

The Moldable Inspector

(Preprint *)

Andrei Chiş Oscar Nierstrasz
Aliaksei Syrel

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

Tudor Gîrba

tudorgirba.com

Abstract

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 ways to visualize and explore objects, depending on both the object and the current developer need. We propose the *Moldable Inspector*, a novel inspector model that enables developers to adapt the inspection workflow to suit their immediate needs by making the inspection context explicit, providing multiple interchangeable domain-specific views for each object, and supporting a workflow that groups together multiple levels of connected objects. We show that the Moldable Inspector can address multiple kinds of development needs involving a wide range of objects.

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

General Terms Tools, Languages, Design

Keywords Object inspector, Domain-specific tools, User interfaces, Programming environments

* In Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2015), October 25–30, 2015, Pittsburgh, PA, USA.
DOI: 10.1145/2814228.2814234

1. Introduction

Understanding the run-time behaviour 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 are an essential category of tools that 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 mainstream integrated development environments (IDEs) incorporate only object inspectors that 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 domain-specific objects like, folders/files, parsers, HTML/XML, database connections, compiled methods, users, accounts, graphical components, *etc.*, leads to a wide range of contextual questions. (For example: *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 [11] 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 [12] 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. 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 navigation mechanism and groups together multiple levels of connected objects.

To validate the proposed approach and show that it has practical applicability we implemented the Moldable Inspector concept in Pharo¹ as part of the Glamorous Toolkit project.² Based on this concrete implementation, we created more than 131 custom views for 84 objects belonging to 15 applications, requiring, on average, 9.2 lines of code per view. We also integrated the prototype implementation into the alpha version of the Pharo 4 IDE, 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 IDE in their day-to-day activi-

ties; the Moldable Inspector is now part of the Pharo 4 main release. Current feedback indicates that while the Moldable Inspector can take some getting used to, as it has a different navigation mechanism than traditional object inspectors, it can significantly improve the inspection process.

In a previous workshop paper we sketched the Moldable Inspector approach [7]. This paper extends that work with the following new contributions:

- Presenting an exploratory study into how developers perceive and use object inspectors resulting in four developer needs for object inspectors;
- Presenting an extended version of the Moldable Inspector model that takes into account the findings of the previous exploratory study;
- Showing 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.

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 [1], or by writing queries against the state [15, 17]. These approaches complement, and do not replace, the object inspector.

We selected a *sequential exploratory design* [10] approach for conducting our study. This is a mixed research methods strategy consisting of a qualitative investigation followed by a quantitative validation.

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.

2.1.1 Setup

We performed semi-structured interviews with software developers based on the template questions presented in Table 1. 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 conference³. Sixteen software developers attending the conference agreed to participate, on a voluntary basis. We collected 2 hours of recording with an interview lasting $7.6 \pm 2.6 (M \pm$

¹<http://pharo.org>

²<http://gt.moosetechnology.org>

³<http://www.esug.org/Conferences/2014>

Table 1: Questions for structuring the interviews.

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

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.

2.1.2 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.

We further proceeded to analyse the interviews using open coding [18]: 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 use 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 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 signifi-

cant time. Furthermore, losing the history of how one got to an object forces developers to repeatedly retrace their steps. Four participants expressed this concerned:

“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 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.

2.2 Quantitative Investigation

2.2.1 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 survey⁴. 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 (pharo.org and moosetechnology.org) 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 to software engineering (Table 2 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 — DN3 and 61 answers for DN4 from respondents whose

practical knowledge with object-oriented programming is shown in Table 2. 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.

Table 2: Background data about the survey respondents.

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

2.2.2 Analysis

Table 3 summarizes the results of the survey. Overall there was a strong tendency towards the *Strongly agree* and *Agree* answers; no respondent chose *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 use 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 only 77% 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 needed to explore dependencies based on more than object state included callbacks on graphical widgets and pointers.

