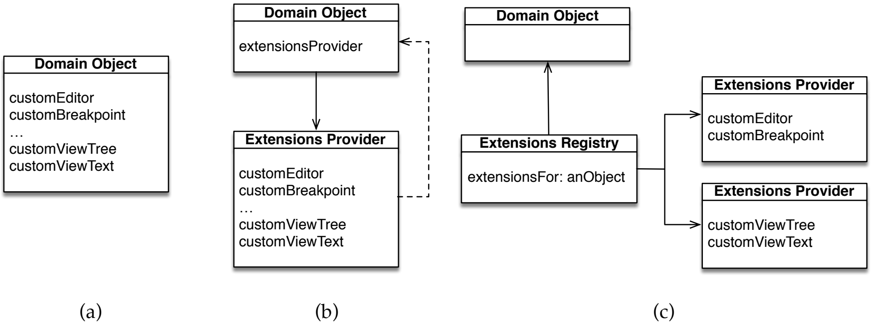


Figure 3.2: Associating extensions with objects: (a) extensions are defined in the same class as the object; (b) extensions are defined in an external extensions provider linked with the object; (c) a registry links extensions with objects.

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 that provide an interface (*i.e.*, API, internal DSLs) for configuring the extension. Developers specify an extension by creating and configuring an object representing an extension using the provided API. To reduce the cost of creating common domain-specific extensions a moldable tool can provide predefined objects modeling custom extensions. Developers can create new types of extensions by creating new objects that adhere to an extension's API. This solution favors a design in which developers create custom extensions by using custom snippets of code to configure predefined components. Related work focusing on similar ideas indicate that using snippets of code to customize tools reduces the cost of creating extensions [Ousterhout 1998; Taeumel et al. 2014].

3.2.3 Associating Extensions with Objects and Tools

Another aspect a builder of a moldable tool needs to take into account is how to associate snippets that configure an extension for an object with both that object and the tool. Regardless of the mechanism, for a moldable tool, there should be a bidirectional relation between objects and extensions: an object should know its extensions and an extension should know the object to which it is attached.

In object-oriented languages that use classes, a tool builder can associate the snippet with the object by allowing developers to define in the class of the object a method that contains the snippet (Figure 3.2a (p.38)). Alternatively, a tool builder can design a tool so that extensions are created in a different object (*i.e.*, an extensions provider). The domain object can then only contain a method that indicates the object responsible for creating its extensions (Figure 3.2b (p.38)). Instead of a single extensions provider for all tools, each tool can have its own provider.

These two solutions require that developers add methods to objects from other libraries or applications. Many languages provide mechanisms that support this use case (*e.g.*, extension methods in Smalltalk, partial classes in C#). If this is not possible, a tool builder can implement a registry that associates extensions with objects (Figure 3.2c (p.38)). In object-oriented languages that do not have classes, tool builders can associate extensions with objects by directly adding methods to those objects; if this is not possible, a registry can be used. To associate extensions with tools, a tool builder can rely on naming conventions or annotations. For example, all methods implementing a custom view for an object can have a predefined prefix (*e.g.*, `customView` in Figure 3.2 (p.38)) or be marked with a predefined annotation.

When extensions are defined as methods in the class of an object, developers can view and edit the code of the object and that of all extensions in the same editor. This favors co-evolution of extensions and objects. If extensions are defined in other classes, developers can still take advantage of this if the environment is designed to hide this separation from the developer: for example the code editor can display extensions together with the code of the objects. This solution however requires that the entire environment is adapted to support moldable tools.

3.2.4 Context-aware Extensions

Domain objects provide one dimension for selecting domain-specific extensions: a moldable tool first selects extensions for those domain objects currently investigated in that tool. Each extension object further has an activation predicate specified when the extension is created; the extension object should provide an interface for adding and retrieving its activation predicate. Activation predicates are objects that express a boolean condition applied on the current development context.

Section 2.1.4 (p.18) defined the development context as *the actual application domain and a developer's previous interactions with the domain*. For object oriented-applications, by application domain we refer to the object associated with an extension, as well as any other domain object accessible from the tool. By interactions with the domain we refer to both interaction data generated as a developer is using the tool and data explicitly added by a developer to the context. This requires a tool to provide a solution for recording relevant interactions within that tool and for allowing developers to add explicit data to the context. Explicit data can consist of, for example, in particular objects, annotations or external resources.

Activation predicates access the context to determine if an extension is applicable. However, the same extension should be able to behave differently depending on the context, if for example the extension is applicable in multiple contexts. To support this moldable tools also allow an extension to access the development context. One solution for implementing this behavior is to model the development context as an explicit object that is made available to the code that creates the extension. For example, it can be stored as an attribute in the extension object or passed to the method creating the extension.

3.3 Conclusion

Based on our previous analysis of related works, we identified a set of requirements for building domain-aware tools. We then proposed moldable tools, a tool building approach following those requirements that focuses on inexpensive creation of custom domain-specific extensions and selection of appropriate extensions based on the development context. In this section we further explored how to apply moldable tools to development tools for object-oriented software, by leveraging the fact that object-oriented programming already provides a way to model a domain in terms of objects and message sends.

Moldable tools propose a high-level approach to tool building. To validate the approach we need to apply it to concrete development tools. Next we set out to show that moldable tools improve program comprehension by applying it to several tools for developing object-oriented applications. Starting from limitations of current tools in *(i)* reasoning about run-time objects, *(ii)* searching for domain-specific artifacts, and *(iii)* reasoning about run-time behavior, we explore how designing tools following the moldable tools approach can improve over the state of the art for performing the aforementioned activities.

Design and programming are human activities; forget that and all is lost.
Bjarne Stroustrup

4 Moldable Inspector

Object inspectors are an essential category of tools that allow developers to comprehend the run-time of object-oriented systems. Traditional object inspectors favor a generic view that focuses on the low-level details of the state of single objects. Based on 16 interviews with software developers and a follow-up survey with 62 respondents we identified a need for object inspectors that support different high-level mechanisms to visualize and explore objects, depending on both the object and the current developer need. In this chapter we investigate how to design an object inspector that addresses these requirements following the moldable tools approach.

4.1 Introduction

Understanding the run-time behavior of object-oriented applications entails the comprehension of run-time objects. While debuggers reify the execution stack and allow developers to reason about the control flow of an application, object inspectors give developers direct access to the actual objects. To better understand what software developers expect from an object inspector we performed an exploratory investigation consisting of a series of interviews with software developers and a follow-up survey. We observed a need for object inspectors that:

- support multiple custom views for an object, not limited to text;
- allow developers to easily create new custom views for objects;
- allow developers to explore objects based on more than object state;
- maintain a working-set of inspected objects.

Nevertheless, most of today's object inspectors favor generic approaches to display and explore the state of arbitrary objects. In most cases they represent objects using tree or table views that only contain a textual representation for each object attribute. While universally applicable, these approaches do not take into account the varying needs of developers that could benefit from tailored views and exploration possibilities. We refer to this as the current *inspection problem*.

For example, encountering during debugging objects like folders, files, parsers, HTML/XML documents, database connections, compiled methods, users, accounts, graphical components, *etc.*, leads to a wide range of contextual questions. (*e.g.*, *What files are contained in this folder? What does this graphical component look like? How does this HTML object render in a browser?*) Approaching these contextual questions using generic object inspectors focusing on the state of individual objects leads to an inefficient inspection effort as the information of interest is not directly available or even not accessible from within the inspector.

Mainstream IDEs such as Eclipse, NetBeans, VisualStudio, or IntelliJ allow developers to customize the textual representation of objects, or the layout used to display the state of an object (*e.g.*, show a dictionary as a list of key-value pairs). They further allow developers to run custom code on objects during debugging to construct custom views. The problem with this approach is that it is not reusable: developers have to manually execute the code every time they want to see that view. Developers also have to manually associate views with objects and keep track of the existing views. jGRASP [Cross et al. 2009] improves the inspection process by allowing objects to have visual representations that are not limited to text, and by automatically constructing custom views for objects based on the internal structure of objects (*e.g.*, showing an object representing a tree using a tree view). Debugger Canvas [DeLine et al. 2012] extends the navigation mechanism of traditional object inspectors by displaying each object in a bubble and linking objects in an exploration session: hence, developers can always reason about how they got to an object. Nevertheless, each of these IDEs addresses only parts of the problem as they do not provide developers with a unified workflow for exploring multiple objects using views tailored to their own contextual needs.

To address the overall inspection problem we propose the Moldable Inspector, an object inspector based on the moldable tools approach. 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 provides multiple domain-specific views for each object, makes the inspection context explicit, and uses the inspection context to automatically find, at run-time, views appropriate for the current developer needs. Furthermore, instead of focusing on individual objects, it supports a workflow that does not hardcode the set of reachable objects and groups together multiple levels of connected objects.

This chapter has the following contributions:

- Presenting an exploratory study into how developers perceive and use object inspectors resulting in four developer needs for object inspectors;
- Showing how the moldable tools approach can be applied to create the Moldable Inspector, an inspector model that takes into account the findings of the previous exploratory study;
- Illustrating how the concepts of the Moldable Inspector map to a concrete implementation and discussing various alternatives;
- Real-world examples illustrating the usage of the Moldable Inspector.

Structure of the Chapter

In Section 4.2 (p.43) we describe our investigation into what developers expect from an object inspector. In Section 4.3 (p.49) we introduce the Moldable Inspector model and in Section 4.4 (p.54) we propose a user interface for object inspectors following this model. We illustrate custom workflows made possible by the Moldable Inspector in Section 4.5 (p.56). We discuss implementation aspects and the cost of custom extensions in Section 4.6 (p.65). In Section 4.7 (p.71) we compare our approach with related works in the field of object inspectors and conclude this chapter in Section 4.8 (p.74) by looking at the Moldable Inspector from the point of view of moldable tools.

4.2 Exploratory Study

To better understand how object inspectors should support developer workflows we performed an exploratory study. We designed this exploratory study with the goal of eliciting requirements for improving object inspectors. One can imagine various other approaches for inspecting the state of a program execution, for instance, by visualizing the entire heap and using zoom to get to the object level [Aftandilian et al. 2010], or by writing queries against the state [Lencevicius et al. 1997; Martin et al. 2005]. These approaches complement, and do not replace, the object inspector.

We selected a *sequential exploratory design* [Creswell and Vicki 2006] approach for conducting our study. This is a mixed research methods strategy consisting of a qualitative investigation followed by a quantitative validation.

4.2.1 Qualitative Investigation

The aim of the qualitative investigation was to gain an understanding into what software developers understand by an object inspector and the features they expect from one.

Experience	How many years of experience with object-oriented programming do you have and what object-oriented languages and IDEs did you use?
Definition	What is an object inspector for you?
Features	What features do you need in an ideal object inspector?
Examples	Can you give a few examples of objects that you inspected recently or situations where you used an object inspector?

Table 4.1: Questions for structuring the inspector interviews.

Setup

We performed semi-structured interviews with software developers based on the template questions presented in Table 4.1 (p.44). Based on a set of test runs we agreed on short 10 minute interviews, as the first three questions required only short answers. We also did not require any preparation from the interviewees.

We performed the interviews during the ESUG 2014 conference[1]. Sixteen software developers attending the conference agreed to participate, on a voluntary basis. We collected 2 hours of recordings with an interview lasting $7.6 \pm 2.6(M \pm SD)$ minutes. Participants reported $17.8 \pm 8.2(M \pm SD)$ years of experience with object-oriented programming. Participants also reported using $3.1 \pm 1.9(M \pm SD)$ object-oriented languages until now (*i.e.*, Smalltalk — 100%, Java — 69%, C++ — 31%, Objective C — 31%, and other languages). It is noteworthy that, given the venue, participants were currently working with various Smalltalk dialects; nevertheless, only three participants worked until now solely with Smalltalk dialects. Given the exploratory nature of this phase, we view this as an advantage: in Smalltalk IDEs the object inspector is both a standalone tool and an integral part of the debugger. In other IDEs, for example Eclipse, the object inspector is just a view of the debugger, often not perceived as a distinct tool.

Analysis

The first finding that became clear after a few interviews, and recurred through the rest, was that while participants provided simple definitions for an object inspector (*e.g.*, *"a tool that allows me to inspect the object" — P5*, *"a way to see inside an object" — P9*), they came up with complex features and usage scenarios when asked to give concrete examples. This explains, to some degree, why mainstream IDEs have object inspectors that focus only on the state of single objects: they conform to the perceived definition of what an object inspector is (e.g. *"see all the fields" — P7*). All answers are available in Appendix A (p.161).

[1] http://www.esug.org/Conferences/2014

We further proceeded to analyze the interviews using open coding [Miles and Huberman 1994]: we first transcribed the interviews, and then for each sentence, or groups of sentences closely related to the same topic, attached a label that best described the need or inspector feature mentioned by the developer. We then used the identified concepts to infer a set of high-level developer needs for object inspectors. During the analysis process we aimed to identify a set of developer needs that covered as many of the individual features and examples as possible. We extracted four developer needs detailed in the remainder of this section.

DN1 — I need different ways to view an object depending on my task. All participants expressed, in various forms, the need of having dedicated views for certain types of objects, like collections, dictionaries, tree maps, streams, caches, and graphical elements (*e.g., "If I inspect a color I see RGB values, which is completely unhelpful" — P14*). Six participants further gave examples that required task specific views:

"One thing I really like and I depend on is context sensitive presentations, such as choosing base for numbers. In the VM [...] hexadecimal makes sense, it's what's embedded in the instructions [...], decimal is too hard to parse." — P15

"There is a method that is troublesome, so I inspect it, and the method is just a bunch of bytes. First of all I need a view on the bytecode level [...]. Then I would like to have another view that shows me the source code, then a view that shows me control flow structures." — P5

DN2 - I need to easily extend the inspector with new views for objects. Given the large diversity of objects from today's applications a predefined set of views cannot capture all relevant aspect of all objects. Six participants reported that they actually extended the inspector with custom views for various types of objects, in order to better understand those objects:

"In the beginning I build a special inspector for collection which had a table view. I find it quite useful." — P2

"My thing is graphics, PDFs and especially charts and all my graphical artefacts have special inspector views so that I can see them directly" — P14

"I made so many changes in the inspector to make my life easier so I do not know what a normal object inspector looks like." — P13

DN3 - I need to explore objects connected both explicitly through direct references and implicitly through code logic. Navigating objects solely by following objects attributes can be a laborious process, especially when there is there is no connection between two relevant objects. Six participants gave examples that required navigating to objects not stored in an instance variable of an object already accessible from the inspector:

"It is very often that I expected a kind of way of just following the pointers: which objects point to myself and reverse [...]. This object is well-formed but there is a crash: who uses it?" — P12

"I've written an interface to a storage system in the cloud and it would have been easy to inspect my remote files from the inspector" — P11

DN4 - I need to keep track of the objects that I inspected while working on a given task. Answering run-time questions requires developers to search for relevant objects. Repetitively searching for the same object can cost significant time. Furthermore, losing the history of how one got to an object forces developers to repeatedly retrace their steps. Four participants expressed this concern:

"What costs time is that I usually look at the same objects repeatedly [...] and that I'm always interested in one or two properties of those objects." — P3

"I would like better navigation going to objects and back and remembering where I came from [...]. I might want to mark points [objects] where I say 'this is an interesting point I might want to go back there." — P4

While these four developer needs were the result of analyzing the interviews, they are not necessarily novel, if taken individually, as current object inspectors implement them to some degree. Nevertheless, current object inspectors focus on some of these developer needs while neglecting others. For example, on the one hand, DebuggerCanvas focuses on enabling an easy exploration of multiple objects, while putting the ability to view objects through tailored presentations in the background. On the other hand, the HTML inspector from Firebug[2] allows each HTML element to be viewed through multiple views (*e.g.*, style, layout, DOM), while focusing less on preserving the navigation history. We view the combination of these four developer needs as a novel requirement for object inspectors.

4.2.2 Quantitative Investigation

Setup

To confirm on a larger scale the validity of the previously identified developer needs for object inspectors we conducted a second quantitative investigation consisting of an online survey[3]. We asked respondents to rate each requirement from full disagreement to full agreement on a 5-point Likert scale. For each requirement, we added an example illustrating that requirement and an optional text field where respondents could give personal examples involving that requirement or indicate if they did not understand the requirement. We asked five pre-survey questions about respondents' background.

We advertised the survey on mailing lists of interest for software developers[4] and through social media (*i.e.*, voluntary sampling method). We collected 70 answers over a period of one month from respondents who reported various jobs related

[2] `http://getfirebug.com`

[3] `http://scg.unibe.ch/research/moldableinspector/survey`

[4] `pharo.org` and `moosetechnology.org`

Professional experience	*Respondents*		*Respondents' current job*
4 - 10 years	8	(12.9%)	Software engineer
	6	(9.7%)	Software researcher
	2	(3.2%)	Other
> 10 years	9	(14.5%)	Project manager
	20	(32.3%)	Software engineer
	15	(24.2%)	Software researcher
	2	(3.2%)	Other

Table 4.2: Background data about the survey respondents.

to software engineering (Table 4.2 (p.47) column 3). We discarded all answers from students (7 answers) or from respondents reporting under 1 year of experience with object-oriented programming (1 response), as we wanted to get feedback from respondents with at least some experience in object-oriented programming. We further discarded one answer for *DN4* where the respondent indicated that she did not understand the requirement. This left 62 answers for *DN1*, *DN1* and *DN3*, and 61 answers for *DN4* from respondents whose practical knowledge with object-oriented programming is shown in Table 4.2 (p.47). These respondents also reported using $4.5 \pm 2.2(M \pm SD)$ object-oriented languages until now (*i.e.*, Smalltalk – 89%, Java – 79%, C++ – 45%, Python – 39%, Javascript – 27%, C# – 27%, Ruby – 21%, PHP – 18%). We only take these responses into account in the analysis.

Analysis

Table 4.3 (p.48) summarizes the results of the survey. Overall there was a strong tendency towards the *Strongly agree* and *Agree* answers; no respondent chose the answer *Strongly disagree*. While respondents considered multiple views to be an essential need (100% of respondents agreed or strongly agreed with *DN1*) they considered that easily adding views to an object inspector (*DN2*) is of less importance (72% agreed or strongly agreed, 23% were neutral, while 5% disagree). 23 respondents further used the optional text field of *DN1* to give concrete examples of objects for which they need specific views: list/tree structures, matrices, dictionaries, UI elements, file objects, SQL results, *etc.* The same tendency can be seen for the remaining two developer needs: 90% of respondents agreed or strongly agreed that they need to keep track of the inspected objects (*DN4*), while 77% of respondents agreed or strongly agreed that they need to discover new objects during inspection based on something other than an object's state (*DN3*). Examples of objects for which respondents indicated the need to explore dependencies based on more than object state included callbacks on graphical widgets and pointers.

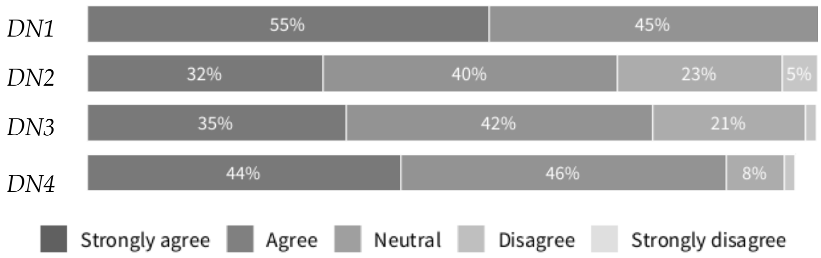


Table 4.3: The results of the quantitative investigation.

4.2.3 Threats to Validity

Internal Validity

The semi-structured interviews were partially moderated by the interviewer. Furthermore, the interviewer knew six interviewees from previous meetings or discussions on mailing lists. While the interviewer did his best not to lead or influence the interviewees we cannot exclude the existence of biased answers (*e.g.*, a slight change in the tone of voice of the interviewer can influence the answer of the interviewee). To minimize the effects of this threat we only included in our analysis developer needs that were explicitly mentioned by four different interviewees. In the survey participants could choose to remain anonymous by not providing an email address (31% of respondents chose to remain anonymous). Nevertheless, respondents had to provide background information about their current job and their experience with object-oriented programming.

External Validity

The software developers interviewed during the first phase were currently working with Smalltalk IDEs, however, just three interviewees had only worked with Smalltalk. Overall, they had a great deal of experience with object-oriented programming ($17.8 \pm 8.2(M \pm SD)$ years) and were exposed to several OO languages ($3.1 \pm 1.9(M \pm SD)$). 89% of the survey respondents marked Smalltalk as one of the languages with which they worked until now, however, these respondents also reported working with $4.6 \pm 2.2(M \pm SD)$ different OO languages; all had more than 4 years of programming experience. Given the vast experience with OO programming of both interviewees and survey respondents, as well as the fact that just three interviewees had only been exposed to Smalltalk, we consider that our findings can apply to other object-oriented programming languages and IDEs rather than only to Smalltalk. Nevertheless, we cannot exclude a bias towards Smalltalk.

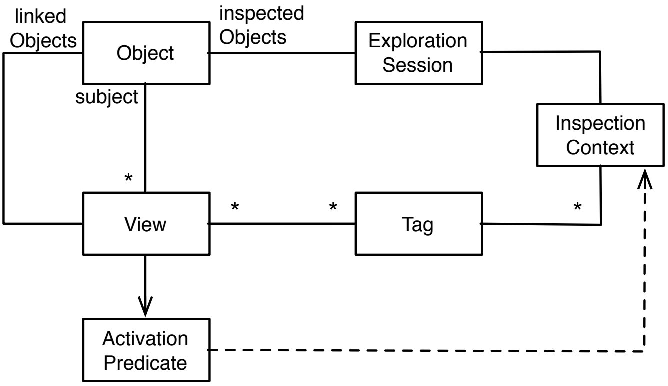


Figure 4.1: The structure of the Moldable Inspector model: an object can have multiple views grouped using tags, and filtered using activation predicates; inspected objects are grouped in an exploration session; the inspection context consists of multiple tags and an exploration session; the inspection context is used to filter views.

4.2.4 Summary

While developers define an object inspector in simple terms they actually expect a lot from an object inspector. Based on 16 interviews with software developers we have identified four developer needs regarding object inspectors. Through an online survey we saw a level of agreement with these developer needs ranging from 72% to 100%. This exploratory study indicates a need for object inspectors that better support developers in reasoning about and exploring specific aspects of their own domain objects.

4.3 The Moldable Inspector in a Nutshell

To address the aforementioned developer needs we propose the Moldable Inspector, a model (Figure 4.1 (p.49)) for constructing object inspectors that can be adapted during the inspection process to suit the immediate needs of developers. Following the moldable tools approach this is achieved in two main steps:

(i) developers create custom extensions for viewing and exploring their domain objects;

(ii) at run time the Moldable Inspector selects extensions (*i.e.*, views) appropriate for the current objects and developer needs.

4.3.1 Running Example

Consider that during debugging a developer has to interact with an object representing a widget (graphical component). A generic state view only showing object attributes — size, bounds, visual properties, *etc.* — helps the developer reason about the internal representation of that object. However, depending on her current needs she could ask more specific questions like:

What does this widget look like?

What keyboard shortcuts are associated with this widget?

How to properly initialize and use this widget?

What objects hold a reference to this widget?

What other widgets are contained inside this widget?

Furthermore, depending on the context, a developer could need to explore other objects useful in reasoning about that widget, not necessarily referred through an attribute of that widget (*e.g.*, canvas — the graphical surface on which the widget is rendered, renderer — the object rendering the widget on a canvas).

4.3.2 Enabling Customization

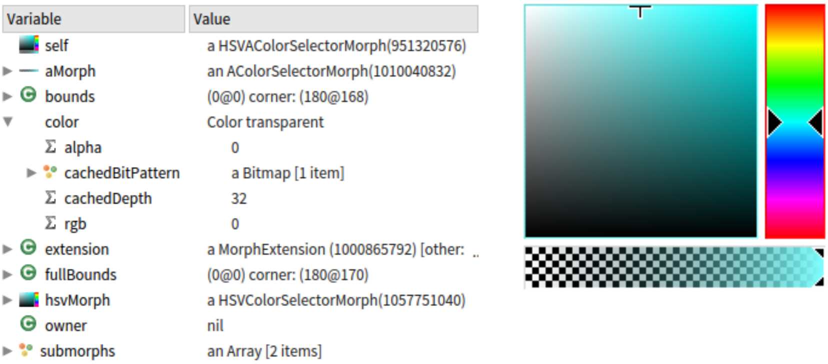
The Moldable Inspector model enables custom extensions through two operators:

multiple views: allow each object to have multiple custom views;

flexible navigation: discover new objects by either direct or indirect object references.

Multiple Views

Most inspectors represent an object generically by displaying its state as a tree or a table, but even in these cases there exists at least one custom string representation that is left to the developer to specify (*e.g.*, `toString()` in Java). However, given that objects model domain concepts they can also have domain-specific representations. *DN1* further enforces the need for multiple views. To address this the Moldable Inspector allows each object to have multiple custom views. For example, a widget object can have views that directly show: its state (Figure 4.2a (p.51)), the code of its class, its visual representation (Figure 4.2b (p.51)), its keyboard shortcuts, the other graphical objects that it contains, or what objects hold a reference to it.



Variable	Value
self	a HSVAColorSelectorMorph(951320576)
▶ aMorph	an AColorSelectorMorph(1010040832)
▶ bounds	(0@0) corner: (180@168)
▼ color	Color transparent
Σ alpha	0
▶ cachedBitPattern	a Bitmap [1 item]
Σ cachedDepth	32
Σ rgb	0
▶ extension	a MorphExtension (1000865792) [other: ..
▶ fullBounds	(0@0) corner: (180@170)
▶ hsvMorph	a HSVColorSelectorMorph(1057751040)
owner	nil
▶ submorphs	an Array [2 items]

(a) (b)

Figure 4.2: Two views for displaying a widget: (a) state; (b) visual representation.

Flexible Navigation

Developers need support for navigating through object models not only by following object attributes but also by considering other types of dependencies (*DN3*). The Moldable Inspector allows each view to specify a set of related objects, together with the mechanism for navigating to those objects. For example, a view showing the graphical representation of a widget can allow developers to navigate to any sub-widget by clicking it. Furthermore, a view can allow developers to navigate to new objects by either constructing or locating those objects using snippets of code executed in the context of the displayed object (*e.g.*, In Figure 4.3 (p.52) a developer navigates from a widget to the context menu of that widget, by using a custom code snippet — `self getMenu: false` — to create the menu).

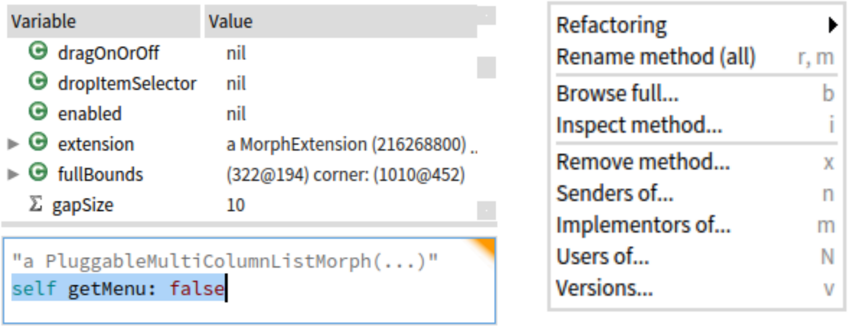
4.3.3 Inspection Context

One problem is still not addressed: *How does a developer select the right views for her current needs?* As argued in Section 1.4 (p.6), moldable tools should select extensions based on the development context. We refer to the development context of an object inspector as an *inspection context*. The Moldable Inspector explicitly models the current inspection context and uses it to determine what views to show. This is achieved using three operators:

tag: groups together views applicable for a development task;

exploration session: groups all objects inspected during an inspection session;

activation predicate: determines if a view is valid or not in the current context.



Variable	Value
dragOnOrOff	nil
dropItemSelector	nil
enabled	nil
extension	a MorphExtension (216268800) ..
fullBounds	(322@194) corner: (1010@452)
Σ gapSize	10

```
"a PluggableMultiColumnListMorph(...)"
self getMenu: false
```

(a) (b)

Figure 4.3: Navigating from a widget (a) to its context menu (b). The menu is not stored in an instance variable of the widget. It can only be constructed by invoking a method of the widget.

An inspection context consists of multiple tags and an exploration session. Hence, the inspector context provides a dedicated mechanism to model the development context for an object inspector: the exploration session contains interaction data; tags are pieces of informations explicitly added by a developer to the context. Activation predicates filter views based on the inspection context.

Tags

Depending on the current developer needs not all available views are of interest; this is the main requirement captured by *DN1*. To help developers discover useful views (and filter unneeded ones) the Moldable Inspector proposes the use of *tags* to identify and group together views applicable for certain types of development tasks. For example, when a developer is interested in the visual representation of a widget she can select to see only those views tagged as showing visual representations, and not those views that show more technical details about the widget (*e.g.*, keyboard shortcuts, pointers, code). When looking for example on how to use a widget she can select the *examples* tag to only see views showing explicit usage examples. Given that different types of development tasks can have overlapping needs the Moldable Inspector allows each view to have multiple tags.

Exploration Session

Inspection sessions can be extensive and can involve many steps. In these situations, developers need to keep track of and go back to previously inspected objects that are relevant for their current development task (*DN4*). To support this use case the

Moldable Inspector stores all inspected objects in an *exploration session*, together with the order in which they were inspected and the type of views selected for each object. This enables developers to determine how they got to the current object, and to go back to any of the previously inspected objects and try alternative exploration paths. If during an exploration a developer inspects by mistake objects not related to the current task, they are kept in the session unless the developer removes them.

Activation Predicates

Tags offer a solution to filter views based on the development task. Nevertheless, the state of the current object, as well as all the previously inspected objects can have an impact on what views are appropriate for the current object. Consider an object representing a file from disk. The same type of object is usually used to refer to files representing text, pictures, HTML documents, executables, *etc.* A view showing a visual representation of a file only applies to files that have a visual representation (*e.g.*, jpeg, png, gif); this view is not applicable to executables. To enable this use case, the Moldable Inspector attaches to each view an *activation predicate*. As discussed in Section 3.2.4 (p.39), an activation predicate captures a boolean condition applied on the current inspection context. The Moldable Inspector used activation predicates to decide whether to display a view. An activation predicate can filter views based on the currently inspected object as well as based on the entire exploration session.

While this feature can make the interface less cluttered, it may also surprise the programmer if she cannot easily tell why the Moldable Inspector decided (not) to enable a particular view. Hence, while necessary for certain objects, this feature should not be abused.

4.3.4 Addressing the Initial Developer Needs

In this section we indicate how the Moldable Inspector addresses the four developer needs identified in Section 4.2.1 (p.43).

DN1: Every object can have multiple views; based on their task developers can filter views using the inspection context. Displaying multiple views is discussed in Section 4.4 (p.54).

DN2: The presented model enables developers to add any kind of view to an object. The actual mechanism for constructing and adding views to objects directly influences the ease of these activities. We investigate these aspects in details inSection 4.6 (p.65).

DN3: Each view can offer an appropriate mechanism for navigating to the next object (*e.g.*, select an object attribute, execute code, write a query).

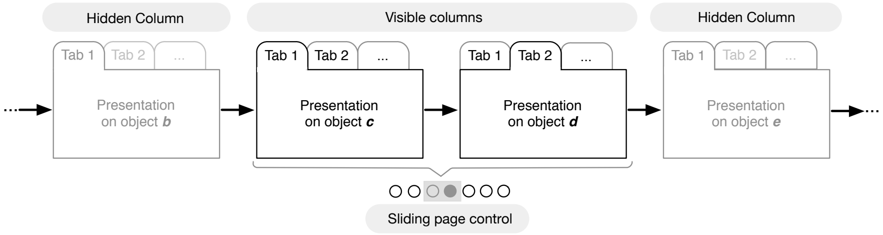

Hidden Column
Visible columns
Hidden Column
Tab 1
Tab 2
...
Presentation on object *b*
Presentation on object *c*
Presentation on object *d*
Presentation on object *e*
Sliding page control

Figure 4.4: The Object Pager user interface in a nutshell: each object is displayed in a single column as a tabbed widget that groups together a set of views; new objects are displayed to the right; an augmented scrollbar improves navigation. Objects are displayed in columns of equal size and it is not possible to reach a situation where a column is partially displayed, as the sliding bar always repositions to show full columns.

DN4: Previously inspected objects are grouped into an exploration session. Section 4.4 (p.54) discusses UI decisions concerning navigating and displaying an exploration session.

4.4 Compact, Efficient Object Exploration

A concrete inspector requires a concrete user interface. The decisions taken to realize that interface, such as how to show multiple views and how to navigate through objects, can have a significant influence on the utility of the inspector. In this section, we present the Object Pager, a user interface for object inspectors that implement the Moldable Inspector model. The Object Pager proposes a compact means to explore a space of run-time objects that aims to minimize screen real estate and reduce spatial maintenance effort.

4.4.1 Displaying Multiple Views for an Object

Baldonado *et al.* introduced eight rules for the design of systems having multiple views [Wang Baldonado et al. 2000]. The fifth rule, *Space/Time Resource optimization*, comments that *"it is easy to forget to account for the display and computation time required to present multiple views side by side; likewise, it is easy to account for the time saved by side-by-side views if the user's goal is to compare views"* [Wang Baldonado et al. 2000].

Considering how much information is displayed in current debuggers and IDEs, screen real estate is a scarce resource. Furthermore, given that each view of an object highlights a specific aspect, we do not consider that directly comparing two views of the same object is an essential activity; what should rather be optimised is the process of finding the right view. Taking into account these arguments, the Object

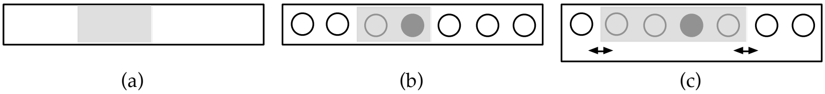


Figure 4.5: Scrollbars for navigating through an exploration session: (a) standard scrollbar with limited navigation support; (b) improved scrollbar with overview support; (c) increasing / decreasing the number of visible objects in the improved scrollbar.

Pager shows only one view for each object at a time and groups all available views of an object using tabs (Figure 4.4 (p.54)).

4.4.2 Representing an Exploration Session

Approaches displaying complete exploration sessions (*e.g.*, by using tree / graph-based structures or matrices) can take considerable screen real estate and require developers to explicitly remove paths that are no longer of interest. Consider DebuggerCanvas [DeLine et al. 2012], a debugger promoting a user interface based on the CodeBubble paradigm [Bragdon et al. 2010b]. While DebuggerCanvas can display complete exploration sessions it occupies the whole display of the IDE.

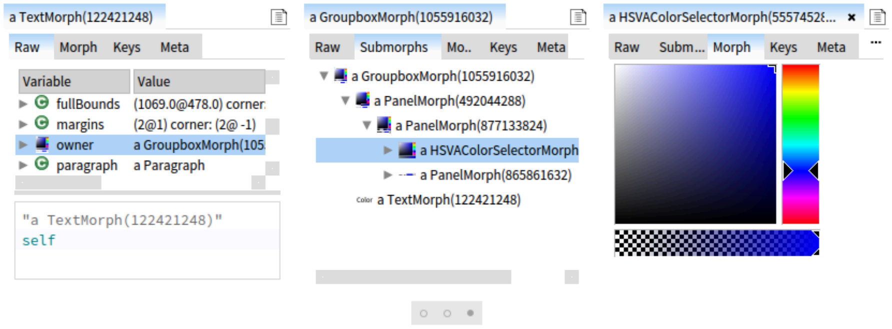
To minimize the usage of screen real estate and reduce interaction overhead, the Object Pager displays only one exploration path at a time and automatically arranges the inspected objects using Miller columns,[5] a technique for navigating hierarchical structures on a horizontal boundless tape, where multiple levels of the hierarchy can be seen at once and each new level is opened in a new column to the right. Figure 4.4 (p.54) shows how several objects are displayed using this approach: the order in which these objects were inspected is given by their positioning from left to right.

4.4.3 Navigating Through an Exploration Path

From a dedicated view of an object in one tab of a Miller column, one can navigate to a view of another object in the next column by selecting a given object, or constructing an object in the view. Whenever a developer selects an object in a dedicated view from a Miller column all the columns to the right of that column are removed, and a new column displaying the selected object is spanned to the right. This ensures that only one exploration path is displayed at a time.

Given that exploration paths can entail a large number of objects [Minelli et al. 2014], navigation back and forth through an exploration path becomes an explicit issue. Simple scrollbars, while enabling fast movement between columns, have several

[5] `http://en.wikipedia.org/wiki/Miller_columns`



Variable	Value
fullBounds	(1069.0@478.0) corner:
margins	(2@1) corner: (2@ -1)
owner	a GroupboxMorph(105
paragraph	a Paragraph

```
"a TextMorph(122421248)"
self
```

```
▼ a GroupboxMorph(1055916032)
  ▼ a PanelMorph(492044288)
    ▼ a PanelMorph(877133824)
      ▶ a HSVAColorSelectorMorph
      ▶ a PanelMorph(865861632)
    a TextMorph(122421248)
```

Figure 4.6: An exploration session involving multiple graphical components.

shortcomings [Alexander et al. 2009] when navigating through Miller columns. For example, it is difficult to tell that the scrollbar from Figure 4.5a (p.55) supports navigation through an exploration path containing seven objects, where two objects are currently visible.

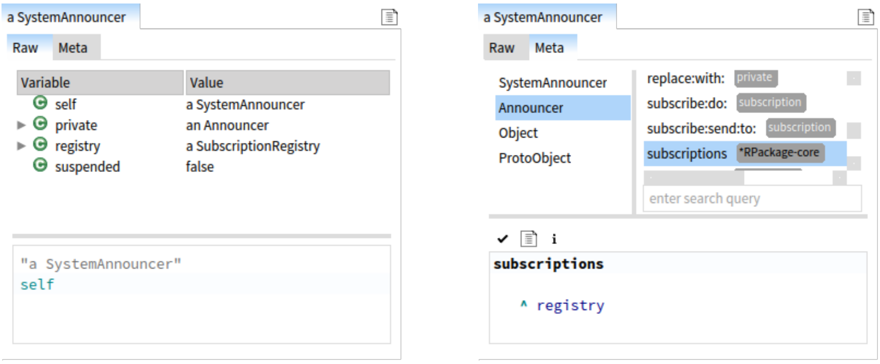
To address this problem Object Pager proposes an augmented scrollbar [Alexander et al. 2009; Chimera 1992] (following the *overview+detail* approach [Plaisant et al. 1995]) that incorporates an icon for each object, highlights the icon of the currently selected object, and enables the developer to change the number of visible objects by expanding the sliding bar (Figure 4.5c (p.55)); the sliding bar indicates the visible objects. Figure 4.5b (p.55) shows how this approach is used to navigate through the same exploration path as in Figure 4.5a (p.55); now a developer can immediately see that the path has seven objects and that two objects are currently visible.

4.5 Custom Workflows

The main goal of the Moldable Inspector is to support custom workflows. To show that this is indeed the case, we present concrete cases of how the model together with Object Pager user interface enables several such workflows, and how this is accomplished by relying exclusively on the previously identified developer needs. In doing this we show that the Moldable Inspector addresses *DN1*, *DN3* and *DN4*.

4.5.1 Multiple Views for Every Object

The Moldable Inspector enables each object to have multiple views (*DN1*). We give examples of views common to all objects and look in detail at views for two specific objects.



Variable	Value
self	a SystemAnnouncer
private	an Announcer
registry	a SubscriptionRegistry
suspended	false

```
"a SystemAnnouncer"
self
```

```
subscriptions

    ^ registry
```

(a)

(b)

Figure 4.7: Generic inspector views for every object: (a) the *Raw* view shows the state of the object; (b) the *Meta* view gives developers access to the class hierarchy of the inspected object. The inspected object in this example is an announcer object used by the Pharo IDE to deliver internal events to registered subscribers.

Generic Views

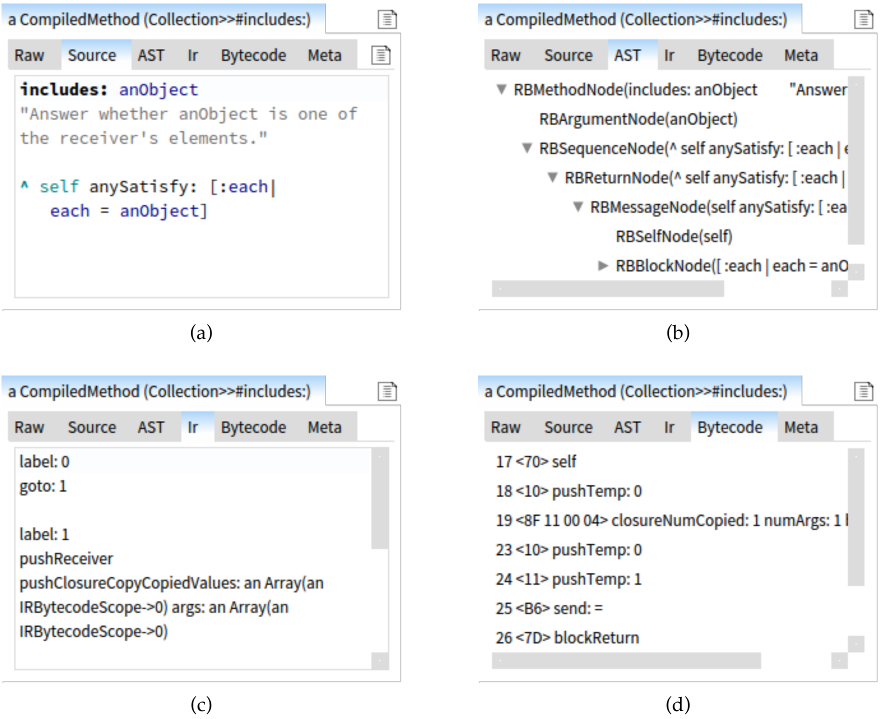
Every object has a *Raw* view (Figure 4.7a (p.57) — first object) that gives access to the state of the object (*i.e.*, object attributes). This view corresponds to what a traditional inspector focusing only on object state offers. Besides state, an object also knows its class. Thus, another generic view offers a source code editor of the corresponding class (*Meta* view, Figure 4.7b (p.57)).

