The Moldable Inspector: a framework for domain-specific object inspection

Andrei Chiş
University of Bern, Switzerland
andrei@iam.unibe.ch

Tudor Gîrba
CompuGroup Medical Schweiz AG
tudor@tudorgirba.com

Oscar Nierstrasz
University of Bern, Switzerland
scg.unibe.ch/oscar

Abstract

Answering run-time questions in object-oriented systems involves reasoning about and exploring connections between multiple objects. Developer questions exercise various aspects of an object and require multiple kinds of interactions depending on the relationships between objects, the application domain and the differing developer needs. Nevertheless, traditional object inspectors, the essential tools often used to reason about objects, favor a generic view that focuses on the low-level details of the state of individual objects. This leads to an inefficient effort, increasing the time spent in the inspector. To improve the inspection process, we propose the *Moldable Inspector*, a novel approach for an extensible object inspector. The Moldable Inspector allows developers to look at objects using multiple interchangeable presentations and supports a workflow in which multiple levels of connecting objects can be seen together. Both these aspects can be tailored to the domain of the objects and the question at hand. We further exemplify how the proposed solution improves the inspection process, introduce a prototype implementation and discuss new directions for extending the Moldable Inspector.

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

1. Introduction

Objects integrate data and behavior to model relevant concepts from application domains. Computation is further expressed in terms of interactions between objects. Therefore, understanding objects along with their relationships is critical for developing and evolving object-oriented applications.

Due to their nature, object-oriented applications have a dual representation: static, in terms of source code, and dynamic, in terms of objects. Developers often focus on the static source code to gain insight into the dynamic representation, however, due to mechanisms like inheritance, polymorphism and dynamic binding, understanding objects and relations between objects based only on a static view of the code poses many difficulties [5].

Object inspectors offer a better alternative as they enable developers to explore the actual run-time objects. Inspectors offer generic mechanisms to display and explore the state of an arbitrary object, however they do not take into account the varying needs of developers that could benefit from tailored ways to view and explore object state.

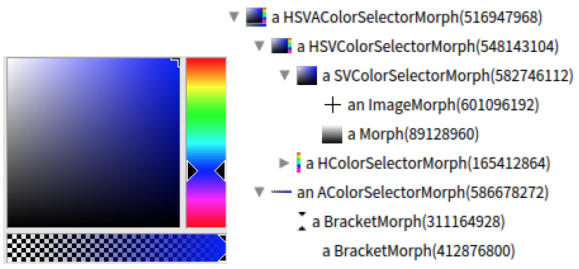
Consider the question of determining whether or not a graphical component that has a certain visual characteristic is present within a list of graphical components. A visual representation of the graphical component provides more insight than just looking at the state of that component. A different question from a different domain consists in determining if an object representing a directory contains a particular file. An object inspector showing the list of files contained by the directory object can provide a straightforward answer.

A generic solution focusing only on object state, while universally applicable, fails to highlight what aspects of an object are important in a given development context. By a development context we understand a specific run-time question from a specific domain (*e.g.*, fixing a performance problem in a parser, finding a memory leak in a graphical framework). This mismatch increases the inspection time as developers have to manually search for what is relevant for their particular contexts.

Traditional object inspectors exhibit this problem since:

- They rely on predefined, generic state-based presentations for displaying objects, thus ignoring significant differences between objects in different domains;
- They focus on individual objects, thus providing only rigid mechanisms for exploring relations between objects.

These problems can be solved if instead of using a generic object inspector a developer relies on an object inspector that can easily be adapted to the development context at hand (*i.e.*, both the application domain and the developer question). We consequently propose the Moldable Inspector, a



(a) (b)

Figure 1: Two distinct way to look at a morph object for choosing colors depending on the developer needs: (a) presentation showing the visual appearance; (b) presentation showing the structure (the tree of submorphs).

novel approach for an extensible object inspector that (1) allows developers to inspect at objects using multiple interchangeable presentations, and (2) provides a workflow in which multiple levels of connecting objects can be seen together, and navigation between objects is guided by the domain and the question. The Moldable Inspector further relies on the idea of using code to both steer the inspection process and to extend the existing presentations at inspection time.