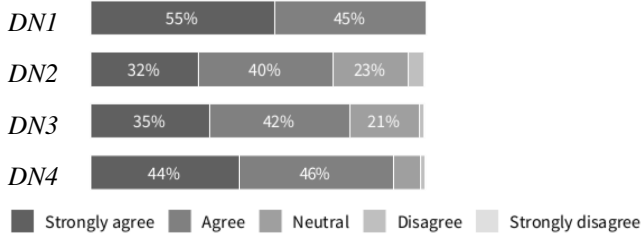
2.3 Threats to Validity

2.3.1 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

⁴<http://scg.unibe.ch/research/moldableinspector/survey>

Table 3: The results of the quantitative investigation.



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.

2.3.2 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.

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.

3. The Moldable Inspector in a Nutshell

To address the aforementioned developer needs we propose the Moldable Inspector, a model (Figure 1) for constructing object inspectors that can be adapted during the inspection

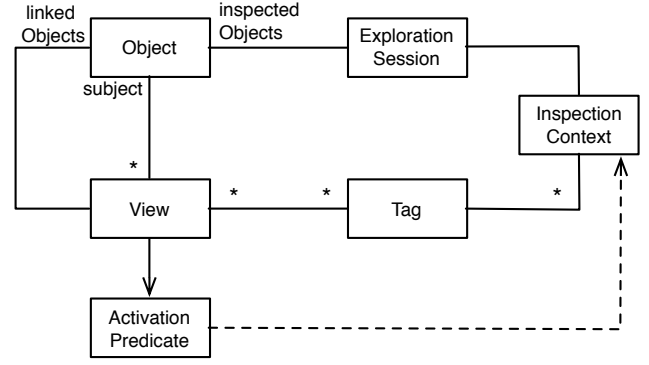


Figure 1: The structure of the Moldable Inspector model: an object can have multiple views grouped using tags, and filtered using activation predicates; inspected objects are grouped in an exploration session; the inspection context consists of multiple tags and an exploration session; the inspection context is used to filter views.

process to suit the immediate needs of developers. This is achieved in two steps:

- (i) developers create custom extensions for viewing and exploring their domain objects;
- (ii) at run time the Moldable Inspector selects extensions appropriate for the current objects and developer needs.

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.*, renderer, canvas).

3.2 Enabling Customization

The Moldable Inspector model enables custom extensions through two operators:

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

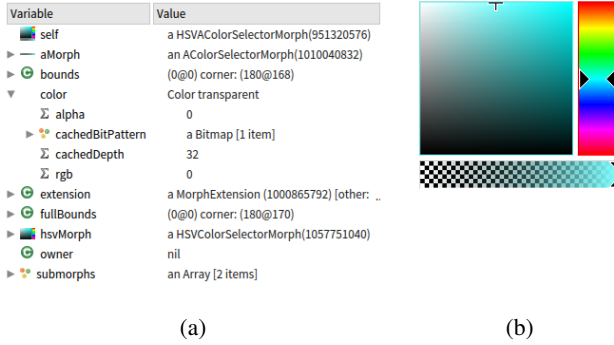


Figure 2: Two views for a widget: (a) state; (b) visual representation.

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

3.2.1 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. *DNI* 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 2a), the code of its class, its visual representation (Figure 2b), its keyboard shortcuts, the other graphical objects that it contains, or what objects hold a reference to it.

3.2.2 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 constructing/locating those objects using snippets of code executed in the context of the displayed object (e.g., In Figure 3 a developer navigates from a widget to the context menu of that widget, by using a code snippet to create the menu).

3.3 Inspection Context

One problem is still not addressed: *How does a developer select the right views for her current needs?* To solve it the Moldable Inspector explicitly models the current inspection context and uses it to determine what views to show. This is achieved using three operators:

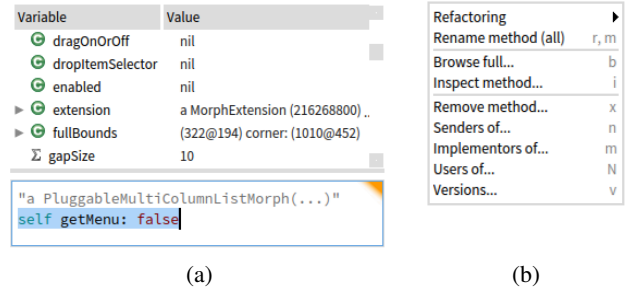


Figure 3: Navigating from a widget (a) to its menu (b). The menu is not stored in an instance variable of the widget. It can only be constructed by invoking a method of the widget.

tag: group together views applicable for a development task;

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

activation predicate: determine if a view is valid or not based on the exploration session.