Multiple Views for Graphical Objects

As highlighted in Section 4.3 (p.49), several aspects of a widget can be of interest to a developer depending on the task, such as:

- the state when examining the implementation;
- the visual representation when fixing a rendering bug.

As a concrete example we use Morphic [Maloney and Smith 1995] the main library for creating user interfaces in Pharo, the target language for our current implementation. In Morphic, graphical objects are instances of the class `Morph` and are referred to as *morphs*. A morph can further contain other morphs (referred to as *submorphs*) forming a tree structure. The state of a morph can be accessed using the aforementioned *Raw* view. To support the visual aspect we added two specific views to every morph object, showing their visual representation (*Morph* view, Figure 4.6 (p.56)) and structure of submorphs (*Submorphs* view, Figure 4.6 (p.56)).



```
includes: anObject
"Answer whether anObject is one of
the receiver's elements."

^ self anySatisfy: [:each|
  each = anObject]
```

(a)

```
▼ RBMethodNode(includes: anObject     "Answer
    RBArgumentNode(anObject)
  ▼ RBSequenceNode(^ self anySatisfy: [ :each | e
    ▼ RBReturnNode(^ self anySatisfy: [ :each |
      ▼ RBMessageNode(self anySatisfy: [ :ea
          RBSelfNode(self)
        ▶ RBBlockNode([ :each | each = anO
```

(b)

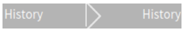
```
label: 0
goto: 1

label: 1
pushReceiver
pushClosureCopyCopiedValues: an Array(an
IRBytecodeScope->0) args: an Array(an
IRBytecodeScope->0)
```

(c)

```
17 <70> self
18 <10> pushTemp: 0
19 <8F 11 00 04> closureNumCopied: 1 numArgs: 1 b
23 <10> pushTemp: 0
24 <11> pushTemp: 1
25 <B6> send: =
26 <7D> blockReturn
```

(d)

Figure 4.8: Domain-specific views for `CompiledMethod` objects: (a) Source code; (b) Abstract syntax tree; (c) Intermediate representation; (d) Bytecode.

A visual representation for a morph is particularly useful when investigating rendering bugs. Consider the following drawing glitch from an implementation of a breadcrumb:  (when there are multiple elements in the breadcrumb, due to rounding errors in calculating the width of each element, there can be a one pixel gap between some elements[6]). To investigate this bug a developer can inspect the breadcrumb morph, use the *Meta* view to edit the code that computes the width, and use the *Morph* view to check if the gap is still there.

Multiple Views for Compiled Code

Methods are represented in Pharo as instances of the `CompiledMethod` class and they hold the corresponding bytecode needed by the virtual machine. A common task when working with these objects (*e.g.*, for developing tools like compilers or debuggers) is to understand how source code maps to bytecode and vice-versa. Bugs in this kind of code can be particularly difficult to debug[7]without proper tool support, as

[6] `http://pharo.fogbugz.com/f/cases/15227`

the mapping involves several steps: parsing the source code into an abstract syntax tree (AST), translating the AST into an intermediate representation (IR), performing various optimizations at the level of the IR and finally translating the IR into the actual bytecode. Inspecting just the attributes of a `CompiledMethod` object provides little help as they only give details about the format in which bytecode is represented (header, literals, trailer). To address this and improve the inspection of compiled code we created, together with the developers of the Pharo compiler, four specific views to `CompiledMethod` objects:

- *Source code*: the original source code from which the `CompiledMethod` object was generated (*Source* view, Figure 4.8a (p.58));
- *AST*: the AST obtained by parsing the source code (*AST* view, Figure 4.8b (p.58));
- *IR*: the intermediate representation (IR) obtained from the AST (*Ir* view, Figure 4.8c (p.58));
- *Bytecode*: the bytecode instructions stored by the method object (*Bytecode* view, Figure 4.8d (p.58)).

4.5.2 Navigating Through Connected Objects

Since navigation between objects based only on object state reduces the available space to accessible objects the Moldable Inpector allows each view to specify its own navigation mechanism (*DN2*). We show that developers can navigate to objects not stored in the current object, use code to guide their navigation and track their exploration history (*DN4*).

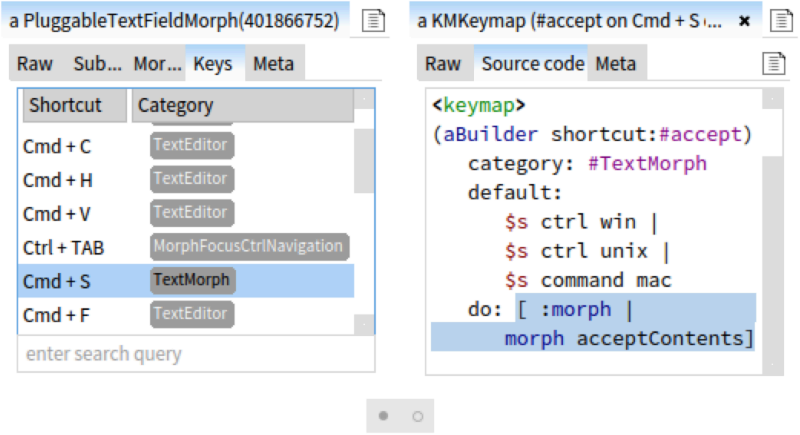
Browsing Indirectly Connected Objects

Each morph object can have a list of key bindings that map keyboard shortcuts to anonymous functions to be executed when the associated shortcut is invoked and the morph has the focus (*e.g.*, pressing CMD+S in a text editor morph triggers an action for saving the content from that editor).

Debugging bugs related to wrong key bindings requires developers to first determine what key bindings are associated with a morph and what code gets executed when a key binding is invoked. However, key bindings are not stored within the morph, but within a global object managing all key bindings for all morphs. Hence, it is often not trivial to determine the key bindings of a morph object as they cannot be accessed using the state view[8]. To address this we added a dedicated view showing a list of keyboard shortcuts associated with the morph (*Keys* view, Figure 4.9 (p.60)). By

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

selecting a shortcut in this view a developer navigates to the `KMKeymap` object that maps the shortcut with the anonymous function executed when the shortcut is invoked; this object has a *Source code* view showing and highlighting the source code of the anonymous function (Figure 4.9 (p.60)).


a PluggableTextFieldMorph(401866752)
Raw
Sub...
Mor...
Keys
Meta
enter search query
a KMKeymap (#accept on Cmd + S ...
Raw
Source code
Meta

Shortcut	Category
Cmd + C	TextEditor
Cmd + H	TextEditor
Cmd + V	TextEditor
Ctrl + TAB	MorphFocusCtrlNavigation
Cmd + S	TextMorph
Cmd + F	TextEditor

```
<keymap>
(aBuilder shortcut:#accept)
   category: #TextMorph
   default:
      $s ctrl win |
      $s ctrl unix |
      $s command mac
   do: [ :morph |
      morph acceptContents]
```

Figure 4.9: Using specific views to browser the code that gets executed when a user presses CMD+S.

Navigating to New Objects

While a `CompiledMethod` object has views showing its bytecodes and source code, addressing the bugs mentioned in Section 4.5.1 (p.58) further requires developers to repeatedly determine what source code corresponds to what bytecode. Due to the complexity of the compilation process this is not an easy task. To directly support this task when a developer selects a bytecode in the *Bytecode* view, a `SymbolicBytecode` object representing that bytecode is created and opened in a new view to the right; each object representing a `SymbolicBytecode` has a view showing the entire source code of the method and highlighting the part of the source code that corresponds to that bytecode (Figure 4.12 (p.63)).

Using Code to Guide the Navigation Process

Constructing and previewing queries over relational databases is typically done in dedicated database client tools that are far away from the development environment. However, when working with relational data, querying is a common activity during software development.

[8] `http://pharo.fogbugz.com/f/cases/14845/`
`http://forum.world.st/How-to-browse-a-given-keymap-category-td4803022.html`

In Pharo, Opening a connection to a Postgres[9] database creates an object of type `PGConnection`. Viewing this object in a traditional object inspector only shows details about the state of the connection (*e.g.*, location, port number, start time). Yet, a typical use case is to interact with the content of the underlying database. To address this, a `PGConnection` object has a dedicated view that allows developers to write and execute SQL queries on that connection (*SQL* view, first object — Figure 4.10 (p.61)). Developers can use the query result to continue navigation. At any time a developer can reason about how she got to the current object, as all previously inspected objects are available in the inspector.

Not only SQL queries can be used to guide navigation, but any other piece of code. For example, in Figure 4.10 (p.61) after a developer executes a SQL query, she uses a snippet of code to create a visual representation of the query result. The snippet of code returns an object of type `GET2DiagramBuilder` which has a view showing a graphical representation of the constructed visualization. This enables workflows that can seamlessly incorporate custom visualizations.

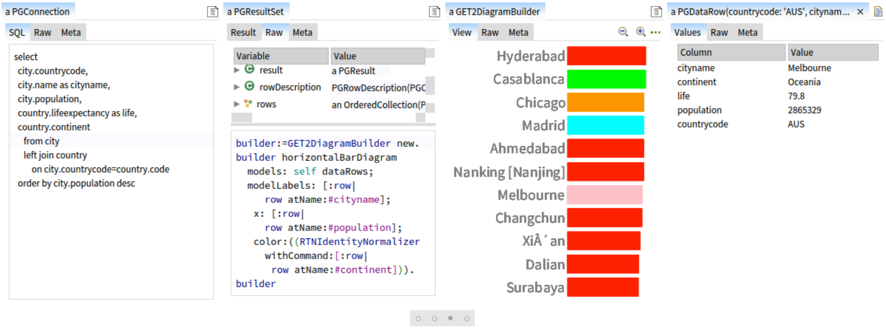
4.5.3 Selecting Views Based on the Inspection Context

Not all views of an object are of interest all the time (*DN1*). The Moldable Inspector filters views based on the inspection context. We show how each component of the inspection context is used to filter views.

Selecting Views Using Activation Predicates

Viewing the internal representation of an object modeling a file or a folder from disk does not provide any insight into the content of that file or folder. For example, in Pharo, objects of type `FileReference` represent files and folders. The state of a

[9] `http://www.postgresql.org`



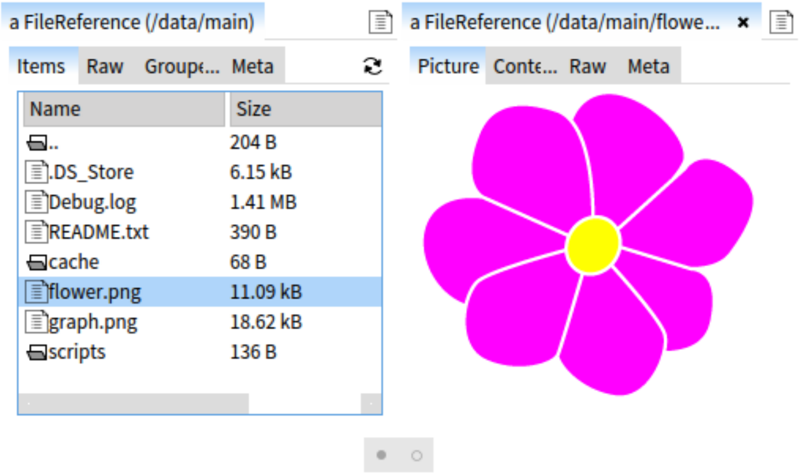
```
select
 city.countrycode,
 city.name as cityname,
 city.population,
 country.lifeexpectancy as life,
 country.continent
   from city
   left join country
     on city.countrycode=country.code
 order by city.population desc
```

Variable	Value
▶ result	a PGResult
▶ rowDescription	PGRowDescription(PGC
▶ rows	an OrderedCollection(P

```
builder:=GET2DiagramBuilder new.
builder horizontalBarDiagram
  models: self dataRows;
  modelLabels: [:row|
     row atName:#cityname];
   x: [:row|
     row atName:#population];
   color:((RTNIdentityNormalizer
     withCommand:[:row|
      row atName:#continent])).
builder
```

Column	Value
cityname	Melbourne
continent	Oceania
life	79.8
population	2865329
countrycode	AUS

Figure 4.10: Exploring the content of a database.


a FileReference (/data/main)
Items
Raw
Groupe...
Meta
a FileReference (/data/main/flowe...
Picture
Conte...
Raw
Meta

Name	Size
..	204 B
.DS_Store	6.15 kB
Debug.log	1.41 MB
README.txt	390 B
cache	68 B
flower.png	11.09 kB
graph.png	18.62 kB
scripts	136 B

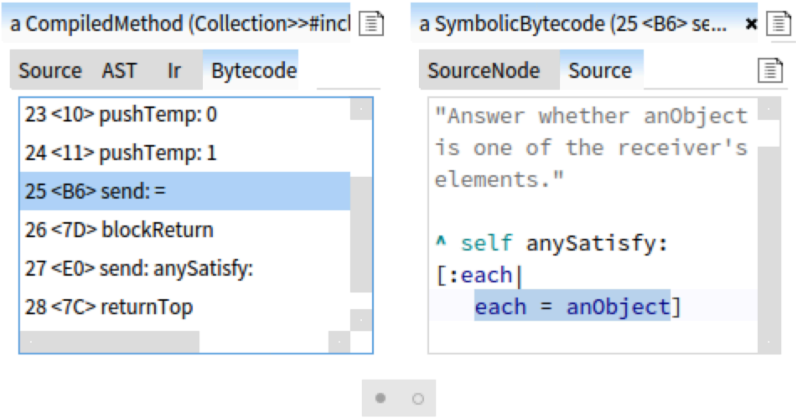
Figure 4.11: Browsing the content of a folder.

`FileReference` object only gives information about the location of the file / folder; the content is not accessible. To address this issue we attached a *Content* view to each `FileReference` object that is not a directory displaying the content of that file in text form. We further attach to objects of type `FileReference` that represent folders a view that shows the list of files and folders from that folder and allows developers to navigate to any of them. This turns the object inspector into a file browser (Figure 4.11 (p.62)).

While the *Content* view is applicable for all file types, it is not appropriate for all file types (*e.g.*, photos, mp3 or executable files). To overcome this limitation we further added several views, each applicable to a `FileReference` object only if that object has a particular extension. For example, a file object storing a picture (*i.e.*, .png, .gif, .jpg) has a view that shows the actual picture (*Picture* view, Figure 4.11 (p.62)), a file object containing a Pharo script (*i.e.*, .st) has a view that shows the script using syntax highlighting, an archive object (*i.e.*, .zip) has views that show the archived files / folders and the compressed content in hexadecimal, *etc.* This is achieved by relying on activation predicates that check the extension of the file object, and turns the object inspector into a dedicated file browser. Activation predicates are modeled as anonymous functions that return a boolean value.

Selecting Views Using Tags

The *Raw* and *Meta* views, showing object state and the source code of an object's class, get in the way if a developer is interested just in a domain-specific aspect of an object, such as the files in a folder or the visual representation of a morph. The same can be said if a developer is only interested in the object's state: showing several


a CompiledMethod (Collection>>#incl
Source
AST
Ir
Bytecode
a SymbolicBytecode (25 <B6> se...
SourceNode
Source

```
23 <10> pushTemp: 0
24 <11> pushTemp: 1
25 <B6> send: =
26 <7D> blockReturn
27 <E0> send: anySatisfy:
28 <7C> returnTop
```

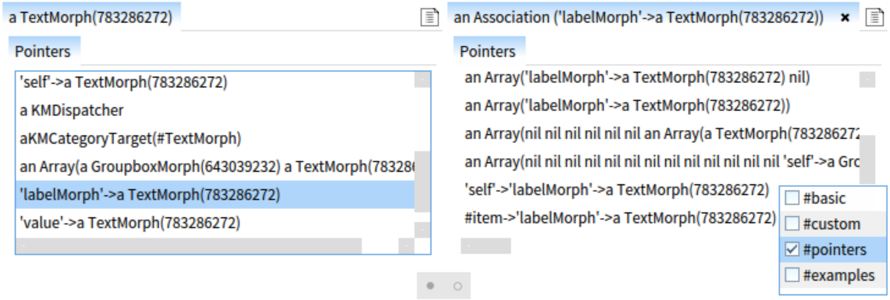
```
"Answer whether anObject
is one of the receiver's
elements."

^ self anySatisfy:
[:each|
    each = anObject]
```

Figure 4.12: Exploring how bytecode maps to source code. The object inspector only contains the *custom* tag in the inspection context. Hence, it displays only domain-specific views.

specific views can get in the way. To make it possible to dynamically select only those views that are currently of interest we group them using tags.

By default, the Moldable Inspector comes with two tags: *basic* and *custom*. The *basic* tag groups generic views applicable for all objects; currently these are the *Raw* and *Meta* views. The *custom* tag groups domain-specific views. Developers can further create and use their own custom tags instead of just relying on the *basic* and *custom* tags. The Moldable Inspector only displays views that have at least a tag present in the current inspection context. For example, the object inspector from Figure 4.17 (p.68) has only the *basic* tag in the current inspection context; the one from Figure 4.12 (p.63) only has the *custom* tag.

Using custom tags we can transform the object inspector into a tool addressing spe-

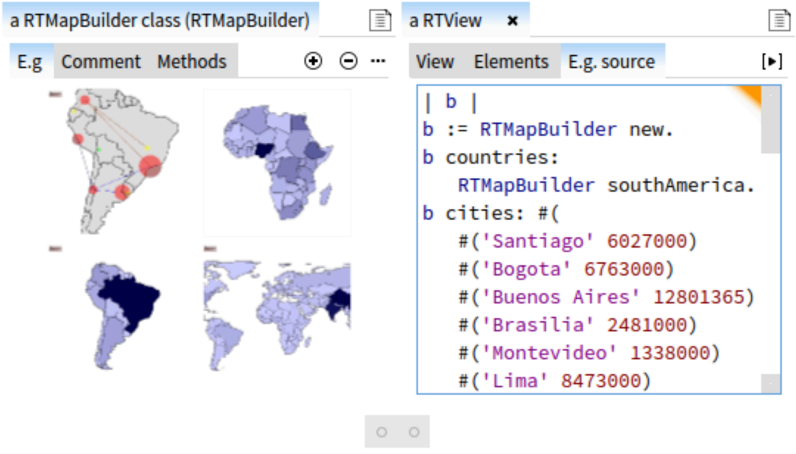


```
'self'->a TextMorph(783286272)
a KMDispatcher
aKMCategoryTarget(#TextMorph)
an Array(a GroupboxMorph(643039232) a TextMorph(78328
'labelMorph'->a TextMorph(783286272)
'value'->a TextMorph(783286272)
```

```
an Array('labelMorph'->a TextMorph(783286272) nil)
an Array('labelMorph'->a TextMorph(783286272))
an Array(nil nil nil nil nil nil an Array(a TextMorph(78328627
an Array(nil nil nil nil nil nil nil nil nil nil nil nil 'self'->a Grc
'self'->'labelMorph'->a TextMorph(783286272)
#item->'labelMorph'->a TextMorph(783286272)
```

□ #basic
□ #custom
☑ #pointers
□ #examples

Figure 4.13: Exploring points-to relations between objects; only the tag *pointers* is in the current inspection context.



```
| b |
b := RTMapBuilder new.
b countries:
   RTMapBuilder southAmerica.
b cities: #(
   #('Santiago' 6027000)
   #('Bogota' 6763000)
   #('Buenos Aires' 12801365)
   #('Brasilia' 2481000)
   #('Montevideo' 1338000)
   #('Lima' 8473000)
```

Figure 4.14: Browsing the examples of class.

cific kinds of development tasks. For example, when investigating memory leaks[10], tools that allow developers to track points-to relations between objects (*i.e.*, what objects point to a given objects) can provide valuable insight. We added a view to all objects that shows all the objects holding a reference to the displayed object. As this view is only useful in certain cases we assigned a dedicated tag to it (*i.e.*, *pointers*). When only this tag is in the inspection context, the inspector becomes a tool for exploring points-to relations. This can be seen in Figure 4.13 (p.63), where a developer investigates what objects point to a given morph using only the *Pointers* view. Apart from the *pointers* tag, the current set of extensions for the Moldable Inspector contains one more custom tag: *examples*. This tag is attached to views that display examples or the source code of an example (*e.g.*, the views *E.g.* and *E.g. source* in Figure 4.14 (p.64)).

Selecting Views Based on the Exploration Session

Examples are useful for developers when looking for how to instantiate objects of a certain type. Nevertheless, examples are not always easy to find.

To provide usage examples for a class we take advantage of the fact that in Pharo classes are also objects and we add to every class a view that shows a list of examples of how to instantiate that class. Developers can add examples as methods in the class object (this corresponds to static methods in Java). When a developer selects an example in the view, the associated method is executed and the constructed object is displayed to the right. The developer can then inspect the state and any specific aspect of the created object. However, in this case, the code that created the example

[10] http://forum.world.st/Some-Memory-Leak-td4814779.html

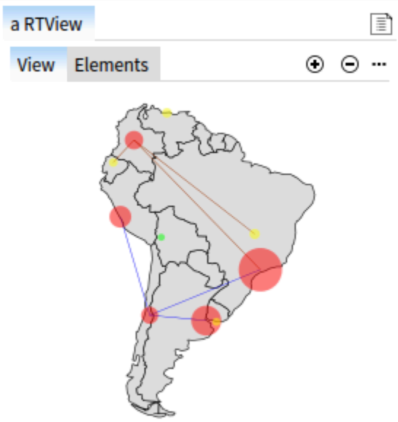
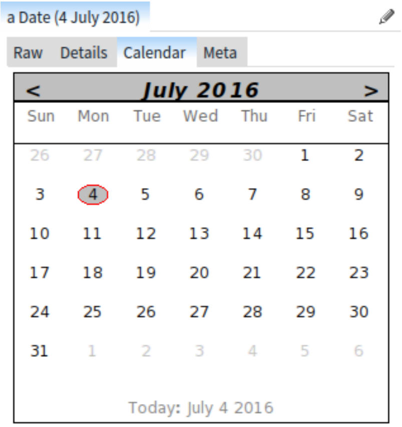


Figure 4.15: Inspecting an example object in isolation.


a Date (4 July 2016)

Raw Details Calendar Meta

< July 2016 >

Sun	Mon	Tue	Wed	Thu	Fri	Sat
26	27	28	29	30	1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31	1	2	3	4	5	6

Today: July 4 2016

Figure 4.16: Viewing a date object using a calendar widget.

is the most important part. To show it in the inspector, we add, to every object, a view whose activation predicate checks if the previously inspected object in the current exploration path is a class displayed using a view showing examples.

For example, in Figure 4.14 (p.64) a developer inspects the class `RTMapBuilder` providing support for building visualizations containing maps. She then switches to the *E.g.* view and clicks on the icon of an example. This executes that example and opens the resulting object in a new column to the right. She can then select the *E.g. source* view showing the source code that created that example. Inspecting the same object in isolation will not show the *E.g. code* view as the object is not inspected in the context of an example (Figure 4.15 (p.65)).

4.6 Discussion

In this section we explore aspects related to implementing the Moldable Inspector, discuss the cost for creating a view in terms of lines of code, and analyze the types of views currently used to render objects in Pharo.

4.6.1 Implementation Aspects

To validate the proposed approach and show that it has practical applicability, we implemented the Moldable Inspector concept in Pharo as part of the Glamorous Toolkit project. We also integrated the prototype implementation into the alpha version of Pharo 4, replacing the previous object inspector, and we iteratively improved the implementation as we obtained feedback from developers relying on the alpha version of Pharo 4 in their day-to-day activities. The Moldable Inspector became the default object inspector in the Pharo 4 release. Initial feedback indicated that while the Moldable Inspector can take some getting used to, as it has a different navigation mechanism than previous inspectors from Pharo, it can significantly improve the inspection process.

Constructing a Specific View for an Object

In the current implementation we aimed for an object inspector that allows developers to use any graphical object (*i.e.,* morph) as a view. To this end we enable developers to manually construct views using code snippets that return a view object. Given that developers reported the need of easily extending the inspector, we provide an internal domain-specific language (*i.e.,* an API) that can be used to directly instantiate several types of basic views such as list, tree, table, text and code. Each view is then modeled as an object. When creating a view developers first have to indicate the type of the view and then configure its properties using anonymous functions. The proposed DSL also makes it easy to integrate more elaborate views created using a visualization library (Roassal [Araya et al. 2013]) and a data browser library (Glamour [Bunge 2009]).

For example, lines 4-9 from Listing 4.1 (p.66) illustrate how to instantiate a tree view showing the submorphs of a morph (*Submorphs* view, Figure 4.6 (p.56)). This view should only be available if the morph has any submorphs. To check this the extension specifies an activation predicate using the `#when:` method (line 10). The actual activation predicate consists in an anonymous method returning a boolean value.

Listing 4.1: Creating a view showing the structure of a graphical component.

```
1  Morph>>#gtInspectorDisplaySubmorphsOn: aCanvas
2      <gtInspectorPresentationOrder: 80>
3      <gtInspectorTag: #custom>

4      ↑ aCanvas tree
5          title: 'Submorphs';
6          rootsExpanded;
7          display: [ self ];
8          format: [:morph | morph printString];
9          children: [:morph | morph submorphs];
10         when: [:morph | morph submorphs notEmpty]
```

Apart from all these specialized types of views, developers can use any graphical widget available in Pharo as a view. Consider a date object: Pharo provides a calendar widget for selecting dates (*i.e.,* `CalendarMorph`). We can reuse this widget and add a view showing a preview using this calendar to date objects. Figure 4.15 (p.65) shows the result. Lines 13–15 create this view by reusing the calendar widget.

Listing 4.2: Creating a view displaying a date object using a calendar widget.

```
Date>>#gtInspectorPreviewIn: aCanvas
    <gtInspectorPresentationOrder: 30>

    ↑ aCanvas morph
        title: 'Calendar';
        morph: [ CalendarMorph on: self ]
```

Another use case supported by the Moldable Inspector is that of reusing views between objects. For example, Figure 4.9 (p.60) showed that when selecting a key binding of a morph object, the inspector offers developers a view showing the source code associated with that key binding. This view is attached to objects of type `KMKeymap`, however, the source code associated with key binding is stored in a `BlockClosure` object; objects of this type also define a view for displaying their source code. Hence, the implementation of this view attached to key map objects, showed in Listing 4.3 (p.67), delegates the creation of the view to the corresponding `BlockClosure` object.

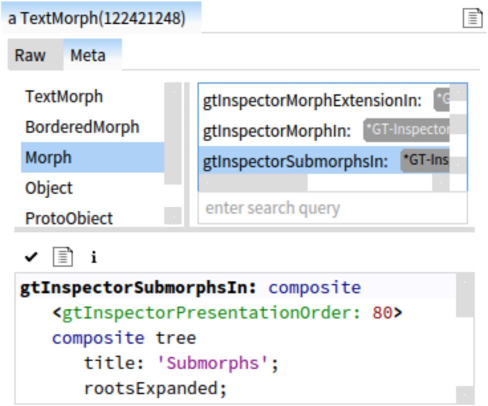
Listing 4.3: Creating a view displaying the source code attached to a key binding by reusing the source view of a block closure object.

```
KMKeymap>>#gtInspectorSourceCodeIn: aCanvas
    <gtInspectorPresentationOrder: 30>

    ↑self action gtInspectorSourceCodeIn: aCanvas
```

Attaching Multiple Views to an Object

Following the discussion from Section 3.2.3 (p.38) we make an object responsible for representing itself in multiple ways by defining methods that construct specific views within its class. This keeps view code together with that of the objects. For example, the extension defined in Listing 4.1 (p.66) is added to the class `Morph`, while the one from Listing 4.2 (p.67) is added to the class `Date`. Given that the target language for our implementation supports extension methods, this allows developers to add views to any existing object while packaging them separately. These methods are marked with a predefined parametrizable annotation (`gtInspectorPresentationOrder:` — line 2, Listing 4.1 (p.66); the annotation parameter is used to order views). The inspector follows the superclass chain when searching for annotated methods.

A side effect of this design is that a developer can use the code editor view to modify the inspector from within the inspector during inspection time. For example, Figure 4.17 (p.68) shows an editor opened on a method defining the *Submorphs* view of a `Morph`. Changing the code in the editor refreshes the inspector and provides a live extension mechanism. In fact, most extensions were created from within the inspector as doing so provides fast feedback and enables quick iterations.


a TextMorph(122421248)
Raw
Meta
TextMorph
BorderedMorph
Morph
Object
ProtoObject
gtInspectorMorphExtensionIn:
gtInspectorMorphIn: *GT-Inspector
gtInspectorSubmorphsIn: *GT-Ins
enter search query

```
gtInspectorSubmorphsIn: composite
   <gtInspectorPresentationOrder: 80>
   composite tree
      title: 'Submorphs';
      rootsExpanded;
```

Figure 4.17: Accessing an object's source code. This also gives access to the source code custom views for that object.

Supporting Tags

We defined tags using parametrized annotations: a view is added to a tag by marking the method creating that view with the tag's annotation (line 3 in Listing 4.1 (p.66) specifies that the view is added to the tag labeled *custom*). This enables a view to have multiple tags and maintains the mapping between tags and views, within the view. Another approach consists of maintain this mapping independent of the view definition (*e.g.*, in a configuration file), however, this would require developers to find and update the tag definition when adding/changing views.

Inspection Context

The Moldable Inspector models the inspection context as an object. Developers can access the context by adding a second method parameter to the method constructing the view. In line 19 in Listing 4.4 (p.69) this parameter is named `aContext`. The method signatures for creating the previous two extensions (line 1, Listing 4.1 (p.66); line 11, Listing 4.2 (p.67)) lack a second parameter, hence, those methods will not be able to access the inspection context. The context stores the previously inspected objects in the navigation (*i.e.*, the exploration session) using a linked list; they can be accessed by extensions using API calls on the context object.

As an example where accessing previously inspected objects is needed, consider the *E.g. source* view from Figure 4.14 (p.64). This view shows the source code that created the current object, and should only by available if in the previous step a developer selected an object modeling an example; this limitation exists as only example objects have the source code used to create those objects attached. To enforce this, the extension can use an activation predicate, shown in lines 26 – 28. The activation predicate checks if a previous step exists (`#hasPreviousStep`) and if the entity selected in that step is an example object. If indeed the selected object in the previous step is an example the extension displays its source code.

Listing 4.4: Creating a view showing the source code of an example object.

```
Object>>#gtInspectorExampleSourceIn: aCanvas inContext: aContext
    <gtInspectorPresentationOrder: 100>

    ↑ aCanvas smalltalkCode
        beForScripting;
        title: 'Source';
        display: [ aContext previousEntity ]
        format: [ :anExample | anExample formattedScript ];
        when: [
            aContext hasPreviousStep and: [
                aContext previousStep rawSelection isGTExample ] ]
```

The Moldable Inspector in Other Languages

The Moldable Inspector is a generic model that is not dependent on any particular object-oriented programming language or IDE. The current version was developed in Pharo. Features from Pharo, like the ability to directly ask an object for the values of its attributes or a class for its defined attributes, simplify the internal implementation of the inspector. Nevertheless, following the discussion from this section, we see no technical difficulties that would impede an implementation in other OO languages and IDEs (*e.g.*, in Eclipse or IntelliJ for Java, or VisualStudio for C#). The lack of extension methods however requires a different solution for attaching views to objects (Section 3.2.3 (p.38)).

4.6.2 A Taxonomy of Views

To investigate how the Moldable Inspector is extended by developers, we analyze 131 views present in the Pharo IDE. These views cover 84 distinct types of objects from more than 15 different applications, frameworks and libraries, including most basic data types from the language (*e.g.*, Integer, Character, Float, String, Collection, Time, Date, Calendar). These views are further grouped using 4 tags: *basic*, *custom*, *pointers* and *examples*. On average a type of object has $1.6 \pm 1.1 (M \pm SD)$ new views. However, considering that an object is displayed using the views from both its class and all its superclasses, an object has on average in all tags a total of $6.1 \pm 1.5 (M \pm SD)$ views (the *basic*, *pointers* and *examples* tags add four views to every object). If we only take into account the *custom* tag, an object has on average $2.1 \pm 1.5 (M \pm SD)$ custom views. The object with the highest number of custom views is `FileReference` (8 views).

To understand what types of views are needed for representing objects we classified all 131 views into 8 types of views based on the API for creating views (*i.e.*, list, tree,

View type	*Example*	*Number of views*
List	*Pointers* view, Figure 4.13 (p.63)	29
Tree	*Submorphs* view, second object — Figure 4.6 (p.56)	9
Table	*Keys* view, first object — Figure 4.9 (p.60)	26
Text	*SQL* view, first object — Figure 4.10 (p.61)	13
Source code	*Source code* view, second object — Figure 4.17 (p.68)	12
Morph	*Morph* view, third object — Figure 4.6 (p.56)	10
Roassal view	*View* view, Figure 4.15 (p.65)	18
Glamour view	*Raw* view, first object — Figure 4.6 (p.56)	14

Table 4.4: Types of views used to display objects in the current implementation.

table, text, source code, morph, glamour and roassal). The first five categories reflect simple textual views. The *Glamour* and *Roassal* categories contain views created using these libraries; the *Morph* category contains visual views directly created using the Morphic framework. 67.9% of the views are textual (Table 4.4 (p.70)), with the list and table views being used the most. The tree view has the lowest usage. We consider this to be the case because with Object Pager new objects can be displayed to the right, rather than discovered by expanding a tree. Visual views represent 21.3% of all views.

4.6.3 The Cost of Creating a View

Creating a view requires an average of $9.2 \pm 6.6(M \pm SD)$ lines of code. This measurement includes the signature of the method containing the view code, code comments and annotations; it excludes empty lines. Figure 4.18 (p.71) shows the size distribution of this extensions. We consider that these numbers attest to the fact that building a custom view is indeed inexpensive. Combined with the ability of creating these views live directly from within the inspector, the Moldable Inspector provides a new workflow that makes custom inspection accessible. The low cost for creating a view also addresses the second developer need identified in Section 4.2 (p.43).

In recent mailing list discussions, several developers confirmed a low learning curve for creating custom views, as long as they had examples of how to use the API[11]. Currently, we offer a browser for exploring all extensions present in the IDE, as well as tutorials on how to extend the inspector[12].

[11] http://bit.ly/1FRfDed, http://bit.ly/1R4DToY, http://bit.ly/1f1a0Fi, http://bit.ly/1dxEmxA

[12] Available at http://www.humane-assessment.com

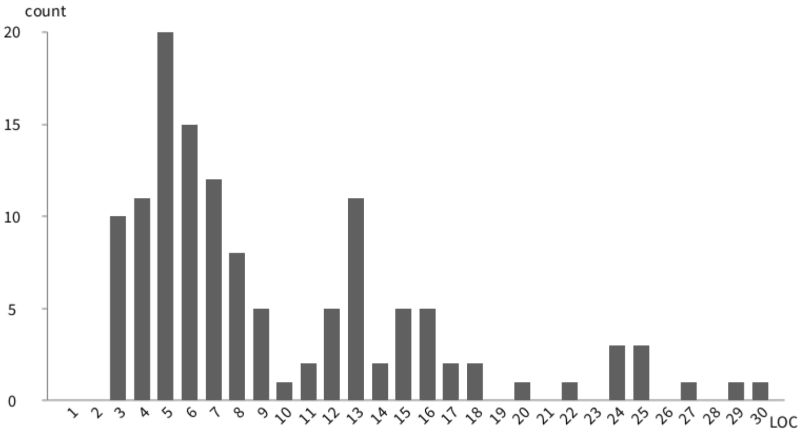


Figure 4.18: Size distribution in lines of code (LOC) for 131 custom views.

4.7 Related Work

There exists a wide body of research looking at how to improve development tools by improving navigation and the representation of various software artefacts. We further look just at approaches that focus on objects and data structures.

Self [Smith et al. 1995] allows objects to have a custom representation. However, in Self, the focus is on having a unique view for each object so that developers can easily identify objects. The Moldable Inspector promotes multiple tailored views. Smalltalk X[13] proposes an object inspector that allows objects to have multiple views and groups them using tabs. The previous object inspector from Pharo (*i.e.*, EyeInspector) also supports multiple views for an object, grouped using a drop-down menu. These approaches do not support workflows that group together multiple objects, nor do they allow developers to filter views based on their current task.

The Eclipse IDE[14] incorporates an object inspector that uses a tree view to show object state and that enables developers to customize the representation of objects through *Detail Formatters* and *Logical Structures*. Each class can have a *Detail Formatter* consisting of a snippet of code that constructs a custom string used to represent instances of that class anywhere in the debugger. Each class can further have a *Logical Structure* that can return an alternative list of key-value pairs to be displayed in the inspector instead of the current object attributes (*e.g.*, the `Map$Entry` class has a logical structure that displays the key and value from the map instead of the actual implementation). In Eclipse each class can have a single *Detail Formatter* and *Logical Structure*. There is no possibility to have multiple *Detail Formatters* or *Logical Structures*

[13] http://www.exept.de/en/products/smalltalk-x.html

[14] http://eclipse.org/ide

and dynamically select one based on a given property of an object. The Moldable Inspector allows each object to have multiple views.

NetBeans[15] offers the possibility to define multiple custom views for an object using *Variable Formatters*. Nevertheless, only one variable formatter can be active at a time; developers have to manually select which one by changing their order in the settings page. IntelliJ[16] also supports multiple custom views for an object using *Data Type Renderers*. IntelliJ further allows developers to switch between renderers using a context menu. Nevertheless, neither Eclipse, NetBeans nor IntelliJ allows views to be selected automatically at run time based on properties of the inspected objects. The Moldable Inspector enables this behavior through activation predicates.

Eclipse, NetBeans and IntelliJ rely on textual representations constructed using either tree or table views. The Moldable Inspector supports graphical representations not limited to trees or tables. While Eclipse and NetBeans allow only one object to be inspected at a time through a tree view, IntelliJ makes it possible to open multiple objects in multiple inspector windows. Nevertheless, it does not provide an explicit way to manage an exploration session, nor control the number of visible objects. *Visualizers* from Visual Studio[17] remove the limitation of textual representations, allowing objects to also have graphical views. However, like IntelliJ, they do not provide an explicit way to manage an exploration session.

A different category of object inspector consists of those integrated in current web browsers for inspecting the structure of web pages, like *HTML tab* in Firebug[18] or *Elements tab* in Chrome DevTools[19]. These inspectors allow developers to navigate the structure of a page using a tree view. When an HTML element is selected in the tree view a pane is spawned to the right displaying the element using multiple views grouped together using tabs; these include views for CSS properties, graphical layout, or the DOM object of the selected element. Nevertheless, these inspectors limit the number of objects from an exploration session to two and do not provide an easy way to customize the inspector with tailored views.

jGRASP is an integrated development environment providing object viewers that automatically generates graphical views for objects based on their structure [Cross et al. 2009]. *jGRASP* displays an object using the view that best matches its structure. Unlike *jGRASP* the Moldable Inspector aims to support views that show more than just the state of an object, and thus cannot be associated with an object only based on its structure. Furthermore, the Moldable Inspector allows views to be grouped based on their intent (*i.e.*, using tags) and proposes a workflow that automatically arranges the inspected objects.

DoodleDebug [Schwarz 2011] allows objects to have two custom representations (a summary view and a detailed view). Vebugger [Rozenberg and Beschastnikh

[15] http://netbeans.org

[16] http://jetbrains.com/idea

[17] http://visualstudio.com

[18] http://getfirebug.com

[19] http://developer.chrome.com/devtools

2014] allows developers to define templates that can create views for objects having a certain type. Nevertheless, the template that will be used to represent an object is discovered only based on the type of the object, without taking into account the state of that object. Both these approaches also focus only on representing individual objects.

Alsallakh *et al.* [Alsallakh et al. 2012] present an extension to the Eclipse IDE that uses multiple types of views to display object representing arrays. The Moldable Inspector is applicable to any type of object, not just to arrays.

Several approaches further propose the use of graphs to visualize various relations between objects [Aftandilian et al. 2010; Savidis and Koutsopoulos 2011]. These approaches scale well; they can even display the entire content of the heap. While the Moldable Inspector supports navigation between objects we do not consider that an inspection session can involve hundred of objects that need to be displayed all at once. Hence, we proposed a user interface that displays objects using a list instead of a tree, and is applicable for navigating a significantly smaller number of objects.

Debugger Canvas [DeLine et al. 2012] brings the Code Bubbles [Bragdon et al. 2010b] idea to debugging. The approach shows related entities next to one another and allows the developer to manipulate and store them in sessions. However, this approach relies on single representations for each entity regardless of the context, and object inspection is offered through a classic tree like view. The Code Bubbles interface also requires the developer to organize the bubbles. Our user interface relies on a Miller columns design that requires small space and little spatial maintenance effort.

Korn and Appel [Korn and Appel 1998] propose a technique called *traversal-based visualization* in which the debugger traverses a data-structure and creates a visualization based on a set of patterns given by a user indicating how to display particular parts of the data structure. The Moldable Inspector uses activation predicates to automatically select views based on object state; activation predicates do not directly indicate how to display an object; they are used to decide if a view is applicable for a given object.

LIVE [Campbell et al. 2003] creates visualizations for data structures automatically from ASTs: a developer first enters a program; the program is then parsed by LIVE into an AST; the AST is then used to create an animated visualization showing the evolution of the data structure. LIVE provides live editing of the visualization in the sense that users can make changes to the visualization (*e.g.*, add a node in a data structure representing a list) which are immediately reflected back to the code that created that visualization. The current implementation of the Moldable Inspector also incorporates this idea: developers can create views directly from within the inspector; any time they save the view the inspector updates.

4.8 Conclusions

Through an empirical study we observed a need for object inspectors that focus on more than the state of single objects. We proposed the Moldable Inspector, a model for object inspectors that can adapt to both the inspected objects and the immediate developer needs. We further introduced the Object Pager, a user interface for navigating through objects having multiple views.