To validate the proposed approach and show that it has practical applicability, we are developing a prototype implementation in Pharo[1], a modern Smalltalk environment. The current version of our implementation features most of the core functionality of the proposed framework in less than 500 lines of code. Furthermore, it has been used to create more than 70 extensions requiring, on average, 8 lines of code per extension. Its small size has two practical advantages: on the one hand it makes it easy to understand; on the other hand it makes the adaptation of the inspector to new run-time questions and domain objects affordable.

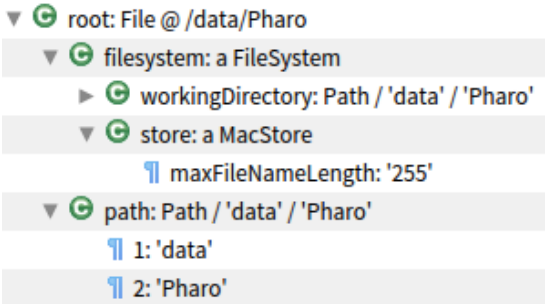
The contributions of this paper are as follows:

- Introducing the Moldable Inspector framework for defining a context-aware object inspector;
- Presenting the current prototype instantiation of the Moldable Inspector and discussing several implementation aspects;
- Proposing new directions for extending the Moldable Inspector framework.

2. Why basic inspectors are not enough

To successfully answer a run-time question in an object-oriented system, developers have to identify which objects are relevant for that question and understand those objects along with their interactions. One can anticipate neither what objects will be needed, nor what aspect of an object (*e.g.*, state, code, memory usage, dependencies) will be important for a given question. Thus, object inspectors viewing objects through generic state-based presentations and offering fixed

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



(a)

Name	Size	Creation
..	136	2013-04-02 14:10:55
pharo-vm	272	2014-06-18 15:27:47
.DS_Store	6148	2014-06-18 15:27:36
pharo	382	2014-06-18 15:27:49
pharo-ui	371	2014-06-18 15:27:49
Pharo.changes	17689307	2014-06-18 02:21:46
Pharo.image	43604012	2014-06-18 02:20:36
PharoDebug.log	20717	2014-06-18 15:30:02
vizualization.png	5876	2014-06-18 15:29:14
Workspace.st	73	2014-06-18 15:32:25

(b)

Figure 2: Two distinct ways to look at a directory object depending on the developer needs: (a) presentation showing the state of the object; (b) presentations showing the contained files and directories

mechanisms for navigating between objects are less suitable for answering run-time questions in object-oriented systems. This section exemplifies these problems.

2.1 Limitations of viewing objects through single views

In Pharo the Morph class is the root class for all graphical components. A morph can contain other morphs (*i.e.*, submorphs). If a developer wants to locate a morph object within a collection, she would benefit from its visual appearance (Figure 1a), but if she wants to debug issues related to the structure of the morph, she needs to see and explore the tree of submorphs (Figure 1b). Both of these representations are valid, but they serve different interests. This type of problem is not isolated and can be found in various other situations.

A different use case consists in inspecting objects modeling various resources. Consider instances of the class FileReference that can refer to a concrete file or directory. During inspection these objects require different representations depending on the developer needs. For example, a state view is sufficient for determining the path of a directory/file (Figure 2a). However, if a developer needs to explore the content of a file/directory, looking just at the state of the corresponding object is not appropriate; a presentation showing the content serves the purpose better. A presentation for a directory (Figure 2b) can show the list of files from

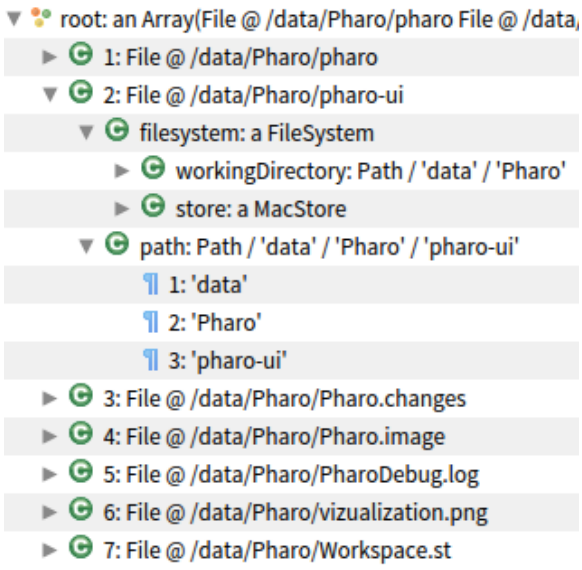


Figure 3: Inspecting a list of files/directories with an object inspector relying on a tree for exploring object state. It is not possible to immediately access the content of a file/directory.

that directory, while one for a file can display its content; the presentation could differ from a file to another depending on the type of the file (*e.g.*, image, text, xml, html). Even if objects representing files and directories are instances of the same class, exploring their content requires different presentations. An object inspector that focuses solely on the state of an object does not support this use case.