An inspection context consists of multiple tags and an exploration session. Activation predicates filter views based on the context.

3.3.1 Tags

Depending on the current developer needs not all available views are of interest; this is the main requirement captured by *DNI*. 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.

3.3.2 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

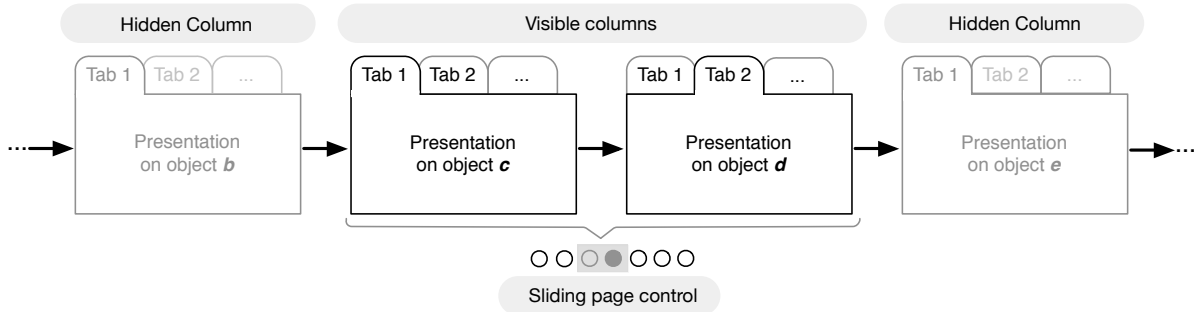


Figure 4: The Object Pager user interface in a nutshell: each object is displayed in a single column as a tabbed widgets that groups together a set of views; new objects are displayed to the right; an augmented scrollbar improves navigation. Objects are displayed in columns of equal size and it is not possible to reach a situation where a column is partially displayed, as the sliding bar always repositions to show full columns.

not related to the current task, they are kept in the session unless the developer removes them.

3.3.3 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*, that is, a boolean condition applied on the current inspection context before showing that view. Hence, an activation predicate can filter views based on the currently inspected object as well as based on the entire exploration session. This approach is similar to a solution proposed previously for activating custom debuggers at run time [8].

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.

3.4 Addressing the Initial Developer Needs

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.

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 discuss these issues in Section 6 and Section 7.

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).

DN4: Previously inspected objects are grouped into an exploration session. Section 4 discusses UI decisions concerning navigating and displaying an exploration session.

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 real-screen estate and reduce spatial maintenance effort.

4.1 Displaying Multiple Views for an Object

Baldonado *et al.* introduced eight rules for the design of systems having multiple views [25]. 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” [25].

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 Pager shows only one view for each object at a time and groups all available views of an object using tabs (Figure 4).

4.2 Representing an Exploration Session

Approaches displaying complete exploration sessions (e.g., by using tree/graph-based structures or matrices) can take considerably screen real estate and require developers to explicitly remove paths that are no longer of interest. Consider DebuggerCanvas [12], a debugger promoting a user interface based on the CodeBubble paradigm [5]. 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 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.

4.3 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 [19], navigation back and forth through an exploration path becomes an explicit issue. Simple scrollbars, while enabling fast movement between columns, have several shortcomings [2] when navigating through Miller columns. For example, it is difficult to tell that the scrollbar from Figure 5a 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 [2, 9] (following the *overview+detail* approach [20]) 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 5c); the sliding bar indicates the visible objects. Figure 5b shows how this approach is used to navigate through the same exploration path as in Figure 5a; now a developer can immediately see that the path has seven objects and that two objects are currently visible.