We designed the Moldable Inspector by applying the designed principles for moldable tools targeting object-oriented development introduced in Section 3.1 (p.33) in the following way:

DP 1. Identify common types of tool-specific adaptations: The Moldable Inspector enables domain-specific adaptations by allowing developers to customize the views through which they look at objects and the set of objects accessible from within a view. The navigation mechanism itself is controlled through a dedicated user interface (*i.e.*, Object Pager).

DP 2. Simplify the creation of common adaptations: Developers customize the inspector by attaching to objects methods returning custom views. To support inexpensive creation of views, we applied the Moldable Inspector to more than 84 distinct types of objects and identified a set of common view types, detailed in Table 4.4 (p.70). We model these views as objects and we provide an internal DSL developers can use to directly configure the view in a few lines of code.

DP 3. Do not limit the types of possible adaptations only to common ones: The internal DSL for creating views allows developers to also use any graphical widget from the environment as a view. For example, we showed how to reuse a calendar widget to create a custom view for a `Date` object.

DP 4. Attach activation predicates to adaptations: The Moldable Inspector maintains an inspection context consisting of previously inspected objects and explicit data (*i.e.*, tags). Developers can then attach to each view an activation predicate that can access the inspection context and decide whether the view is applicable in that context. The activation context is set using the provided DSL.

DP 5. Update adaptations based on a development context: The Moldable Inspector updates the set of visible views when developers add or remove tags from the context. Currently, the inspector does not refresh the set of views for previous objects when a new objects is added to the exploration session. We made this decision to avoid situations in which the exploration path between objects is lost because a view from the path is removed.

While simple, the Moldable Inspector enables a wide range of different workflows and shows that an object inspector can be more than a simple tool for looking at the state of single objects. An object inspector can instead be a central tool during debugging that gives developers immediate access to contextual information. We showed that the Moldable Inspector can be used to understand various scenarios such as manipulating graphical objects, understanding compiled code, following pointers, exploring databases, navigating the file system, or browsing examples.

Some of these features are usually addressed within IDEs using dedicated tools, without these tools being connected to the actual run-time objects. Developers have to fragment their debugging activities, look for these tools elsewhere and then bring the desired information back to the inspector and debugger. The Moldable Inspector removes this gap. By adapting the displayed views to the current development needs it immediately provides the desired data right in the inspector.

Our solution relies on developers constructing custom views. To be practical, the cost associated with creating these views should be small. Through our concrete implementation we showed that this is indeed achievable.

The key to performance is elegance, not battalions of special cases.

Jon Bentley and Doug McIlroy

5 Moldable Spotter

Searching is a pervasive activity employed by developers during program comprehension to deal with the vast amount of data from today's software systems. These data often consist of many different kinds of domain-specific and interrelated software entities. Nevertheless, most integrated development environments support searching through generic and disconnected search tools. This impedes search tasks over domain-specific entities, as considerable effort is wasted by developers locating and linking data and concepts relevant to their application domains. To address this problem, in this chapter we explore how a moldable tool can enable developers to directly search through domain concepts.

5.1 Introduction

Program comprehension requires developers to reason about many kinds of interconnected software entities (*e.g.*, code, annotations, packages/namespaces, documentation, configuration files, resource files, bugs, change sets, run-time data structures) [Sillito et al. 2008] often stored in different locations [Eichberg and Schäfer 2004]. Dealing with this reality prompts developers to form and maintain task contexts [Murphy et al. 2005] by continuously searching for relevant entities and navigating their dependencies [Robillard et al. 2004; Ko et al. 2006; Fritz et al. 2014]. Cognitive task analyses describe this process as a foraging loop in which developers seek, understand, and relate information [Pirolli and Card 2005].

Depending on the application domain, software entities are further assigned domain-specific concepts. For example, an event-based system can use run-time objects to model events, a server can rely on XML files to model descriptors for web services, and a parser can model grammars using methods. Hence, instead of reasoning just in generic and low-level terms (*e.g., What files named* `web.xml` *contain within a* <`security-role`> *tag a* <`role-name`> *tag with the value* ``` `` ```manager`''`*?*), developers commonly formulate their queries using abstractions from their application domains (*e.g., What web applications use the security role* "manager"*?*).

Nevertheless, although searching is pervasive in software development and maintenance tasks, it is supported in IDEs mainly by means of disconnected and generic search tools. On the one hand, the lack of search tool integration forces developers to manually locate and construct domain abstractions by piecing together information from various sources (*e.g., What XML tags represent security roles? In what files are they defined?*). On the other hand, it impedes discoverability: one has to be aware of a domain abstraction to know what to look for. Nevertheless, given the size of today's systems, awareness of all domain abstractions is not feasible [Petrenko et al. 2008]. Hence, a generic and disconnected approach of integrating searching into IDEs leads to information foraging loops where significant effort is wasted recovering concepts instead of directly reasoning in terms of those concepts.

To address this problem and improve program comprehension during information foraging loops we propose that search tools directly enable developers to discover and search through domain concepts. This goal can be achieved if IDEs support developers in creating and managing custom ways to search through their domains. Towards this goal we propose Moldable Spotter, a moldable tool for enabling contextual domain-aware searching in IDEs by putting customization in the foreground and enabling developers to:

(i) easily create custom searches for domain objects;

(ii) automatically discover searches for domain objects.

Moldable Spotter targets the foraging loop described by Beck *et al.* (*i.e.*, Search and Filter, Read and Extract, Follow Relations) [Beck et al. 2015] and leverages a simple object-oriented model for expressing search tools by composing search processors. First, a search processor is a run-time object that expresses an individual search query. Second, every search processor is associated with a software entity from an application (*i.e.*, its target object). To achieve this, all software entities from an application that can have an associated search processor are modeled as run-time objects. Following the discussion from Section 3.2.1 (p.36) this covers domain objects, source code entities and external resources. A developer creates new custom search processors by configuring the following attributes of a processor object:

Provider: extracts the data on which the processor operates from the system(*e.g.*, the productions from a parser, the shapes from a visualization).

Preprocessor: alters, if needed, the user-supplied query (*e.g.*, fixes typos, removes white spaces, compiles a regular expression) or improves the query by following various heuristics (*e.g.*, based on natural language processing).

Query engine: extracts a subset of elements from the data provider based on the preprocessed query (*e.g.*, substring matching, regular expressions, similarity threshold).

Sorter (optional): can reorder the filtered results (*e.g.*, based on the frequency of their usage [Spasojević et al. 2014]).

Hence, a search processor is an object that knows how certain domain concepts related to its target object are reflected in an application and can restrict textual searches to software entities appropriate for those domain concepts. However, a search processor models just an individual query. To model a complete foraging loop, Moldable Spotter relies on search steps and exploration sessions: a *search step* captures a step in a foraging loop; an *exploration session* consists of a series of connected search steps. Each step takes as input an object and loads all processors associated with that object. When a user opens a step or enters a textual query all loaded search processors are executed in parallel. For example, opening a search step on a web server loads processors for searching through exposed services and security roles. Search results are displayed using a user interface that follows the guidelines for improving search tools proposed by Starke *et al.*: *(i)* skimming through search results, *(ii)* ranking and grouping of results, and *(iii)* exploring result sets [Starke et al. 2009].

The contributions of this chapter are as follows:

- Extracting and motivating, based on previous related works and use-cases, a set of requirements for enabling domain-aware searching within an IDE;
- Showing how the moldable tools approach can be applied to create Moldable Spotter, a model for integrating domain-aware searching within an IDE;
- Discussing the practical applicability of Moldable Spotter in providing domain-aware searching and improving foraging loops based on real-world examples;
- An analysis investigating the cost of creating custom extensions for Moldable Spotter based on a pilot user study and a survey looking into how developers perceive and use Moldable Spotter in practice.

Structure of the Chapter

In Section 5.2 (p.80) we motivate requirements for enabling domain-aware searching in an IDE and explore how they are addressed in related works. In Section 5.3 (p.85) we introduce the Moldable Spotter model, and in Section 5.4 (p.91) we show how it improves information foraging loops. We discuss implementation aspects in Section 5.5 (p.96). We analyze in detail the cost of creating custom extensions in Section 5.6 (p.98) and we look at how developers use Moldable Spotter in practice in Section 5.7 (p.106). In Section 5.8 (p.108) we conclude this chapter and dissect how Moldable Spotter instantiates the moldable tools approach.

5.2 Requirements

To illustrate how generic approaches lead to wasted effort during information foraging loops, we start with two motivating examples. We then propose and motivate a set of requirements for addressing this problem and discuss how they are currently supported in related approaches.

5.2.1 Motivating Scenarios

Searching Through a Parser Grammar

PetitParser is a framework for creating parsers that makes it easy to dynamically reuse, compose, transform and extend grammars [Renggli et al. 2010a]. Developers create parsers by specifying a set of grammar productions in a class or in a class hierarchy. 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. Productions are referred to in other productions by accessing object attributes. Developers can add other helper methods and attributes to a parser class.

Finding a grammar production is a common task during the development of a parser. As grammar productions have associated methods, one way to find a production within a parser class is to use a generic search for methods. Nevertheless, this approach will find methods defined in that class that are not productions; developers need to further check that an attribute with that exact name also exists. If the parser is organized in a class hierarchy, developers are required to browse through the superclass chain when a production is not found in the current class. Another task that often arises when working with grammars is to locate those productions using a given production. A developer can use a generic approach and start browsing all methods that access an attribute, however, as in the previous situation, she will have to check if the accessing methods are indeed grammar productions.

These issues can be directly addressed with the help of two domain-specific searches that allow developers to dis- cover and search through productions in a parser and through productions using a given production. For example, a search through the productions of a PetitParser parser can be instantiated as follows using a search processor:

> *Provider*: extracts from a parser class those methods that are grammar productions (*i.e.*, methods where there exists an attribute with the same name as the method);
>
> *Preprocessor*: parses and compiles the query supplied by the user into a regular expression;

Query engine: extracts those productions whose name matches the regular expression;

Sorter: orders productions based on the frequency of their usage within the current grammar.

The results of using this processor to search for productions containing the string *"hex"* in a parser for Java code is displayed in Figure 5.1a (p.85). The presented scenario is not unique to PetitParser. Similar situations arise every time code elements (*e.g.*, methods, classes, annotations) have a domain specific semantic, since generic search tools cannot filter out unrelated entities.

Searching Through a Visualization

Roassal is an engine for building visualizations defined in terms of objects and their relations [Araya et al. 2013]. Developers create a new visualization starting from a set of domain objects by *(i)* mapping different types of shapes and relations to those objects, *(ii)* choosing a layout algorithm, and *(iii)* specifying how properties of shapes and of the layout are computed from the domain objects.

When reasoning about how a model object is rendered, a common task consists in locating the parts of a visualization responsible for rendering that model object. A visualization in Roassal is a run-time object consisting of a composite (*i.e.*, tree) of shape objects. Answering the previous question requires a developer to search through the composite and locate shapes that render that object. As a visualization is an object, one way to address this question is to navigate through object state using a generic object inspector. Nevertheless, Roassal visualizations are complex objects containing many other attributes, unrelated to the task at hand, leading to a significant effort just for navigating through the object graph.

Providing a domain-specific search that enables developers to directly determine what shapes render a domain object can reduce navigation overhead. This search can be instantiated using a search processor as follows:

Provider: extracts, from a Roassal visualization, all graphical objects that render the target object associated with the processor;

Preprocessor: parses and compiles the query supplied by the user into a regular expression;

Query engine: extracts those graphical objects whose class name matches the regular expression;

Sorter: orders graphical objects alphabetically based on their class name.

Figure 5.5b (p.95) shows an example for this search processor. Such types of searches are not limited to Roassal; they are common when domain objects of interest for a developer are spread across an object graph.

5.2.2 Requirements Discussion

The scenarios presented in Section 5.2.1 (p.80) cover different types of developer questions that can be efficiently addressed through custom domain-specific searches. For this approach to be possible, IDEs should enable developers to create and work with domain-specific searches. Starting from the presented scenarios we propose the following as a set of minimum requirements towards this goal: *inexpensive creation of search processors*, *support for multiple data sources* and *context-aware searches*.

Inexpensive Creation of Search Processors (REQ1)

Given the wide rage of development tasks and applications, foreseeing all usage contexts of a tool is not possible [Sillito et al. 2008]. A fixed set of searches limits the applicability of a search tool. Enabling developers to create custom searches for their domain entities addresses this problem. Nevertheless, the difficulty of creating a custom search directly influences the usability of such an approach. On the one hand, a domain-specific language for creating custom searches can significantly reduce the cost for certain types of extensions. On the other hand, supporting custom searches through a general-purpose programming language allows for any type of extension. To provide a quick entry point and not limit the types of possible extensions, an infrastructure for domain-specific searching should *support inexpensive creation of common types of searches, while allowing developers to fall back to a general-purpose programming language when advanced extensions are needed.*

Multiple Data-sources Support (REQ2)

The two questions discussed in the previous section require information from two different data sources: source code and runtime. External data (*e.g.*, files) is another form of data source also frequently encountered in developer questions. Given the wide range of heterogenous data used in software applications, enabling successful domain-specific searching requires *to integrate and present data from multiple sources to developers.*

Context-aware Searches (REQ3)

The questions discussed in Section 5.2.1 (p.80) are sometimes addressed in IDEs through standalone search tools (*e.g.*, tools for query-based debugging, dedicated tools for working with parsers). To take advantage of them, developers have to be aware of their presence and know when they are applicable. An infrastructure encouraging developers to create and work with custom searches should support this by *enabling developers to automatically find custom searches applicable for a domain entity, based on the entity, the task at hand, and the developer's task context.*

Tool	Data model	Extension language	Requirement		
			RQ1	*RQ2*	*RQ3*
JQuery	logic database	TyRuBa	✓*	—	✓*
SEXTANT	XML database	XQuery	✓*	—	✓*
Ferret	sphere model	algebra	?	✓	✓*
Sando	text-based files	internal DSL	✓	—	—
SPOOL	OO model	internal DSL	✓	—	—
Moldable Spotter	reuses the IDE model	internal DSL	✓	✓	✓

— no support, ✓ full support, ✓* partial support, ? unknown

Table 5.1: Feature comparison for tools enabling custom extensions for searching.

5.2.3 Current Approaches

There exists a wide range of software tools focusing on improving program comprehension by combining and integrating multiple search tools and techniques. In this section we present and discuss several tools that support custom extensions for searching through a software system:

a) JQuery is a code browsing tool that combines the advantages of query-based and hierarchical browser tools [Janzen and de Volder 2003]. JQuery relies on a knowledge database generated dynamically using the Eclipse API and queried using TyRuBa, a logical programming language augmented with a library of helper predicates for searching through source code.

b) SEXTANT is a software exploration tool that leverages a custom graph-based model [Schafer et al. 2006]. In SEXTANT all sources of a project are transformed to XML, stored in a database and queried using XQuery.

c) Ferret is a tool for answering conceptual queries that integrates different sources of information, referred to as spheres, into a queryable knowledge-base [de Alwis and Murphy 2008]. Each sphere performs its queries using a component/plugin of the Eclipse IDE (*e.g.*, static Java searches are executed using JDT).

d) Sando is code search tool and framework that embodies a general and extensible local code search model [Shepherd et al. 2012]. Sando focuses on enabling researches to easily implement and compare approaches for local code search.

e) SPOOL is a reverse engineering environment combining searching and browsing. SPOOL has at its core a repository that stores source code models and provides a query mechanism through which a user can query the model [Robitaille et al. 2000].

Customization

JQuery and SEXTANT address customization by selecting a language that best fits the model they use to represent the queried data. Nevertheless, this requires developers to write queries for object-oriented programs using a language based on a different paradigm (*i.e.*, logical programming, functional programming). The query language used in JQuery, while allowing for expressive queries, makes the creation of complex queries difficult even for advanced users. Sando and SPOOL use a different approach: developers create custom extensions by implementing a predefined interface and writing code in the host language (*i.e.*, Java, C#) leveraging a given API. This does not require developers to become familiar with another language. Sando shows that with this approach complex searches can be implemented in less than 100 lines of code. We were unable to find a discussion regarding the ease of creating conceptual queries for Ferret.

Data-sources

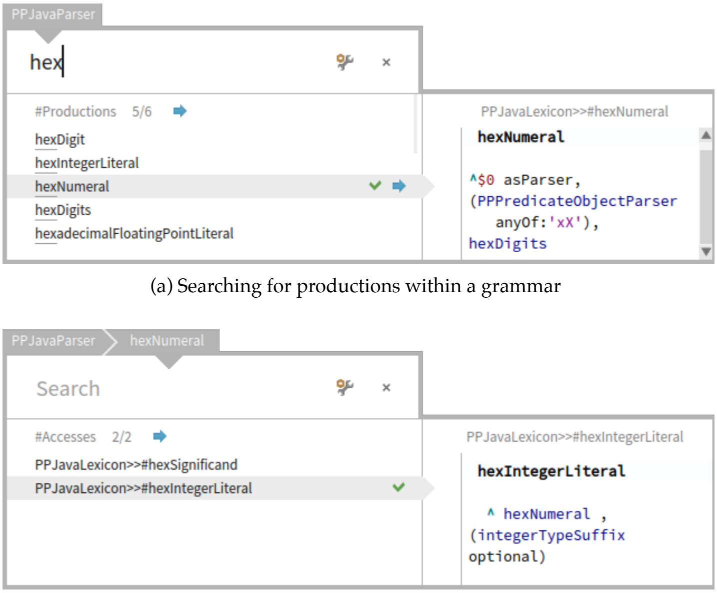
Ferret is the only one of the five selected tools that fully addresses *REQ2* by taking source code, dynamic and historical data into account; SEXTANT, JQuery, Sando and SPOOL are targeted towards the analysis of source code artifacts and do not take the run-time into account. Apart from the discussed tools, many other approaches from the area of feature location combine multiple types of information to improve their results but do not focus on customization [Dit et al. 2012]. Search tools from current IDEs also allow developers to search through multiple data. For example, *Global Search* in IntelliJ makes it possible for developers to search through files, methods, preferences, tools, menus, *etc.*

Context-aware Searches

SPOOL only allows custom searches to be selected based on the type of an entity (*e.g.*, method, file). SEXTANT, JQuery and Ferret go one step further and attach searches to software entities; applicable searches can than be dynamically selected based on various properties of those entities. They however do not attempt to model the development context and select searches also by taking into that context. A different approach is taken by recommender systems which aim to suggest to developers useful tools by recording and mining usage histories of software tools [Murphy-Hill et al. 2012]; this however requires usage history information.

Summary

The requirements identified in Section 5.2.2 (p.82) are addressed to various degrees in current approaches that focus on integrating searches over multiple data types.



hex

#Productions 5/6

hexDigit
hexIntegerLiteral
hexNumeral
hexDigits
hexadecimalFloatingPointLiteral

PPJavaLexicon>>#hexNumeral

```
hexNumeral

^$0 asParser,
(PPPredicateObjectParser
   anyOf:'xX'),
hexDigits
```

(a) Searching for productions within a grammar

#Accesses 2/2

PPJavaLexicon>>#hexSignificand
PPJavaLexicon>>#hexIntegerLiteral

PPJavaLexicon>>#hexIntegerLiteral

```
hexIntegerLiteral

  ^ hexNumeral ,
(integerTypeSuffix
optional)
```

(b) Searching through the users of a production.

Figure 5.1: Exploring a PetitParser grammar for Java code.

This indicates a need for a search framework that focuses on unifying search support within an IDE by addressing all three requirements.

5.3 Moldable Spotter in a Nutshell

In this section we show how Moldable Spotter addresses the foraging loop described by Beck *et al.*, we introduce a user interface to support the presented model, and we discuss how Moldable Spotter models a search context.

5.3.1 Search Context

As discussed in Section 5.2.2 (p.82), developers need support for automatically selecting searches relevant for their current needs. In Chapter 4, Section 4.3.3 (p.51), in the context of domain-specific object inspection, we introduced a solution for modeling a development context based on *tags* and *exploration sessions*; extensions were filtered using *activation predicates*. We propose reusing the same approach for enabling Moldable Spotter to automatically select relevant search processors. For completeness, we briefly present these operators and show how they apply to Moldable Spotter.

Tags identify and group together processors applicable for a development task or application domain. For example, the processors related to PetitParser have the *parsing* tag. Generic processors are grouped using the *default* tag. Only processors that have a tag currently present in the search context are made available to developers. An *exploration session* stores the objects found by a developer using Moldable Spotter, together with the order in which they were searched for and the search processors used to find those objects. Each search processor can then have an *activation predicate* applied by Moldable Spotter on the current search context before loading that processor. For example, the search processor for searching through production in a parser class described in Section 5.2.1 (p.80), should only be available if the developer opened Moldable Spotter on a class that represents a PetitParser parser; this processor needs an activation predicate that checks the type of the class.

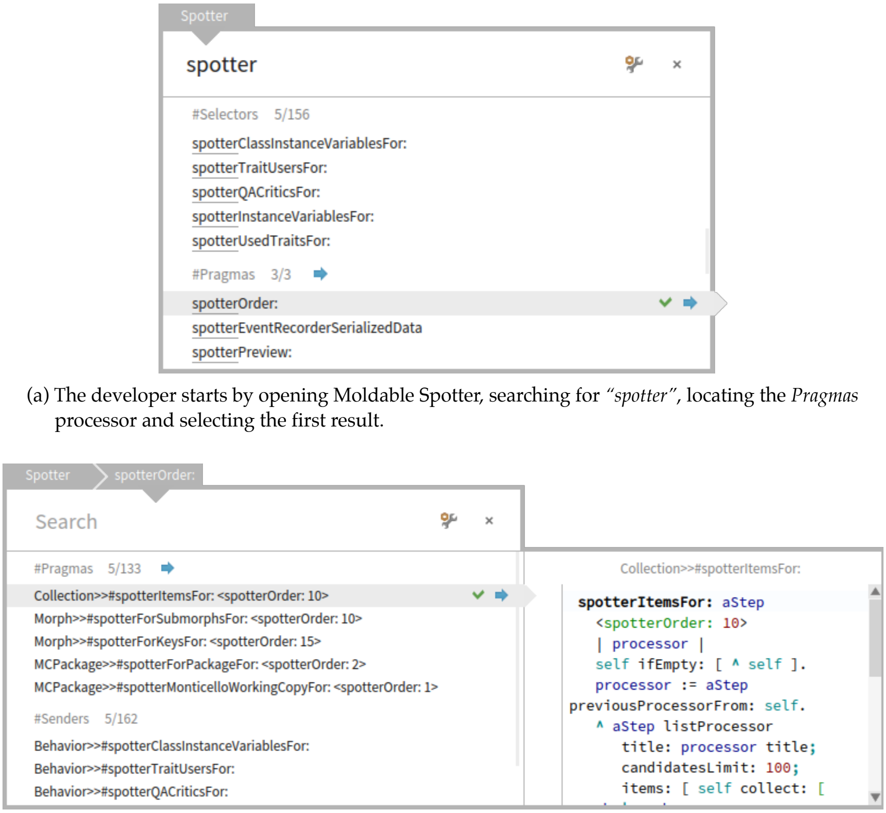
5.3.2 Supporting an Information Foraging Loop

Starting from the foraging loop for intelligence analysis proposed by Pirolli and Card [Pirolli and Card 2005], Beck *et al.* describe a foraging loop for feature location [Beck et al. 2015] having three main activities: *search and filter*, *read and extract* and *follow relations*. We show next how Moldable Spotter supports these activities.

Search and Filter

To support this activity, Moldable Spotter uses *search steps*. A step encapsulates a search on an object in a given context. Each step takes a target object as input and loads all processors that apply to objects of that type in the current context. For example, a search step opened on a method (*i.e.*, on a method object) can load processors for searching through both the callers and the callees of that method. A step opened on a class representing a PetitParser parser in a context that contains the *parsing* tag loads specific processors related to PetitParser (*e.g.*, a search through grammar productions — Figure 5.1a (p.85)). If the *default* tag is also in the current context, then generic processors related to classes are loaded (*e.g.*, searches through subclasses, superclasses or instances when the runtime is available).

When a developer opens a step on an object all data providers from the loaded processors are executed and their results are presented to the user according to the order defined by the sorter. For example, when a developer enters the query *"hex"* in Figure 5.1a (p.85), all loaded processors receive the query and update their results. This is the search phase of the foraging loop; developers do not have to select a priori what processors they want to execute. To implement the filter phase, each time the developer provides a textual query, the query engines are used to filter the presented data in each processor.



#Selectors 5/156
spotterClassInstanceVariablesFor:
spotterTraitUsersFor:
spotterQACriticsFor:
spotterInstanceVariablesFor:
spotterUsedTraitsFor:
#Pragmas 3/3
spotterOrder:
spotterEventRecorderSerializedData
spotterPreview:

(a) The developer starts by opening Moldable Spotter, searching for *"spotter"*, locating the *Pragmas* processor and selecting the first result.

#Pragmas 5/133
Collection>>#spotterItemsFor: <spotterOrder: 10>
Morph>>#spotterForSubmorphsFor: <spotterOrder: 10>
Morph>>#spotterForKeysFor: <spotterOrder: 15>
MCPackage>>#spotterForPackageFor: <spotterOrder: 2>
MCPackage>>#spotterMonticelloWorkingCopyFor: <spotterOrder: 1>
#Senders 5/162
Behavior>>#spotterClassInstanceVariablesFor:
Behavior>>#spotterTraitUsersFor:
Behavior>>#spotterQACriticsFor:

Collection>>#spotterItemsFor:

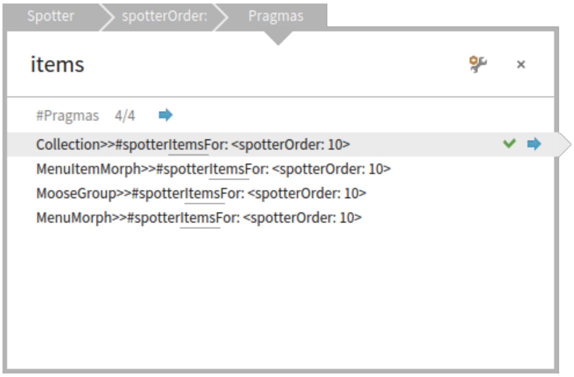
```
spotterItemsFor: aStep
  <spotterOrder: 10>
  | processor |
  self ifEmpty: [ ^ self ].
  processor := aStep
previousProcessorFrom: self.
  ^ aStep listProcessor
     title: processor title;
     candidatesLimit: 100;
     items: [ self collect: [
```

(b) She then dives into the selected annotation (`spotterOrder:`) by clicking on the arrow to the right of that annotation. This creates a new step allowing her to search through methods having that annotation and methods calling a method with that name.

Figure 5.2: Locating a method having the `spotterOrder:` annotation (in Pharo annotations are referred to as *pragmas*)

Read and Extract

Developers can extract initial information by reading a short textual description of each element present in the results list. As this only gives a limited amount of information, Moldable Spotter supports a contextual preview for each type of result (*e.g.*, source code for a method, graphical representation for a graphical object or a *png* file). Hence, developers can choose to view each element using the preview. For example when searching for a production in a parser (Figure 5.1a (p.85)) the preview of each shape shows the source code of that production. If the preview is not enough, developer can open a result in another tool from the IDE; these tools are selected based on the type of the result entity: code editor for methods/classes, object inspector for objects, *etc.*

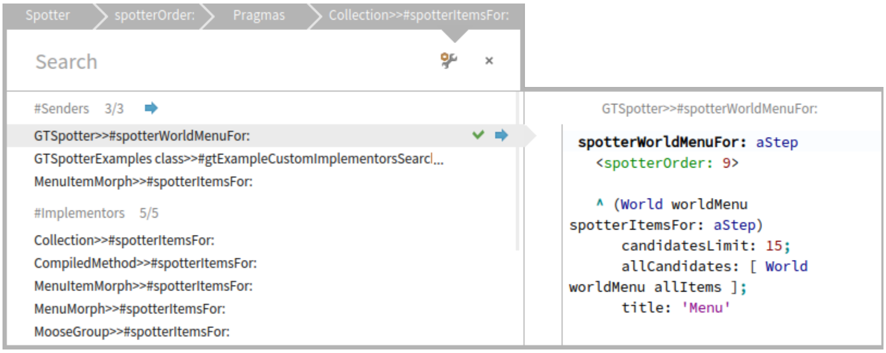


items

#Pragmas 4/4

Collection>>#spotterItemsFor: <spotterOrder: 10>
MenuItemMorph>>#spotterItemsFor: <spotterOrder: 10>
MooseGroup>>#spotterItemsFor: <spotterOrder: 10>
MenuMorph>>#spotterItemsFor: <spotterOrder: 10>

(a) Continuing the navigation from Figure 5.2 (p.87) the developer expands the *Pragmas* processor and filters results using the *"items"* query.



#Senders 3/3

GTSpotter>>#spotterWorldMenuFor:
GTSpotterExamples class>>#gtExampleCustomImplementorsSearc|...
MenuItemMorph>>#spotterItemsFor:

#Implementors 5/5

Collection>>#spotterItemsFor:
CompiledMethod>>#spotterItemsFor:
MenuItemMorph>>#spotterItemsFor:
MenuMorph>>#spotterItemsFor:
MooseGroup>>#spotterItemsFor:

GTSpotter>>#spotterWorldMenuFor:

```
spotterWorldMenuFor: aStep
  <spotterOrder: 9>

  ^ (World worldMenu
spotterItemsFor: aStep)
      candidatesLimit: 15;
      allCandidates: [ World
worldMenu allItems ];
      title: 'Menu'
```

(b) She finishes by opening a method in a new step which loads processors for searching through senders of that method, as well as other methods with that name (*Implementors*).

Figure 5.3: Exploring callers of methods having the `spotterOrder:` annotation.

Follow Relations

From a search step a developer can choose to continue navigation by selecting a search result. This adds the selected element to the search session and opens a new search step on the selected result. The creation of a new search step is exemplified in Figure 5.1 (p.85): initially the developer searched for productions containing the text *"hex"* and selected the production *hexNumeral* (Figure 5.1a (p.85)). Then the developer dived into the *hexNumeral* production, creating a new search step (Figure 5.1b (p.85)); in the new search step she can search for productions using the *hexNumeral* production. The search session preserves the navigation history through multiple search steps, and allows developers to reason about how they got to the current step, as well as to go back to previous steps and try alternative searches. Furthermore, processors in the current step can also be filtered based on objects from the previous search steps, not only based on the current object.

5.3.3 A User Interface for the Moldable Spotter Model

While the Moldable Spotter model can express a wide range of searches, the user interface (UI) plays a crucial role in the usability of a search tool. Hence, in this section we introduce and discuss a UI design for Moldable Spotter based on the three guidelines proposed by Starke *et al.*: *(i)* group and rank results, *(ii)* support rapid skimming through result sets, and *(iii)* enable in-depth exploration of result sets [Starke et al. 2009]. To illustrate the UI we use a running example in which a developer is interested in exploring callers of methods having a certain annotation (Figures 5.2 and 5.3).

Grouping and Ranking Results

Running all search processors loaded in a step is at the core of the Moldable Spotter model. This raises the need of displaying multiple heterogenous result sets at once. We address this by displaying result sets using one level trees: the roots of the trees are the processors that return results and the children are the actual results; results are presented in the order retuned by the processor (Figure 5.2a (p.87)).

Skimming Through Result Sets

Given that processors can produce a large number of results, for each processor we display the first *n* results; *n* is customizable per processor (Figure 5.2a (p.87)) and root nodes are expanded automatically. The label of each result set includes the number of displayed results and the total number of results from that result set. To help developers reason about why a result is displayed, the querying engine can embed visual cues when displaying results (*e.g.*, highlight the text matching the query).

Exploring Result Sets

Once a developer has found one or more interesting entities she can investigate them in more detail. Moldable Spotter supports this through two mechanisms:

Preview pane Developers can obtain more information about a result by opening the contextual preview associated with the current selected result in a pane to the right of the search pane (Figure 5.2b (p.87) and Figure 5.3b (p.88)).

Navigation Navigation is supported using a simple visual language that consists of two main commands:

Dive in: adds the selected object to the search session and creates a new step having that object as target entity. A developer can invoke this action after selecting an element using a keyboard shortcut or the arrow to the right of the selected element (➡). For example, after selecting the annotation `spotterOrder:` in Figure 5.2a (p.87), the developer executes the dive in action by pressing the arrow; this creates a new search step (Figure 5.2b (p.87)). The new search step has as target entity the selected annotation and allows the developer to search through methods having that annotation.

Expand: creates a new search step containing a single processor for searching only through the selected result set. This processors shows the entire result set. A developer can invoke this action after selecting a result from a result set using a keyboard shortcut or the arrow next to the label of a result set. For example, after diving into the `spotterOrder:` annotation (Figure 5.2b (p.87)), the developer decides she is only interested in what methods have that annotation. Hence, she expands the result set returned by the *Pragmas* processor (Figure 5.2b (p.87)). In the next search step (Figure 5.3a (p.88)) she can see all methods having that annotation, explore them in more details and refine her search.

Figure 5.2b (p.87) illustrates the final step of the exploration. The developer created this step by diving in a method having the `spotterOrder:` annotation. In this step she can search through data related to that method, like callers or methods with the same name from other classes. To maintain orientation a breadcrumb shows previous steps.

UIs for Feature Location and Exploration Tools

For completeness, we present here a comparison with UIs used in other feature location and exploration tools.

Apatite, a tool for searching and navigating through five levels of an API's hierarchy (packages, classes, methods, actions (methods containing verbs) and properties (getters and setters)) [Eisenberg et al. 2010] relies on a similar interface for displaying the results of a search step: the first 5 results from each category are automatically presented and developers can see more results on demand. The same UI is also used by *Global Search* from IntelliJ. Nevertheless, Apatite relies on Miller columns to display a navigation session which takes considerably more screen-real estate than the UI used by Moldable Spotter; *Global Search* does not provide any navigation mechanism.

Ferret relies on a tree view where users need to manually expand each node to see the actual result. JQuery and SEXTANT allow users to discover available searches through a context menu and display the result in a tree or graph.

UIs for feature location focus on allowing users to reason about why a result is displayed. For example, *I3*, a novel user interface for feature location providing searches through methods, highlights query terms in the code editor and uses visualizations to convey the similarity of a result with the query as well as show co-change patterns [Beck et al. 2015]. The Moldable Spotter UI highlights the text matching the query and uses a contextual preview to convey more information about a result.

Global

1) What {method, class, annotation, package, project} names match a regular expression?
2) What methods {call a method, access an attribute} whose name match a regular expression?
3) What {files, folders} are in the {current working directory, a selected file/folder}?

Packages, classes and methods

4) What packages have changes that need to be committed?
5) What are this package's {classes, extension methods, tags}?
6) What are this class' {methods, attributes, static method, static attributes, superclass methods, subclass methods}?
7) What are this class' siblings, super-classes, sub-classes?
8) What classes implement a method with the same name?
9) What attributes/fields does this method access?
10) What methods does this method call?
11) What methods call this method?
12) What methods reference this class by name?
13) What methods have this annotation?

History

14) What are the previous versions of this method, package?
15) Who edited this method before?

Run-time

16) What are the fields of this object?
17) What are the elements of this collection object?
18) What are the keys of this dictionary object?

Table 5.2: Generic searches supported by Moldable Spotter.

5.4 Improving Information Foraging Loops

Moldable Spotter aims at enabling direct searches through domain-specific concepts during information foraging loops. In this section we show that Moldable Spotter addresses this aspect by applying it to the motivating examples discussed in Section 5.2.1 (p.80) and we highlight how the presented model supports this goal. Generic code related searches are still an integral part of an IDE (*e.g., What methods call this method?*, *What are the attributes of this class?*). Not taking them into account would require developers to decide between Moldable Spotter and a generic tool when they need to perform a search. To avoid this we also extended Moldable Spotter with support for generic searches (Table 5.2 (p.91)) based on previous literature, common searches from IDEs and our own developer experience. An example was presented in Figures 5.2 and 5.3 where to determine the senders of a method having a certain annotation the developer *(i)* started by searching for the desired annotation, *(ii)* continued by searching through methods having that annotation, and *(iii)* finished by exploring the senders of one of those methods. This enables foraging loops that combine generic with domain specific searches (Section 5.4.4 (p.94)).

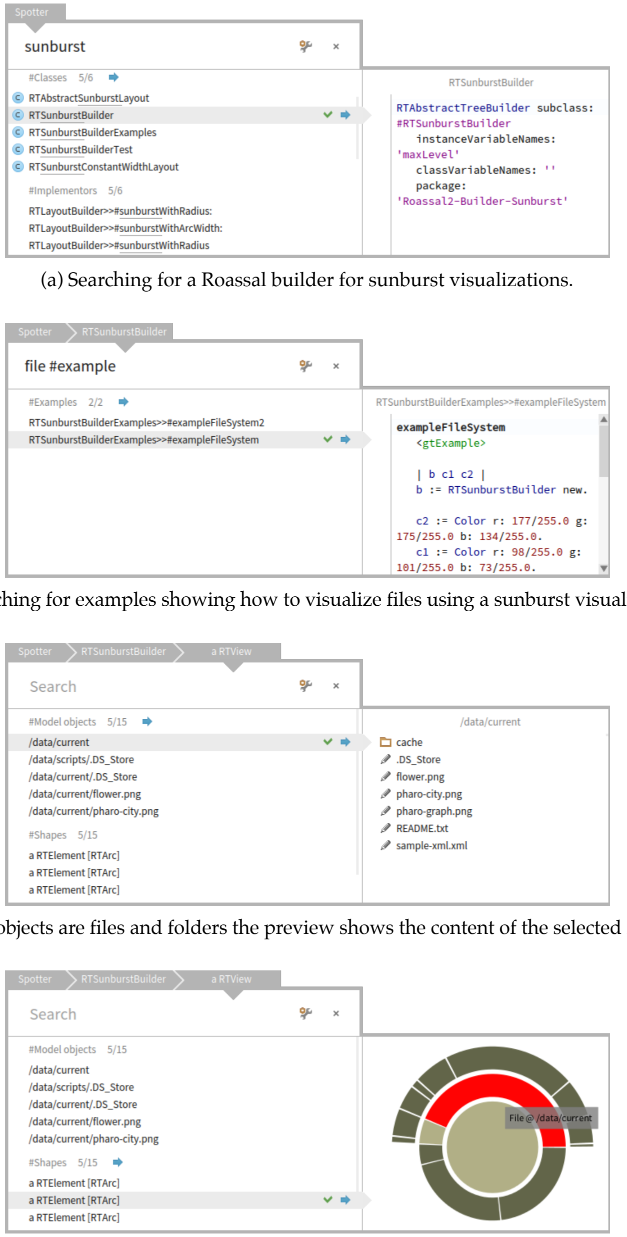
5.4.1 Starting Moldable Spotter

To perform a search the developer has to first open Moldable Spotter on a target object. When an explicit target object is missing, Moldable Spotter selects the current workspace as target object. This loads global processors for searching through software entities within the current workspace, including processors for searching through classes, methods, pragmas, history, files, folders, *etc.* Moldable Spotter registers shortcuts and menu options for performing this action. For example, in Figure 5.2a (p.87) the developer opened Moldable Spotter this way and entered the query text *'spotter'*. To allow developers to specify an explicit target object, Moldable Spotter further registers shortcuts and menus with other tools from the IDE. For example, a developer can open Moldable Spotter on any object available in the debugger or the object inspector; this way Moldable Spotter gets access to run-time state. The code editor makes it possible to open Moldable Spotter on code entities (*e.g.*, classes, annotations).

5.4.2 Finding Productions Within a Parser

Moldable Spotter supports the two developer questions related to PetitParser discussed in Section 5.2.1 (p.80) through two processors. The first enables searches through the productions of a parser and has the structure described in Section 5.2.1 (p.80). As PetitParser parsers are specified using classes, this processor is associated with classes, and has an activation predicate that checks whether the target class is a PetitParser parser (*i.e.*, The class must have `PPCompositeParser` in the superclass chain). Figure 5.1a (p.85) shows a scenario in which a developer has opened Moldable Spotter on a PetitParser parser for Java code and searched for productions containing the word *"hex"* in their name. The preview gives direct access to the source code of the production.

Searching through the uses of a production is achieved using a processor attached to methods valid when the target method is a PetitParser production. Developers can use this processor by opening Moldable Spotter on a production or adding a production to the current search session. Figure 5.1b (p.85) shows the latter situation where, after searching for a production, a developer uses the *dive-in* feature to add the selected production to the current search session. The preview pane shows the source code of a selected production.



Spotter
sunburst
#Classes 5/6
RTAbstractSunburstLayout
RTSunburstBuilder
RTSunburstBuilderExamples
RTSunburstBuilderTest
RTSunburstConstantWidthLayout
#Implementors 5/6
RTLayoutBuilder>>#sunburstWithRadius:
RTLayoutBuilder>>#sunburstWithArcWidth:
RTLayoutBuilder>>#sunburstWithRadius
RTSunburstBuilder

```
RTAbstractTreeBuilder subclass:
#RTSunburstBuilder
   instanceVariableNames:
'maxLevel'
   classVariableNames: ''
   package:
'Roassal2-Builder-Sunburst'
```

(a) Searching for a Roassal builder for sunburst visualizations.

Spotter
RTSunburstBuilder
file #example
#Examples 2/2
RTSunburstBuilderExamples>>#exampleFileSystem2
RTSunburstBuilderExamples>>#exampleFileSystem
RTSunburstBuilderExamples>>#exampleFileSystem

```
exampleFileSystem
   <gtExample>

   | b c1 c2 |
   b := RTSunburstBuilder new.

   c2 := Color r: 177/255.0 g:
175/255.0 b: 134/255.0.
   c1 := Color r: 98/255.0 g:
101/255.0 b: 73/255.0.
```

(b) Searching for examples showing how to visualize files using a sunburst visualization.

(c) As model objects are files and folders the preview shows the content of the selected file or folder.

(d) Selecting a graphical shape shows highlights it in the preview with red.

Figure 5.4: Finding an example of how to use a sunburst visualization to display files and folders.