2.2 Limitations when focusing on individual objects

On the one hand, visually searching for certain objects within lists (*e.g.*, spotting a particular file/directory inside a directory) requires presentations that make it easy to identify the desired objects (*e.g.*, show the content of a file/directory). On the other hand, this task also requires a developer to simultaneously interact with two objects: the list and an element from that list. Object inspectors that focus on the state of single objects lead to a time consuming exploration effort. Consider using an object inspector representing an object as a tree to look for a particular file/directory within a given directory (Figure 3). This inspector does not provide easy access to the content of a file/directory and further requires a developer to permanently expand and collapse the elements of the target list, increasing the time spent in the inspector.

Another limitation of an object inspector providing only a fixed navigation based on object state is that it does not allow a developer to reach objects not stored in an instance variable of an object already accessible from the inspector. Consider a developer wishing to navigate from a morph representing a list to its context menu (*i.e.*, to check if the context menu has the correct structure). Unfortunately, the context menu is generated on demand every time a user right-clicks on an element of the list and is never stored in an instance variable of the morph, thus, it won't be accessible if we can only navigate between objects based on their state. This is an example of a more general problem where references to objects that we wish to navigate to are not actually stored in instance variables of the objects we are currently inspecting.

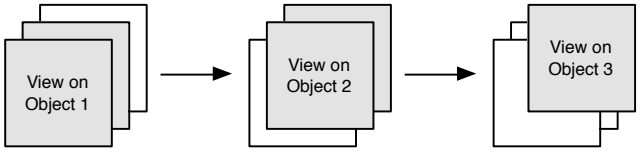


Figure 4: An inspection session consisting of three objects, where each object defines three basic presentations. Gray presentations are valid in one development context, while white presentations are valid in another development context. A *moldable presentation* selects only those presentations that are relevant for the current development context.

3. The Moldable Inspector framework

The Moldable Inspector supports developers in reasoning about run-time questions in specific application domains by providing *moldable presentations* and *moldable navigation*. Moldable presentations make it possible for an object to have multiple interchangeable presentations tailored to the domain and the question at hand. Moldable navigation provides a workflow in which multiple levels of connecting objects can be seen together and navigation between objects is guided by the domain and the question at hand.

3.1 Moldable presentations

Reiss argues that software understanding requires custom visualizations tailored to the problems at hand [8]. In the context of object inspectors we argue that understanding objects requires presentations tailored to both the domain and the question at hand (*i.e.*, the development context). While different objects require different presentations, given that they model different entities, the same object requires multiple presentations that depend on multiple usage contexts.

To address this, the Moldable Inspector allows an object to define a set of multiple interchangeable presentations capturing interesting aspects of that object in various development contexts. We will refer to these presentations as *basic presentations*. A developer can then inspect an object using a *moldable presentation* that selects only those basic presentations that are suitable for the current development context.

Moldable presentations are made possible by the Moldable Inspector reifying the current development context (*i.e.*, the domain and the question). An object can thus define a wide set of presentations, not all relevant to a particular context, however, in a given development context a moldable presentation only shows those presentations relevant for that context (Figure 4).

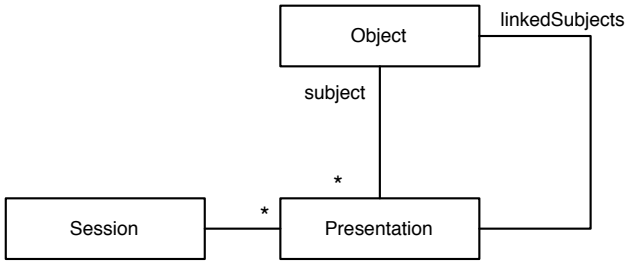


Figure 5: The model behind the Moldable Inspector framework: the *Session* objects reifies the development context; objects define multiple presentations; presentations can indicate relevant objects; when used, only those presentations relevant to the current development context are selected.