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 Ob-

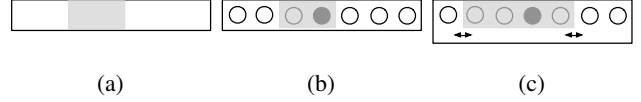


Figure 5: (a) Standard scrollbar with limited navigation support; (b) Improved scrollbar with overview support; (c) Increasing/decreasing the number of visible objects.

ject 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*.

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 details at views for two specific objects.

5.1.1 Generic Views


Every object has a *Raw* view (Figure 6 — 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 9).

5.1.2 Multiple Views for Graphical Objects

As highlighted in Section 3, 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 [16] 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 to every morph object two specific views showing their visual representation (*Morph* view, Figure 6) and structure of submorphs (*Submorphs* view, Figure 6).

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 elements⁶). 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.

⁵http://en.wikipedia.org/wiki/Miller_columns

⁶<http://pharo.fogbugz.com/f/cases/15227>

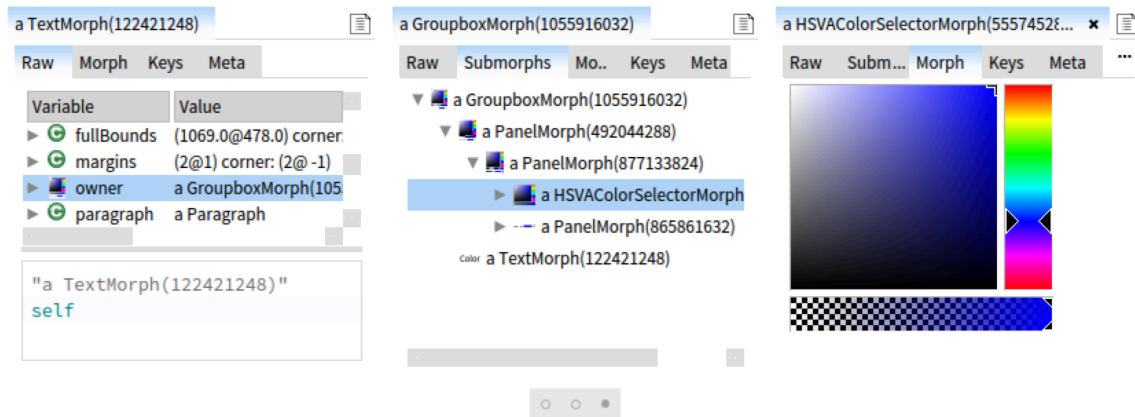


Figure 6: An exploration session involving multiple graphical components (*i.e.*, morphs).

5.1.3 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 debug⁷ without proper tool support, as 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).

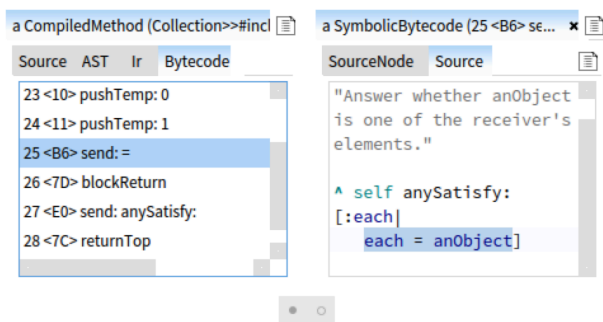


Figure 7: Exploring how bytecode maps to code.

To improve the inspection of compiled code we created, together with the developers of the Pharo compiler, four specific views to `CompiledMethod` objects:

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

- *Source code*: the original source code from which the `CompiledMethod` object was generated;
- *AST*: the AST obtained by parsing the source code;
- *IR*: the intermediate representation (IR) obtained from the AST;
- *Bytecode*: the bytecode stored by the method object (*Bytecode* view, Figure 7).