5.4.3 Searching Within a Visualization

To improve the user experience when developing Roassal visualizations, we created together with the developers of the Roassal framework, two dedicated processors for searching through visualization objects (*i.e.*, objects of type *RTView*). The first processor (*Model objects*, Figure 5.5a (p.95)) enables a search through model objects used by graphical elements; the preview pane shows the state of the model object. The second (*Shapes*, Figure 5.5a (p.95)) targets the actual shape objects used in the visualization. As a concrete example, Figure 5.5 (p.95) shows Moldable Spotter opened on a visualization for comparing two metrics about population in several US states using a horizontal double bar chart.

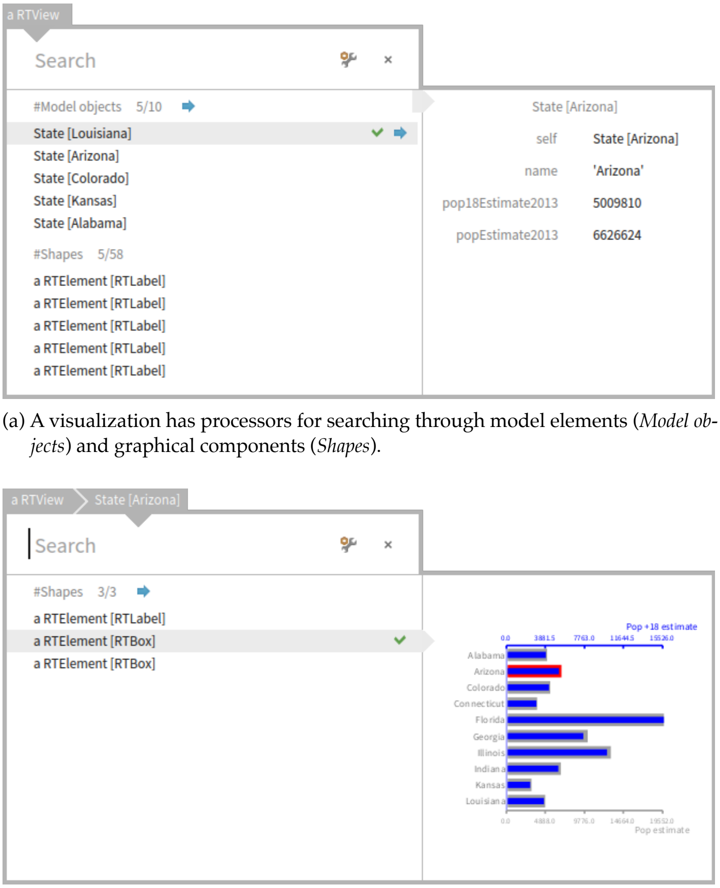
To determine what graphical shapes are used to render a model object a developer can dive in and add the model object to the search session. Doing this loads a processor showing the list of shapes used to render that object. This processor is attached to all objects and has an activation predicate that checks whether the previous object in a search session is a Roassal visualization; if so, the *provider* locates the shapes in that visualization that have the current object as a model. Hence, this processor would not be available when opening Moldable Spotter on the same object in isolation. In Figure 5.5b (p.95) a developer can see that there are three shapes that render the state Arizona: two *RTBox*es and a *RTLabel*.

5.4.4 Finding Examples

Roassal comes with a large number of examples. They are organized in classes linked with the graphical classes for which they provide examples. Hence, for each graphical class, if it has an example class, we can provide a processor that displays and searches through its examples. A developer can find and execute an example by *(i)* opening Moldable Spotter on the current workspace and searching for the desired Roassal class and *(ii)* diving into the class and locating the *Examples* processor (Figure 5.4b (p.93)). The preview of an example shows the source code of that example. To get to the actual visualization a developer can then execute the example and dive into the result. As this is a visualization object the developer can continue her investigation using the custom processors presented in Section 5.4.3 (p.94) (Figure 5.4c (p.93)).

5.4.5 Summary

Apart from the presented examples we also added Moldable Spotter support to other applications from the Pharo ecosystem: GUI libraries (Glamour, Spec, Morphic), Opal compiler, MongoDB bindings for Pharo, Metacello package management system, Monticello versioning system, *etc.* We further added IDE related searches for preferences, menus, help topics, shortcuts, clipboard history, and files and folders.



a RTView

Search

#Model objects 5/10

State [Louisiana] ✓

State [Arizona]

State [Colorado]

State [Kansas]

State [Alabama]

#Shapes 5/58

a RTElement [RTLabel]

a RTElement [RTLabel]

a RTElement [RTLabel]

a RTElement [RTLabel]

a RTElement [RTLabel]

State [Arizona]

self	State [Arizona]
name	'Arizona'
pop18Estimate2013	5009810
popEstimate2013	6626624

(a) A visualization has processors for searching through model elements (*Model objects*) and graphical components (*Shapes*).

a RTView

State [Arizona]

Search

#Shapes 3/3

a RTElement [RTLabel]

a RTElement [RTBox] ✓

a RTElement [RTBox]

(b) Diving into a model object from a step opened on a visualization shows what shapes use that object.

Figure 5.5: Exploring an object modeling a Roassal visualization.

As we aim to support custom searches through domain concepts, providing an exhaustive list of searches is not feasible. Instead, Moldable Spotter enables domain-specific adaptations. This section showed that the proposed model can support searches through relevant concepts from different domains that spawn over source code and run-time objects.

By providing both domain-specific and generic searches Moldable Spotter offers a unified interface for embedding search support within an IDE: developers open Moldable Spotter on a selected entity and can immediately discover available searches. By using the search session they can further perform targeted explorations that build on previous search results.

5.5 Implementation

To determine the extent to which Moldable Spotter can enable domain-aware searching we have implemented it in Pharo, as part of the GToolkit project. We developed the first version of Moldable Spotter over three months and integrated it in the alpha version of Pharo 4, replacing *Spotlight*, a previous search tool for finding methods and classes. We then continued to develop Moldable Spotter based on developer feedback (Section 5.7 (p.106)) in one-month iterations.

5.5.1 Constructing Custom Search Processors

Moldable Spotter supports the creation of custom search processors through an internal DSL (*i.e.*, API). Developers can configure the predefined blocks of a processor object (Section 5.2.1 (p.80)) using anonymous functions or provide a custom implementation for the processor. When creating processors we started by configuring the predefined blocks and switched to a custom stream-based implementation when performance became an issue. For example, lines 32–40 in Listing 5.1 (p.96) show the code snippet for creating a search processor for PetitParser productions. An activation predicate ensures that this processor is only available to parser classes (*i.e.*, classes that inherit from `PPCompositeParser` — lines 39–40).

Listing 5.1: Search processor for PetitParser productions.

```
29 spotterForProductionsFor: aStep
30     <spotterOrder: 10>
31     <spotterTag: #PetitParser>

32     aStep listProcessor
33         title: 'Productions';
34         allCandidates: [ self productionMethods ];
35         candidatesLimit: 5;
36         itemName: [:aProduction | aProduction selector];
37         filter: GTFilterRegex;
38         itemFilterName: [:aProduction| aProduction selector];
39         when: [ :aClass |
40                 aClass inheritsFrom: PPCompositeParser ];
```

Line 34 configures the provider; this component of the search processor is expressed using an anonymous function. Lines 37 and 38 configure the query engine (*e.g.*, filter productions based on their names using regular expressions). The query engine is responsible for filtering the elements returned by the provider based on the user supplied query. Query engines are modeled as classes that extend a predefined class (*i.e.*, `GTFilter`). Currently Moldable Spotter provides filters for matching elements with a user query based on: substring matching, regular expressions, and approximate matching using thresholds; currently these filters do not support indexing of search results. The filter includes the preprocessor.

When the numbers of items that need to be filtered is large, separating the provider from the query engine can cause performance problems. To account for this, Moldable Spotter also supports searches based on streams that combine the provider and the query engine. For example, lines 43–51 from Listing 5.2 (p.97) show the processor for searching through files, which uses the API method *filter:item:* that provides stream-based searching for a filter. Currently 13 processors use this optimization.

Listing 5.2: Search processor for finding files within a folder.

```
41 spotterForFilesFor: aStep
42     <spotterOrder: 40>

43     aStep listProcessor
44         title: 'Files';
45         itemFilterName: [:aReference| aReference basename ];
46         filter: GTFilterFileReference item: [ :filter :context |
47             self
48                 fileReferencesBy: #files
49                 inContext: context
50                 usingFilter: filter ];
51         when: [ :aReference | aReference isDirectory ];
```

Like in the case of the Moldable Inspector (Section 4.6.1 (p.65)), an extension (*i.e.*, a search processor) is attached to an object by defining in the class of the object a method constructing the extension. The Moldable Spotter uses the annotation `spotterOrder` (lines 30 and 42). The annotation parameter is used to sort processors in a search step. Moldable Spotter also implements tags using annotations: processors are added to tags using the annotation `spotterTag:` (*e.g.*, in line 31 the *Productions* processor is added to the tag *PetitParser*).

5.5.2 Exploration Sessions and Activation Predicates

An exploration session is modeled as a linked list of step objects. The code of a processor can access the exploration session using the method parameter of its containing method (`aStep`, line 52). This parameter gives access to the current step object. Developers can use it to access previous steps. This solution differs from the one used in the Moldable Inspector where developers needed to add an extra parameter to the method. We made this change as it does not require developers to decide in advance if the extension would need access to the context.

For example, Figure 5.5b (p.95) contains a search processor named *Shapes* that is only available when the previous step contains a Roassal visualization. If this is the case, the processor extracts all graphical shapes from that visualization that render the object loaded in the current step. In Figure 5.5b (p.95) we can see that three graphical shapes render the current object. To check if the previous object is a Roassal visualization the processor from Listing 5.3 (p.98) relies on an activation predicate (lines 63-65): using the current step object the activation predicate can access the previous step, if present. The provider can also access the previous objects (*i.e.*, the Roassal visualization) to extract the relevant graphical shapes.

Listing 5.3: Search processor for finding graphical shapes rendering the object loaded by the current step.

```
spotterForRenderingShapesFor: aStep
    <spotterOrder: 5>

    aStep listProcessor
        title: 'Shapes';
        candidatesLimit: 5;
        allCandidates: [ aStep previousStep origin
            elements select: [ :each | each model = self ] ];
        itemName: [ :each | each gtDisplayString ];
        filter: GTFilterSubstring;
        wantsToDisplayOnEmptyQuery: true ];
        when: [
            aStep hasPreviousStep and: [
                aStep previousStep origin isKindOf: RTView ] ]
```

5.5.3 Spotter in Other Languages

Currently Moldable Spotter is developed in Pharo and uses several querying facilities present in Pharo related to code and objects. This includes the ability to directly query code objects for relations (*e.g.*, ask for the methods of a class or for the subclasses of a class) and any object for its attributes. Pharo also provides support for determining relations between methods (*e.g.*, callers/callees of a method). While this simplifies the implementation of Moldable Spotter there is no conceptual limitation that ties Spotter to Pharo and prevents its implementation in other IDEs for object-oriented languages. For example, to integrate Moldable Spotter within IntelliJ, one can start from the *Global Search* tool that already provides the possibility to execute multiple searches in parallel and extend it with the notion of search context and search session. The AST model can then be used to implement queries over code entities.

5.6 The Cost of Custom Search Processors

Moldable Spotter enables developers to improve information foraging loops by creating domain-specific searches. To investigate the cost of creating a custom search we analyze the current search processors present in Pharo based on their size and searched data. We then discuss a user study with software developers and PhD students looking into how difficult it is to create extensions for Moldable Spotter.

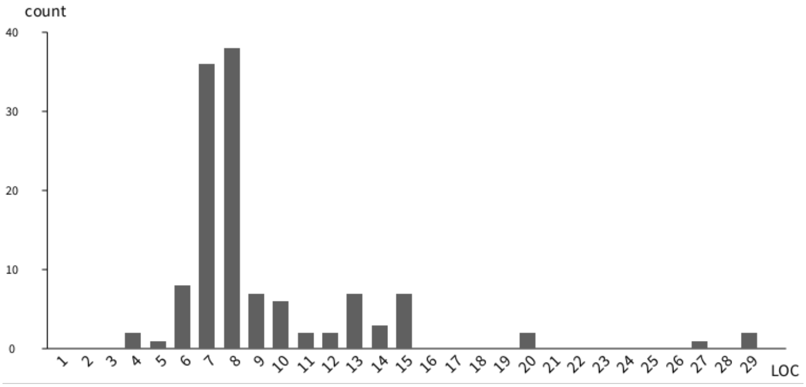


Figure 5.6: Size distribution in lines of code (LOC) for 124 search processors.

5.6.1 Analyzing Existing Extensions

Based on 124 search processors present in the Pharo IDE, the average cost of creating a processor in lines of code is 9.2. This includes the entire source code of the method defining a processor, as well as the source code of helper methods created together with the processor; this does not include methods called from within the processor that existed in the target class before adding the processor. As can be seen in Figure 5.6 (p.99), a large number of extensions require 7 or 8 lines of code. These extensions follow the same pattern as the one from lines 32–40, where the class already provides a way to get the required data and the extension just uses available methods and overrides default values from the processor. When this is not the case, the size of an extension is significantly larger.

5.6.2 A Taxonomy of Moldable Spotter Searches

In Table 5.3 (p.100) we classify all 124 search processors whose cost was analyzed in Section 5.6.1 (p.99), based on the type of searched data. Search processors in the *Global* category correspond to global searches through code entities and files / folders mentioned in Table 5.2 (p.91). Search processors from the category *Code entities* address developers questions from the *Packages, classes and methods* group from Table 5.2 (p.91). *Methods (containment)* groups searches through methods contained by a class (*e.g.*, instance methods, class / static methods, methods from subclasses); *Methods (relations)* covers searches involving relations between methods (*e.g.*, callers / callees). Code critiques are warnings about code returned by the Quality Assistant tool from Pharo. The *IDE* category contains searches though data more related to the IDE like: settings, help topics, plug-ins or external projects that can be loaded into the IDE, and URLs to repositories. Search processors in the *Project* category cover searches for the Metacello package management system and the Monticello versioning system.

Category	Data	Count
Global	Packages	1
	Classes	1
	Annotations	1
	Methods	5
	Global variables	1
	Files/folders	2
Code entities	Packages	5
	Classes	10
	Traits	4
	Annotations	1
	Methods (containment)	9
	Methods (relations)	7
	Attributes/variables	8
	Code critiques (QA)	3
IDE	Settings	2
	Help	2
	Menus	3
	Plug-ins/Projects (Catalog)	1
	Repositories	3
Project	Configurations (Metacello)	6
	Versioning (Monticello)	7
Domain objects	Collection objects	3
	Graphical objects	5
	XML objects	2
	Examples	6
	Parser objects	3
	Bytecode	1
	Moose models [Nierstrasz et al. 2005]	6
	Files	5
Other	Code/text	3
	Extensions	2
	History	4
	Dynamic	2

Table 5.3: Search processors ground based on the type of searched data. *Count* indicates the number of search processors for a data type.

Domain objects groups searches through specific objects like graphical widgets, parsers, XML documents, files, *etc. Dynamic* in the *Other* category covers searches where the results are generated from the query string; this includes a calculator for arithmetic expressions and a processor that given an URL pointing to a Shared Workspace[1] loads the code stored in that workspace. *Extensions* offer developers the possibility of searching through all Moldable Spotter processors as well as domain-specific extensions for other tools.

[1] http://ws.stfx.eu/

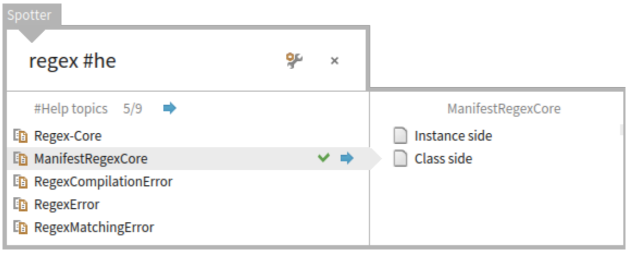


Figure 5.7: Searching for help topics related to regular expressions using a custom processor. Each `HelpTopic` object has a custom preview showing its content.

As indicated in Section 5.5 (p.96), developers attach extensions to an object's class. 64 extensions are attached to objects that represent code entities or to the object that models the current workspace (*e.g.*, all global processors). The remaining 60 extensions are attached to 32 different types of objects. On average an object has $1.85 \pm 1.5 (M \pm SD)$ search processors.

5.6.3 User Study Design

To investigate the creation of custom searches for Moldable Spotter in more detail, we designed and performed a pilot user study. Our main goal was to test the extension mechanism and not the knowledge of a particular application domain or the usability of Moldable Spotter. Towards this goal we decided to focus on simple domain models. In doing this we assume that developers working on an application know their domain model well. Through the evaluation of this study we aimed to better understand how developers behave when creating custom searches. Hence, we structured the evaluation based on the following research questions:

RQ1 How much time does it take to create a custom extension for developers who did not extend Moldable Spotter before?

RQ2 Does previous experience in extending Moldable Spotter reduce the effort required to create a new extension?

RQ3 How do developers approach the task of creating a custom extension for Moldable Spotter?

The user study consisted in implementing two custom searches for Moldable Spotter detailed below. We selected these searches as, while covering small domain models, they give a chance for the creation of relevant search extensions.

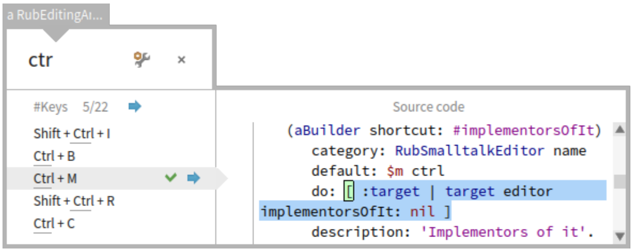
Task 1 – Help Topics *Extend Moldable Spotter with a custom global extension for searching through all help topics from the system by their title using a textual search.* Searches through documentation and other textual data are common tasks during development. With this task we aimed to see if developers can create such searches using Moldable Spotter. We selected the Pharo help as a data source for the search, given that it is a familiar data source for Pharo developers. While searches through help topics are not themselves novel, as they are a common presence in IDEs, they are well understood and provide a good baseline for testing if developers understand the extension mechanism from Moldable Spotter. In the case of Pharo the entire help is organized as a tree of `HelpTopic` objects. A `HelpTopic` has a title, a content and a list of children topics. Participants needed to create a global search extension that extracted all available help topics, filter them using one of the available filters based on their title and open a selected help topic in the Pharo Help Browser. Our default implementation of this extension is presented in Listing 5.4 (p.102). A real-world search using this extension is shown in Figure 5.7 (p.101).

Listing 5.4: Search processor for locating help topics.

```
spotterForHelpTopicFor: aStep
    <spotterOrder: 200>

    aStep listProcessor
        title: 'Help topics';
        allCandidates: [ SystemHelp asHelpTopic allSubtopics ];
        candidatesLimit: 5;
        itemName: [:helpTopic | helpTopic title ];
        itemIcon: [:helpTopic | helpTopic topicIcon ];
        actLogic: [:helpTopic | HelpBrowser openOn: helpTopic];
        filter: GTFilterSubstring;
        wantsToDisplayOnEmptyQuery: true
```

Task 2 - Morph Shortcuts *Extend Spotter with a custom textual search for searching through the shortcuts of a morph.* Morphs are the main graphical components from Pharo. A morph object uses a dispatcher object to store its shortcuts. The dispatcher object is stored in a dictionary together with other properties of a morph. A dispatcher object stores shortcuts as a set of keymap objects. Participants needed to create an extension that extracted all shortcuts from the dispatcher and supported a search based on the text of the shortcut (*e.g.*, 'ctrl+shift+s'). Unlike the previous extension, this is a specialized one that, to our knowledge, is not present in other development tools and IDEs. What we find in other IDEs are global searches through documentation that also take shortcuts into account. We selected this extension to observe how developers apply Moldable Spotter in a domain-specific context. Listing 5.5 (p.103) illustrates our default implementation; Figure 5.8 (p.103) shows a usage example.

The study had three parts. In the first part we gave participants a short introduction about how to use Moldable Spotter. The second part consisted in solving the two tasks and the third part in a survey for collecting impressions about the study and background data. During the introduction phase we only showed participants how to attach a custom search to a domain object. We did not go into any other details about how to extend Moldable Spotter. Instead we pointed participants to



Shift + Ctrl + I
Ctrl + B
Ctrl + M
Shift + Ctrl + R
Ctrl + C

```
    (aBuilder shortcut: #implementorsOfIt)
       category: RubSmalltalkEditor name
       default: $m ctrl
       do: [ :target | target editor
implementorsOfIt: nil ]
       description: 'Implementors of it'.
```

Figure 5.8: Processor for searching through the key bindings of a morph object. In this example the morph objects is a text editor for Pharo code, and the developer is searching for key bindings that use the *Ctrl* key. Each key binding has a custom preview that shows the code that will be executed when the key binding is pressed.

three places where they could get more data: *(i)* the Moldable Spotter documentation available in Pharo explaining Moldable Spotter the extension process in details; *(ii)* the class `GTSpotterCandidatesListProcessor` containing the main API for creating custom search processors; *(iii)* a tool for exploring all Moldable Spotter extensions available in the Pharo IDE.

During the second phase we captured both screen and audio recordings and asked participants to use a think-aloud strategy as they coded. For each task we provided participants with a description of the domain model presenting the main classes and methods. We provided these data to reduce the time needed for participants to learn the domain model. Participants first read the description of the task and of the domain and only afterwards started the coding phase. We only take the coding phase into account in our analysis. We did not impose any time limit for reading the provided material and implementing the task. A task was considered completed when the participant decided that it was good enough. The two tasks followed in immediate succession.

Listing 5.5: Search processor for searching through the key bindings of a morph.

```
Morph>>#spotterForKeysFor: aStep
    <spotterOrder: 15>

    aStep listProcessor
        title: 'Keys';
        allCandidates: [ self kmDispatcher
             allKeymaps asOrderedCollection ];
        candidatesLimit: 5;
        itemName:[:aKeymap| aKeymap shortcut printString];
        filter: GTFilterSubstrings
```

Six participants took part in this user study. Three PhD students, having between 2 and 4 years of experience with Pharo and three software developers, having between 1 and 3 years of experience with Pharo. All knew how to use Moldable Spotter, however, they had no knowledge of its extension mechanism. Two developers reported interacting briefly with the domain model used in the second task in the past.

5.6.4 User Study Results

We first determined the correctness of the extensions. All but one participant created extensions that correctly implemented the given tasks. One participant (*P2*) did not add the functionality for opening the help topics and the shortcut objects in the required tools. In the third phase he reported that he forgot about that part of the task. We also included this participant in our analysis as the implemented parts of the extensions were correct. In analyzing the coding phase of the study we distinguished between three types of activities: *(i)* understanding and implementing a Moldable Spotter extension; *(ii)* understanding the domain model; *(iii)* clarifying the task. We split each coding session into 30 second intervals and assigned them to one of the aforementioned categories; we inferred the category of an interval based on what code or object the participant was looking at and the think-aloud data.

RQ1 – Effort for creating a first extension Participants took, on average, 16 minutes to implement the first task (Table 5.4 (p.105) – *Total time*). Participants spent most of their time during this task (84%) on learning how to implement a Moldable Spotter extension. Participant *P5* was the exception: he decided to implement a different solution for obtaining all help topics than the one presented during the first phase. There was a large difference between participants in the completion time: participant *P2* took 24 minutes while participant *P6* took only 9 minutes. Regarding the size of an extension we obtained similar results to the discussion from Section 5.6.1 (p.99): on average, the size of extension was 8.6 LOC (Table 5.4 (p.105) – *Extension size*). Differences in size were due to extra features added by participants (*e.g.*, Participant *P3* added shortcuts and icons to each extension) and different ways to extract the required data. Given that this was the very first extension implemented by our participants, we consider the cost and the time to be low.

RQ2 – Effort for creating a second extension We explored our second research question by asking participants to create a second extension immediately after the first one for a more complex domain model. We noticed two changes from the first task. On the one hand, participants spent more time understanding the domain model (30% of the total time as opposed to 13% for the first task). On the other hand, the total time for solving this task improved on average by 37%; if we take only the extension implementation time into account the average individual improvement is 48%. There were nevertheless significant differences between individual participants: participant *P2* improved by 15 minutes 30 seconds while participant *P6* only by 30 seconds. In terms of lines of code, participants were consistent with the first extension. Based on this evidence we conclude that previous experience in extending Moldable Spotter reduces the time needed for creating new extensions.

		Extension implementation	Model understanding	Task clarification	Total time	Extension size
P1	Task 1	12m30s (89%)	1m30s (11%)	0 (0%)	14m00s	6 LOC
	Task 2	7m30s (63%)	4m30s (37%)	0	12m00s	7 LOC
P2	Task 1	22m30s (94%)	0	1m30s (6%)	24m00s	6 LOC
	Task 2	5m00s (59%)	3m30s (41%)	0	8m30s	6 LOC
P3	Task 1	17m00s (83%)	3m:00s (14%)	0m30s (3%)	21m00s	11 LOC
	Task 2	6m00s (80%)	1m30s (20%)	0	7m30s	10 LOC
P4	Task 1	16m00s (86%)	1m30s (5%)	1m00s (9%)	18m30s	11 LOC
	Task 2	11m30s (70%)	4m00s (24%)	1m00s (6%)	16m30s	12 LOC
P5	Task 1	8m00s (66%)	4m00s (34%)	0	12m00s	9 LOC
	Task 2	2m30s (62%)	1m30s (38%)	0	4m00s	7 LOC
P6	Task 1	8m00s (88%)	1m00s (12%)	0	9m00s	9 LOC
	Task 2	7m00s (82%)	1m30s (18%)	0	8m30s	10 LOC

Table 5.4: User study results for each participant and task.

RQ3 – Approaching the creation of extensions To understand the differences observed between participants and tasks we analyzed the strategies used by participants to implementing the given tasks. The first observation that holds for all participants is that they started to implement the required extensions by modifying an already existing extension. In the first part of the coding session (first 2-3 minutes) all participants looked through existing examples to understand the extension mechanism and selected an extension that they later modified. No participant started by reading documentation, however, later in the task if they got stuck, participants decided to either read documentation (*P2, P4*), browse Moldable Spotter classes (*P3*) or look at more examples (all participants did this to different extents).

The factor that contributed the most to high reduction in completion times between the first and the second task (participants *P2* and *P3*) was the example selected as a starting point in the first task. To exemplify, participant *P2* selected to start by modifying the global search for methods available in Pharo. Currently this search is an optimized search using streams that combine the provider and the query engine. As discussed in Section 5.5 (p.96), Moldable Spotter currently does not optimize the rapid creation of these kinds of searches. After having problems with the extension, participant *P2* chose to look at three more examples which happen to also also be searches based on streams. In the end participant *P2* moved to reading documentation and found a different way to create an extension (using the API from lines 32–40). At the opposite end participant *P6* started with an example similar to the solution required for the first task. He adapted this example to the first task with ease.

This study revealed that the examples used by participants to learn how to extend Moldable Spotter had an impact on their efficiency. For this study we used all extensions currently available in the Pharo IDE as examples . These examples had no associated documentation giving insight into why they are implemented that way. These made participants choose to start from examples that were not necessarily suited for their requirements. We consider addressing these aspect by providing a collection of curated extensions, better search for examples and better documentation for extensions.

Threats to validity During the coding phase participants needed to think-aloud and were also observed by the person conducting the study; five participants knew the person conducting the study. We cannot exclude that this may have caused changes in the way they approached the tasks. The task also focused on simple domain models that may not reflect the reality of complex software applications.

5.7 Spotter in Practice

Section 5.6 (p.98) explored the cost of extending Moldable Spotter. This gives no insight into how developers use Moldable Spotter in practice. To address this we collected and analyzed usage data and mailing-list discussions and performed an online survey.

5.7.1 Usage Data and Mailing-list Discussions

Moldable Spotter was integrated into the alpha version of Pharo 4 in December 2014. Six months after the initial integration we analysed usage data recorded over a period of two months (April 2014 - May 2015) using a visual language [Kubelka et al. 2015]. To summarize, we noticed that developers did not discover and use the navigation features of Moldable Spotter to their fullest potential. For example, only half of the recorded developers used the dive-in feature at least once. Regarding search data, although we observed developers using 51 search processors, more than 74% of the time developers only used Moldable Spotter to search through classes and implementors of methods. One explanation for this observation is that Moldable Spotter exposes information that is not apparent and users need to be explicitly informed about this.

In the same period we also gathered feedback from discussion on several Pharo mailing-lists. Most feedback gathered from mailing-lists is related to the discoverability of features in the UI, as Moldable Spotter proposes a UI different from other tools in the Pharo IDE. Based on this feedback we added the possibility to select processors directly from the query (*e.g.*, entering the *"spotter #pragma"* in Figure 5.3a

	Very useful	Useful	Sometimes useful	Sometimes irrelevant	Irrelevant	Very irrelevant	Unknown feature
FT1	7 (20%)	7 (20%)	13 (37.2%)	2 (5.7%)	1 (2.8%)	1 (2.8%)	4 (11.4%)
FT2	9 (25.7%)	9 (25.7%)	5 (14.3%)	3 (8.6%)	0 (0%)	1 (2.8%)	8 (22.9%)
FT3	8 (22.9%)	8 (22.9%)	5 (14.3%)	2 (5.7%)	0 (0%)	3 (8.6%)	9 (25.7%)
FT4	7 (20%)	6 (17.4%)	9 (25.7%)	0 (0%)	1 (2.8%)	1 (2.8%)	11 (31.4%)
Moldable Spotter	14 (40%)	12 (34.3%)	8 (22.9%)	0 (0%)	0 (0%)	1 (2.8%)	—

Table 5.5: Survey results related to how respondents perceive Moldable Spotter and its features.

(p.88) only shows the *Pragmas* processor; # filters processors by name) and selection of common processors using keyboard shortcuts.

5.7.2 Survey

To further explore how developers perceive and use Moldable Spotter, one year after the release of Pharo 4, we performed an online survey during March 2016. We advertised the survey on Pharo related mailing-lists and collected 35 answers from software developers (17 responses – 48.6%), software researchers (12 responses – 34.3%), students (5 responses – 14.3%) and others (1 response – 2.8%). Regarding their programming experience with object-oriented languages, 5 respondents (14.3%) reported between 1 and 3 years, 10 respondents (28.6%) between 4 and 10 years and 20 respondents (57.1%) more than 10 years. Respondents used Moldable Spotter until the survey during different lengths of time: 5 respondents (14.3%) less than 3 months, 20 respondents (57.1%) between 4 and 12 months and 10 respondents (28.6%) more than 12 months. When asked how often they used Moldable Spotter 25 respondents (71.4%) answered that during development they use Moldable Spotter at least several times a day; 9 respondents (25.7%) only used Moldable Spotter sometimes; one respondent stopped using Moldable Spotter. Out of the 35 respondents, 8 (22.9%) also extended Moldable Spotter with a custom search until the survey.

Next, respondents were asked to rate how useful they find Moldable Spotter as well as four individual features using a six point scale ranging from *very useful* to *very irrelevant*. The four features were: *dive-in a search result* (*FT1*), *show all results* (*FT2*), *filter search processor using #* (*FT3*) and *preview for the selected element* (*FT4*). For each individual feature, respondents could also indicate that they did not know about that feature. Table 5.5 (p.107) shows an overview of the results. The vast majority of respondents found Moldable Spotter to be at least sometimes useful: 40% very useful, 34.3% useful and 22.9% sometimes useful. Regarding the individual features, for each there were respondents that did not know about that feature. Most respondents did not know about *FT2*, *FT3* and *FT4*. This indicates the need to further improve the user interface of Moldable Spotter to help users discover features. Answers for the individual features followed a similar pattern: most participants found them to be at least sometimes useful, with a few finding them irrelevant.

We also asked respondents to rate the statements below on a 5-point Likert scale:

Spotter reduced the number of tools I used for searching in the Pharo IDE. This statement received the following responses: 2 (5.7%) strongly agree, 13 (37.2%) agree, 7 (20%) neutral, 11 (31.4%) disagree, 2 (5.7%) strongly disagree. This indicates that some respondents use Moldable Spotter alongside the other search tools from the IDE.

Spotter reduced the time I need to perform searches in the Pharo IDE. This statement received the following responses: 10 (28.6%) strongly agree, 12 (34.3%) agree, 8 (22.9%) neutral, 3 (8.6%) disagree, 2 (5.7%) strongly disagree. We observe that more respondents agree than in the case of the previous statement. This suggests that respondents perform queries faster using Moldable Spotter than with other search tools from the Pharo IDE. This aspect still needs to be further investigated.

Threats to validity This survey is prone to both internal and external threats to validity. Respondents could chose to remain anonymous; 17 respondents (48.6%) did so by choosing to not provide an email address; this was observed mostly in respondents giving negative feedback.

5.8 Conclusions

Domain concepts play an important role in program comprehension. Relying only on generic search tools during information foraging loops requires developers to focus on searching for domain concepts instead of reasoning in terms of those concepts. This can be addressed if search tools enable developers to directly search through domain concepts. To support this, we proposed Moldable Spotter, a moldable tool that allows developers to inexpensively incorporate domain concepts in the search process as well as discover searches applicable for their own contexts. We designed the Moldable Spotter by applying the designed principles for moldable tools introduced in Section 3.1 (p.33) in the following way:

DP 1. Identify common types of tool-specific adaptations: The Moldable Spotter enables domain-specific adaptations by allowing developers to create custom searches for their domain objects. Like the Moldable Inspector, the Moldable Spotter proposes a dedicated user interface for browsing search results and navigating through search processors.

DP 2. Simplify the creation of common adaptations: To support inexpensive creation of custom search processors, Moldable Spotter models search processors as objects and provides an internal DSL developers can use to configure the processors in a few lines of code. We showed that this reduces the cost of custom searches for objects modeling different kinds of information.

DP 3. Do not limit the types of possible adaptations only to common ones: The inexpensive approach for creating a search processor optimizes for the case where indexing and obtaining the searched data using streams is not required. When these are needed developers can replace the *Query engine* and *Provider* components of a search processor with custom components that provide those services.

DP 4. Attach activation predicates to adaptations: The Moldable Spotter maintains a search context consisting of previously searched objects and explicit data, using the same approach as the Moldable Inspector. Each search processor can have an activation predicate that decides, depending on the search context, whether the processor is applicable or not.

DP 5. Update adaptations based on a development context: Every time the developer opens Moldable Spotter, updates the query text or creates a new search step, Moldable Spotter updates the search contexts and only takes into account processors valid in that context.

Through usage scenarios we showed that Moldable Spotter can address a wide range of domain-specific questions from various domains (*e.g.*, parsing, GUIs, event-bases systems, profilers, compilers, HTTP servers). A pilot user study further revealed that developers who did not extend Moldable Spotter before can create custom searches with a low effort. We also showed that Moldable Spotter can be extended to also support generic searches through code, classes, methods, bug reports, run-time objects, *etc.* By doing this, Moldable Spotter can provide a single entry point for embedding search support within IDEs.

As soon as we started programming, we found to our surprise that it wasn't as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.

Maurice Wilkes

6 Moldable Debugger

Debuggers are an essential category of tools for reasoning about the dynamic behavior of software systems. Nevertheless, traditional debuggers rely on generic mechanisms to introspect and interact with the running systems, 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. To reduce this gap and to increase the efficiency of the debugging process, we explore in this chapter how to provide debugging support at the level of abstraction of an application through a moldable tool.

6.1 Introduction

Debugging is an integral activity of the software development process, consisting in localizing, understanding and fixing software bugs, with the goal of making software systems behave as expected. Nevertheless, despite its importance, debugging is a laborious, costly and time-consuming activity [Vessey 1986; Tassey 2002]. Given the difficulty and pervasiveness of debugging, numerous debugging techniques have been proposed (*e.g.*, remote debugging, omniscient debugging [Pothier et al. 2007], post-mortem debugging [Yuan et al. 2010; Han et al. 2012], delta debugging [Zeller 1999] – to name a few), each with its own constraints and benefits. These techniques rely on a wide set of tools to locate, extract and analyze data about the run-time behavior of software systems.

Among the multitude of debugging tools, debuggers are an essential category. If loggers[1] or profilers [Ressia et al. 2012b] record run-time data presented to developers post-mortem, debuggers enable developers to directly observe the run-time behavior of software [Roehm et al. 2012; Murphy et al. 2006]. In test-driven development the debugger is used as a development tool given that it provides direct access to the running system [Beck 2002]. This makes the debugger a crucial tool in any programming environment.

Nevertheless, classical debuggers focusing on generic stack-based operations, line breakpoints and generic user interfaces do not allow developers to rely on domain concepts when debugging object-oriented applications. For example, when developing a parser, one common action is to step through the execution until parsing reaches a certain position in the input stream. However, as it has no knowledge of parsing and stream manipulation, a classical generic debugger requires developers to manipulate low-level concepts like message sends or variable accesses. This abstraction gap leads to an ineffective and error-prone effort [Sillito et al. 2008].

Furthermore, classical debuggers are less useful when the bug is far away from its manifestation [Zeller 2005]. Raising the level of abstraction of a debugger by offering object-oriented debugging idioms [Ressia et al. 2012a] solves only part of the problem, as these debuggers cannot capture domain concepts constructed on top of object-oriented programming idioms. Other approaches raise the level of abstraction in different ways: back-in-time debugging, for example, allows one to inspect previous program states and step backwards in the control flow [Lienhard et al. 2008].

When looking at a debugger, there exist two main approaches to address, at the application level, the gap between the debugging needs and debugging support:

- Enable developers to create domain-specific debugging operations for stepping through the execution, setting breakpoints, checking invariants [Olsson et al. 1991; Marceau et al. 2007; Khoo et al. 2013] and querying stack-related information [Lencevicius et al. 1997; Potanin et al. 2004; Martin et al. 2005; Ducasse et al. 2006];
- Allow domain-specific debuggers to have domain-specific user interfaces that do not have the same fixed content or layout [DeLine et al. 2012]. This is needed as, depending on the domain, different information can be of interest. For example, a domain-specific debugger for a parser needs to show information about grammar productions.

Each of these approaches addresses individual debugging problems (*i.e.*, interacting with the runtime at the right level of abstraction and displaying data relevant for the application domain), however, until now there does not exist one comprehensive approach to tackle the overall debugging puzzle. We propose an approach that incorporates both of these directions in one coherent model. We start from the realization

[1] `http://logging.apache.org/log4j`

that the most basic feature of a debugger model is to enable the customization of all aspects, and we design a debugging model around this principle. We call our approach the *Moldable Debugger*.

The Moldable Debugger decomposes a domain-specific debugger into a *domain-specific extension* and an *activation predicate*. The domain-specific extension customizes the user interface and the operations of the debugger, while the activation predicate captures the state of the running program in which that domain-specific extension is applicable. In a nutshell, the Moldable Debugger model follows the moldable tools approach and allows developers to *mold* the functionality of the debugger to their own domains by creating domain-specific extensions. Then, at run time, the Moldable Debugger adapts to the current domain by using activation predicates to select appropriate extensions.

A domain-specific extension consists of *(i)* a set of domain-specific *debugging operations* and *(ii)* a domain-specific *debugging view*, both built on top of *(iii)* a *debugging session*. The debugging session abstracts the low-level details of a domain. Domain-specific operations reify debugging operations as objects that control the execution of a program by creating and combining *debugging events*. We model debugging events as objects that encapsulate *a predicate over the state of the running program* (*e.g.*, method call, attribute mutation) [Auguston et al. 2002]. A domain-specific debugging view consists of a set of graphical widgets offering debugging information. Each widget locates and loads, at run-time, relevant domain-specific operations using an annotation-based approach. The Moldable Debugger incorporates the Moldable Inspector as a widget in the user interface.

The contributions of this chapter are as follows:

- Proposing four requirements that an infrastructure for developing domain-specific debuggers should support;
- Showing how the moldable tools approach can be applied to create the Moldable Debugger, a model for developing domain-specific debuggers following the four proposed requirements that integrates domain-specific debugging operations with domain-specific user interfaces;
- Illustrating, through usage examples, the advantages of the Moldable Debugger model over generic debuggers;
- A prototype of the Moldable Debugger model together with a discussion of three different approaches for implementing domain-specific debugging operations and an in-depth analysis of their performance.

Structure of the Chapter

In Section 6.2 (p.114) we introduce and discuss requirements for working with domain-specific debuggers. In Section 6.3 (p.117) we present the Moldable Debugger, a debugging model following those requirements, and we show, in Section 6.4 (p.120), how to address domain-specific debugging problems by adapting the debugger to specific domains. We analyze implementation aspects in Section 6.5 (p.129), and in Section 6.6 (p.134) we discuss the cost of domain-specific debugger and the limits of the Moldable Debugger. In Section 6.7 (p.136) we explore related works and we conclude in Section 6.8 (p.140) by looking at how the Moldable Debugger applies the moldable tools approach.

6.2 Requirements for Domain-specific Debuggers

Debuggers are comprehension tools. Despite their importance, most debuggers only provide low-level operations that do not capture user intent, and standard user interfaces that only display generic information. These issues can be addressed if developers are able to create domain-specific debuggers adapted to their domain concepts. Domain-specific debuggers can provide features at a higher level of abstraction that match the domain model of software applications. In this section we propose four requirements that an infrastructure for developing domain-specific debuggers should support, namely: *domain-specific user interfaces*, *domain-specific debugging operations*, *automatic discovery* and *dynamic switching*.

6.2.1 Domain-specific User Interfaces

User interfaces of software development tools tend to provide large quantities of information, especially as the size of systems increases. This, in turn, increases the navigation effort of identifying the information relevant for a given task [Ko et al. 2006]. Consider a unit test with a failing equality assertion. In this case, the only information required by the developer is the difference between the expected and the actual value. However, finding the exact difference in non-trivial values can be daunting and can require multiple interactions such as finding the place in the stack where both variables are accessible, and opening separate inspectors for each value. A better approach, if a developer opens a debugger when a test fails, is to show a diff view on the two values directly in the debugger when such an assertion exception occurs, without requiring any further action.