5.2 Navigating Through Connected Objects

Since navigation between objects based only on object state reduces the available space to accessible objects the Moldable Inspector 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*).

5.2.1 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 view⁸. To address this we added a dedicated view showing a list of keyboard shortcuts associated with the morph (*Keys* view, Figure 8). By 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 8).

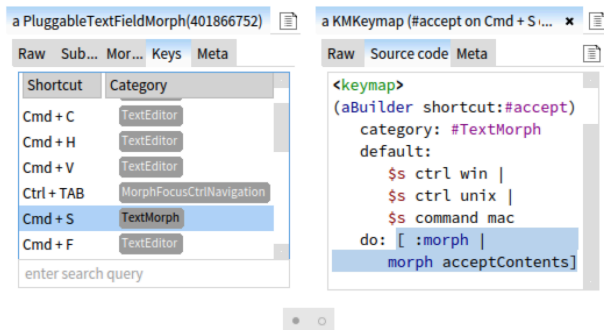


Figure 8: Using specific views to browse the code that gets executed when a user presses CMD+S.

5.2.2 Navigating to New Objects

While a CompiledMethod object has views showing its bytecodes and source code, addressing the bugs mentioned in Section 5.1.3 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 7).

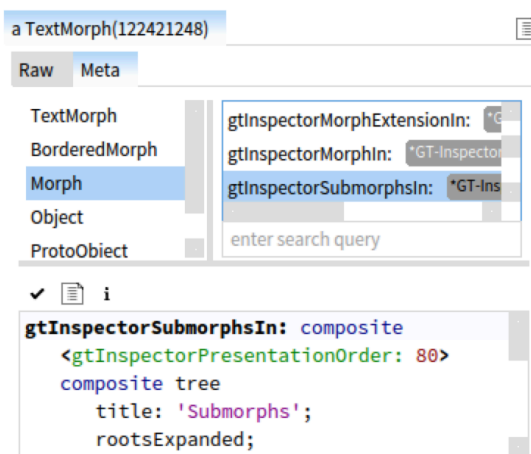


Figure 9: Accessing an object's source code.

5.2.3 Using Code to Guide the Navigation Process

Constructing and previewing queries over relational databases is typically done in dedicated database client tools that

⁸ pharo.fogbugz.com/f/cases/14845/, <http://bit.ly/1JQ216X>

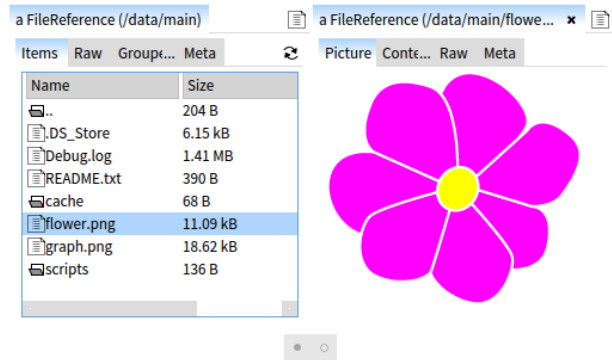


Figure 11: Browsing the content of a folder.

are far away from the development environment. However, when working with relational data, querying is a common activity during software development.

Opening in Pharo a connection to a Postgres⁹ 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 10). 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 10 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.

5.3 Selecting Views Based on the Inspection Context

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

5.3.1 Selecting Views Using Activation Predicates

Viewing the internal representation of an object modelling 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 FileReference object only gives information about the location of the file/folder; the content is not accessible.

⁹ <http://www.postgresql.org>

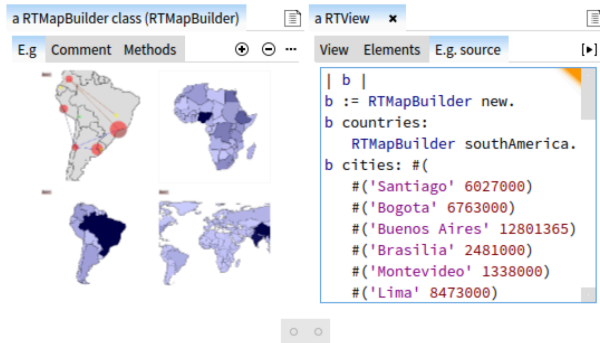


Figure 13: Browsing the examples of class.