This shows that user interfaces that extract and highlight domain-specific information have the power to reduce the overall effort of code understanding [Kersten and Murphy 2005]. However, today's debuggers tend to provide generic user interfaces that cannot emphasize what is important in application domains. To address this concern, an infrastructure for developing domain-specific debuggers should:

- Allow domain-specific debuggers to have *domain-specific user interfaces* displaying information relevant for their particular domains;
- Support the *fast prototyping* of domain-specific user interfaces for debugging.

While other approaches, like *deet* [Hanson and Korn 1997] and *Debugger Canvas* [DeLine et al. 2012], support domain-specific user interfaces for different domains, they do not offer an easy and rapid way to develop such domain-specific user interfaces.

6.2.2 Domain-specific Debugging Operations

Debugging can be a laborious activity requiring much manual and repetitive work. On the one hand, debuggers support language-level operations, while developers think in terms of domain abstractions. As a consequence, developers need to mentally construct high-level abstractions on top of language constructs, which can be time-consuming. On the other hand, debuggers rarely provide support for identifying and navigating through those high-level abstractions. This leads to repetitive tasks that increase debugging time.

Consider a framework for synchronous message passing. One common use case in applications using it is the delivery of a message to a list of subscribers. When debugging this use case, a developer might need to *step to when the current message is delivered to the next subscriber*. One solution is to manually step through the execution until the desired code location is reached. Another consists in identifying the code location beforehand, setting a breakpoint there and resuming execution. In both cases the developer has to manually perform a series of actions each time she wants to execute this high-level operation.

A predefined set of debugging operations cannot anticipate and capture all relevant situations. Furthermore, depending on the domain, different debugging operations are of interest. Thus, an infrastructure for developing domain-specific debuggers should:

- Support the creation of domain-specific debugging operations that allow developers to *express and automate* high-level abstractions from application domains (*e.g.*, creating domain-specific breakpoints, building and checking invariants, altering the state of the running system). Since developers view debugging as an event-oriented process, the underlying mechanism should allow developers to treat the running program as a generator of events, where an event corresponds to the occurrence of a particular action during the program's execution, like: method entry, attribute access, attribute write or memory access.
- Group together those debugging operations that are relevant for a domain and only make them available to developers when they encounter that domain.

This idea of having *customizable* or *programmable* debugging operations that view debugging as an event-oriented activity has been supported in related works [Olsson et al. 1991; Marceau et al. 2007; Khoo et al. 2013; Hanson and Korn 1997]. Mainstream debuggers like GDB[2] have, to some extent, also incorporated it. We also consider

[2] http://www.gnu.org/software/gdb

that debugging operations should be grouped based on the domain and only usable when working with that domain.

6.2.3 Automatic Discovery

Consider an environment that offers a domain-specific debugger for parsers and one for events. If, while stepping through the execution of a program, a developer reaches a parser, the environment should facilitate the developer to discover that a suitable domain-specific debugger is available; if later the execution of the parser completes and the program continues with the propagation of an event, the environment should inform the developer that the current debugger extension is no longer useful and that a better one exists. Hence, infrastructures for developing domain-specific debuggers should help *developers to discover domain-specific debuggers during debugging*. This way, the burden of finding appropriate domain-specific debuggers and determining when they are applicable does not fall on developers.

6.2.4 Dynamic Switching

Even with just two different types of debuggers, DeLine *et al.* noticed that users needed to switch between them at run time [DeLine et al. 2012]. This happened as users did not know in advance in what situation they would find themselves in during debugging. Thus, they often did not start with the appropriate one.

Furthermore, even if one starts with the right domain-specific debugger, during debugging situations can arise requiring a different one. For example, the following scenario can occur: *(i)* while investigating how an event is propagated through the application *(ii)* a developer discovers that it is used to trigger a script constructing a GUI, and later learns that *(iii)* the script uses a parser to read the content of a file and populate the GUI. At each step a different domain-specific debugger can be used. For this to be feasible, *domain-specific debuggers should be switchable at debug time without having to restart the application*.

6.2.5 Summary

Generic debuggers focusing on low-level programming constructs, while universally applicable, cannot efficiently answer domain-specific questions, as they make it difficult for developers to take advantage of domain concepts. Domain-specific debuggers aware of the application domain can provide direct answers. We advocate that a debugging infrastructure for developing domain-specific debuggers should support the four aforementioned requirements (domain-specific user interfaces, domain-specific debugging operations, automatic discovery and dynamic switching). Next we show that a debugging infrastructure following this set of requirements supports developers in creating and working with relevant domain-specific debuggers.

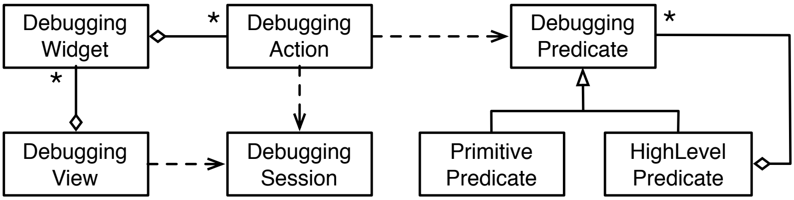


Figure 6.1: The structure of a domain-specific extension.

6.3 A Closer Look at the Moldable Debugger Model

The Moldable Debugger enables domain-specific debuggers that can express and answer questions at the application level. A domain-specific debugger consists of a domain-specific extension encapsulating the functionality and an activation predicate encapsulating the situations in which the extension is applicable. This model directly follows the moldable tools idea and makes it possible for multiple domain-specific debuggers to coexist at the same time.

To exemplify the ideas behind the proposed solution we will instantiate a domain-specific debugger for working with synchronous events[3]. Event-based programming poses debugging challenges as it favors a control flow based on events not supported well by conventional stack-based debuggers.

6.3.1 Modeling Domain-specific Extensions

A domain-specific extension defines the functionality of a domain-specific debugger using multiple debugging operations and a debugging view. Debugging operations rely on debugging predicates to implement high-level abstractions (*e.g.*, domain-specific breakpoints); the debugging view highlights contextual information. To decouple these components from the low-level details of a domain they are built on top of a debugging session.

A *debugging session* encapsulates the logic for working with processes and execution contexts (*i.e.*, stack frames). It further implements common stack-based operations like: *step into*, *step over*, *resume/restart process*, *etc.* Domain-specific debuggers can extend the debugging session to extract and store custom information from the runtime, or provide fine-grained debugging operations. For example, our event-based debugger extends the debugging session to extract and store the current event together with the sender of that event, the receiver of that event, and the announcer that propagated that event.

[3] This section briefly describes this debugger. More details are given in Section 6.4.2 (p.122).

Attribute read	detects when a field of any object of a certain type is accessed
Attribute write	detects when a field of any object of a certain type is mutated
Method call	detects when a given method is called on any object of a certain type
Message send	detects when a specified method is invoked from a given method
State check	checks a generic condition on the state of the running program (*e.g.*, the identity of an object).

Table 6.1: Primitive debugging predicates capturing basic events.

Debugging predicates detect *run-time events*. Basic run-time events (*e.g.*, method call, attribute access) are detected using a set of *primitive predicates*, detailed in Table 6.1 (p.118). More complex run-time events are detected using *high-level predicates* that combine both *primitive predicates* and other *high-level predicates* (Figure 6.1 (p.117)). Both of these types of debugging predicates are modeled as objects whose state does not change after creation. Debugging predicates are related to *coupling invariants* from data refinement, as coupling invariants are traditionally defined as logical formulas that relate concrete variables to abstract variables [Roever and Engelhardt 2008]. Hence, they can detect specific conditions during the execution of a program.

Consider our event-based debugger. This debugger can provide high-level predicates to detect when a sender initiates the delivery of an event, or when the middleware delivers the event to a receiver.

Debugging operations can execute the program until a debugging predicate is satisfied or can perform an action every time a debugging predicate is satisfied. They are modeled as objects that can accumulate state. They can implement breakpoints, log data, watch fields, change the program's state, detect violations of invariants, *etc.* In the previous example a debugging operation can be used to stop the execution when an event is delivered to a receiver. Another debugging operation can log all events delivered to a particular receiver without stopping the execution.

At each point during the execution of a program only a single debugging operation can be active. Thus, debugging operations have to be run sequentially. For example, in the event debugger one cannot activate, at the same time, two debugging operations, each detecting when an event of a certain type is sent to a receiver. One can, however, create a single debugging operation that detects when an event of either type is sent to a receiver. This design decision simplifies the implementation of the model, given that two conflicting operations cannot run at the same time. Hence, no conflict resolution mechanism is required.

The Moldable Debugger models a *debugging view* as a collection of graphical widgets (*e.g.*, stack, code editor, object inspector) arranged using a particular layout. At run time, each widget loads a subset of debugging operations. Determining what operations are loaded by which widgets is done at run time via a lookup mechanism of operation declarations (implemented in practice using annotations). This way, widgets do not depend upon debugging operations, and are able to reload debugging operations dynamically during execution.

Our event-based debugger provides dedicated widgets that display an event together with the sender and the receiver of that event. These widgets load and display the debugging operations for working with synchronous events, like logging all events or placing a breakpoint when an event is delivered to a receiver.

Developers can create domain-specific extensions by:

1. Extending the debugging session with additional functionality;

2. Creating domain-specific debugging predicates and operations;

3. Specifying a domain-specific debugging view;

4. Linking debugging operations to graphical widgets.

6.3.2 Combining Predicates

We support two boolean operators for combining debugging predicates:

and(predicate1, predicate2, ..., predicateN): creates a new predicate that detects a run-time event when all given predicates have detected a run-time event at the same time. This only allows for combining *attribute read*, *attribute write*, *method call* and *message send* with one or more *state check* predicates. For example, detecting when a method is called on an a given object is done by using a *method call* predicate together with a *state check* predicate verifying the identify of the receiver object.

or(predicate1, predicate2, ..., predicateN): creates a new predicate that detects a run-time event when any one of the given predicates have detected a run-time event. For example, detecting when any message is sent from a given method is done by using a *message send* predicate for every message send from the given method.

Given the definition of the *and* predicate, detecting high-level events that only happen when a sequence of events is detected is not possible. For example, one cannot detect, just by combing debugging predicates, a sequence of method calls on a given object. This operation requires persistent state and can be implemented by a debugging action.

6.3.3 Dynamic Integration

The Moldable Debugger model enables each domain-specific debugger to decide if it can handle a debugging situation by defining an activation predicate over the current debugging context. The debugging context captures a development context relevant for debugging. In the case of the Moldable Debugger, the debugging context consists in the execution stack; currently, unlike the Moldable Inspector and Moldable Spotter, the Moldable Debugger does not reify and use interaction data for selecting extensions.

Activation predicates capture the state of the running program in which a domain-specific debugger is applicable. While debugging predicates are applied on an execution context, activation predicates are applied on the entire execution stack. For example, the activation predicate of our event-based debugger will check if the execution stack contains an execution context involving an event. This way, developers do not have to be aware of applicable debuggers a priori. At any point during debugging they can see what domain-specific debuggers are applicable (*i.e.*, their activation predicate matches the current debugging context) and can switch to any of them.

When a domain-specific debugger is no longer appropriate, we do not automatically switch to another one. Instead, all domain-specific widgets and operations are disabled. This avoids confronting users with unexpected changes in the user interface if the new debugging view has a radically different layout. Nevertheless, for complex user interfaces where many widgets need to be disabled this solution can still lead to unexpected changes, though this is not as radical as replacing the user interface with a different one. Designing the disabled widgets in a way that does not confuse users could alleviate part of this issue (*e.g.*, by showing a grayed out version of the widget with no interaction possibilities).

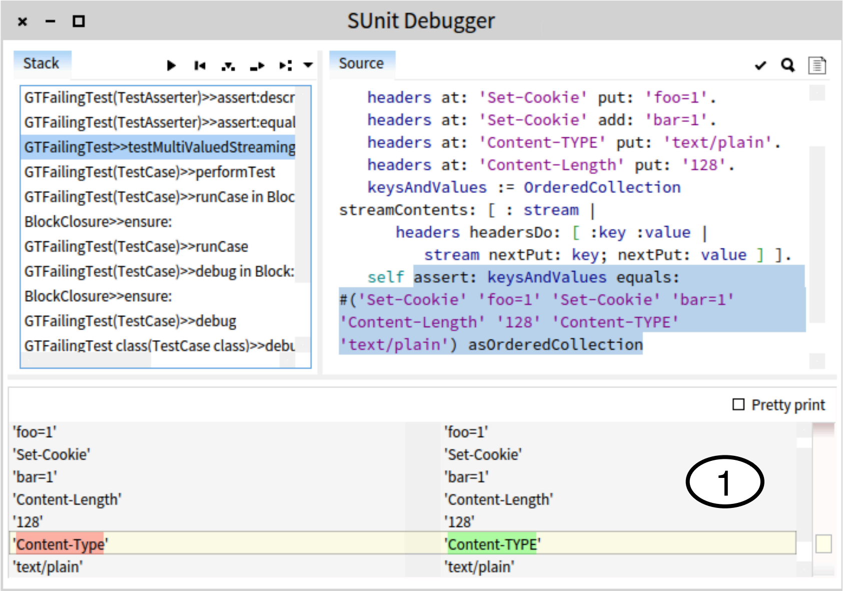
To further improve working with multiple domain-specific debuggers we provide two additional concepts:

A debugger-oriented breakpoint is a breakpoint that when reached opens the domain-specific debugger best suited for the current situation. If more than one view is available the developer is asked to choose one.

Debugger-oriented steps are debugging operations that resume execution until a given domain-specific debugger is applicable. They are useful when a developer knows a domain-specific debugger will be used at some point in the future, but is not sure when or where.

6.4 Addressing Domain-specific Debugging Problems

To demonstrate that the Moldable Debugger addresses the requirements identified in Section 6.2 (p.114), we have instantiated it for six different domains: *testing*, *synchronous*



```
GTFailingTest(TestAsserter)>>assert:descr
GTFailingTest(TestAsserter)>>assert:equal
GTFailingTest>>testMultiValuedStreaming
GTFailingTest(TestCase)>>performTest
GTFailingTest(TestCase)>>runCase in Bloc
BlockClosure>>ensure:
GTFailingTest(TestCase)>>runCase
GTFailingTest(TestCase)>>debug in Block:
BlockClosure>>ensure:
GTFailingTest(TestCase)>>debug
GTFailingTest class(TestCase class)>>debu
```

```
   headers at: 'Set-Cookie' put: 'foo=1'.
   headers at: 'Set-Cookie' add: 'bar=1'.
   headers at: 'Content-TYPE' put: 'text/plain'.
   headers at: 'Content-Length' put: '128'.
   keysAndValues := OrderedCollection
streamContents: [ : stream |
     headers headersDo: [ :key :value |
        stream nextPut: key; nextPut: value ] ].
   self assert: keysAndValues equals:
#('Set-Cookie' 'foo=1' 'Set-Cookie' 'bar=1'
'Content-Length' '128' 'Content-TYPE'
'text/plain') asOrderedCollection
```

```
'foo=1'             'foo=1'
'Set-Cookie'        'Set-Cookie'
'bar=1'             'bar=1'
'Content-Length'    'Content-Length'
'128'               '128'
'Content-Type'      'Content-TYPE'
'text/plain'        'text/plain'
```

Figure 6.2: A domain-specific debugger for SUnit: (1) diff between the textual representation of the expected and obtained value.

events, *parsing*, *internal DSLs*, *profiling* and *bytecode interpretation*. In this section we detail these instantiations.

6.4.1 Testing with SUnit

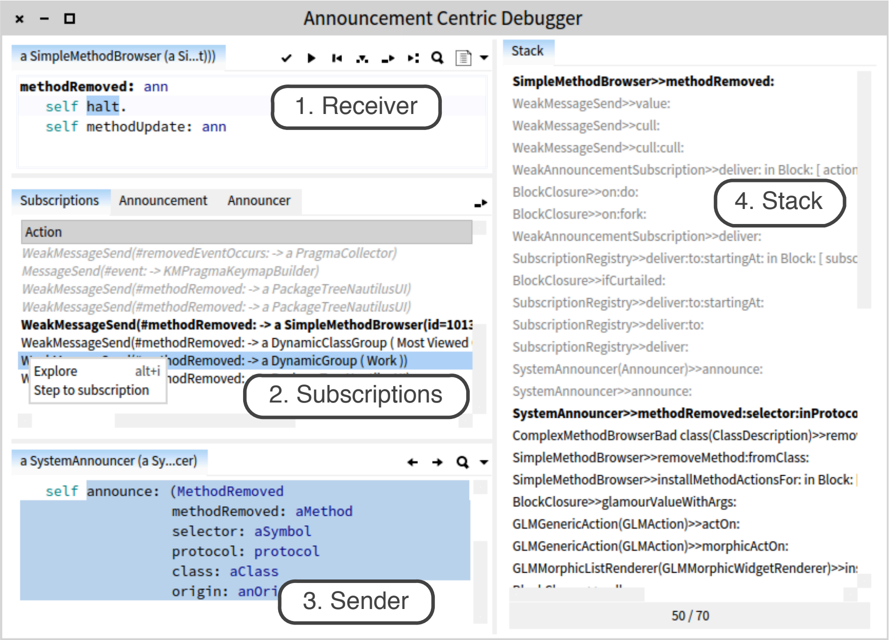
SUnit is a framework for creating unit tests [Beck 1999]. The framework provides an assertion to check if a computation results in an expected value. If the assertion fails the developer is presented with a debugger that can be used to compare the obtained value with the expected one. If these values are complex, identifying the difference may be time consuming. A solution is needed to *facilitate comparison*.

To address this, we developed a domain-specific debugger having the following components:

Session: extracts the expected and the obtained value from the runtime;

View: displays a diff between the textual representation of the two values. The diff view depends on the domain of the data being compared. Figure 6.2 (p.121) shows how it can be used to compare two HTTP headers;

Activation predicate: verifies whether the execution stack contains a failing equality assertion.



```
methodRemoved: ann
    self halt.
    self methodUpdate: ann
```

```
Action
WeakMessageSend(#removedEventOccurs: -> a PragmaCollector)
MessageSend(#event: -> KMPragmaKeymapBuilder)
WeakMessageSend(#methodRemoved: -> a PackageTreeNautilusUI)
WeakMessageSend(#methodRemoved: -> a PackageTreeNautilusUI)
WeakMessageSend(#methodRemoved: -> a SimpleMethodBrowser(id=1013
WeakMessageSend(#methodRemoved: -> a DynamicClassGroup ( Most Viewed
W                  hodRemoved: -> a DynamicGroup ( Work ))
W                  hodRemoved:
```

```
    self announce: (MethodRemoved
                      methodRemoved: aMethod
                      selector: aSymbol
                      protocol: protocol
                      class: aClass
                      origin: anOri
```

```
SimpleMethodBrowser>>methodRemoved:
WeakMessageSend>>value:
WeakMessageSend>>cull:
WeakMessageSend>>cull:cull:
WeakAnnouncementSubscription>>deliver: in Block: [ action
BlockClosure>>on:do:
BlockClosure>>on:fork:
WeakAnnouncementSubscription>>deliver:
SubscriptionRegistry>>deliver:to:startingAt: in Block: [ subsc
BlockClosure>>ifCurtailed:
SubscriptionRegistry>>deliver:to:startingAt:
SubscriptionRegistry>>deliver:to:
SubscriptionRegistry>>deliver:
SystemAnnouncer(Announcer)>>announce:
SystemAnnouncer>>announce:
SystemAnnouncer>>methodRemoved:selector:inProtoco
ComplexMethodBrowserBad class(ClassDescription)>>remo
SimpleMethodBrowser>>removeMethod:fromClass:
SimpleMethodBrowser>>installMethodActionsFor: in Block: |
BlockClosure>>glamourValueWithArgs:
GLMGenericAction(GLMAction)>>actOn:
GLMGenericAction(GLMAction)>>morphicActOn:
GLMMorphicListRenderer(GLMMorphicWidgetRenderer)>>in:
```

Figure 6.3: A domain-specific debugger for announcements: (1) the receiver and (3) the sender of an announcement; (2) subscriptions triggered by the current announcement. The run-time stack (4) highlights the stack frames that correspond to the sender and receiver, and grays out those stack frames internal to the announcements framework.

6.4.2 An Announcement-centric Debugger

The *Announcements* framework from Pharo provides a synchronous notification mechanism between objects based on a registration mechanism and first class announcements (*i.e.*, objects storing all information relevant to particular occurrences of events). Since the control flow for announcements is event-based, it does not match well the stack-based paradigm used by conventional debuggers. For example, Section 6.2.2 (p.115) describes a high-level action for *delivering an announcement to a list of subscribers*. Furthermore, when debugging announcements it is useful to see at the same time *both the sender and the receiver of an announcement*; most debuggers only show the receiver.

To address these problems we have created a domain-specific debugger, shown in Figure 6.3 (p.122). A previous work discusses the need for such a debugger in more detail and looks more closely at the runtime support needed to make the debugger possible [Chiş et al. 2013]. This debugger is instantiated as follows:

Session: extracts from the runtime the announcement, the sender, the receiver and all the other subscriptions triggered by the current announcement;

Predicates:

Detect when the framework initiates the delivery of a subscription: *message send*(`deliver:` in `SubscriptionRegistry>>deliver:to:startingAt:`);

Detect when the framework delivers a subscription to an object: *method call*(`aSubscription action selector`) on the class of the object, *state check* verifying the identity of the target object and *state check* verifying that the message was sent from the announcer holding the given subscription;

Operations:

Step to the delivery of the next subscription;

Step to the delivery of a selected subscription;

View: shows both the sender and the receiver of an announcement, together with all subscriptions served as a result of that announcement;

Activation predicate: verifies whether the execution stack contains an execution context initiating the delivery of an announcement.

6.4.3 A Debugger for PetitParser

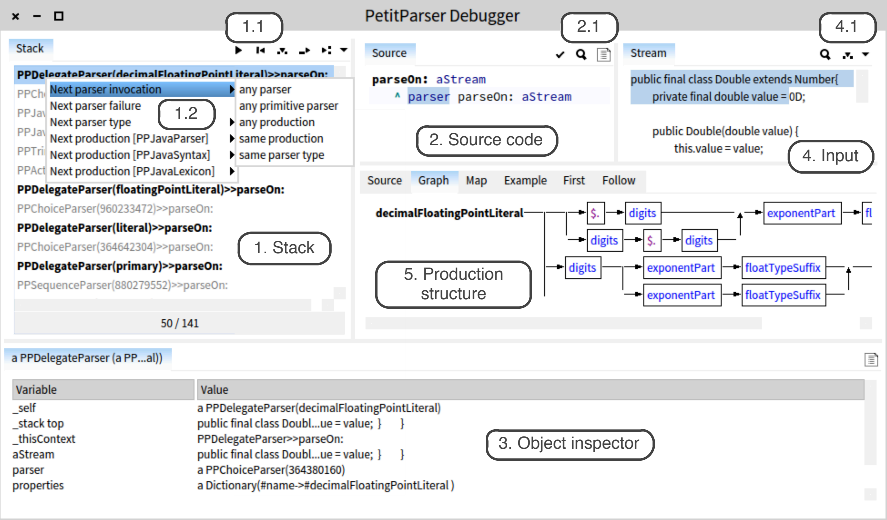
PetitParser is a framework for creating parsers, written in Pharo, introduced in Section 5.2.1 (p.80). Whereas most parser generators instantiate a parser by generating code, when PetitParser instantiates a parser the grammar productions are used to create a tree of primitive parsers (*e.g.*, choice, sequence, negation); this tree is then used to parse the input. Nevertheless, the same issues arise as with conventional parser generators: generic debuggers do not provide debugging operations at the level of the input (*e.g.*, set a breakpoint when a certain part of the input is parsed) and of the grammar (*e.g.*, set a breakpoint when a grammar production is exercised). In addition, generic debuggers do not display the source code of grammar productions nor do they provide easy access to the input being parsed. To overcome these issues, other tools for working with parser generators provide dedicated domain-specific debuggers. For example, ANTLR Studio, an IDE for the ANTLR[4] parser generator [Parr and Quong 1995] provides both breakpoints and views at the level of the grammar[5]. Rebernak *et al.* also give an example of a dedicated debugger for ANTLR [Rebernak et al. 2009].

In the case of PetitParser we have developed a domain-specific debugger by configuring the Moldable Debugger as follows:

Session: extracts the parser and the input being parsed from the runtime;

[4] `http://www.antlr.org/`

[5] `http://www.placidsystems.com/articles/article-debugging/usingdebugger.htm`



```
PPDelegateParser(decimalFloatingPointLiteral)>>parseOn:
PPCh
PPJav
PPJav
PPTri
PPAct
PPDelegateParser(floatingPointLiteral)>>parseOn:
PPChoiceParser(960233472)>>parseOn:
PPDelegateParser(literal)>>parseOn:
PPChoiceParser(364642304)>>parseOn:
PPDelegateParser(primary)>>parseOn:
PPSequenceParser(880279552)>>parseOn:
```

```
parseOn: aStream
    ^ parser parseOn: aStream
```

```
public final class Double extends Number{
    private final double value = 0D;

    public Double(double value) {
        this.value = value;
```

Variable	Value
_self	a PPDelegateParser(decimalFloatingPointLiteral)
_stack top	public final class Doubl...ue = value; } }
_thisContext	PPDelegateParser>>parseOn:
aStream	public final class Doubl...ue = value; } }
parser	a PPChoiceParser(364380160)
properties	a Dictionary(#name->#decimalFloatingPointLiteral)

Figure 6.4: A domain-specific debugger for PetitParser. The debugging view displays relevant information for debugging parsers ((4) Input, (5) Production structure). Each widget loads relevant debugging operations (1.1, 1.2, 2.1, 4.1).

Predicates:

Detect the usage of any type of parser: *method call*(`parseOn:`) predicates combined using *or* on all subclasses of `PPParser` that are not abstract and override the method `parseOn:`;

Detect the usage of any type of production: *method call*(`PPDelegateParser>> parseOn:`);

Detect the usage of a specific primitive parser: *method call*(`parseOn:`) predicates combined using *or* on all subclasses of `PPParser` that represent a primitive parser (*e.g.*, `PPRepeatingParser`);

Detect the usage of a specific production: *method call*(`PPDelegateParser>> parseOn:`) and *state check* verifying that the receiver object is a parser for the given grammar production;

Detect when a parser fails to match the input: *method call*(`PPFailure class>> message:context:`), or *method call*(`PPFailure class>>message:context:at:`);

Detect when the position of the input stream changes: *attribute write*(`#position` from `PPStream`) and *state check* verifying that the attribute value changed;

Detect when the position of the input stream reaches a given value: *attribute write*(`#position` from `PPStream`) and *state check* verifying that the attribute value is set to a given value;

Operations: Navigating through the execution at a higher level of abstraction is supported through the following debugging operations:

Next parser: step until a primitive parser of any type is reached;

Next production: step until a production is reached;

Primitive parser(aPrimitiveParserClass): step until a parser having the given class is reached;

Production(aProduction): step until the given production is reached;

Next failure: step until a parser fails to match the input;

Stream position change: step until the stream position changes (it either increases, if a character was parsed, or decreases if the parser backtracks);

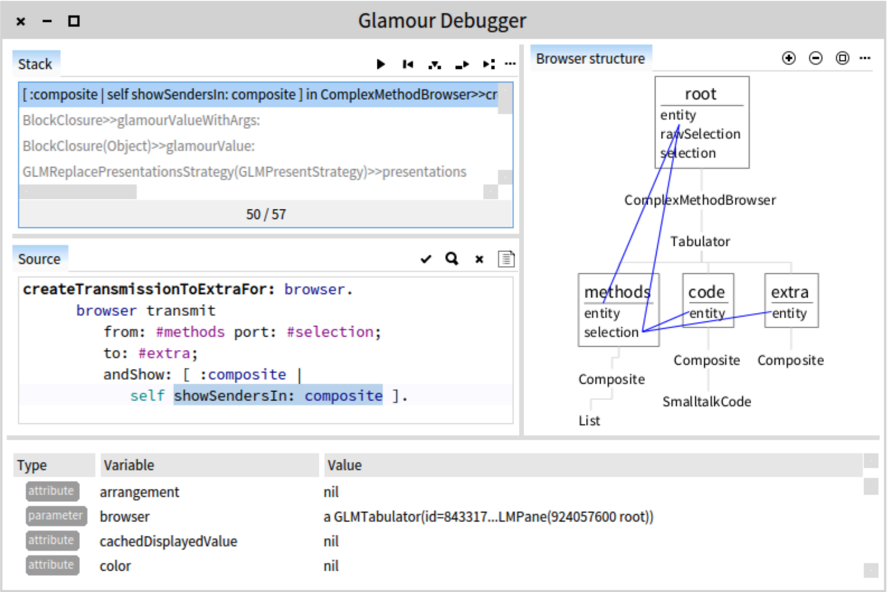
Stream position(anInteger): step until the stream reaches a given position;

View: The debugging view of the resulting debugger is shown in Figure 6.4 (p.124). We can see that now the input being parsed is incorporated into the user interface; to know how much parsing has advanced, the portion that has already been parsed is highlighted. Tabs are used to group six widgets showing different types of data about the current production, like: source code, structure, position in the whole graph of parsers, an example that can be parsed with the production, *etc.* The structure of the parser (*e.g.*, the *Graph* view in Figure 6.4 (p.124)), for example, is generated from the object graph of a parser and can allow developers to navigate a production by clicking on it. The execution stack further highlights those execution contexts that represent a grammar production;

Activation predicate: verifies whether the execution stack contains an execution context created when using a parser.

6.4.4 A Debugger for Glamour

Glamour is an engine for scripting browsers based on a components and connectors architecture [Bunge 2009]. New browsers are created by using an internal domain-specific language (DSL) to specify a set of *presentations* (graphical widgets) along with a set of *transmissions* between those presentations, encoding the information flow. Users can attach various conditions to transmissions and alter the information that they propagate. Presentations and transmissions form a model that is then used to generate the actual browser.



Glamour Debugger

Stack

[:composite | self showSendersIn: composite] in ComplexMethodBrowser>>cr
BlockClosure>>glamourValueWithArgs:
BlockClosure(Object)>>glamourValue:
GLMReplacePresentationsStrategy(GLMPresentStrategy)>>presentations

50 / 57

Source

```
createTransmissionToExtraFor: browser.
      browser transmit
         from: #methods port: #selection;
         to: #extra;
         andShow: [ :composite |
            self showSendersIn: composite ].
```

Type	Variable	Value
attribute	arrangement	nil
parameter	browser	a GLMTabulator(id=843317...LMPane(924057600 root))
attribute	cachedDisplayedValue	nil
attribute	color	nil

Figure 6.5: A domain-specific debugger for Glamour showing the model of the browser currently constructed.

The Moldable Debugger relies on Glamour for creating domain-specific views. Thus, during the development of the framework we created a domain-specific debugger to help us understand the creation of a browser:

Session: extracts from the runtime the model of the browser;

Predicates:

Detect the creation of a presentation:
message send(`glamourValue:` in `GLMPresentStrategy>>presentations`);

Detect when a transmission alters the value that it propagates:
message send(`glamourValue:` in `GLMTransmission>>value`);

Detect when the condition of a transmission is checked:
message send(`glamourValue:` in `GLMTransmission>>meetsCondition`);

Operations:

Step to presentation creation;

Step to transmission transformation;

Step to transmission condition;

View: displays the structure of the model in an interactive visualization that is updated as the construction of the model advances (Figure 6.5 (p.126));

Activation predicate: verifies whether the execution stack contains an execution context that triggers the construction of a browser.

6.4.5 Profiler Framework

Spy is a framework for building custom profilers [Bergel et al. 2011]. Profiling information is obtained by executing dedicated code before or after method executions. Code is inserted into a method by replacing the target method with a new method (*i.e.*, method wrapper [Brant et al. 1998]) that executes the code of the profiler before and after calling the target method. Hence, if a developer needs to debug profiled code, she has to manually skip over the code introduced by the profiler. To address this issue, we created a domain-specific debugger together with the developers of S2py [S2py accessed June 3, 2016], the second version of the Spy framework, that does not expose developers to profiler code when debugging methods are being profiled:

Session: no changes in the default debugging session are required;

Predicates:

Detect when Spy finished executing profiling code for a method
message send(`valueWithReceiver:arguments:` in `S2Method>>run:with:in:`)

Operations:

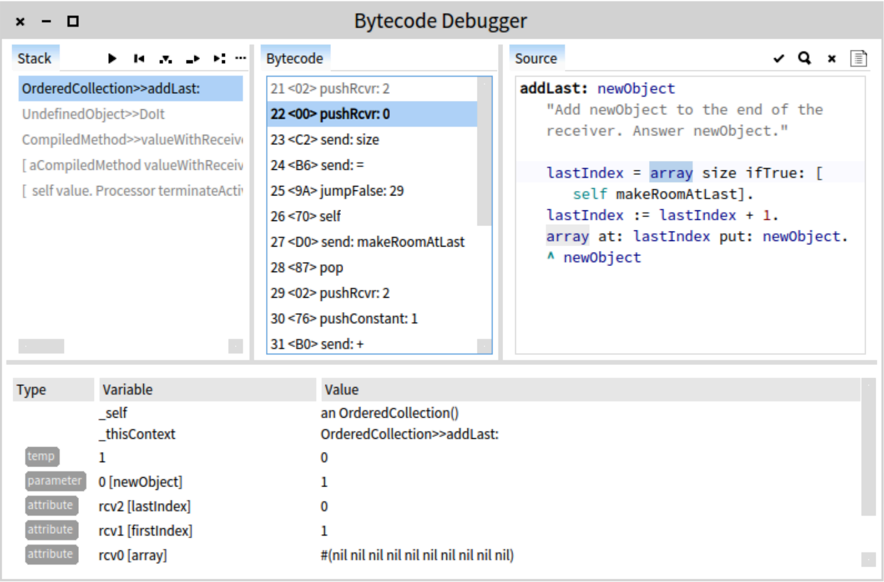
Step into method call ignoring profiled code: debugging action that when stepping into a method containing profiling code, automatically steps over the profiling code and into the code of the original method.

View: has the same widgets as the user interface of the default debugger. However, the stack widget removes all stack frames internal to the Spy framework.

Activation predicate: verifies whether the execution stack contains an execution context that starts a Spy profiler.

6.4.6 Stepping Through Bytecodes

While normally programs are debugged at the source level, building tools like compilers and interpreters requires developers to understand the execution of a program at the bytecode level. However, all debugging actions presented in this section skip over multiple bytecode instructions. For example, *step into message send*, a debugging action present in most debuggers, skips over the bytecode instructions that push method parameters onto the stack. The ability to debug at the bytecode level is especially important when generating or manipulating bytecode directly with bytecode



OrderedCollection>>addLast:
UndefinedObject>>DoIt
CompiledMethod>>valueWithReceiv
[aCompiledMethod valueWithReceiv
[self value. Processor terminateActi

```
21 <02> pushRcvr: 2
22 <00> pushRcvr: 0
23 <C2> send: size
24 <B6> send: =
25 <9A> jumpFalse: 29
26 <70> self
27 <D0> send: makeRoomAtLast
28 <87> pop
29 <02> pushRcvr: 2
30 <76> pushConstant: 1
31 <B0> send: +
```

```
addLast: newObject
	"Add newObject to the end of the
	receiver. Answer newObject."

	lastIndex = array size ifTrue: [
		self makeRoomAtLast].
	lastIndex := lastIndex + 1.
	array at: lastIndex put: newObject.
	^ newObject
```

Type	Variable	Value
	_self	an OrderedCollection()
	_thisContext	OrderedCollection>>addLast:
temp	1	0
parameter	0 [newObject]	1
attribute	rcv2 [lastIndex]	0
attribute	rcv1 [firstIndex]	1
attribute	rcv0 [array]	#(nil nil nil nil nil nil nil nil nil nil)

Figure 6.6: A domain-specific debugger for stepping through the execution of a program at the bytecode level.

transformation tools. In many cases the resulting bytecode cannot be de-compiled to a textual representation. To address this, we developed a debugger for stepping through the execution of a program one bytecode instruction at a time:

Session: customizes the default debugging session to not step over multiple bytecode instructions when performing various initializations;

Predicates: no predicates are required

Operations:

Step over one bytecode instruction;

Step into a bytecode instruction representing a message send;

View: shows the method that is currently executed both as a list of bytecodes and a textual representation; embeds an object inspector that shows the internal stack of the current execution context (Figure 6.6 (p.128));

Activation predicate: uses an activation predicate that always returns `true`.

6.4.7 Summary

PetitParser, Glamour, SUnit, Spy, Announcements framework and bytecode interpretation cover six distinct domains. For each one we instantiated a domain-specific debugger having a contextual debugging view and/or a set of debugging operations capturing abstractions from that domain. This shows the Moldable Debugger framework addresses the first two requirements, domain-specific user interfaces and domain-specific debugging operations.

The two remaining requirements, automatic discovery and dynamic switching, are also addressed. At each point during debugging developers can obtain a list of all domain-specific debuggers applicable to their current context. This does not require them either to know in advance all available debuggers, or to know when those debuggers are applicable. Once the right debugger was found developers can switch to it and continue debugging without having to restart the application. For example, one can perform the scenario presented in Section 6.2.4 (p.116). The cost of creating these debuggers is discussed in Section 6.6.1 (p.134).

6.5 Implementation Aspects

While the Moldable Debugger model can capture a wide range of debugging problems, the actual mechanism for detecting run-time events has a significant impact on the performance and usability of a debugger. In this section we present three approaches for detecting run-time events in the execution of a debugged program based on debugging predicates:

1. step-by-step execution;
2. code-centric instrumentation;
3. object-centric instrumentation.

This section discusses these approaches by looking at their usability. We further performed a detailed investigation into the performance penalty incurred by each of these mechanisms. This investigation is presented in Appendix B.

6.5.1 Controlling the Execution

When an event is detected, a breakpoint is triggered, stopping the execution of the debugged program. The breakpoint notifies the active debugging action (*i.e.*, the action that installed the predicate). The debugging action can then perform an operation, resume the execution or wait for a user action.

We use the following debugging actions from the PetitParser debugger as examples in this section:

- *Production(aProduction)*: step until the given grammar production is reached;
- *Stream position(anInteger)*: step until parsing reaches a given position in the input stream.

Step-by-step execution

Approach Interpret the debugged program one bytecode instruction at a time (*i.e.*, step-by-step execution) and check, after each bytecode instruction, if a debugging predicate matches the current execution context (*i.e.*, stack frame). This approach matches the *while-step* construct proposed by *Crawford et al.* [Crawford et al. 1995].

Implementation Each debugging predicate is transformed to a boolean condition that is applied to the current execution context (*i.e.*, stack frame).

Examples

- *Production(aProduction)*: *(i)* check if the current bytecode instruction is the initial instruction of a method; *(ii)* check if the currently executing method is `PPDelegateParser>>parseOn:`; *(iii)* check if the receiver `PPDelegateParser` object is a parser for the given grammar production;
- *Stream position(anInteger)*: *(i)* check if the current bytecode instruction pushes a value into an object attribute; *(ii)* check if the attribute is named `#stream` and it belongs to an instance of `PPStream`; *(iii)* check if the value that will be pushed into the attribute is equal to the given value.

Code-centric instrumentation

Approach Use basic debugging predicates (*i.e.*, *attribute access*, *attribute write*, *method call* and *message send*) to insert instrumentations into the code of the debugged program. State check predicates can then ensure that a breakpoint is triggered only if further conditions hold when the instrumentation is reached (*e.g.*, the target object is equal to a given one). This approach resembles dynamic aspects [Bonér 2004] and conditional breakpoints.

Implementation We rely on two different mechanisms for handling predicates for attributes and methods.

We implement *attribute access* and *attribute write* predicates using slots [Verwaest et al. 2011]. Slots model instance variables as first-class objects that can generate the code for reading and writting instance variables. We rely on a custom slot that can wrap any existing slot and insert code for triggering a breakpoint before the attribute is read or written in all methods of a given class.

We implement *method call* and *message send* by adding meta-links to AST nodes [Denker et al. 2007]: when compiling an AST node to bytecode, if that AST node has an attached meta-link, that meta-link can generate code to be executed before, after or instead of the code represented by the AST node. We rely on a custom meta-link that inserts code for triggering a breakpoint before the execution of an AST node. We then implement these types of predicates as follows:

- *message send*: locate in the body of a method all AST nodes that represent a call to the target method; add the custom meta-link only to these AST nodes;
- *method call*: add the custom meta-link on the root AST node of the target method.

A different strategy for implementing code-centric instrumentations consists in injecting the debugged concern directly into an existing bytecode version of the code using a bytecode engineering library [Tanter et al. 2002]. The meta-links used have similar properties: code instrumentation happens at runtime and the original code remains unchanged. Direct bytecode manipulation would give a more fined-grained control on the position where and how code inserted into the debugged code. This flexibility is not needed for our debugger and it would come with the cost of having to deal with the complexity of bytecode.

Examples

- *Production(aProduction)*: instrument the root AST node of `PPDelegateParser>> parseOn:` to check if the receiver object is a parser for the given grammar production;
- *Stream position(anInteger)*: add instrumentations before all write instructions of the attribute `#stream` from class `PPStream` to check if the new value of the attribute is equal with a given value.

Object-centric instrumentation

Approach Combine a basic predicate (*i.e., attribute access*, *attribute write*, *method call* or *message send*) with a *state check* predicate verifying the identity of the receiver object against a given object (*i.e., identity check* predicate). Then insert into the debugged program instrumentations only visible to the given object. Thus, a breakpoint is triggered only when the event captured by the basic predicate has taken place on the given instance. This approach matches the object-centric debugging approach [Ressia et al. 2012a], where debugging actions are specified at the object level (*e.g.*, stop execution when the specified object receives a particular message).

Implementation We insert an instrumentation visible to only a single target object as follows:

- create an anonymous subclass of the target object's class; the anonymous subclass is created dynamically, at debug time, when the underlying dynamic action is executed;
- apply code-centric instrumentation to insert the basic debugging event into the anonymous class; code-centric instrumentation is inserted through code generation and recompilation of the anonymous class at debug time;
- change the class of the target object to the new anonymous class.

As access to a run-time object is necessary, this approach can only be used once a debugger is present; it cannot be used to open the initial debugger.