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 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.

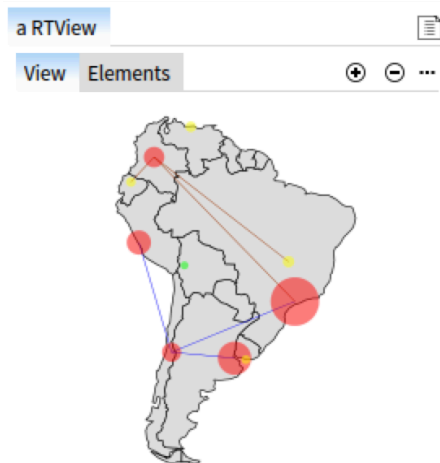


Figure 14: Inspecting an example object in isolation.

For example, in Figure 13 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 14).

6. Implementation Aspects

We integrated the current implementation into Moose in 2013 and into the alpha version of Pharo 4 (the language

on top of which Moose is built) in October 2014, replacing the previous object inspector. We consider the current implementation to be stable given that it is currently used by hundreds of people on a daily basis. In this section we explore various design decisions.

6.1 Constructing a Specific View for an Object

There are numerous approaches to create specific views for objects ranging from fully automatic generation of views based on the structure of objects [11] to manual creation using code snippets [23]. Also, several IDEs allow developers to customize views using dedicated tools (*e.g.*, *Custom Data Viewers* in IntelliJ).

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 such a graphical object. Given that developers reported the need to be able to easily extend 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. For example, lines 4-9 show how to instantiate a tree view showing the submorphs of a morph (*Submorphs* view, Figure 6).

```

1 gtInspectorDisplaySubmorphsOn: aCanvas in: aContext
2 <gtInspectorPresentationOrder: 80>
3 <gtInspectorTag: #custom>
4   ↑ aCanvas tree
5     title: 'Submorphs';
6     rootsExpanded;
7     display: [:rootMorph | {rootMorph}];
8     format: [:morph | morph printString];
9     children: [:morph | morph submorphs];
10    when: [:morph | morph submorphs notEmpty]

```

The proposed API also makes it easy to integrate more elaborate views created using a visualization library (Roassal [4]) and a data browsers library (Glamour [13]).

6.2 Attaching Multiple Views to an Object

In most object-oriented languages objects can represent themselves in a textual form with the help of a method in the class of the object to generate that textual representation (*e.g.*, `toString` in Java). Extending this idea, we make an object responsible for representing itself in multiple ways by defining within its class methods that construct specific views. This keeps view code together with that of the objects. 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; the 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 9 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.

6.3 Supporting Tags

We opted to defined tags using parametrized annotations: a view is added to a tag by marking the method creating that view with the tag's annotation (in the above snippet, line 3 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, that would require developers to find and update the tag definition when adding/changing views.

6.4 The Moldable Inspector in Other Languages

The Moldable Inspector is a generic model that does not dependent on any particular OO programming language and IDE. While the current prototype was developed in Pharo, following the discussion from this section, we view 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 would however require a different solution for grouping views (*e.g.*, putting the views for an object type in a dedicated class).

7. Discussion

7.1 A Taxonomy of Views

To validate the API proposed for creating views (Section 6.1) and show that it has practical applicability we built, together with the developers of several frameworks, 131 views covering 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 currently 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

Table 4: Types of views used to display objects in the current implementation.