Examples

- *Production(aProduction)*: *(i)* locate the `PPDelegateParser` object that represents the given grammar production; *(ii)* replace the class of that object with an anonymous one where the method `parseOn:` has a *method call* predicate inserted using a code-centric instrumentation (*i.e.*, meta-link);
- *Stream position(anInteger)*: *(i)* locate the `PPStream` object holding the input that is being parsed; *(ii)* replace the class of that object with an anonymous one where the attribute `#position:` has an *attribute access* predicate inserted using a code-centric instrumentation (*i.e.*, slot).

6.5.2 Usability

In this section we discuss usability aspects for each of the three aforementioned mechanisms for detecting run-time events.

Step-by-step execution The main advantage of this approach is that it is simple to understand and implement, and it does not alter the source of the debugged program. However, it can slow down the debugged program considerably to the point where it is no longer usable. Despite this shortcoming it can be useful for debugging actions that need to skip over a small number of bytecode instructions. For example, we use this approach to implement the action *Step to next subscription* in the Announcements debugger: we only need to skip over several hundred bytecode instructions internal to the Announcements framework.

Code-centric instrumentation This approach has a much lower performance overhead than step-by-step execution that makes it practically applicable in most situations. While the current overhead is low, it can still prove inconvenient when using complex predicates or when stepping over code that already takes a significant amount of time. For example, we do not use this solution in the Announcements debugger as the Announcements framework is heavily used by the Pharo IDE, and any overhead will apply to the entire IDE.

	Step-by-step execution	Code-centric instrumentations	Object-centric instrumentations
Announcements	✓	✓	✓
Petit Parser	✓	✓	✓
Glamour	✓	✓	
Spy	✓		
Bytecode	✓		

Table 6.2: Feasible approaches for implementing debugging actions for the example debuggers from Section 6.4 (p.120).

Object-centric instrumentation While it imposes no performance overhead, this approach does not work for code that depends on the actual class of the instrumented object. It further requires access to an object beforehand, which is not always possible. We use this solution in the Announcements and PetitParser debuggers; however, in both cases we only instrument objects internal to these frameworks.

Discussion Even if not practically applicable in most situations we used debugging actions based on step-by-step execution to implement the initial version of all our domain-specific debuggers. This allowed us to quickly prototype and refine the interface and the functionality of those debuggers. Later on, whenever performance became a problem we moved to actions based on code-centric instrumentation. We then only changed these actions to object-centric instrumentation in very specific situations where we could take advantage of particular aspects of a framework (*e.g.*, PetitParser uses a single `PPContext` object that is passed to all the parse methods; the Announcements framework creates an internal subscription object each time a subscriber registers with an announcer).

Depending on the particular aspects of a domain, not all three approaches are applicable. Table 6.2 (p.133) indicates which approaches can be used for the example debuggers from Section 6.4 (p.120) (Glamour is a prototype-based framework that relies on copying objects; Spy already instruments the debugged code).

Note that the performance penalty is present only when using the custom debugger, and not when using the regular one.

6.5.3 The Moldable Debugger in Other Languages

The current prototype of the Moldable Debugger is implemented in Pharo. Pharo made it easy to integrate the Moldable Debugger due to its powerful introspection support. For example, the entire run-time stack can be reified on demand and the class of an object can be changed dynamically at run time. Pharo further incorporates support for slots and behavior reflection through AST annotations. Nevertheless, the Moldable Debugger can be ported to other languages as long as:

	Session	Operations	View	Total
Base model	800	700	-	1500
Default Debugger	-	100	400	500
Announcements	200	50	200	450
Petit Parser	100	300	200	600
Glamour	150	100	50	300
SUnit	100	-	50	150
Spy	-	30	30	60
Bytecode	20	50	130	200

Table 6.3: Size of extensions in lines of code (LOC).

- they provide a debugging infrastructure that supports custom extensions/ plugins for controlling the execution of a target program;
- there exists a way to rapidly construct user interfaces for debuggers, either through domain-specific languages or UI builders.

For example, one could implement the framework in Java. Domain-specific debugging operations can be implemented on top of the Java Debugging Interface (JDI) or by using aspects. JDI is a good candidate as it provides explicit control over the execution of a virtual machine and introspective access to its state. Aspect-Oriented Programming [Kiczales et al. 1997] can implement debugging actions by instrumenting only the code locations of interest. Dynamic aspects (*e.g.*, AspectWerkz [Bonér 2004]) can further scope code instrumentation at the debugger level. Last but not least, domain-specific views can be obtained by leveraging the functionality of IDEs, like perspectives in the Eclipse IDE.

6.6 Discussion

6.6.1 The Cost of Creating New Debuggers

The four presented domain-specific debuggers were created starting from a model consisting of 1500 lines of code. Table 6.3 (p.134) shows, for each debugger, how many lines of code were needed for the debugging view, the debugging actions, and the debugging session. Regarding the view column, custom debuggers extend and customize the view of the default debugger; hence, the view of the default debugger is twice the size of any other view, as it provides the common functionality needed in a debugging interface.

In general lines of code (LOC) must be considered with caution when measuring complexity and development effort (Section 2.1.3 (p.18)). Nevertheless, it gives a good indication of the small size of these domain-specific debuggers. This small size makes the construction cost affordable. Similar conclusions can be derived from the work of Kosar *et al.* that shows that with the right setup it is possible to construct a domain-specific debugger for a modeling language with relatively low costs [Kosar et al. 2014]. Hanson and Korn further show that a useful debugger for C can be written in under 2500 lines of code, one order of magnitude smaller than *gdb* [Hanson and Korn 1997].

Nevertheless, depending on the application domain and the actual debugger a developer wants to build, deeper knowledge about program execution may be needed. Hence, depending on the requirements, creating a domain-specific debugger is not an activity suitable for developers lacking this kind of knowledge. This differs from from the Moldable Inspector and Moldable Spotter where less knowledge about how the tool works are required to create an adaptation.

6.6.2 Applicability

Section 6.4 (p.120) shows that the Moldable Debugger can cover a wide range of application domains. While Section 6.4 (p.120) gives particular examples, we consider the Moldable Debugger to be applicable for most types of application domains that build upon an object-oriented model. For example, one could apply the proposed solution to AmbientTalk [Cutsem et al. 2014], an actor-based distributed programming language, by extending the Moldable Debugger with support for actor-based concurrency.

The applicability of the Moldable Debugger, nevertheless, has its limits. An edge case is Monaco, a domain-specific language for reactive systems with imperative programming notation [Prähofer et al. 2013]. While Monaco has a model based on hierarchical components that could be accommodated by the Moldable Debugger, the main goal of Monaco is to develop programs for control systems. As running and debugging programs on live control systems is not a feasible option, simulators, rather then debuggers, provide better support for reasoning about theses types of program. A case where the Moldable Debugger would not be applicable is SymGridPar2, a language for parallel symbolic computation on a large number of cores [Maier et al. 2014]. On the one hand, SymGridPar2 features a functional programming style. On the other hand, it is designed for programs that will run in parallel on tens of thousands of cores. The run-time overhead added by a debugger can significantly influence the behavior of the code. Logging frameworks provide better alternatives as they allow developers to collect information at run time with a very low overhead and analyze it postmortem with more costly analyses.

6.7 Related Work

There exists a wide body of research addressing debugging from various perspectives. In this section we give an overview of several perspectives related to the Moldable Debugger model.

6.7.1 Software Logging

While debuggers support a direct interaction with the run-time state of an application, logging frameworks record dynamic information about the execution of a program. One challenge with logging frameworks, related to the current work, is how to capture the right information about the run-time [Oliner et al. 2012]. Given the large diversity of data that could be relevant about a running application, like the Moldable Debugger, many frameworks for logging run-time information allow developers to customize the logging infrastructure to fit their needs [Gülcü and Stark 2003; Erlingsson et al. 2011; Ressia et al. 2012b]. MetaSpy, for example, makes it possible to easily create domain specific-profilers [Ressia et al. 2012b]. In the context of wireless sensor networks, Cao *et al.* propose declarative tracepoints, a debugging system that allows users to insert action-associated checkpoints using a SQL-like language. Like activation predicates it allows users to detect and log run-time events using *condition predicates* over the state of the program [Cao et al. 2008].

6.7.2 Specifying Domain-specific Operations

There is a wide body of research in programmable and scriptable debugging allowing developers to automate debugging tasks by creating high-level abstractions. *MzTake* [Marceau et al. 2007] is a scriptable debugger enabling developers to automate debugging tasks, inspired by *functional reactive programming*. MzTake treats a running program as a stream of run-time events that can be analyzed using operators like *map* and *filter*; streams can be combined to form new streams. For example, one can create a stream in which a new value is added every time a selected method is called from the debugged program. Selecting only method calls performed on objects that are in a certain state is achieved using the *filter* operator; this operator creates a new stream that contains only the method call events that matched the filter's condition. Unlike MzTake we propose an approach for detecting run-time events based on object-oriented constructs: run-time events are specified by combining predicate objects, instead of combining streams. A debugging action can then use a predicate to detect a run-time event (*e.g.*, method call on a given object) and put the event in a stream.

Coca [Ducassé 1999] is an automated debugger for C using Prolog predicates to search for events of interest over program state. Events capture various language constructs (*e.g.*, function, return, break, continue, goto) and are modeled as C structures; a sequence of events is grouped in a trace. Developers write, using Prolog, queries that search for an event matching a pattern in a trace. To perform a query a developer has to provide an *event pattern* and call a primitive operation for performing the actual search. An event pattern consists of any combination of 3-tuples of the form *'<attributename> <operation> <attribute-value>'* connected with *and*, *or* or *not*. In our approach we express high-level run-time events by combining objects (*i.e.*, debugging predicates) instead of combining tuples. We further use predicates as a specification of run-time events and employ different implementations to detect those events at run-time.

Dalek [Olsson et al. 1991] is a C debugger employing a dataflow approach for debugging sequential programs: developers create high-level events by combining primitive events and other high-level events. Developers enter breakpoints into the debugged code to generate primitive events. A high-level event is created by specifying, in the definition of that event, what events activate that event; when an event is triggered all other events that depend on it are also triggered. High-level events can maintain state and execute code when triggered (*e.g.*, print, test invariants). Thus, high-level events map to debugging actions in our approach. However, we do not require developers to explicitly trigger primitive events from the debugged code; developers provide a specification of the run-time event using debugging predicates, outside of the debugged program's code.

Auguston *et al.* present a framework that uses declarative specifications of debugging actions over event traces to monitor the execution of a program [Auguston et al. 2002]. Several types of run-time events, corresponding to various actions encountered in the debugged program, are generated directly by the virtual machine. Events are grouped together in traces that conform to an event grammar, defining the valid traces of events. An execution monitor loads a target program, executes it, obtains a trace of run-time events from the program and performs various computations over the event trace (*e.g.*, check invariants, profile). We do not have an explicit concept of monitor in our approach and do not directly provide operations for manipulating event traces. Our model only associates run-time events (predicates) with various operations. Event traces can be implemented on top of this model, by having debugging actions that store and manipulate events.

Expositor [Khoo et al. 2013] is a scriptable time-travel debugger that can check temporal properties of an execution: it views program traces as immutable lists of time-annotated program state snapshots and uses an efficient data structure to manage them. *Acid* [Winterbottom 1994] makes it possible to write debugging operations, like breakpoints and step instructions, in a language designed for debugging that reifies program state as variables.

The aforementioned approaches focus on improving debugging by enabling developers to create commands, breakpoints or queries at a higher level of abstraction. Nevertheless, while they encapsulate high-level abstractions into scripts, programs or files, developers have to manually find proper high-level abstractions for a given debugging context. We propose an approach for automatically detecting relevant high-level abstractions (*e.g.*, debugging actions) based on the current debugging context. Furthermore, only some of the aforementioned approaches incorporate at least ad hoc possibilities of visually exploring data by using features from the host platform. We propose a domain-specific view for each domain-specific debugger that displays and groups relevant widgets for the current debugging context.

Object-centric debugging [Ressia et al. 2012a] proposes a new way to perform debugging operations by focusing on objects instead of the execution stack; while it increases the level of abstraction of debugging actions to object-oriented idioms, the approach does not enable developers to create and combine debugging actions to capture domain concepts instead of just object-oriented idioms. *Reverse watchpoints* use the concept of *position* to automatically find the last time a target variable was written and move control flow to that point [Maruyama and Terada 2003]. *Whyline* is a debugging tool that allows developer to ask and answer *Why* and *Why Not* questions about program behavior [Ko and Myers 2008]. *Query-based debugging* facilitates the creation of queries over program execution and state using high-level languages [Lencevicius et al. 1997; Potanin et al. 2004; Martin et al. 2005; Ducasse et al. 2006]. *Duel* [Golan and Hanson 1993] is a high-level language on top of GDB for writing state exploration queries. These approaches are complementary to our approach as they can be used to create other types of debugging operations.

Omniscient debugging provides a way to navigate backwards in time within a program's execution trace [Pothier et al. 2007]. Bousse *et al.* introduce an omniscient debugger targeting an executable Domain-Specific Modeling Language (xDSML) [Bousse et al. 2015] that, while still partially generic, can generate domain-specific traces tuned to the actual xDSML. This debugger is partially generic as it treats all xDSMLs in a uniform way. Like our approach this debugger is based on an object-oriented model and aims to improve the debugging process by presenting domain-specific information to the user. The Moldable Debugger, however, is not omniscient but allows developers to fine-tune the debugger to the particular aspects of a domain-specific language or application.

Lee *et al.* propose a debugger model for composing portable mixed-environment debuggers [Lee et al. 2009]. Their current implementation, *Blink*, is a full-featured debugger for both Java and C. While the Moldable Debugger model does not depend on a particular object-oriented language, we do not provide an approach for cross-language debugging (*e.g.*, between an object-oriented and a non object-oriented language).

6.7.3 User Interfaces for Debugging

Another category of approaches looks at how to improve the user interface of debuggers instead of their actions. *Debugger Canvas* [DeLine et al. 2012] proposes a novel type of user interface for debuggers based on the *Code Bubbles* [Bragdon et al. 2010b] paradigm. Rather than starting from a user interface having a predefined structure, developers start from an empty one, on which they add bubbles as they step through the execution of the program. Our approach requires developers to create custom user interfaces (views) beforehand. *The Data Display Debugger (DDD)* [Zeller and Lütkehaus 1996] is a graphical user interface for GDB providing a graphical display for representing complex data structures as graphs that can be explored incrementally and interactively. *jGRASP* supports the visualization of various data structure by means of dynamic viewers and a *structure identifier* that automatically select suitable views for data structures [Cross et al. 2009]. *x*DIVA is a 3-D debugging visualization system where complex visualization metaphors are assembled from individual ones, each of which is independently replaceable [Cheng et al. 2008].

Each of these approaches introduces different improvements in the user interface of a debugger. To take advantage of this, our approach does not hardcode the user interface of the debugger: each domain-specific debugger can have a dedicated user interface. Given that domain-specific debuggers are switchable at run time, when multiple debuggers are applicable a developer can select the one whose user interface she finds appropriate. By focusing only on the user interface, these approaches do not provide support for adding custom debugging operations. Our approach addresses both aspects.

6.7.4 Unifying Approaches

deet [Hanson and Korn 1997] is a debugger for ANSI C written in *tksh*[6] that, like our approach, promotes simple debuggers having few lines of code, and allows developers to extend the user interface and add new commands by writing code in a high-level language (*i.e., tksh*). Commands are directly embedded in the user interface. Our approach decouples debugging actions from user-interface components (*i.e.,* widgets): each widget dynamically loads at run time debugging actions that have a predefined annotation. If in *deet* run-time events are detected by attaching *tksh* code to conditional breakpoints, we provide a finer model based on combining debugging predicates. Last but not least, we propose modeling the customization of a debugger (*i.e.,* debugging actions and user interface) through explicit domain-specific extensions and provide support for automatically detecting appropriate extensions at run time. In *deet*, developers have to manually select appropriate debuggers.

TIDE is a debugging framework focusing on the instantiation of debuggers for formal languages (ASF+ SDF, in particular) [van den Brand et al. 2005]; developers can implement high-level debugging actions like breakpoints and watchpoints, extend the user interface by modifying the Java implementation of TIDE, and use *debugging rules* to state which debugging actions are available at which logical breakpoints. *LISA* is a grammar-based compiler generator that can automatically generate debuggers,

[6] An extension of Korn shell including the graphical support for Tcl/Tk.

inspectors and visualizers for DSLs that have a formal language specification [Henriques et al. 2005]. Debuggers are obtained by constructing, from the grammar, transformations mapping code from the DSL to the generated GPL code. Wu *et al.* present a grammar-driven technique for automatically generating debuggers for DSLs implemented using source-to-source translation (a line of code from a DSL is translated into multiple consecutive lines of GPL code); this makes it possible to reuse an existing GPL debugger [Wu et al. 2008]. Other language workbenches [Faith et al. 1997; Lindeman et al. 2011; Pavletic et al. 2015] for developing DSLs or language extensions follow similar ideas: they enable the creator of a DSL or language extension to provide extra specifications during the development of the DSL or language extension that are then used to generate a specialized debugger. Our approach targets object-oriented applications where a formal specification is missing and not programs written in domain-specific languages that have a grammar or another formal specification. Furthermore, if domain concepts are built on top of a DSL, then DSL debuggers suffer from the same limitations as generic debuggers. Our approach directly supports debuggers aware of application domains.

6.7.5 Debugging in Domain-specific Modeling

Domain-specific modeling (DSM) enables domain-experts to directly work with familiar concepts instead of manually mapping concepts from the problem domain to the solution domain. Debuggers that work at a lower level of abstraction than that of the model limit the ability of a domain-expert to properly debug a model. To address this, the goal of domain-specific modeling is to automatically generate debuggers from a meta-model. While most meta-modeling tools do not automatically generate debuggers, several approaches target this goal. Mannadiar and Vangheluwe propose a conceptual mapping between debugging concepts from the programming languages (*e.g.*, breakpoints, assertions) and concepts from domain-specific modeling languages that use rule-based approaches for model transformations [Mannadiar and Vangheluwe 2011]. Kosar *et al.* discuss debugging facilities for a modeling environment for measurement systems [Kosar et al. 2014]. Kolomvatsos *et al.* present a debugger architecture for a domain-specific language used to model autonomous mobile nodes [Kolomvatsos et al. 2012]. These approaches take advantage of, and integrate with meta-modeling tools. The approach proposed in this paper is for object-oriented applications where the model consists of objects and relations between them. This model is created by application developers using idioms provided directly by object-oriented programming without the use of a meta-modeling tool.

6.8 Conclusions

Developers encounter domain-specific questions. Traditional debuggers relying on generic mechanisms, while universally applicable, are less suitable to handle domain-specific questions. To address this contradiction we identified four requirements that need to be supported by debuggers. We then proposed the Moldable Debugger, a debugging framework following those requirements that allows developers to create, with little effort, domain-specific debuggers that enable custom debugging actions through custom user interfaces. We designed the Moldable Debugger by applying the designed principles for moldable tools introduced in Section 3.1 (p.33) in the following way:

DP 1. Identify common types of tool-specific adaptations: The Moldable Debugger enables domain-specific adaptations by allowing developers to customize the user interface and provide debugging actions like domain-specific breakpoints. Unlike the previous two tools, the Moldable Debugger does not force all custom adaptations to conform to the same user interface. This is because debuggers can vary significantly between domains.

DP 2. Simplify the creation of common adaptations: To support inexpensive creation of custom debuggers, the Moldable Debugger provides a model for the user interface and the run-time that developers can reuse in their debuggers. As a validation, we implemented the Moldable Debugger model and created six different debuggers in fewer than 600 lines of code each. In this case the cost of an adaptation is greater than in the previous two tools as the scope of an adaptation is larger (*i.e.*, the entire debugger).

DP 3. Do not limit the types of possible adaptations only to common ones: Developers can reuse the provided model as well as the three provided strategies for controlling the execution. Nevertheless, developers can create new user interfaces from scratch or provide their own instrumentation strategies.

DP 4. Attach activation predicates to adaptations: Each domain-specific debugger can have an activation predicate indicating in what debugging contexts it is applicable. When we talk about debugging context we refer to the run-time stack of the process being debugged. Unlike the previous two tools, currently the Moldable Debugger does not use interaction data to select adaptations.

DP 5. Update adaptations based on a development context: Every time the developer performs a debugging action, the Moldable Debugger verifies if the current domain-specific debugger is still applicable and if there exist any other applicable domain-specific debuggers. If the current debugger is no longer applicable, developers need to explicitly switch to another one.

Given the large costs associated with debugging activities, improving the workflow and reducing the cognitive load of debugging can have a significant practical impact. Through the Moldable Debugger we showed that developers can create their own debuggers to address recurring problems, and thus, reduce the abstraction gap between the debugging needs and debugging support leading to a more efficient and less error-prone debugging effort.

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. It demands the same skill, devotion, insight, and even inspiration as the discovery of the simple physical laws which underlie the complex phenomena of nature.

C.A.R. Hoare

7

Moldable Development

The previous chapters introduced the moldable tools approach and discussed the design and implementation of three moldable tools. The value of these tools, and hence of the moldable tools approach, comes from developers extending moldable tools to address their own contextual problems. 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 by reading and exploring source code or using external tools. This vision, however, requires that developers see an advantage in extending moldable tools. In this chapter we explore this aspect by looking at the evolution of extensions for moldable tools in Pharo. We then exemplify how, by continuously adapting their development tools, developers can create custom environments to support them in their work. We conclude by giving guidelines, based on our experience, for improving the creation and adoption of moldable tools.

7.1 The Evolution of Domain-specific Extensions

We integrated the Moldable Inspector in the alpha version of Pharo 4. The previous inspector from Pharo 3 provided a basic extension possibility, and the Pharo image shipped with 8 such extensions. The Pharo 4 release, however, shipped with 138 extensions for the Moldable Inspector, built together with the development team of Pharo. One year later, the Pharo 5 release contained 165 extensions for the Moldable Inspector. This does not include extensions from external projects. For example, the Moose 6.0 project, which is based on Pharo 5, contains a total of 227 extensions. Similar observations can be made about the Moldable Spotter: it was integrated in

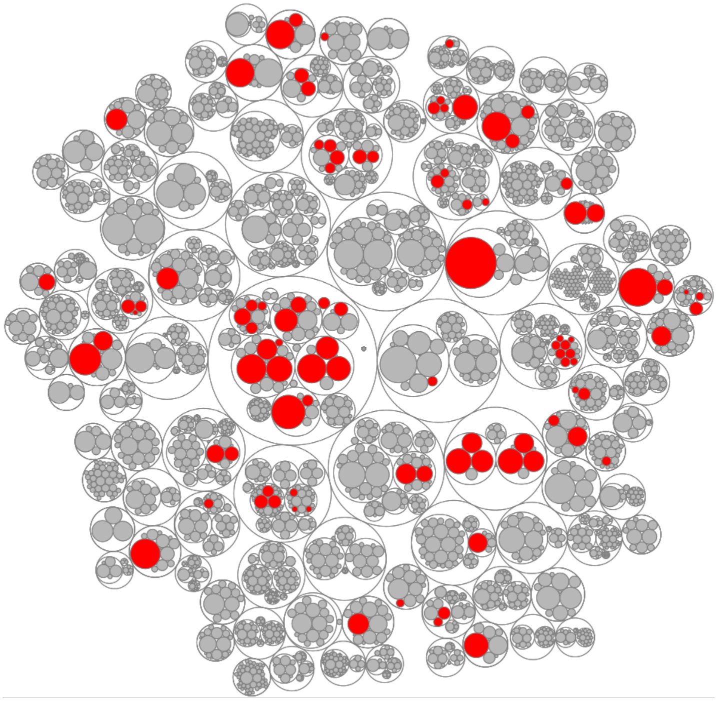


Figure 7.1: Spread of classes having at least an extension for Moldable Inspector or Moldable Spotter within packages from the Pharo 5 release. Classes are grouped per package; packages and classes are arranged using a circular tree layout. A class is highlighted in red if it has at least one domain-specific extensions.The size of a class is given by its number of methods.

the alpha version of Pharo 4, and the Pharo 4 release shipped with 92 extensions. In the Pharo 5 release there were 122 extensions.

Furthermore, out of a total of 298 packages present in Pharo 5 that are not test packages or packages holding Monticello configurations[1], 47 packages (15.8%) contained a class having an extension for Moldable Inspector or Moldable Spotter. To communicate the spread of extensions among packages, Figure 7.1 (p.144) shows the classes from the aforementioned packages grouped by package, highlighting in red those classes that have at least one extension and in gray classes having no extensions. This increase in the number and spread of extensions for both Moldable Inspector and Moldable Spotter indicates that there was an explicit need to have such extensions, and that these tools address this need. This provides one axis for validating the moldable tools approach.

[1] The convention in Pharo is to put each Monticello configuration in a separate package that only contains the configuration.

Regarding the Moldable Debugger, the Pharo 5 release integrates two out of the six domain-specific extensions discussed in Section 6.4 (p.120): the bytecode and the SUnit extensions. Currently, apart from these extensions, two other projects built using Pharo started to build domain-specific debuggers. Unlike extensions for Moldable Inspector and Moldable Spotter, the cost of a debugger is measured in hundreds of lines of code. Also the granularity of an extension is larger as it encompases the entire debugger. Hence, we did not expect to see as many debugger extensions as in the case of the previous two moldable tools.

7.2 Exemplifying Moldable Development

Instead of focusing on the spread of extensions through multiple domains, another axis for validating the moldable tools approach consists in focusing on applying it to specific application domains. This can show what custom environments developers can create when continuously adapting and evolving moldable tools to take their application domains into account . We refer to this consistent creation of extensions during the development and evolution of an application as *moldable development*. In this section we exemplify it using the Opal compiler and Petit Parser. We previously used these frameworks to show various features of the three moldable tools discussed in this dissertation. Now we bring these examples together; we only give an overview of the already presented features and discuss in detail aspects that were not already introduced.

7.2.1 Opal Compiler

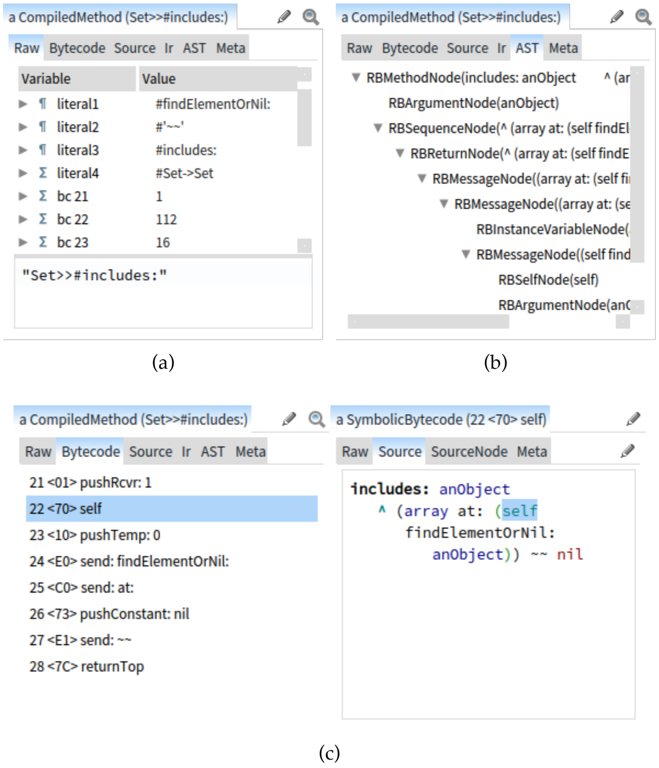
Opal[2] is a compiler infrastructure focusing on customizability that has been part of Pharo since the Pharo 3 release (May 2014)[3]. Initially Opal was developed using the standard development tools of Pharo. Developing a compiler is, however, a challenging activity due to the interplay between source code, ASTs, intermediate representations and bytecode. Section 4.5 (p.56) and Section 6.4.6 (p.127) explored these issues in details. To improve the development of Opal we extended the three moldable tools detailed in this dissertation together with the Opal team while Opal itself was being developed.

Moldable Inspector Extensions

To improve the inspection of compiled bytecode we gradually attached several custom views to `CompiledMethod` and `SymbolicBytecode` objects, detailed in Section 4.5 (p.56). These include views for displaying a human-friendly representation of the bytecode

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

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



(a)

a CompiledMethod (Set>>#includes:)

Raw Bytecode Source Ir AST Meta

Variable	Value
▶ ¶ literal1	#findElementOrNil:
▶ ¶ literal2	#'~~'
▶ ¶ literal3	#includes:
▶ Σ literal4	#Set->Set
▶ Σ bc 21	1
▶ Σ bc 22	112
▶ Σ bc 23	16

```
"Set>>#includes:"
```

(b)

a CompiledMethod (Set>>#includes:)

Raw Bytecode Source Ir AST Meta

```
▼ RBMethodNode(includes: anObject      ^ (ar
      RBArgumentNode(anObject)
    ▼ RBSequenceNode(^ (array at: (self findEl
        ▼ RBReturnNode(^ (array at: (self findE
            ▼ RBMessageNode((array at: (self fi
                ▼ RBMessageNode((array at: (se
                      RBInstanceVariableNode(
                    ▼ RBMessageNode((self find
                          RBSelfNode(self)
                          RBArgumentNode(an(
```

(c)

a CompiledMethod (Set>>#includes:)

Raw Bytecode Source Ir AST Meta

```
21 <01> pushRcvr: 1
22 <70> self
23 <10> pushTemp: 0
24 <E0> send: findElementOrNil:
25 <C0> send: at:
26 <73> pushConstant: nil
27 <E1> send: ~~
28 <7C> returnTop
```

a SymbolicBytecode (22 <70> self)

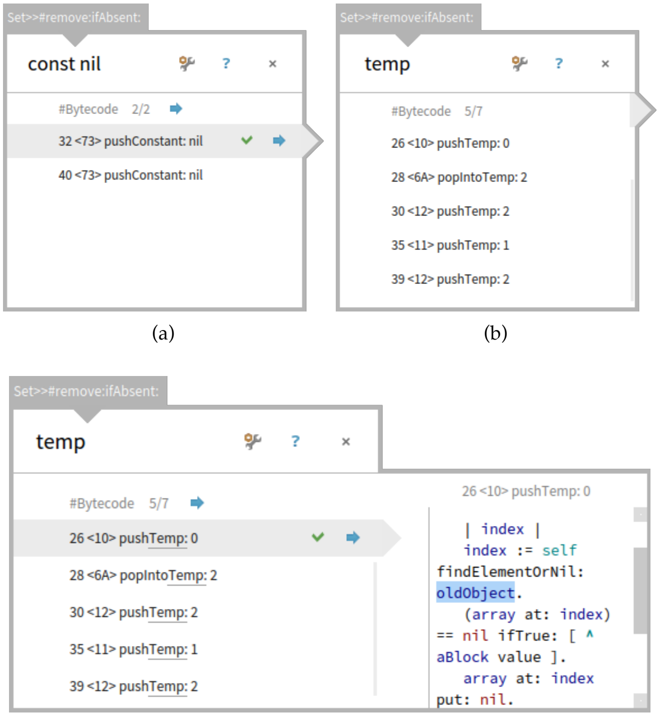
Raw Source SourceNode Meta

```
includes: anObject
    ^ (array at: (self
      findElementOrNil:
         anObject)) ~~ nil
```

Figure 7.2: Using the Moldable Inspector to visualize compiled code: *(a)* The *Raw* view shows the actual structure of the `CompiledMethod` 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.

(Figure 7.2c (p.146), left side), the source code of the method, the AST (Figure 7.2b (p.146)), and the IR. Figure 7.2 (p.146) summarizes these views. As an example of a workflow that becomes possible with these views consider Figure 7.2a (p.146) where 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. Using the bytecode view, the developer can see that the bytecode at index 22 corresponds to pushing `self`[4] onto the top of the stack.

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



Set>>#remove:ifAbsent:

const nil

```
#Bytecode 2/2
32 <73> pushConstant: nil
40 <73> pushConstant: nil
```

(a)

Set>>#remove:ifAbsent:

temp

```
#Bytecode 5/7
26 <10> pushTemp: 0
28 <6A> popIntoTemp: 2
30 <12> pushTemp: 2
35 <11> pushTemp: 1
39 <12> pushTemp: 2
```

(b)

Set>>#remove:ifAbsent:

temp

```
#Bytecode 5/7
26 <10> pushTemp: 0
28 <6A> popIntoTemp: 2
30 <12> pushTemp: 2
35 <11> pushTemp: 1
39 <12> pushTemp: 2
```

26 <10> pushTemp: 0

```
    | index |
    index := self
findElementOrNil:
oldObject.
    (array at: index)
== nil ifTrue: [ ^
aBlock value ].
    array at: index
put: nil.
```

(c)

Figure 7.3: Searching through bytecode using Moldable Spotter: *(a)* searching for instructions that access the value `nil`; *(b)* searching for instructions accessing temporary variables; *(c)* When selecting a bytecode the mapping with the source code is shown.

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`). To support these tasks we attached to `CompiledMethod` objects a custom search through a human-friendly representation of bytecode. This extension supports all the aforementioned searches as well as others, such as looking for when a constant is pushed to the stack (Figure 7.3a (p.147)) or finding all instructions that access temporary variables (local variables and method parameters; Figure 7.3b (p.147)). 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 7.3c (p.147)).

Moldable Debugger Extensions

One cannot easily use source-level debuggers to reason about bytecode. To address this we developed an extension to the Moldable Debugger that gives direct access to the bytecode and supports stepping through the execution of a program one bytecode instruction at a time. This extension is covered in Section 6.4.6 (p.127).

7.2.2 Petit Parser

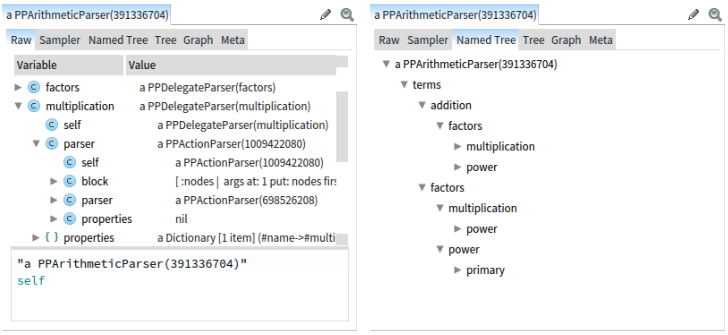
PetitParser is a framework for creating parsers presented in Section 5.2.1 (p.80) and Section 6.4.3 (p.123). PetitParser is meant to be used by developers other than just its creators to specify parsers. To ease the creation of parsers the PetitParser developers initially built a custom code editor that allowed the creators of a parser to 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.

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 7.4a (p.149), 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 7.4b (p.149) contains a view showing the structure of the grammar using a tree list.

Moldable Spotter Extensions

Parser classes can also contain other methods and attributes apart from those used to model grammar productions. Following the motivating example from Section 5.2.1 (p.80), 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 searches that work at the level of grammar productions to classes representing parsers (Section 5.4.2 (p.92)).


a PPArithmeticParser(391336704)
Raw Sampler Named Tree Tree Graph Meta
a PPArithmeticParser(391336704)
Raw Sampler Named Tree Tree Graph Meta

Variable	Value
▶ factors	a PPDelegateParser(factors)
▼ multiplication	a PPDelegateParser(multiplication)
self	a PPDelegateParser(multiplication)
▼ parser	a PPActionParser(1009422080)
self	a PPActionParser(1009422080)
▶ block	[:nodes | args at: 1 put: nodes firs
▶ parser	a PPActionParser(698526208)
▶ properties	nil
▶ { } properties	a Dictionary [1 item] (#name->#multi

```
"a PPArithmeticParser(391336704)"
self
```

```
▼ a PPArithmeticParser(391336704)
  ▼ terms
    ▼ addition
      ▼ factors
        ▶ multiplication
        ▶ power
    ▼ factors
      ▼ multiplication
        ▶ power
      ▼ power
        ▶ primary
```

(a)

(b)

Figure 7.4: Using the 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 Debugger Extensions

Debugging a parser using a generic debugger also poses challenges. On 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. To overcome these challenges we developed a custom extension for the debugger (Section 6.4.3 (p.123)) that offers debugging operations at the level of the input (*e.g.*, setting a breakpoint when a certain part of the input is parsed) and a dedicated user interface that presents information about the grammar and the input being parsed.

7.3 The Path to Successful Moldable Development

The previous two sections discussed existing extensions to validate the practical applicability of moldable tools. Nevertheless, this gives no insight into the factors that facilitate the creation of extensions and of moldable tools. To address this, based on our experience in developing moldable tools and observing how developers adapt moldable tools within the Pharo community, we present several guidelines that facilitate these activities.

7.3.1 Discover Extensions by Identifying Repetitive Tasks

Once a developer has started using a moldable tool, to take advantage of that tool, she needs to decide what domain-specific extensions to create. This is not a straightforward activity even when the domain model entities are well defined, as it might not be clear what information is relevant. We observed that an approach that works well is to identify repetitive tasks during development and maintenance and to automate them using domain-specific extensions. For example, when introducing a new text editor in Glamour, a library previously discussed, there were several problems related to wrong key mappings in the editor. While addressing those bugs we observed that we repeatedly needed to inspect the key bindings of a text editor. Hence, we extended the Moldable Inspector (Figure 4.9 (p.60)) and Moldable Spotter with appropriate extensions for displaying the key bindings of graphical objects.

7.3.2 Enable Precise and Inexpensive Extensions

When the aspect that needs to be customized in a moldable tool is clear and the creation of an extension takes only a few lines of code we observed that developers create many extensions. This is evident in the case of the Moldable Inspector where developers can customize the visual representation of an object with few lines of code using an internal DSL. This led to the creation of a large number of extensions. The same observation can also be made for the Moldable Spotter.

To reduce the cost of a domain-specific extension in terms of lines of code, we designed internal DSLs where developers first select a predefined type of extension, modeled as an object, and then configure its properties by sending messages to the extension object. Messages are sent to the extension object through cascading; the extension object can provide default values for properties. All code examples from Chapter 4 and Chapter 5 follow this approach. For example, the code from Listing 4.1 (p.66) shows that the developer selected a tree view and then configured several of its properties through direct message sends. To improve the learnability of the API, each tool uses the same message names to configure similar properties in extensions. In the case of the Moldable Inspector, for example, the message `display:` is used in all extensions to set the objects or group of objects rendered by the extension and `when:` to set the activation predicate.

In the case of the Moldable Debugger, however, the unit of extension is the entire debugger. This allows for more types of extensions. Nevertheless, this also increases the cost of creating a full domain-specific debugger. For example, the domain-specific debuggers presented in Section 6.4 (p.120) took between 60 and 600 lines of code to build. Creating an extension for the Moldable Inspector requires, on average, only 9 lines of code. Hence, we saw a much smaller number of extensions for the Moldable Debugger than for the other two moldable tools.

7.3.3 Reduce the Entry Barrier for Creating Extensions

We further observed that not only the cost of a domain-specific extension is important, but also the entry barrier for creating an extension. For example, the previous object inspector from Pharo 3 also allowed developers to create domain-specific extensions. The cost of an extension was also low: 19 lines of code, on average. However, each extension required a dedicated class. With the Moldable Inspector we removed this requirement and allowed developers to create extensions by directly adding methods in the class of an object. For the Moldable Debugger we kept the need for a dedicated class for creating a domain-specific debugger. We took this decision as there are multiple aspects that need to be customized in a debugger. Nevertheless, this might also have influenced the small number of extensions that we observed for the Moldable Debugger. A future research direction can look into removing the need for a dedicated class for a domain-specific debugger and observing its effects.

Another factor that influences the barrier for creating an extension is the language used to specify the extension. For the three moldable tools discussed in this thesis we selected to rely on the general-purpose programming language of the IDE, and optimize extension creation through internal DSLs. This solution does not require developers to learn a new language for creating extensions. However, we did not explore how this solution works for heterogenous applications written in more than one programming language.

7.3.4 Build Workflows Using Extensions

To increase the value of a domain-specific extension in the Moldable Inspector and Moldable Spotter, domain-specific extensions are not used in isolation. Due to the design of these tools, developers can combine extensions to form custom workflows. For example, as exemplified in Figure 7.2c (p.146), by adding a custom extension to the classes `CompileMethod` and `SymbolicBytecode`, developers can obtain a custom browser for viewing how bytecode maps to source code. We observed that many extensions were created to support such workflow, rather than to be used in isolation.

7.3.5 Bootstrap Using Initial Examples

For all tree moldable tools we observed from discussions over the Pharo mailing list that developers started to create new extensions by exploring already available extensions from the system. This observation was confirmed for Moldable Spotter in the user study presented in Section 5.6.3 (p.101). There, all participants decided to create a new extension for Moldable Spotter by modifying an existing extension, instead of starting from scratch. This indicates that providing a good set of sample extensions helps developers to learn how to extend a moldable tool.

7.4 Summary

In this chapter we looked at how moldable tools were applied within Pharo. We explored the evolution of domain-specific extensions and showed what kind of domain-specific environments are created when developers extend the three moldable tools discussed in this thesis. To improve the adoption of moldable tools we presented several guidelines based on our experience. These guidelines indicate that a low cost for creating precise extensions, together with the ability to use extensions to build custom workflows and examples showing how to create extensions, have a positive impact on the adoption and usage of a moldable tool.

Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

Antoine de Saint-Exupéry

8 Conclusions

Evolving software requires developers to continuously understand the state of their software systems and change them according to new requirements. Through this work we explored how to better support developers in performing this activity. Our solution consists in enabling developers to craft their own domain-aware development tools, as these tools can directly answer relevant domain-specific questions. In this chapter we summarize the contributions made by this dissertation for supporting this activity and discuss future research directions.

8.1 Contributions

In this dissertation we argued that supporting inexpensive adaptations and context-awareness are crucial aspects for enabling domain-aware development tools. As a validation of this thesis statement we made several contributions that improve the state of the art in the field of tool building:

1. We proposed and motivated the *moldable tools* approach for tool building. This approach describes how to incorporate domain concepts into development tools through inexpensive adaptations.

2. We demonstrated the practical applicability of the moldable tools approach by applying it to solve three existing problems in the development of object-oriented applications:

    - Traditional object inspectors give developers access to the state of run-time objects. Through an empirical investigation we observed a need for object inspectors that support a unified workflow for exploring multiple objects using views tailored to their own contextual needs. We addressed this research problem by designing, following the moldable tools approach, the *Moldable Inspector*, an inspector that allows developers to customize the views through which they look at objects and the set of objects accessible from within a view.

- Searching, while a pervasive activity during software development, is mainly supported in IDEs through generic and disconnected search tools. Finding relevant domain concepts often requires developers to use multiple generic tools and manually link their results. We addressed this problem by applying the moldable tools approach to design *Moldable Spotter*, a search infrastructure allowing developers to create custom searches for their domain objects and automatically discover appropriate searches.
- Traditional debuggers do not allow developers to debug their applications directly in terms of domain concepts; instead they provide stack-based operations, line breakpoints, and generic user interfaces. To address this problem we investigated solutions that enable developers to create their own domain-specific debuggers. We designed and implemented *Moldable Debugger* a framework that makes is possible for developers to customize the user interface and provide debugging actions like domain-specific breakpoints.

The availability of a moldable infrastructure opens new possibilities:

1. The developers of a library or a framework can create and ship dedicated extensions for moldable tools together with the code, to help users work with and better reason about that framework or library. This can have a practical impact due to the reuse of the library or the framework in many applications. For example, the developers of PetitParser, Glamour, Opal and Roassal built themselves custom extensions for the moldable tools discussed in this dissertation and shipped them together with the frameworks;
2. Developers can extend moldable tools for their own applications, during the development process, to help them solve bugs or better understand the application. This can make considerable economical sense when working on a long-lived system.

8.2 Future Research Directions

Having introduced the moldable tools approach and showed how it can be applied to address relevant issues in current development tools, we identify scope of further work in this area. We split future work in the following three parts: the first part addresses how to further improve the creation of moldable tools, the second part looks at how to group moldable tools into a moldable environment and the third section introduces wider questions in this research field.

8.2.1 Improving and Evolving Moldable Tools

Keeping extensions and objects synchronized Properties and responsibilities of concepts from a domain can change as the software system evolves; this leads to changes in the code that implements those domain concepts. Given that we propose a manual approach for constructing domain-specific extensions, one needs to make sure that new changes in an application do not break assumptions made when developing an extension. Currently, for each object that provides an extension we require that extension builders also provide code that constructs an instance of that type. During testing we can then verify if there is an error while creating an extension on that instance. This gives a minimum safety-net for detecting changes that break an extension. We also see automatic test generation based on more meta-information attached to extensions as a possible solution.

Characterizing changes based on how they impact extensions As software systems undergo changes, domain-specific extensions written for those systems must also evolve. Nevertheless, not all changes from an application can directly impact the extensions created for that application. For example, adding a new object attribute would not break existing domain-specific views for an object. Introducing code instrumentation or data accesses with side effects (as in a stream), however, can lead to significant changes in an existing domain-specific debugger. To support updating extensions as the code changes, a future research direction could look into classifying code changes based on how they affect domain-specific extensions. This could help detect changes that break domain-specific extensions and notify developers of this fact.

Automatic and semi-automatic generation of extensions Currently moldable tools rely on manual creation of extensions using a dedicated API. An alternative consists in automatically, or semi-automatically, generating extensions. Referring to the Moldable Inspector, this was shown to work well when generating views based only on the internal representation of an object [Cross et al. 2009]. For example, a tree view can be used to represent all objects whose internal structure matches a tree. Nevertheless, not all the views supported by the Moldable Inspector follow this approach. Some, like the *Bytecode* view of a compiled method object display information that is not clear how to extract automatically. Others, like the *Picture* view (Figure 4.13 (p.63)) of a graphical object are not related directly to the structure of an object. To enable automatic generation of extensions in moldable tools a future research direction needs to look in more detail at what kind of extensions can be created automatically.

Application characteristics that simplify extension creation In the field of software testing, high coupling makes the creation of unit tests difficult (by increasing the number of dependencies that need to be taken into account) and thus decreases the testability of a software system. The same idea can also be applied to the creation of domain-specific extensions. Hence, a future research direction can investigate what characteristics of an application ease, or exacerbate the creation of domain-specific extensions for moldable tools.

Live development of extensions Live programming aims to give developers immediate feedback, ideally, after every change to the code. To move towards live programming, moldable tools should enable developers to extend a tool directly from within that tool and show a live preview of the extension. Currently only the Moldable Inspector supports this: developers can extend the inspector from within the inspector at run time by creating an extension for an object directly in the inspector, and getting a live preview of the resulting extension whenever they save the extension. Nevertheless, even in this case to get the preview the developer has to make sure the extension is valid whenever saving it. A future research direction can look at how to better support live development of extensions for moldable tools so that developers always have a working extension as they develop that extension. This would enable developers to get live feedback after every change and do not get in a situation where they have an invalid extension.

8.2.2 Towards a Moldable Environment

Developers often complain about loose integration of tools that forces them to look for relevant information in multiple places [Maalej 2009]. To avoid this problem the three aforementioned moldable tools are integrated into Pharo and essentially replace the previous tools. Nevertheless, Pharo contains 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. Below are two main research directions for supporting such an environment.

Unified model of a development context Each of the three aforementioned moldable tools maintains its own development context. Adding more moldable tools, each maintaining its own context, could result in duplicated functionality between tools and introduce confusion when creating extensions, if development contexts are modeled in different ways. To address this, a moldable environment can provide a global development context where tools can add interaction and explicit data. This would enable activation predicates to filter extensions within a tool based on developer interactions from another tool, a feature currently not supported.

Collaborative development and sharing of extensions In the examples presented in this dissertation, developers create custom extensions which they can share together with their applications. This solution works well if all developers that want to create extensions can add them to the target application. This is however not the case with frameworks or libraries, where external developers may have difficulties in integrating their extensions. To address this an infrastructure for moldable tools can further provide a complementary mechanism for developers to share and discover extensions.

8.2.3 Wider Research Questions

Application of moldable tools to other domains This dissertation exemplifies the moldable tools approach by applying it to several application domains including parsers, compilers, profilers, event-based systems, UI frameworks, *etc.* This still only covers a small range of existing application domains, and does not take into account applications from other research fields like biology, physics, astronomy, to name a few. Future work is needed to explore how moldable tools can be applied to applications from these and other domains.

Empirical investigations into the impact of tool building on program comprehension Moldable tools promote domain-specific tool building as a solution for improving program comprehension: if developers incorporate domain-specific information into their tools they will need to read less code to extract the information of interest and perform fewer repetitive tasks. Based on our interaction with the Pharo community we observed that indeed developers extended moldable tools to improve how they reason about their systems. Nevertheless, an empirical investigation assessing the impact of adapting moldable tools on program comprehension is currently missing. One direction for performing this evaluation is by comparing the effort required to build an extension for solving a domain-specific task with the effort required to solve that task with existing tools. This, however, only applies if an extension is thrown away after use. A more extensive study should further look into how consistent adaptation of tools in development teams, combined with extension reuse during the evolution of a software application, affects comprehension.

Extensions as program comprehension aids More often than not domain-specific development tools are designed as black boxes, or make it difficult for developers to understand the domain-specific inner workings of the tools. With moldable tools, domain-specific extensions are explicit. Hence, if a developer would want to know the meaning of a custom extension, she has the opportunity to do so by browsing the code of that extension. The small size associated with custom extensions further makes understanding the inner workings of the extension affordable. This could be leveraged to support program comprehension: developers could reason about an application by studying the domain-specific extensions available for an application. As an analogy, while initially designed to verify the correctness of applications, tests are now often employed as a mechanism to understand how to use a library or an API.

8.3 Summary

Development tools are a prerequisite for crafting software and represent the lens through which developers perceive software. If this lens is generic and does not allow developers to reason about their applications in terms of domain concepts, developers will repeatedly waste time and effort manually recovering those concepts from the code. To address this problem we proposed that developers continuously incorporate domain abstractions into their development tools through inexpensive extensions. For this activity to be feasible, development tools need to be designed to support inexpensive domain-specific extensions. To show that this is indeed achievable we introduced the moldable tools approach for designing development tools. We then showed how tool builders can apply it to enable adaptations that solve relevant problems in several types of development tools.

8.4 Closing Words

Software is contextual by design. Stakeholders, developers, technology and the randomness of everyday life make each software system unique. Yet, when crafting software our tools act as if all software systems are the same.

This entire work is about showing that this does not have to be the case. Tools are the means to an end. We should shape them, they should not shape us.

August 29, 2016
Andrei Chiş

Appendices

A

What is an Object Inspector?

Answers provided by the interview participants to the question: *What is an object inspector for you?* Two participants did not provide an answer.

Participant	*Answer*
P1	—
P2	A tool to look inside an object
P3	Something to get a good idea about what the state of the object is
P4	Something that allows me to take a look at an object
P5	A tool that allows me to inspect the object
P6	A tool that allows me to inspect objects
P7	See all the fields
P8	A tool that helps you to inspect data at runtime
P9	A way to see inside an object
P10	A tool to understand which are the components/status/relations of an object
P11	The tool where I can inspect live objects and I can dive and inspect the state
P12	A reflective tool that allows me to see all the structure of an object and change it
P13	I can see the state in which an object is
P14	A tool showing the state of an object which is logged in memory
P15	—
P16	An easy way to see and manipulate what's inside an object at its most fundamental level

B

Performance Overhead of Detecting Run-time Events

Section 6.5 (p.129) presented three mechanisms for detecting run-time events: step-by-step execution, code-centric instrumentation and object-centric instrumentation. The performance overhead of these mechanisms plays an important role when a developers needs to decide which one to use. To investigate the performance overhead of these mechanisms we performed a series of micro-benchmarks. We performed these benchmarks using the implementation of the Moldable Debugger[1] in Pharo, on an Apple MacBook Pro, 2.7 GHz Intel Core i7 in Pharo 4 with the jitted Pharo VM[2]. We ran each benchmark 5 times and present the average time of these runs in milliseconds. All presented times exclude garbage collection time.

Step-by-step execution

Basic benchmarks We performed, for each basic predicate, a benchmark checking for an event (*i.e.*, method call, message send, attribute access, attribute/condition over the state) not present in the measured code. Given that a predicate never matches a point in the execution, the boolean condition will be checked for every bytecode instruction, giving us the worst-case overhead of these predicates on the measured code. The measured code (lines 89–90)[3] consists of ten million calls to the method `Counter>>#increment` (lines 86–87). We selected this method as it has only one message send, attribute access and attribute write, making it possible to use it for all predicates (for *method call* we instrument the call to `increment`).

```
86 Counter>>#increment
87     counter ← counter + 1
```

```
88 targetObject ← Counter new.
89 10000000 timesRepeat: [
90     targetObject increment]
```

[1] The version of the Moldable Debugger used to perform the measurements, including the code of the benchmarks, can be found at `http://scg.unibe.ch/download/moldabledebugger/moldabledebugger.zip`

[2] `http://files.pharo.org/vm/pharo`

[3] In all code snippets showing code on which we performed a measurement, only the underlined line is the one that is actually being measured; the other lines represent just the setup and are not taken into account in the measurement.

Predicate	Normal execution	Step-by-step execution	Overhead
attribute access (`#counter`)	11 ms	13473 ms	1225×
attribute write (`#counter`)		13530 ms	1230×
method call (`#increment`)		14137 ms	1285×
message send (+)		14302 ms	1300×
identity check		12771 ms	1161×
empty state check		12627 ms	1148×

Table B.1: Performance measurements for step-by-step execution on basic examples.

Benchmark	Normal execution	Step-by-step execution	Overhead
factorial	1094 ms	2716 ms	2.6×
merge sort	4 ms	4530 ms	1120×
parser initialization	192 ms	5881 ms	37×
parser execution	18 ms	10334 ms	613×
announcement delivery	19 ms	16419 ms	864×

Table B.2: Performance measurements done on real-world examples for step-by-step execution.

As expected, this approach introduces a significant overhead of more than three orders of magnitude for all predicates, when the event of interest is not detected (Table B.1 (p.164)). The high overhead is due to the step-by-step execution rather than to the actual condition being checked: verifying the identity of an object using a *state check* predicate (Table B.1 (p.164) – identity check) has almost the same overhead as a *state check* predicate that performs no check (Table B.1 (p.164) – empty state check).

Advanced benchmarks To determine if this high overhead is present when dealing with real-world code, rather than a constructed example, we performed five more benchmarks presented in Listings B.1 – B.5. In each benchmark we used a *method call* predicate detecting the execution of a method not called in the measured code. Like in the previous benchmarks this gives us the maximal performance impact for this predicate. We only use one type of basic predicate given that all types of basic predicates exhibit a similar overhead.

Listing B.1: Factorial.

```
25000 factorial
```

Listing B.2: Merge sort.

```
collection ← 2000 factorial asString.
collection sorted
```

Listing B.3: Parser initialization: initialize a PetitParser parser for Java code.

```
PPJavaParser new
```

Listing B.4: Parser execution: parse the source code of the interface `Stack` from Java 6 using a parser for java code.

```
parser ← PPJavaParser new.
parserContext ← PPContext new.
parserContext stream: self getStackJava.
parser parseWithContext: parserContext
```

Listing B.5: Announcement delivery.

```
announcer ← Announcer new.
10000 timesRepeat: [
    announcer
        when: Announcement
        send: #execute:
        to: AnnouncementTarget new ].
announcer announce: Announcement
```

We obtained different results (Table B.2 (p.164)) than in the previous set of benchmarks ranging from an overhead of only 2.6× to an overhead of 1120×. These diverging results can be explained by looking at one particular aspect of the measured code: the time spent in method calls that are implemented directly by the VM (*i.e.*, primitive operations) and thus cannot by executed in a step-by-step manner by a bytecode interpreter. For example, on the one hand, when computing factorial most time is spent doing additions, an operation implemented directly by the VM. Merge sort, on the other hand, spends little time in primitives, thus exhibits similar worst-case overhead to the example code from the previous benchmarks.

Code-centric instrumentation

This approach does not introduce any runtime overhead when using basic predicates to detect attribute reads/writes, message sends and method calls. The overhead comes from combining these predicates with state check predicates, and from the actual implementation mechanism used to check the condition. Given that we use two approaches for instrumenting code (*i.e.*, slots, AST annotations) we performed measurements that combine *attribute access* and *method call* predicates with *state check* predicates.

Basic benchmarks We first combine the aforementioned predicates with an *identity check* predicate. For each situation we perform a benchmark on only the operation we are interested in (*i.e.*, attribute write – lines 106-107, method call to `returnOne` – lines 108-109) and on the `#increment` method used in the previous section. We execute each method ten million times on one object and use an *identity check* predicate that never detects an event in the measured code (*i.e.*, checks for another object).

```
106 Counter>>#initializeWithOne
107     counter ← 1
```

```
108 Counter>>#returnOne
109     ↑ 1
```

Instrumented method	Predicate	Normal execution	Instrumented execution	Overhead one check	Overhead three checks
`#initializeWithOne`	attribute write	81 ms	1317 ms	16×	17×
`#returnOne`	method call	83 ms	7664 ms	95×	98×
`#increment`	attribute write	103 ms	1350 ms	13×	14×
`#increment`	method call	103 ms	7560 ms	75×	77×
`#initializeWithOne`	attribute mutation	81 ms	645 ms	8×	-
`#increment`	attribute mutation	103 ms	652 ms	6×	-

Table B.3: Performance measurements done on simple examples for code-centric instrumentation.

As seen from Table B.3 (p.166) the overhead is significantly lower than the one introduced by step-by-step execution. Regardless of the predicate, the highest overhead is obtained for the methods `initializeWithOne` and `returnOne` where, given that the methods have almost no functionality, any extra instrumentation increases execution time significantly. The overhead for the `increment` method is lower as this method performs more operations than the previous two. Nevertheless, the *method call* predicate has an overhead six times higher than *attribute write* predicate. While for both implementations we reify the current stack frame before checking any associated condition, Reflectivity, the framework used for code instrumentation, has an expensive mechanism for detecting recursive calls from meta-links (*i.e.*, detect when a meta-link is added in code called from the meta-link). Repeating these measurements when the basic predicates are combined with five identity check predicates results in only slightly higher overheads for all benchmarks. This indicates that most of the overhead comes from reifying the execution context every time a condition needs to be checked.

Based on the previous observation a further improvement can be done when combining an *attribute access*/*attribute write* predicate with a *state check* predicate that only accesses the new and old value of the instance variable: given that we use slots for instrumenting attributes accesses/writes we can directly get the new and old values of the attribute from the slot without reifying the current stack frame. This leads to a performance overhead just x8 when monitoring changes in the value of an attribute.

A further improvement in performance can be achieved by removing altogether the need for reifying the current stack frame. A *method call* predicate combined with an *identity check* predicate can directly insert a condition comparing `self` (`this` in Java) with a given object in the code of the target method. Our current prototype does not support these kinds of instrumentations. Nevertheless, the performance overhead

Debugging action	Normal execution	Step-by-step execution	Overhead
Production(aProduction)	486ms	56ms	8.5×
Parser(aParser)	1553ms	56	27.7×
Stream position(anInteger)	92ms	56ms	1.65×
Subscription(aSubscription)	225ms	968ms	4.2×

Table B.4: Performance measurements done on real-world examples for code-centric instrumentation.

required to reify the stack frame is small enough to have practical applicability. This approach further allows developers to test any kind of property of the stack frame (*e.g.*, the position of the program counter).

Advanced benchmarks We performed four benchmarks on the following domain-specific actions presented in Section 6.4 (p.120): *Production(aProduction)*, *Stream position(anInteger)*, *Parser(aParserClass)* and *Subscription(aSubscription)*. For the first three we used the code from Listing B.4 (p.165), while for the last one we used the code from Listing B.5 (p.165). For each debugging action we look for an event not present in the measured code (*e.g.*, a production not present in the parser).

For all debugging actions, we get a runtime overhead lower than the one from the basic benchmarks ranging from 1.6x to 27.7x (Table B.4 (p.167)). This is expected because in this case the event that triggers a breakpoint is encountered far less often. The debugging action *Parser(aParserClass)* has the largest overhead as it introduces a high number of instrumentations.

Object-centric instrumentation

Using this approach there is no longer any runtime overhead in detecting when an attribute read/write, message send or method call happens on a given object. Runtime overhead is introduced by adding conditions that need to be checked when the target event is detected. For example, checking if a method is called on a target object that satisfies an extra condition only incurs runtime overhead for checking the extra condition every time the method is called on the target object.

Basic benchmarks Given that we use code-centric instrumentations to insert those checks into the unique anonymous class of the target object, the performance overhead will always be lower than or equal to the overhead of just code-centric instrumentations. Consider the two situations below:

```
targetObject ← Counter new.
10000000 timesRepeat: [
    targetObject initializeWithOne]
```

```
targetObjects ← OrderedCollection
    new: 10000000.
10000000 timesRepeat: [
    targetObjects addLast: Counter new].
targetObjects do: [:aCounter|
    aCounter initializeWithOne]
```

In the code on the left, detecting when `targetObject` is initialized with value 2 has the same overhead as using code-centric instrumentations given that the condition must be checked on every write of the `counter` attribute (as seen in the previous section verifying the identity of one or more objects incurs a similar overhead, given that what takes the most time is reifying the execution context).

In the code on the right, the runtime overhead when checking that one object from the collection is initialized with 2 is negligible, as the condition is checked only once. Installing the predicate on every object will lead to a similar runtime overhead as in the previous case, given that the condition will be checked ten million times.

Advanced benchmarks The same observations from *Basic benchmarks* apply. On the one hand, in the action *Stream position(anInteger)*, detecting when the stream has reached a certain position using an object-centric instrumentation has the same overhead as a code-centric instrumentation given that there is a single stream object shared by all the parsers. On the other hand, applying the action *Subscription(aSubscription)* on the code from Listing B.5 has a negligible overhead as each announcement is delivered to a different object.

Bibliography

[Aftandilian et al. 2010] Aftandilian, Edward E. ; Kelley, Sean ; Gramazio, Connor ; Ricci, Nathan ; Su, Sara L. ; Guyer, Samuel Z.: Heapviz: interactive heap visualization for program understanding and debugging. In: *Proceedings of the 5th international symposium on Software visualization*. New York, NY, USA : ACM, 2010 (SOFTVIS '10), S. 53–62. – URL `http://doi.acm.org/10.1145/1879211.1879222`. – ISBN 978-1-4503-0028-5

[Alexander et al. 2009] Alexander, Jason ; Cockburn, Andy ; Fitchett, Stephen ; Gutwin, Carl ; Greenberg, Saul: Revisiting Read Wear: Analysis, Design, and Evaluation of a Footprints Scrollbar. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM, 2009 (CHI '09), S. 1665–1674. – URL `http://doi.acm.org/10.1145/1518701.1518957`. – ISBN 978-1-60558-246-7

[Alsallakh et al. 2012] Alsallakh, Bilal ; Bodesinsky, Peter ; Miksch, Silvia ; Nasseri, Dorna: Visualizing Arrays in the Eclipse Java IDE. In: *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*. Washington, DC, USA : IEEE Computer Society, 2012 (CSMR '12), S. 541–544. – URL `http://dx.doi.org/10.1109/CSMR.2012.71`. – ISBN 978-0-7695-4666-7

[de Alwis and Murphy 2008] Alwis, Brian de ; Murphy, Gail C.: Answering conceptual queries with Ferret. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. New York, NY, USA : ACM, 2008, S. 21–30. – ISBN 978-1-60558-079-1

[Araya et al. 2013] Araya, Vanessa P. ; Bergel, Alexandre ; Cassou, Damien ; Ducasse, Stéphane ; Laval, Jannik: Agile Visualization with Roassal. In: *Deep Into Pharo*. Square Bracket Associates, September 2013, S. 209–239. – ISBN 978-3-9523341-6-4

[Auguston et al. 2002] Auguston, Mikhail ; Jeffery, Clinton ; Underwood, Scott: A Framework for Automatic Debugging. In: *ASE*, IEEE Computer Society, 2002, S. 217–222. – ISBN 0-7695-1736-6

[Ayewah et al. 2008] Ayewah, N. ; Hovemeyer, D. ; Morgenthaler, J.D. ; Penix, J. ; Pugh, William: Using Static Analysis to Find Bugs. In: *Software, IEEE* 25 (2008), September, Nr. 5, S. 22–29. – ISSN 0740-7459

[Bahlke and Snelting 1986] Bahlke, Rolf ; Snelting, Gregor: The PSG System: From Formal Language Definitions to Interactive Programming Environments. In: *ACM Trans. Program. Lang. Syst.* 8 (1986), August, Nr. 4, S. 547–576. – URL `http://doi.acm.org/10.1145/6465.20890`. – ISSN 0164-0925

[Ballance et al. 1992] Ballance, Robert A. ; Graham, Susan L. ; Van De Vanter, Michael L.: The Pan Language-based Editing System. In: *ACM Trans. Softw. Eng. Methodol.* 1 (1992), Januar, Nr. 1, S. 95–127. – URL http://doi.acm.org/10.1145/125489.122804. – ISSN 1049-331X

[Beck et al. 2015] Beck, Fabian ; Dit, Bogdan ; Velasco-Madden, Jaleo ; Weiskopf, Daniel ; Poshyvanyk, Denys: Rethinking User Interfaces for Feature Location. In: *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. Piscataway, NJ, USA : IEEE Press, 2015 (ICPC '15), S. 151–162. – URL http://dl.acm.org/citation.cfm?id=2820282.2820304

[Beck 1999] Beck, Kent: *Kent Beck's Guide to Better Smalltalk*. Sigs Books, 1999. – ISBN 0-251-64437-2

[Beck 2002] Beck, Kent: *Test Driven Development: By Example*. Addison-Wesley Longman, 2002. – ISBN 978-0321146533

[Bergel et al. 2008] Bergel, Alexandre ; Ducasse, Stéphane ; Putney, Colin ; Wuyts, Roel: Creating Sophisticated Development Tools with OmniBrowser. In: *Journal of Computer Languages, Systems and Structures* 34 (2008), Nr. 2-3, S. 109–129

[Bergel et al. 2011] Bergel, Alexandre ; nados, Felipe B. ; Robbes, Romain ; Röthlisberger, David: Spy: A flexible Code Profiling Framework. In: *Journal of Computer Languages, Systems and Structures* 38 (2011), Dezember, Nr. 1. – URL http://bergel.eu/download/papers/Berg10f-Spy.pdf

[Bézivin and Gerbé 2001] Bézivin, Jean ; Gerbé, Olivier: Towards a Precise Definition of the OMG/MDA Framework. In: *ASE'01: Proceedings of 16th International Conference on Automated Software Engineering*. San Diego, CA, USA : IEEE Computer Society, November 2001, S. 273–282. – URL http://www.sciences.univ-nantes.fr/lina/atl/www/papers/ASE01.OG.JB.pdf. – ISSN 1527-1366

[Bonér 2004] Bonér, Jonas: What are the key issues for commercial AOP use: how does AspectWerkz address them? In: *Proceedings of the 3rd international conference on Aspect-oriented software development*. New York, NY, USA : ACM, 2004 (AOSD '04), S. 5–6. – ISBN 1-58113-842-3

[Borras et al. 1988] Borras, P. ; Clement, D. ; Despeyroux, Th. ; Incerpi, J. ; Kahn, G. ; Lang, B. ; Pascual, V.: Centaur: The System. In: *SIGPLAN Not.* 24 (1988), November, Nr. 2, S. 14–24. – URL http://doi.acm.org/10.1145/64140.65005. – ISSN 0362-1340

[Bousse et al. 2015] Bousse, Erwan ; Corley, Jonathan ; Combemale, Benoit ; Gray, Jeff ; Baudry, Benoit: Supporting Efficient and Advanced Omniscient Debugging for xDSMLs. In: *8th International Conference on Software Language Engineering (SLE)* . Pittsburg, United States, Oktober 2015. – URL https://hal.inria.fr/hal-01182517

[Bragdon et al. 2010a] Bragdon, Andrew ; Reiss, Steven P. ; Zeleznik, Robert ; Karumuri, Suman ; Cheung, William ; Kaplan, Joshua ; Coleman, Christopher ; Adeputra, Ferdi ; LaViola, Joseph J.: Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA : ACM, 2010 (ICSE '10), S. 455–464. – URL http://doi.acm.org/10.1145/1806799.1806866. – ISBN 978-1-60558-719-6

[Bragdon et al. 2010b] Bragdon, Andrew ; Zeleznik, Robert ; Reiss, Steven P. ; Karumuri, Suman ; Cheung, William ; Kaplan, Joshua ; Coleman, Christopher ; Adeputra, Ferdi ; LaViola, Joseph J.: Code bubbles: a working set-based interface for code understanding and maintenance. In: *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*. New York, NY, USA : ACM, 2010, S. 2503–2512. – ISBN 978-1-60558-929-9

[Brand et al. 2001] Brand, M. G. J. ; Deursen, A. ; Heering, J. ; Jong, H. A. ; Jonge, M. ; Kuipers, T. ; Klint, P. ; Moonen, L. ; Olivier, P. A. ; Scheerder, J. ; Vinju, J. J. ; Visser, E. ; Visser, J.: *Compiler Construction: 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2–6, 2001 Proceedings*. Kap. The ASF+SDF Meta-environment: A Component-Based Language Development Environment, S. 365–370. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001. – URL http://dx.doi.org/10.1007/3-540-45306-7_26. – ISBN 978-3-540-45306-2

[van den Brand et al. 2005] Brand, M.G.J. van den ; Cornelissen, B. ; Olivier, P.A. ; Vinju, J.J.: TIDE: A Generic Debugging Framework — Tool Demonstration —. In: *Electronic Notes in Theoretical Computer Science* 141 (2005), Nr. 4, S. 161 – 165. – URL http://www.sciencedirect.com/science/article/pii/S1571066105051789. – Proceedings of the Fifth Workshop on Language Descriptions, Tools, and Applications (LDTA 2005) Language Descriptions, Tools, and Applications 2005. – ISSN 1571-0661

[Brant et al. 1998] Brant, John ; Foote, Brian ; Johnson, Ralph ; Roberts, Don: Wrappers to the Rescue. In: *Proceedings European Conference on Object Oriented Programming (ECOOP'98)* Bd. 1445 Springer-Verlag (Veranst.), 1998, S. 396–417

[Bunge 2009] Bunge, Philipp: *Scripting Browsers with Glamour*, University of Bern, Master's Thesis, April 2009. – URL http://scg.unibe.ch/archive/masters/Bung09a.pdf

[Campbell et al. 2003] Campbell, Alistair E. R. ; Catto, Geoffrey L. ; Hansen, Eric E.: Language-independent interactive data visualization. In: *SIGCSE Bull.* 35 (2003), Januar, Nr. 1, S. 215–219. – ISSN 0097-8418

[Cao et al. 2008] Cao, Qing ; Abdelzaher, Tarek ; Stankovic, John ; Whitehouse, Kamin ; Luo, Liqian: Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks. In:

Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems. New York, NY, USA : ACM, 2008 (SenSys '08), S. 85–98. – URL `http://doi.acm.org/10.1145/1460412.1460422`. – ISBN 978-1-59593-990-6

[Caracciolo 2016] Caracciolo, Andrea: *A Unified Approach to Architecture Conformance Checking*, University of Bern, PhD thesis, März 2016. – URL `http://scg.unibe.ch/archive/phd/caracciolo-phd.pdf`

[Charles et al. 2007] Charles, Philippe ; Fuhrer, Robert M. ; Sutton, Stanley M.: IMP: A Meta-tooling Platform for Creating Language-specific IDEs in Eclipse. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA : ACM, 2007 (ASE '07), S. 485–488. – URL `http://doi.acm.org/10.1145/1321631.1321715`. – ISBN 978-1-59593-882-4

[Cheng et al. 2008] Cheng, Yung P. ; Chen, Jih F. ; Chiu, Ming C. ; Lai, Nien W. ; Tseng, Chien C.: xDIVA: a debugging visualization system with composable visualization metaphors. In: *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. New York, NY, USA : ACM, 2008 (OOPSLA Companion '08), S. 807–810. – ISBN 978-1-60558-220-7

[Chiş et al. 2015a] Chiş, Andrei ; Denker, Marcus ; Gîrba, Tudor ; Nierstrasz, Oscar: Practical domain-specific debuggers using the Moldable Debugger framework. In: *Computer Languages, Systems & Structures* 44, Part A (2015), S. 89–113. – URL `http://scg.unibe.ch/archive/papers/Chis15c-PracticalDomainSpecificDebuggers.pdf`. – Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014). – ISSN 1477-8424

[Chiş et al. 2016a] Chiş, Andrei ; Gîrba, Tudor ; Kubelka, Juraj ; Nierstrasz, Oscar ; Reichhart, Stefan ; Syrel, Aliaksei: Exemplifying Moldable Development. In: *Proceedings of the 1st Edition of the Programming Experience Workshop*. New York, NY, USA : ACM, 2016 (PX/16), S. to appear. – URL `http://scg.unibe.ch/archive/papers/Chis16b-ExemplifyingMoldableDevelopment.pdf`. – ISBN 978-1-4503-4776-1/16/07

[Chiş et al. 2016b] Chiş, Andrei ; Gîrba, Tudor ; Kubelka, Juraj ; Nierstrasz, Oscar ; Reichhart, Stefan ; Syrel, Aliaksei: Moldable, context-aware searching with Spotter. In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York, NY, USA : ACM, 2016 (Onward! 2016), S. to appear. – URL `http://scg.unibe.ch/archive/papers/Chis16a-MoldableContextAwareSearchingWithSpotter.pdf`

[Chiş et al. 2016c] Chiş, Andrei ; Gîrba, Tudor ; Kubelka, Juraj ; Nierstrasz, Oscar ; Reichhart, Stefan ; Syrel, Aliaksei: Moldable Tools for Object-oriented Development. In: Meyer, Bertrand (Hrsg.): *PAUSE: Present And Ulterior Software Engineering*. Springer Verlag, 2016, S. to appear

[Chiş et al. 2014a] Chiş, Andrei ; Gîrba, Tudor ; Nierstrasz, Oscar: The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In: Combemale, Benoît (Hrsg.) ; Pearce, DavidJ. (Hrsg.) ; Barais, Olivier (Hrsg.) ; Vinju, Jurgen J. (Hrsg.): *Software Language Engineering* Bd. 8706, Springer International Publishing, 2014, S. 102–121. – URL http://scg.unibe.ch/archive/papers/Chis14b-MoldableDebugger.pdf. – ISBN 978-3-319-11244-2

[Chiş et al. 2014b] Chiş, Andrei ; Gîrba, Tudor ; Nierstrasz, Oscar: The Moldable Inspector: a framework for domain-specific object inspection. In: *Proceedings of International Workshop on Smalltalk Technologies (IWST 2014)*, URL http://scg.unibe.ch/archive/papers/Chis14a-MoldableInspector.pdf, 2014

[Chiş et al. 2015b] Chiş, Andrei ; Gîrba, Tudor ; Nierstrasz, Oscar: Towards moldable development tools. In: *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*. New York, NY, USA : ACM, 2015 (PLATEAU '15), S. 25–26. – URL http://scg.unibe.ch/archive/papers/Chis15d_TowardsMoldableDevelopmentTools.pdf. – ISBN 978-1-4503-3907-0

[Chiş et al. 2015c] Chiş, Andrei ; Gîrba, Tudor ; Nierstrasz, Oscar ; Syrel, Aliaksei: GTInspector: A Moldable Domain-Aware Object Inspector. In: *Proceedings of the Companion Publication of the 2015 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*. New York, NY, USA : ACM, 2015 (SPLASH Companion 2015), S. 15–16. – URL http://scg.unibe.ch/archive/papers/Chis15b-GTInspector.pdf. – ISBN 978-1-4503-3722-9

[Chiş et al. 2015d] Chiş, Andrei ; Gîrba, Tudor ; Nierstrasz, Oscar ; Syrel, Aliaksei: The Moldable Inspector. In: *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York, NY, USA : ACM, 2015 (Onward! 2015), S. 44–60. – URL http://scg.unibe.ch/archive/papers/Chis15a-MoldableInspector.pdf. – ISBN 978-1-4503-3688-8

[Chiş et al. 2013] Chiş, Andrei ; Nierstrasz, Oscar ; Gîrba, Tudor: Towards a Moldable Debugger. In: *Proceedings of the 7th Workshop on Dynamic Languages and Applications*. New York, NY, USA : ACM, 2013 (DYLA '13), S. 2:1–2:6. – URL http://scg.unibe.ch/archive/papers/Chis13a-TowardsMoldableDebugger.pdf. – ISBN 978-1-4503-2041-2

[Chimera 1992] Chimera, Richard: Value Bars: An Information Visualization and Navigation Tool for Multi-attribute Listings. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM, 1992 (CHI '92), S. 293–294. – URL http://doi.acm.org/10.1145/142750.142817. – ISBN 0-89791-513-5

[Clayberg and Rubel 2008] Clayberg, Eric ; Rubel, Dan: *Eclipse Plug-ins*. 3. Addison-Wesley Professional, 2008. – ISBN 0321553462, 9780321553461

[Crawford et al. 1995] Crawford, Richard H. ; Olsson, Ronald A. ; Ho, W. W. ; Wee, Christopher E.: Semantic issues in the design of languages for debugging.

In: *Comput. Lang.* 21 (1995), April, Nr. 1, S. 17–37. – URL http://dx.doi.org/10.1016/0096-0551(94)00015-I. – ISSN 0096-0551

[Creswell and Vicki 2006] Creswell, John W. ; Vicki: *Designing and Conducting Mixed Methods Research*. 1. Sage Publications, Inc, August 2006. – URL http://www.worldcat.org/isbn/1412927927. – ISBN 9781412927925

[Cross et al. 2009] Cross, James H. ; Hendrix, T. D. ; Umphress, David A. ; Barowski, Larry A. ; Jain, Jhilmil ; Montgomery, Lacey N.: Robust Generation of Dynamic Data Structure Visualizations with Multiple Interaction Approaches. In: *Trans. Comput. Educ.* 9 (2009), Juni, Nr. 2, S. 13:1–13:32. – URL http://doi.acm.org/10.1145/1538234.1538240. – ISSN 1946-6226

[Cuadrado et al. 2006] Cuadrado, Jesús S. ; Molina, Jesús G. ; Tortosa, Marcos M.: RubyTL: A Practical, Extensible Transformation Language. In: *Proceedings of the Second European Conference on Model Driven Architecture: Foundations and Applications*. Berlin, Heidelberg : Springer-Verlag, 2006 (ECMDA-FA'06), S. 158–172. – URL http://dx.doi.org/10.1007/11787044_13. – ISBN 3-540-35909-5, 978-3-540-35909-8

[Cutsem et al. 2014] Cutsem, Tom V. ; Boix, Elisa G. ; Scholliers, Christophe ; Carreton, Andoni L. ; Harnie, Dries ; Pinte, Kevin ; Meuter, Wolfgang D.: AmbientTalk: programming responsive mobile peer-to-peer applications with actors. In: *Computer Languages, Systems and Structures* 40 (2014), Nr. 3–4, S. 112–136. – URL http://www.sciencedirect.com/science/article/pii/S1477842414000335. – ISSN 1477-8424

[Dahl and Nygaard 1966] Dahl, Ole-Johan ; Nygaard, Kristen: SIMULA: An ALGOL-based Simulation Language. In: *Commun. ACM* 9 (1966), September, Nr. 9, S. 671–678. – URL http://doi.acm.org/10.1145/365813.365819. – ISSN 0001-0782

[Dawson and Straub 2016] Dawson, Chris ; Straub, Ben: *Building tools with GitHub: Customize Your Workflow*. 1st. O'Reilly Media, 2016. – ISBN 978-1491933503

[De Volder 2006] De Volder, Kris: JQuery: A Generic Code Browser with a Declarative Configuration Language. In: *Proceedings of the 8th International Conference on Practical Aspects of Declarative Languages*. Berlin, Heidelberg : Springer-Verlag, 2006 (PADL'06), S. 88–102. – URL http://dx.doi.org/10.1007/11603023_7. – ISBN 3-540-30947-0, 978-3-540-30947-5

[DeLine et al. 2012] DeLine, Robert ; Bragdon, Andrew ; Rowan, Kael ; Jacobsen, Jens ; Reiss, Steven P.: Debugger canvas: industrial experience with the code bubbles paradigm. In: *Proceedings of the 2012 International Conference on Software Engineering*. Piscataway, NJ, USA : IEEE Press, 2012 (ICSE 2012), S. 1064–1073. – URL http://dl.acm.org/citation.cfm?id=2337223.2337362. – ISBN 978-1-4673-1067-3

[Denker et al. 2007] Denker, Marcus ; Ducasse, Stéphane ; Lienhard, Adrian ; Marschall, Philippe: Sub-Method Reflection. In: *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007* Bd. 6/9, ETH, Oktober 2007, S. 231–251. – URL http://www.jot.fm/contents/issue_2007_10/paper14.html. – ISSN 1660-1769

[Devanbu 1992] Devanbu, Premkumar T.: GENOA: A Customizable Language- and Front-end Independent Code Analyzer. In: *Proceedings of the 14th International Conference on Software Engineering*. New York, NY, USA : ACM, 1992 (ICSE '92), S. 307–317. – URL http://doi.acm.org/10.1145/143062.143148. – ISBN 0-89791-504-6

[Devanbu et al. 1996] Devanbu, Premkumar T. ; Rosenblum, David S. ; Wolf, Alexander L.: Generating Testing and Analysis Tools with Aria. In: *ACM Trans. Softw. Eng. Methodol.* 5 (1996), Januar, Nr. 1, S. 42–62. – URL http://doi.acm.org/10.1145/226155.226157. – ISSN 1049-331X

[Dey 2001] Dey, Anind K.: Understanding and Using Context. In: *Personal Ubiquitous Computing* 5 (2001), Nr. 1, S. 4–7. – ISSN 1617-4909

[Dijkstra 1972] Dijkstra, Edsgar W.: The Humble Programmer. In: *Commun. ACM* 15 (1972), Nr. 10, S. 859–866. – URL http://www.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html. – ISSN 0001-0782

[Dit et al. 2012] Dit, Bogdan ; Revelle, Meghan ; Gethers, Malcom ; Poshyvanyk, Denys: Feature location in source code: a taxonomy and survey. In: *Journal of Software: Evolution and Process* (2012), S. n/a–n/a. – URL http://www.cs.wm.edu/~denys/pubs/JSME-FL-SurveyCRCV1.pdf. – ISSN 2047-7481

[Donzeau-Gouge et al. 1980] Donzeau-Gouge, Véronique ; Huet, Gérard ; Lang, Bernard ; Kahn, Gilles: Programming environments based on structured editors : the Mentor experience / INRIA. URL https://hal.inria.fr/inria-00076535, 1980 (RR-0026). – Research Report

[Ducassé 1999] Ducassé, Mireille: Coca: An Automated Debugger for C. In: *International Conference on Software Engineering*, 1999, S. 154–168

[Ducasse et al. 2006] Ducasse, Stéphane ; Gîrba, Tudor ; Wuyts, Roel: Object-Oriented Legacy System Trace-based Logic Testing. In: *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, IEEE Computer Society Press, 2006, S. 35–44. – URL http://scg.unibe.ch/archive/papers/Duca06aTestLogtestingCSMR.pdf

[Ducasse et al. 2004] Ducasse, Stéphane ; Lienhard, Adrian ; Renggli, Lukas: Seaside — a Multiple Control Flow Web Application Framework. In: *Proceedings of 12th International Smalltalk Conference (ISC'04)*, URL http://scg.unibe.ch/archive/papers/Duca04eSeaside.pdfhttp://www.iam.unibe.ch/publikationen/techreports/2004/iam-04-008, September 2004, S. 231–257

[Eichberg and Schäfer 2004] Eichberg, Michael ; Schäfer, Thorsten: XIRC: Cross-artifact Information Retrieval [GPCE]. In: *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA : ACM, 2004 (OOPSLA '04), S. 43–44. – URL http://doi.acm.org/10.1145/1028664.1028688. – ISBN 1-58113-833-4