View type	Example	Number of views
<i>List</i>	<i>Pointers</i> view, Figure 12	29
<i>Tree</i>	<i>Submorphs</i> view, second object — Figure 6	9
<i>Table</i>	<i>Keys</i> view, first object — Figure 9	26
<i>Text</i>	<i>SQL</i> view, first object — Figure 10	13
<i>Source code</i>	<i>Source code</i> view, second object — Figure 9	12
<i>Morph</i>	<i>Morph</i> view, third object — Figure 6	10
<i>Roassal view</i>	<i>View</i> view, Figure 14	18
<i>Glamour view</i>	<i>Raw</i> view, first object — Figure 6	14

views based on the API for creating views (*i.e.*, list, tree, 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), 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.

7.2 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. 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 addressed the second developer need identified in Section 2.

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 API¹¹. Currently we offer a browser for exploring all extensions present in the IDE, as well as tutorials on how to extend the inspector¹².

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>

7.3 Implications

This paper 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 directly in the debugger/inspector. By reducing the cost of creating views we enable developers to adapt the inspector to their own domain-objects. Currently several libraries from the Pharo ecosystem took advantage of this possibility (e.g., OpalCompiler¹³, MongoTalk¹⁴).

7.4 Open Questions

In the current prototype we rely on manual creation of views using a dedicated API. While the cost of creating views is low in terms of lines of code, developers have to manually update views as they make changes to code. This raises the following research question: *How to update a view as the object it represents evolves?* One approach for addressing this research question is to automatically generate views. This was shown to work well when generating views based only on the internal representation of an object [11]. Nevertheless, not all the views that are in our current implementation follow this approach; some display domain-specific aspects that have nothing to do with the internal representation of an object (e.g., *Picture* view, Figure 12). This raises another research question: *How to automatically generate views that capture domain-specific aspects?*

While we showed that the Moldable Inspector can support a wide range of different workflows, an empirical evaluation measuring the effectiveness of our approach is currently missing. We consider that one can evaluate the Moldable Inspector on two directions starting from the following research questions: (i) *Does the Moldable Inspector increase the efficiency of developers when performing program comprehension tasks?* and (ii) *Can developers easily create custom views for exploring domain objects?* In answering these questions we are currently looking into instrumenting the tool to get a better understanding of how developers use the inspector in practice and performing controlled experiments with software developers.

8. 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 [24] 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 X¹⁵ 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.

Eclipse IDE¹⁶ 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* and dynamically select one based on a given property of an object. The Moldable Inspector allows each object to have multiple views.

NetBeans¹⁷ 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. IntelliJ¹⁸ 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 behaviour 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 Studio¹⁹ 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.

¹³<http://www.smalltalkhub.com/#!/~Pharo/Opal>

¹⁴<http://www.smalltalkhub.com/#!/~MongoTalkTeam/mongotalk>

¹⁵<http://www.exept.de/en/products/smalltalk-x.html>

¹⁶<http://eclipse.org/ide>

¹⁷<http://netbeans.org>

¹⁸<http://jetbrains.com/idea>

¹⁹<http://visualstudio.com>

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 Firebug²⁰ or *Elements tab* in Chrome DevTools²¹. 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 includes 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 [11]. *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 [23] allows objects to have two custom representations (a summary view and a detailed view). Debugger [21] 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.* [3] present an extension to 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 [1, 22]. 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 [12] brings the Code Bubbles [5] 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 [14] 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, patterns 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 only used to decide if a view is applicable for a given object.

LIVE [6] 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.

9. 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.

While simple, the Moldable Inspector enables a wide range of different workflows. We showed that it 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/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 indeed this is achievable.

²⁰<http://getfirebug.com>

²¹<http://developer.chrome.com/devtools>

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Agile Software Assessment” (SNSF project Nr. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We thank the anonymous reviewers for their suggestions in improving this paper. We also gratefully acknowledge the financial support of the Swiss Group for ObjectOriented Systems and Environments (CHOOSE) and the European Smalltalk User Group (ESUG).