[Eisenberg et al. 2010] Eisenberg, Daniel S. ; Stylos, Jeffrey ; Myers, Brad A.: Apatite: A New Interface for Exploring APIs. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA : ACM, 2010 (CHI '10), S. 1331–1334. – URL http://doi.acm.org/10.1145/1753326.1753525. – ISBN 978-1-60558-929-9

[Erlingsson et al. 2011] Erlingsson, Úlfar ; Peinado, Marcus ; Peter, Simon ; Budiu, Mihai: Fay: Extensible Distributed Tracing from Kernels to Clusters. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. New York, NY, USA : ACM, 2011 (SOSP '11), S. 311–326. – URL http://doi.acm.org/10.1145/2043556.2043585. – ISBN 978-1-4503-0977-6

[Faith et al. 1997] Faith, Rickard E. ; Nyland, Lars S. ; Prins, Jan F.: KHEPERA: a system for rapid implementation of domain specific languages. In: *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*. Berkeley, CA, USA : USENIX Association, 1997, S. 19–19. – URL http://www.cs.unc.edu/~faith/faith-dsl-1997.ps

[Favre 2001] Favre, Jean-Marie: GSEE: a Generic Software Exploration Environment. In: *Proceedings of the 9th International Workshop on Program Comprehension*, IEEE, Mai 2001, S. 233–244

[Fayad et al. 1999] Fayad, Mohamed ; Schmidt, Douglas ; Johnson, Ralph: *Building Application Frameworks: Object Oriented Foundations of Framework Design*. Wiley and Sons, 1999

[Fischer et al. 1984] Fischer, C. N. ; Johnson, Gregory F. ; Mauney, Jon ; Pal, Anil ; Stock, Daniel L.: The Poe Language-based Editor Project. In: *SIGSOFT Softw. Eng. Notes* 9 (1984), April, Nr. 3, S. 21–29. – URL http://doi.acm.org/10.1145/390010.808245. – ISSN 0163-5948

[Fowler 2010] Fowler, Martin: *Domain-Specific Languages*. Addison-Wesley Professional, September 2010. – ISBN 0321712943

[Fritz et al. 2014] Fritz, Thomas ; Shepherd, David C. ; Kevic, Katja ; Snipes, Will ; Bräunlich, Christoph: Developers' Code Context Models for Change Tasks. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM, 2014 (FSE 2014), S. 7–18. – URL http://doi.acm.org/10.1145/2635868.2635905. – ISBN 978-1-4503-3056-5

[Golan and Hanson 1993] Golan, Michael ; Hanson, David R.: DUEL — A Very High-Level Debugging Language. In: *USENIX Winter*, 1993, S. 107–118

[Goldberg 1984] Goldberg, Adele: *Smalltalk 80: the Interactive Programming Environment*. Reading, Mass. : Addison Wesley, 1984. – ISBN 0-201-11372-4

[Grant 1999] Grant, Calum A. M.: *Software Visualization in Prolog*, Queens' College, Cambridge, Dissertation, Dezember 1999

[Gülcü and Stark 2003] Gülcü, Ceki ; Stark, Scott: *The complete log4j manual*. QOS. ch, 2003

[Habermann and Notkin 1986] Habermann, A. N. ; Notkin, David: Gandalf: Software development environments. In: *IEEE Transactions on Software Engineering* 12 (1986), Nr. 12, S. 1117–1127. – ISSN 0098-5589

[Han et al. 2012] Han, Shi ; Dang, Yingnong ; Ge, Song ; Zhang, Dongmei ; Xie, Tao: Performance Debugging in the Large via Mining Millions of Stack Traces. In: *Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA : IEEE Press, 2012 (ICSE '12), S. 145–155. – URL http://dl.acm.org/citation.cfm?id=2337223.2337241. – ISBN 978-1-4673-1067-3

[Hanson and Korn 1997] Hanson, David R. ; Korn, Jeffrey L.: A Simple and Extensible Graphical Debugger. In: *IN WINTER 1997 USENIX CONFERENCE*, 1997, S. 173–184

[Henriques et al. 2005] Henriques, P. R. ; Pereira, M. J. V. ; Mernik, M. ; Lenic, M. ; Gray, J. ; Wu, H.: Automatic generation of language-based tools using the LISA system. In: *Software, IEE Proceedings -* 152 (2005), Nr. 2, S. 54–69. – URL http://dx.doi.org/10.1049/ip-sen:20041317

[Hirschfeld et al. 2008] Hirschfeld, Robert ; Costanza, Pascal ; Nierstrasz, Oscar: Context-Oriented Programming. In: *Journal of Object Technology* 7 (2008), März, Nr. 3. – URL http://www.jot.fm/contents/issue_2008_03/article4.htmlhttp://www.jot.fm/issues/issue_2008_03/article4.pdf

[Ingalls et al. 2008] Ingalls, Daniel ; Palacz, Krzysztof ; Uhler, Stephen ; Taivalsaari, Antero ; Mikkonen, Tommi: The Lively Kernel A Self-supporting System on a Web Page. In: *Self-Sustaining Systems, First Workshop, S3 2008, Potsdam, Germany, May 15-16, 2008, Revised Selected Papers*, URL http://dx.doi.org/10.1007/978-3-540-89275-5_2, 2008, S. 31–50

[Ingalls 1981] Ingalls, Daniel H.: Design Principles Behind Smalltalk. In: *Byte* 6 (1981), August, Nr. 8, S. 286–298

[IntelliJSDK 2016] *IntelliJ Platform SDK DevGuide*. 2016. – URL http://www.jetbrains.org/intellij/sdk/docs/index.html. – http://www.jetbrains.org/intellij/sdk/docs/index.html

[Janzen and de Volder 2003] Janzen, Doug ; Volder, Kris de: Navigating and Querying Code Without Getting Lost. In: *AOSD'03: Proceedings of the 2nd International Conference on Aspect-oriented Software Development*. New York, NY, USA : ACM, 2003, S. 178–187. – ISBN 1-58113-660-9

[Johnson et al. 1989] Johnson, J. ; Roberts, T. L. ; Verplank, W. ; Smith, D. C. ; Irby, C. H. ; Beard, M. ; Mackey, K.: The Xerox Star: a retrospective. In: *Computer* 22 (1989), September, Nr. 9, S. 11–26. – ISSN 0018-9162

[Joy 1980] Joy, William: An Introduction to Display Editing with Vi. In: *In UNIX User's Manual Supplementary Documents,*, USENIX Association, 1980

[Kahn et al. 1983] Kahn, G. ; Lang, B. ; Mélèse, B. ; Morcos, E.: Metal: a formalism to specify formalisms". In: *Science of Computer Programming* 3 (1983), Nr. 2, S. 151 – 188. – URL http://www.sciencedirect.com/science/article/pii/0167642383900096. – ISSN 0167-6423

[Kersten and Murphy 2005] Kersten, Mik ; Murphy, Gail C.: Mylar: a degree-of-interest model for IDEs. In: *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. New York, NY, USA : ACM Press, 2005, S. 159–168. – ISBN 1-59593-042-6

[Khoo et al. 2013] Khoo, Yit P. ; Foster, Jeffrey S. ; Hicks, Michael: Expositor: scriptable time-travel debugging with first-class traces. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA : IEEE Press, 2013 (ICSE '13), S. 352–361. – URL http://dl.acm.org/citation.cfm?id=2486788.2486835. – ISBN 978-1-4673-3076-3

[Kiczales et al. 1997] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: Aksit, Mehmet (Hrsg.) ; Matsuoka, Satoshi (Hrsg.): *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming* Bd. 1241. Jyvaskyla, Finland : Springer-Verlag, Juni 1997, S. 220–242

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

[Klint et al. 2009] Klint, Paul ; Storm, Tijs van der ; Vinju, Jurgen: RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In: *Source Code Analysis and Manipulation, 2009. SCAM '09. Ninth IEEE International Working Conference on*, URL http://homepages.cwi.nl/~jurgenv/papers/SCAM-2009.pdf, 2009, S. 168–177

[Kniesel and Koch 2004] Kniesel, Günter ; Koch, Helge: Static Composition of Refactorings. In: *Sci. Comput. Program.* 52 (2004), August, Nr. 1-3, S. 9–51. – URL http://dx.doi.org/10.1016/j.scico.2004.03.002. – ISSN 0167-6423

[Ko et al. 2006] Ko, A.J. ; Myers, B.A. ; Coblenz, M.J. ; Aung, H.H.: An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. In: *Software Engineering, IEEE Transactions on* 32 (2006), Dezember, Nr. 12, S. 971 –987. – ISSN 0098-5589

[Ko et al. 2005] Ko, Andrew J. ; Aung, Htet ; Myers, Brad A.: 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*, 2005, S. 125–135. – ISBN 1-59593-963-2

[Ko and Myers 2008] Ko, Andrew J. ; Myers, Brad A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In: *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA : ACM, 2008 (ICSE '08), S. 301–310. – URL http://doi.acm.org/10.1145/1368088.1368130. – ISBN 978-1-60558-079-1

[Kolomvatsos et al. 2012] Kolomvatsos, Kostas ; Valkanas, George ; Hadjiefthymiades, Stathes: Debugging Applications Created by a Domain Specific Language: The IPAC Case. In: *J. Syst. Softw.* 85 (2012), April, Nr. 4, S. 932–943. – URL http://dx.doi.org/10.1016/j.jss.2011.11.1009. – ISSN 0164-1212

[Korn and Appel 1998] Korn, Jeffrey L. ; Appel, Andrew W.: Traversal-Based Visualization of Data Structures. In: *Proceedings of the 1998 IEEE Symposium on Information Visualization*. Washington, DC, USA : IEEE Computer Society, 1998 (INFOVIS '98), S. 11–18. – ISBN 0-8186-9093-3

[Kosar et al. 2014] Kosar, Tomaz ; Mernik, Marjan ; Gray, Jeff ; Kos, Tomaz: Debugging measurement systems using a domain-specific modeling language. In: *Computers in Industry* 65 (2014), Nr. 4, S. 622 – 635. – URL http://www.sciencedirect.com/science/article/pii/S0166361514000293. – ISSN 0166–3615

[Kubelka et al. 2015] Kubelka, Juraj ; Bergel, Alexandre ; Chiş, Andrei ; Gîrba, Tudor ; Reichhart, Stefan ; Robbes, Romain ; Syrel, Aliaksei: On Understanding How Developers Use the Spotter Search Tool. In: *Proceedings of 3rd IEEE Working Conference on Software Visualization - New Ideas and Emerging Results*, IEEE, September 2015 (VISSOFT-NIER'15), S. 145–149. – URL http://scg.unibe.ch/archive/papers/Kube15a-OnUnderstandingHowDevelopersUseTheSpotterSearchTool.pdf

[Lamb 1987] Lamb, David A.: IDL: Sharing Intermediate Representations. In: *ACM Trans. Program. Lang. Syst.* 9 (1987), Juli, Nr. 3, S. 297–318. – URL http://doi.acm.org/10.1145/24039.24040. – ISSN 0164-0925

[LaToza and Myers 2010] LaToza, Thomas D. ; Myers, Brad A.: Developers Ask Reachability Questions. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA : ACM, 2010 (ICSE '10), S. 185–194. – URL http://doi.acm.org/10.1145/1806799.1806829. – ISBN 978-1-60558-719-6

[Lee et al. 2009] Lee, Byeongcheol ; Hirzel, Martin ; Grimm, Robert ; McKinley, Kathryn S.: Debug All Your Code: Portable Mixed-environment Debugging. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. New York, NY, USA : ACM, 2009 (OOPSLA

'09), S. 207–226. – URL http://doi.acm.org/10.1145/1640089.1640105. – ISBN 978-1-60558-766-0

[Lehman 1980] Lehman, Manny: Programs, Life Cycles, and Laws of Software Evolution. In: *Proceedings of the IEEE* 68 (1980), September, Nr. 9, S. 1060–1076. – URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1456074

[Lencevicius et al. 1997] Lencevicius, Raimondas ; Hölzle, Urs ; Singh, Ambuj K.: Query-Based Debugging of Object-Oriented Programs. In: *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming (OOPSLA'97)*. New York, NY, USA : ACM, 1997, S. 304–317. – ISBN 0-89791-908-4

[Li and Thompson 2012] Li, Huiqing ; Thompson, Simon: Let's Make Refactoring Tools User-extensible! In: *Proceedings of the Fifth Workshop on Refactoring Tools*. New York, NY, USA : ACM, 2012 (WRT '12), S. 32–39. – URL http://doi.acm.org/10.1145/2328876.2328881. – ISBN 978-1-4503-1500-5

[Lienhard et al. 2008] Lienhard, Adrian ; Gîrba, Tudor ; Nierstrasz, Oscar: Practical Object-Oriented Back-in-Time Debugging. In: *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)* Bd. 5142, Springer, 2008, S. 592–615. – URL http://scg.unibe.ch/archive/papers/Lien08bBackInTimeDebugging.pdf. – ECOOP distinguished paper award. – ISBN 978-3-540-70591-8

[Lincke and Hirschfeld 2013] Lincke, Jens ; Hirschfeld, Robert: User-evolvable Tools in the Web. In: *Proceedings of the 9th International Symposium on Open Collaboration*. New York, NY, USA : ACM, 2013 (WikiSym '13), S. 19:1–19:8. – URL http://doi.acm.org/10.1145/2491055.2491074. – ISBN 978-1-4503-1852-5

[Lindeman et al. 2011] Lindeman, Ricky T. ; Kats, Lennart C. ; Visser, Eelco: Declaratively Defining Domain-specific Language Debuggers. In: *SIGPLAN Not.* 47 (2011), Oktober, Nr. 3, S. 127–136. – URL http://doi.acm.org/10.1145/2189751.2047885. – ISSN 0362-1340

[Littman et al. 1987] Littman, David C. ; Pinto, Jeannine ; Letovsky, Stanley ; Soloway, Elliot: Mental models and software maintenance. In: *J. Syst. Softw.* 7 (1987), Dezember, Nr. 4, S. 341–355. – URL http://dx.doi.org/10.1016/0164-1212(87)90033-1. – ISSN 0164-1212

[Lozano et al. 2015] Lozano, Angela ; Mens, Kim ; Kellens, Andy: Usage contracts: Offering immediate feedback on violations of structural source-code regularities. In: *Science of Computer Programming* 105 (2015), S. 73 – 91. – URL http://www.sciencedirect.com/science/article/pii/S016764231500012X. – ISSN 0167-6423

[Maalej 2009] Maalej, Walid: Task-First or Context-First? Tool Integration Revisited. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2009

(ASE '09), S. 344–355. – URL http://dx.doi.org/10.1109/ASE.2009.36. – ISBN 978-0-7695-3891-4

[Maier et al. 2014] Maier, P. ; Stewart, R. ; Trinder, P.W.: Reliable scalable symbolic computation: The design of SymGridPar2. In: *Computer Languages, Systems and Structures* 40 (2014), Nr. 1, S. 19–35. – URL http://www.sciencedirect.com/science/article/pii/S1477842414000049. – Special issue on the Programming Languages track at the 28th {ACM} Symposium on Applied Computing. – ISSN 1477-8424

[Maloney and Smith 1995] Maloney, John H. ; Smith, Randall B.: Directness and liveness in the Morphic user interface construction environment. In: *Proceedings of the 8th annual ACM symposium on User interface and software technology*. New York, NY, USA : ACM, 1995 (UIST '95), S. 21–28. – URL http://selflanguage.org/documentation/published/directness.pdf. – ISBN 0-89791-709-X

[Mannadiar and Vangheluwe 2011] Mannadiar, Raphael ; Vangheluwe, Hans: Debugging in Domain-Specific Modelling. In: Malloy, Brian (Hrsg.) ; Staab, Steffen (Hrsg.) ; Brand, Mark van den (Hrsg.): *Software Language Engineering* Bd. 6563. Springer Berlin Heidelberg, 2011, S. 276–285. – URL http://dx.doi.org/10.1007/978-3-642-19440-5_17. – ISBN 978-3-642-19439-9

[Marceau et al. 2007] Marceau, Guillaume ; Cooper, Gregory H. ; Spiro, Jonathan P. ; Krishnamurthi, Shriram ; Reiss, Steven P.: The Design and Implementation of a Dataflow Language for Scriptable Debugging. In: *Automated Software Engg.* 14 (2007), März, Nr. 1, S. 59–86. – URL http://dx.doi.org/10.1007/s10515-006-0003-z. – ISSN 0928-8910

[Martin et al. 2005] Martin, Mickael ; Livshits, Benjamin ; Lam, Monica S.: Finding application errors and security flaws using PQL: a program query language. In: *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. New York, NY, USA : ACM Press, 2005, S. 363–385

[Maruyama and Terada 2003] Maruyama, Kazutaka ; Terada, Minoru: Debugging with Reverse Watchpoint. In: *Proceedings of the Third International Conference on Quality Software (QSIC'03)*. Washington, DC, USA : IEEE Computer Society, 2003, S. 116. – ISBN 0-7695-2015-4

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

[Miles and Huberman 1994] Miles, Matthew B. ; Huberman, Michael: *Qualitative Data Analysis: An Expanded Sourcebook(2nd Edition)*. 2nd. Sage Publications, Inc, 1994. – ISBN 0803955405

[Minelli et al. 2015] Minelli, Roberto ; and, Andrea M. ; Lanza, Michele: 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*. Piscataway, NJ, USA : IEEE Press, 2015 (ICPC '15), S. 25–35. – URL `http://dl.acm.org/citation.cfm?id=2820282.2820289`

[Minelli et al. 2014] Minelli, Roberto ; Mocci, Andrea ; Lanza, Michele ; Kobayashi, Takashi: Quantifying Program Comprehension with Interaction Data. In: *Quality Software (QSIC), 2014 14th International Conference on*, Oktober 2014, S. 276–285. – ISSN 1550-6002

[Minelli et al. 2016] Minelli, Roberto ; Mocci, Andrea ; Robbes, Romain ; Lanza, Michele: Taming the IDE with Fine-grained Interaction Data. In: *Proceedings of ICPC 2016 (*24^{th} *International Conference on Program Comprehension)*, 2016, S. to appear

[Murphy et al. 2006] Murphy, Gail C. ; Kersten, Mik ; Findlater, Leah: How are Java software developers using the Eclipse IDE? In: *IEEE Softw.* 23 (2006), Juli, Nr. 4, S. 76–83. – URL `http://dx.doi.org/10.1109/MS.2006.105`. – ISSN 0740-7459

[Murphy et al. 2005] Murphy, Gail C. ; Kersten, Mik ; Robillard, Martin P. ; Čubranić, Davor: The Emergent Structure of Development Tasks. In: *Proceedings of the 19th European Conference on Object-Oriented Programming*. Berlin, Heidelberg : Springer-Verlag, 2005 (ECOOP'05), S. 33–48. – URL `http://dx.doi.org/10.1007/11531142_2`. – ISBN 3-540-27992-X, 978-3-540-27992-1

[Murphy-Hill et al. 2012] Murphy-Hill, Emerson ; Jiresal, Rahul ; Murphy, Gail C.: Improving Software Developers' Fluency by Recommending Development Environment Commands. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA : ACM, 2012 (FSE '12), S. 42:1–42:11. – URL `http://doi.acm.org/10.1145/2393596.2393645`. – ISBN 978-1-4503-1614-9

[Murphy-Hill et al. 2009] Murphy-Hill, Emerson ; Parnin, Chris ; Black, Andrew P.: How we refactor, and how we know it. In: *Proceedings of the 31st International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2009 (ICSE '09), S. 287–297. – ISBN 978-1-4244-3453-4

[Nayyeri 2008] Nayyeri, Keyvan: *Professional Visual Studio Extensibility*. Birmingham, UK, UK : Wrox Press Ltd., 2008. – ISBN 0470230843, 9780470230848

[Nierstrasz et al. 2005] Nierstrasz, Oscar ; Ducasse, Stéphane ; Gîrba, Tudor: The Story of Moose: an Agile Reengineering Environment. In: *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*. New York, NY, USA : ACM Press, September 2005, S. 1–10. – URL `http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf`. – Invited paper. – ISBN 1-59593-014-0

[Oliner et al. 2012] Oliner, Adam ; Ganapathi, Archana ; Xu, Wei: Advances and Challenges in Log Analysis. In: *Commun. ACM* 55 (2012), Februar, Nr. 2, S. 55–61. – URL `http://doi.acm.org/10.1145/2076450.2076466`. – ISSN 0001-0782

[Olsson et al. 1991] Olsson, Ronald A. ; Crawford, Richard H. ; Ho, W. W.: A Dataflow Approach to Event-based Debugging. In: *Softw. Pract. Exper.* 21 (1991), Februar, Nr. 2, S. 209–229. – URL http://dx.doi.org/10.1002/spe.4380210207. – ISSN 0038-0644

[Oualline 2001] Oualline, Steven: *Vi iMproved*. Thousand Oaks, CA, USA : New Riders Publishing, 2001. – ISBN 0735710015

[Ousterhout 1998] Ousterhout, John K.: Scripting: Higher Level Programming for the 21st Century. In: *IEEE Computer* 31 (1998), März, Nr. 3, S. 23–30. – URL http://www.cs.indiana.edu/classes/c102/read/Ousterhout.pdf

[Parr and Quong 1995] Parr, Terence J. ; Quong, Russell W.: ANTLR: A predicated-LL(k) parser generator. In: *Software Practice and Experience* 25 (1995), S. 789–810. – URL http://www.antlr.org/article/1055550346383/antlr.pdf

[Pavletic et al. 2015] Pavletic, Domenik ; Voelter, Markus ; Raza, SyedAoun ; Kolb, Bernd ; Kehrer, Timo: Extensible Debugger Framework for Extensible Languages. In: Puente, Juan A. de la (Hrsg.) ; Vardanega, Tullio (Hrsg.): *Reliable Software Technologies – Ada–Europe 2015* Bd. 9111. Springer International Publishing, 2015, S. 33–49. – URL http://dx.doi.org/10.1007/978-3-319-19584-1_3. – ISBN 978-3-319-19583-4

[Pawson 2004] Pawson, Richard: *Naked Objects*, Trinity College, Dublin, Ph.D. Thesis, 2004. – URL http://www.nakedobjects.org/downloads/Pawson%20thesis.pdf

[Petrenko et al. 2008] Petrenko, M. ; Rajlich, V. ; Vanciu, R.: Partial Domain Comprehension in Software Evolution and Maintenance. In: *The 16th IEEE Int'l Conf. on Program Comprehension*, IEEE, Juni 2008, S. 13–22

[Pirolli and Card 2005] Pirolli, Peter ; Card, Stuart: The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In: *Proceedings of International Conference on Intelligence Analysis*, 2005, S. 2–4

[Plaisant et al. 1995] Plaisant, Catherine ; Carr, David ; Shneiderman, Ben: Image-Browser Taxonomy and Guidelines for Designers. In: *IEEE Softw.* 12 (1995), März, Nr. 2, S. 21–32. – URL http://dx.doi.org/10.1109/52.368260. – ISSN 0740-7459

[Potanin et al. 2004] Potanin, Alex ; Noble, James ; Biddle, Robert: Snapshot Query-Based Debugging. In: *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*. Washington, DC, USA : IEEE Computer Society, 2004, S. 251. – ISBN 0-7695-2089-8

[Pothier et al. 2007] Pothier, Guillaume ; Tanter, Éric ; Piquer, José: Scalable Omniscient Debugging. In: *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)* 42 (2007), Nr. 10, S. 535–552. – ISSN 0362-1340

[Prähofer et al. 2013] Prähofer, Herbert ; Schatz, Roland ; Wirth, Christian ; Hurnaus, Dominik ; Mössenböck, Hanspeter: Monaco–A domain-specific language solution for reactive process control programming with hierarchical components. In: *Computer Languages, Systems and Structures* 39 (2013), Nr. 3, S. 67–94. – URL http://www.sciencedirect.com/science/article/pii/S1477842413000031. – ISSN 1477-8424

[Rajlich and Wilde 2002] Rajlich, Václav ; Wilde, Norman: The Role of Concepts in Program Comprehension. In: *Proceedings of the 10th International Workshop on Program Comprehension*. Washington, DC, USA : IEEE Computer Society, 2002 (IWPC '02), S. 271–. – URL http://dl.acm.org/citation.cfm?id=580131.857012. – ISBN 0-7695-1495-2

[Raymond 2003] Raymond, Eric S.: *The Art of UNIX Programming*. Pearson Education, 2003. – ISBN 0131429019

[Rebernak et al. 2009] Rebernak, Damijan ; Mernik, Marjan ; Wu, Hui ; Gray, Jeffrey G.: Domain-specific aspect languages for modularising crosscutting concerns in grammars. In: *IET Software* 3 (2009), Nr. 3, S. 184–200. – URL http://dx.doi.org/10.1049/iet-sen.2007.0114

[Reiss 1990] Reiss, S. P.: Interacting with the FIELD Environment. In: *Softw. Pract. Exper.* 20 (1990), Juni, Nr. S1, S. 89–115. – URL http://dx.doi.org/10.1002/spe.4380201308. – ISSN 0038-0644

[Reiss 1985] Reiss, S.P.: PECAN: Program Development Systems that Support Multiple Views. In: *IEEE Transactions on Software Engineering* SE-11 (1985), März, Nr. 3, S. 276–285

[Reiss 2012] Reiss, Steven P.: Plugging in and into Code Bubbles. In: *Proceedings of the Second International Workshop on Developing Tools As Plug-Ins*. Piscataway, NJ, USA : IEEE Press, 2012 (TOPI '12), S. 55–60. – URL http://dl.acm.org/citation.cfm?id=2667062.2667072. – ISBN 978-1-4673-1820-4

[Renggli et al. 2010a] Renggli, Lukas ; Ducasse, Stéphane ; Gîrba, Tudor ; Nierstrasz, Oscar: Practical Dynamic Grammars for Dynamic Languages. In: *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*. Malaga, Spain, Juni 2010, S. 1–4. – URL http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf

[Renggli et al. 2010b] Renggli, Lukas ; Gîrba, Tudor ; Nierstrasz, Oscar: Embedding Languages Without Breaking Tools. In: D'Hondt, Theo (Hrsg.): *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming* Bd. 6183. Maribor, Slovenia : Springer-Verlag, 2010, S. 380–404. – URL http://scg.unibe.ch/archive/papers/Reng10aEmbeddingLanguages.pdf. – ISBN 978-3-642-14106-5

[Reps and Teitelbaum 1984] Reps, Thomas ; Teitelbaum, Tim: The Synthesizer Generator. In: *SIGSOFT Softw. Eng. Notes* 9 (1984), April, Nr. 3, S. 42–48. – URL http://doi.acm.org/10.1145/390010.808247. – ISSN 0163-5948

[Ressia et al. 2012a] Ressia, Jorge ; Bergel, Alexandre ; Nierstrasz, Oscar: Object-Centric Debugging. In: *Proceedings of the 34rd international conference on Software engineering*, URL http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf, 2012 (ICSE '12)

[Ressia et al. 2012b] Ressia, Jorge ; Bergel, Alexandre ; Nierstrasz, Oscar ; Renggli, Lukas: Modeling Domain-Specific Profilers. In: *Journal of Object Technology* 11 (2012), April, Nr. 1, S. 1–21. – URL http://www.jot.fm/issues/issue_2012_04/article5.pdf. – ISSN 1660-1769

[Riel 1996] Riel, Arthur: *Object-Oriented Design Heuristics*. Boston MA : Addison Wesley, 1996. – 400 S

[Robillard et al. 2004] Robillard, Martin P. ; Coelho, Wesley ; Murphy, Gail C.: How Effective Developers Investigate Source Code: An Exploratory Study. In: *IEEE Trans. Softw. Eng.* 30 (2004), Dezember, Nr. 12, S. 889–903. – URL http://dx.doi.org/10.1109/TSE.2004.101. – ISSN 0098-5589

[Robitaille et al. 2000] Robitaille, S. ; Schauer, R. ; Keller, R.K.: Bridging program comprehension tools by design navigation. In: *Software Maintenance, 2000. Proceedings. International Conference on*, 2000, S. 22–32. – ISSN 1063-6773

[Roehm et al. 2012] Roehm, Tobias ; Tiarks, Rebecca ; Koschke, Rainer ; Maalej, Walid: How do professional developers comprehend software? In: *Proceedings of the 2012 International Conference on Software Engineering*. Piscataway, NJ, USA : IEEE Press, 2012 (ICSE 2012), S. 255–265. – ISBN 978-1-4673-1067-3

[Roever and Engelhardt 2008] Roever, Willem-Paul d. ; Engelhardt, Kai: *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. 1st. New York, NY, USA : Cambridge University Press, 2008. – ISBN 9780521103503

[Rozenberg and Beschastnikh 2014] Rozenberg, Daniel ; Beschastnikh, Ivan: Templated Visualization of Object State with Vebugger. In: *Proc. VISSOFT*, September 2014, S. 107–111

[S2py accessed June 3, 2016] *S2py Profiling Framework*, accessed June 3, 2016. – URL http://www.smalltalkhub.com/#!/~ObjectProfile/S2py. – http://www.smalltalkhub.com/#!/ÕbjectProfile/S2py

[Salvaneschi et al. 2012] Salvaneschi, Guido ; Ghezzi, Carlo ; Pradella, Matteo: Context-oriented programming: A software engineering perspective. In: *J. Syst. Softw.* 85 (2012), August, Nr. 8, S. 1801–1817. – URL http://dx.doi.org/10.1016/j.jss.2012.03.024. – ISSN 0164-1212

[Savidis and Koutsopoulos 2011] Savidis, Anthony ; Koutsopoulos, Nikos: Interactive Object Graphs for Debuggers with Improved Visualization, Inspection and Configuration Features. In: *Proceedings of the 7th International Conference on Advances in Visual Computing - Volume Part I*. Berlin, Heidelberg : Springer-Verlag, 2011 (ISVC'11), S. 259–268. – URL `http://dl.acm.org/citation.cfm?id=2045110.2045139`. – ISBN 978-3-642-24027-0

[Schafer et al. 2006] Schafer, Thorsten ; Eichberg, Michael ; Haupt, Michael ; Mezini, Mira: The SEXTANT Software Exploration Tool. In: *IEEE Trans. Softw. Eng.* 32 (2006), September, Nr. 9, S. 753–768. – URL `http://dx.doi.org/10.1109/TSE.2006.94`. – ISSN 0098-5589

[Schwarz 2011] Schwarz, Niko: DoodleDebug, Objects Should Sketch Themselves For Code Understanding. In: *Proceedings of the TOOLS 2011, 5th Workshop on Dynamic Languages and Applications (DYLA'11).*, URL `http://scg.unibe.ch/archive/papers/Schw11bDoodleDebug.pdf`, 2011

[Seidewitz 2003] Seidewitz, Ed: What Models Mean. In: *IEEE Software* 20 (2003), September, Nr. 5, S. 26–32

[Shepherd et al. 2012] Shepherd, David ; Damevski, Kostadin ; Ropski, Bartosz ; Fritz, Thomas: Sando: An Extensible Local Code Search Framework. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. New York, NY, USA : ACM, 2012 (FSE '12), S. 15:1–15:2. – URL `http://doi.acm.org/10.1145/2393596.2393612`. – ISBN 978-1-4503-1614-9

[Sillito et al. 2008] Sillito, Jonathan ; Murphy, Gail C. ; De Volder, Kris: Asking and Answering Questions during a Programming Change Task. In: *IEEE Trans. Softw. Eng.* 34 (2008), Juli, S. 434–451. – URL `http://portal.acm.org/citation.cfm?id=1446226.1446241`. – ISSN 0098-5589

[Sloane et al. 2014] Sloane, Anthony M. ; Roberts, Matthew ; Buckley, Scott ; Muscat, Shaun: Monto: A Disintegrated Development Environment. In: Combemale, Benoît (Hrsg.) ; Pearce, David J. (Hrsg.) ; Barais, Olivier (Hrsg.) ; Vinju, Jurgen J. (Hrsg.): *Proceedings of the Seventh International Conference on Software Language Engineering* Bd. 8706, Springer International Publishing, 2014, S. 211–220. – ISBN 978-3-319-11244-2

[Smith et al. 2015] Smith, Edward K. ; Bird, Christian ; Zimmermann, Thomas: 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, Mai 2015. – URL `http://research.microsoft.com/apps/pubs/default.aspx?id=238936`

[Smith et al. 1995] Smith, Randall B. ; Maloney, John ; Ungar, David: The Self-4.0 user interface: manifesting a system-wide vision of concreteness, uniformity, and flexibility. In: *SIGPLAN Not.* 30 (1995), Oktober, Nr. 10, S. 47–60. – ISSN 0362-1340

[Snodgrass and Shannon 1986] Snodgrass, Richard ; Shannon, Karen: Supporting flexible and efficient tool integration. In: *Advanced Programming Environments: Proceedings of an International Workshop*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1986, S. 290–313. – URL http://dx.doi.org/10.1007/3-540-17189-4_104. – ISBN 978-3-540-47347-3

[Spasojević et al. 2014] Spasojević, Boris ; Lungu, Mircea ; Nierstrasz, Oscar: Overthrowing the Tyranny of Alphabetical Ordering in Documentation Systems. In: *2014 IEEE International Conference on Software Maintenance and Evolution (ERA Track)*, URL http://scg.unibe.ch/archive/papers/Spas14b.pdf, September 2014, S. 511–515

[Stallman 1981] Stallman, Richard M.: EMACS the Extensible, Customizable Self-documenting Display Editor. In: *ACM SIGOA Newsletter* 2 (1981), April, Nr. 1-2, S. 147–156. – URL http://doi.acm.org/10.1145/1159890.806466. – ISSN 0737-819X

[Starke et al. 2009] Starke, J. ; Luce, C. ; Sillito, J.: Searching and skimming: An exploratory study. In: *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, September 2009, S. 157–166. – ISSN 1063-6773

[Syrel et al. 2015] Syrel, Aliaksei ; Chiş, Andrei ; Gîrba, Tudor ; Kubelka, Juraj ; Nierstrasz, Oscar ; Reichhart, Stefan: 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*. New York, NY, USA : ACM, 2015 (SPLASH Companion 2015), S. 54–55. – URL http://scg.unibe.ch/archive/papers/Syre15a-SpotterPosterAbstract.pdf. – ISBN 978-1-4503-3722-9

[Szyperski 2002] Szyperski, Clemens A.: *Component Software — Beyond Object-Oriented Programming*. Second Edition. Addison Wesley, 2002. – ISBN 0-201-74572-0

[Taeumel et al. 2014] Taeumel, Marcel ; Perscheid, Michael ; Steinert, Bastian ; Lincke, Jens ; Hirschfeld, Robert: 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*. New York, NY, USA : ACM, 2014 (Onward! 2014), S. 185–200. – URL http://doi.acm.org/10.1145/2661136.2661150. – ISBN 978-1-4503-3210-1

[Taeumel et al. 2012] Taeumel, Marcel ; Steinert, Bastian ; Hirschfeld, Robert: The VIVIDE programming environment: connecting run-time information with programmers' system knowledge. In: *Onward! '12*. New York, NY, USA : ACM, 2012, S. 117–126. – URL http://doi.acm.org/10.1145/2384592.2384604. – ISBN 978-1-4503-1562-3

[Tanter et al. 2002] Tanter, Éric ; Ségura-Devillechaise, Marc ; Noyé, Jacques ; Piquer, José: Altering Java Semantics via Bytecode Manipulation. In: *Proceedings of GPCE'02* Bd. 2487, Springer-Verlag, 2002, S. 283–89

[Tassey 2002] Tassey, G.: The economic impacts of inadequate infrastructure for software testing / National Institute of Standards and Technology. 2002. – Forschungsbericht

[Teitelbaum and Reps 1981] Teitelbaum, Tim ; Reps, Thomas: The Cornell Program Synthesizer: A Syntax-directed Programming Environment. In: *Commun. ACM* 24 (1981), September, Nr. 9, S. 563–573. – URL http://doi.acm.org/10.1145/358746.358755. – ISSN 0001-0782

[Tenma et al. 1988] Tenma, Takao ; Tsubotani, Hideaki ; Tanaka, Minoru ; Ichikawa, Tadao: A System for Generating Language-Oriented Editors. In: *IEEE Trans. Softw. Eng.* 14 (1988), August, Nr. 8, S. 1098–1109. – URL http://dx.doi.org/10.1109/32.7620. – ISSN 0098-5589

[Thomas and Nejmeh 1992] Thomas, Ian ; Nejmeh, Brian A.: Definitions of Tool Integration for Environments. In: *IEEE Softw.* 9 (1992), März, Nr. 2, S. 29–35. – URL http://dx.doi.org/10.1109/52.120599. – ISSN 0740-7459

[Ungar and Smith 1987] Ungar, David ; Smith, Randall B.: Self: The Power of Simplicity. In: *Proceedings OOPSLA '87, ACM SIGPLAN Notices* Bd. 22, Dezember 1987, S. 227–242

[Verwaest et al. 2011] Verwaest, Toon ; Bruni, Camillo ; Lungu, Mircea ; Nierstrasz, Oscar: 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*. New York, NY, USA : ACM, 2011 (OOPSLA '11), S. 959–972. – URL http://scg.unibe.ch/archive/papers/Verw11bFlexibleObjectLayouts.pdf. – ISBN 978-1-4503-0940-0

[Vessey 1986] Vessey, I: Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols. In: *IEEE Trans. Syst. Man Cybern.* 16 (1986), September, Nr. 5, S. 621–637. – URL http://dx.doi.org/10.1109/TSMC.1986.289308. – ISSN 0018-9472

[Visser 2005] Visser, Eelco: A Survey of Strategies in Rule-Based Program Transformation Systems. In: *Journal of Symbolic Computation* 40 (2005), Nr. 1, S. 831–873. – Special issue on Reduction Strategies in Rewriting and Programming

[Voelter 2014] Voelter, Markus: *Generic tools, specific languages*, Delft University of Technology, Ph.D. Thesis, 2014

[Wang Baldonado et al. 2000] Wang Baldonado, Michelle Q. ; Woodruff, Allison ; Kuchinsky, Allan: Guidelines for Using Multiple Views in Information Visualization. In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. New York, NY, USA : ACM, 2000 (AVI '00), S. 110–119. – URL http://doi.acm.org/10.1145/345513.345271. – ISBN 1-58113-252-2

[Wasserman 1990] Wasserman, Anthony I.: Tool Integration in Software Engineering Environments. In: *Proceedings of the International Workshop on Environments on Software Engineering Environments*. New York, NY, USA : Springer-Verlag New York, Inc., 1990, S. 137–149. – URL http://dl.acm.org/citation.cfm?id=111335.111346. – ISBN 3-540-53452-0

[Whittle et al. 2013] Whittle, Jon ; Hutchinson, John ; Rouncefield, Mark ; Burden, Håkan ; Heldal, Rogardt: *Model-Driven Engineering Languages and Systems: 16th International Conference, (MODELS 2013)*. Kap. Industrial Adoption of Model-Driven Engineering: Are the Tools Really the Problem?, S. 1–17. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – URL http://dx.doi.org/10.1007/978-3-642-41533-3_1. – ISBN 978-3-642-41533-3

[Winterbottom 1994] Winterbottom, Phil: ACID: A Debugger Built From a Language. In: *USENIX Technical Conference*, 1994, S. 211–222

[Wu et al. 2008] Wu, Hui ; Gray, Jeff ; Mernik, Marjan: Grammar-driven Generation of Domain-specific Language Debuggers. In: *Softw. Pract. Exper.* 38 (2008), August, Nr. 10, S. 1073–1103. – URL http://dx.doi.org/10.1002/spe.v38:10. – ISSN 0038-0644

[Wuyts 1996] Wuyts, Roel: Class-management using Logical Queries, Application of a Reflective User Interface Builder. In: Polak, I. (Hrsg.): *Proceedings of GRONICS '96*, URL http://scg.unibe.ch/archive/papers/Wuyt96a.pdf, 1996, S. 61–67

[Yang and Jiang 2007] Yang, Z. ; Jiang, M.: Using Eclipse as a Tool-Integration Platform for Software Development. In: *IEEE Software* 24 (2007), März, Nr. 2, S. 87–89. – ISSN 0740-7459

[Yuan et al. 2010] Yuan, Ding ; Mai, Haohui ; Xiong, Weiwei ; Tan, Lin ; Zhou, Yuanyuan ; Pasupathy, Shankar: SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In: *SIGARCH Comput. Archit. News* 38 (2010), März, Nr. 1, S. 143–154. – URL http://doi.acm.org/10.1145/1735970.1736038. – ISSN 0163-5964

[Zeller 1999] Zeller, Andreas: Yesterday, my program worked. Today, it does not. Why? In: *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*. London, UK : Springer-Verlag, 1999, S. 253–267. – ISBN 3-540-66538-2

[Zeller 2005] Zeller, Andreas: *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oktober 2005. – ISBN 1558608664

[Zeller and Lütkehaus 1996] Zeller, Andreas ; Lütkehaus, Dorothea: DDD — a free graphical front-end for Unix debuggers. In: *SIGPLAN Not.* 31 (1996), Nr. 1, S. 22–27. – ISSN 0362-1340

Curriculum Vitæ

Personal Information

Name	Andrei Chiş
Date of Birth	September 18, 1987
Place of Birth	Reşiţa, Romania
Nationality	Romanian

Education

2012–2016	Ph.D. in Computer Science, University of Bern, Switzerland Thesis: *Moldable Tools*
2010–2012	MSc in Software Engineering, "*Politehnica*" University of Timişoara, Romania Thesis: *Polymetric Maps*
2006–2010	BSc in Software Engineering, "*Politehnica*" University of Timişoara, Romania Thesis: *A Language for the Specification of Software Design Visualization*

Employment

2012–2016	Research Assistant, *University of Bern*, Switzerland
2011–2012	Software Engineer, *Intooitus, Ltd.*, Romania
2008–2009	Web Developer, *Epoint, Ltd.*, Romania

My complete Curriculum Vitæ is available at `www.andreichis.com`

Last update: September 19, 2016