References

- [1] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proc. SOFTVIS*, pages 53–62, 2010.
- [2] J. Alexander, A. Cockburn, S. Fitchett, C. Gutwin, and S. Greenberg. Revisiting read wear: Analysis, design, and evaluation of a footprints scrollbar. In *Proc. SIGCHI, CHI*, pages 1665–1674. ACM, 2009.
- [3] B. Alsallakh, P. Bodesinsky, S. Miksch, and D. Nasser. Visualizing Arrays in the Eclipse Java IDE. In *Proc. CSMR*, pages 541–544, March 2012.
- [4] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. Agile visualization with Roassal. In *Deep Into Pharo*, pages 209–239. Square Bracket Associates, Sept. 2013.
- [5] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code Bubbles: a working set-based interface for code understanding and maintenance. In *Proc. CHI*, pages 2503–2512, 2010.
- [6] A. E. R. Campbell, G. L. Catto, and E. E. Hansen. Language-independent interactive data visualization. *SIGCSE Bull.*, 35 (1):215–219, Jan. 2003.
- [7] A. Chiş, T. Gîrba, and O. Nierstrasz. The Moldable Inspector: a framework for domain-specific object inspection. In *Proc. IWST*, 2014.
- [8] A. Chiş, T. Gîrba, and O. Nierstrasz. The Moldable Debugger: A framework for developing domain-specific debuggers. In *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 102–121. Springer International Publishing, 2014.
- [9] R. Chimera. Value bars: An information visualization and navigation tool for multi-attribute listings. In *Proc. CHI*, pages 293–294, 1992.
- [10] J. W. Creswell and Vicki. *Designing and Conducting Mixed Methods Research*. Sage Publications, Inc, Aug. 2006.
- [11] J. H. Cross, II, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, and L. N. Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *Trans. Comput. Educ.*, 9(2):13:1–13:32, June 2009.
- [12] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. P. Reiss. Debugger Canvas: industrial experience with the code bubbles paradigm. In *Proc. ICSE*, pages 1064–1073. IEEE Press, 2012.
- [13] T. Gîrba, A. Bergel, D. Cassou, S. Ducasse, and J. Laval. Glamour. In *Deep Into Pharo*, pages 191–207. Square Bracket Associates, Sept. 2013.
- [14] J. Korn and A. Appel. Traversal-based visualization of data structures. In *Information Visualization, 1998. Proceedings. IEEE Symposium on*, pages 11–18, Oct 1998.
- [15] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proc. OOPSLA*, pages 304–317, 1997.
- [16] J. H. Maloney and R. B. Smith. Directness and liveness in the Morphic user interface construction environment. In *Proc. UIST*, pages 21–28. ACM, 1995.
- [17] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proc. OOPSLA*, pages 363–385. ACM, 2005.
- [18] M. B. Miles and M. Huberman. *Qualitative Data Analysis: An Expanded Sourcebook(2nd Edition)*. Sage Publications, Inc, 2nd edition, 1994.
- [19] R. Minelli, A. Mocci, M. Lanza, and L. Baracchi. Visualizing developer interactions. In *Proc. VISSOFT*, pages 147–156, 2014.
- [20] C. Plaisant, D. Carr, and B. Shneiderman. Image-browser taxonomy and guidelines for designers. *Software, IEEE*, 12 (2):21–32, Mar 1995.
- [21] D. Rozenberg and I. Beschastnikh. Templated visualization of object state with Vebgger. In *Proc. VISSOFT*, pages 107–111, Sept 2014.
- [22] A. Savidis and N. Koutsopoulos. Interactive object graphs for debuggers with improved visualization, inspection and configuration features. In *Proc. ISVC*, pages 259–268, 2011.
- [23] N. Schwarz. DoodleDebug, objects should sketch themselves for code understanding. In *Proc. DYLA*, 2011.
- [24] R. B. Smith, J. Maloney, and D. Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *Proc. OOPSLA*, pages 47–60, 1995.
- [25] M. Q. Wang Baldonado, A. Woodruff, and A. Kuchinsky. Guidelines for using multiple views in information visualization. In *Proc. AVI*, pages 110–119. ACM, 2000.

A. Appendix A

<i>Participant</i>	<i>Answer</i>
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

Table 5: Answers provided by the interview participants to the question: *What is an object inspector for you?* Two participants did not provide an answer.