3.2 Moldable navigation

Understanding a run-time question involves reasoning about multiple objects and exploring connections between objects. Different types of questions require different kinds of interactions depending on the relationships between objects, the domain and the differing developer needs. Viewing one object at a time and hardcoding the reachable objects by only supporting state-based navigation does not support well this activity.

To overcome these limitations the Moldable Inspector offers *moldable navigation*, *i.e.*, a workflow in which multiple levels of connecting objects can be seen together and navigation between objects is guided by the question and the domain. This navigation mechanism is obtained by allowing each basic presentation to indicate a set of relevant objects that could be inspected next (Figure 5). Since objects are displayed during an inspection session using moldable presentations, only basic presentations relevant to the current question and domain are accessible; thus, the objects indicated by those basic presentations will also be relevant in that situation. However, as one cannot anticipate all developer needs, code can be used to steer the inspection process on-the-fly and increase the set of reachable objects.

4. Implementation aspects

In this section we present the current prototype instantiation of the Moldable Inspector, called the *GTInspector*, and discuss several implementation aspects. The GTInspector is integrated into Moose[2], a platform for data and software analysis [6].

4.1 Supporting moldable presentations

A moldable presentation consists of a set of basic presentations selected according to the current development context. This feature requires mechanisms to associate basic presentations with objects, specify a development context, and filter basic presentations based on the development context.

[2] `http://moosetechnology.org`

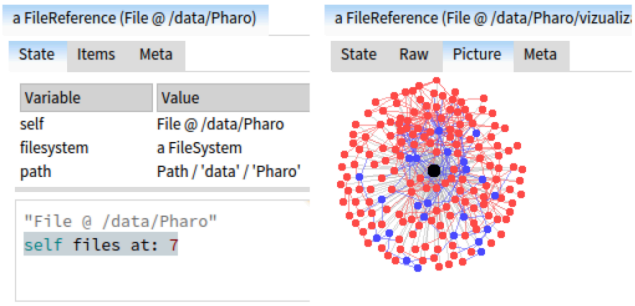


Figure 6: GTInspector displaying a directory and one contained file. Both objects have presentations to display the state and the source code of their class. The directory further has a presentation for showing the contained files/directories ("Items"). The file represents an image and has a presentation that shows its content as text ("Raw") and one that displays the actual image contained in that file ("Picture").

Once a moldable presentation has been computed a solution that can display multiple basic presentations is required.

In the current implementation a basic presentation is associated with an object by defining in the class of the object a method constructing the presentation and marking it with a predefined annotation. Currently defining a development context and filtering presentations based on it is not supported. Instead, regardless of the context, an object is represented using all the available presentations. An object with multiple basic presentations is displayed using a tabulator widget, where each presentation is added as a tab. This allows for interchangeable presentations and also gives an overview of all the basic presentations available for an object. Figure 6 shows how a directory and a file object are represented using this approach.

By default two basic presentations are added to every object: one shows the state of the object while the other gives access to the source code of the object class. By making the state of every object available, we support the classic way of using an inspector. Furthermore, by making the class of the object immediately browsable the inspector supports a common use case of looking implementation up during inspection time.

4.2 Supporting moldable navigation

Providing support for navigating between objects requires a mechanism to show connections between objects. GTInspector shows connections between objects using the Miller columns technique[3]: the next object is always shown to the right. This preserves the entire logical flow of how the developer got to an object. At any moment a predefined number of columns (*i.e.*, objects) is visible and a scroll bar is used to access previous columns.

To support *moldable navigation* we need to allow basic presentations to indicate relevant objects and support developers in using code to guide the inspection process. To sup-

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

port the first aspect GTInspector allows developers to continue the navigation by selecting any object available in the current presentation. The second aspect is also supported as the presentation showing the state, available for every object, can be used to write code executed in the context of an object. This can be seen in Figure 6 where the object on the right was obtained by executing the code "self files at: 7" on the object on the left.

4.3 The cost of new presentations

GTInspector has been used to create over 70 basic presentations for 40 different types of objects. On average a basic presentation requires 8 lines of code. Their small size makes them easy to understand and makes the creation of new basic presentations an affordable activity. To give a feeling of the amount of code required to create a new presentation the following lines show how to specify a tree presentation displaying the structure of a morph:

```
composite tree
  title: 'Submorphs';
  rootsExpanded;
  display: [:rootMorph | {rootMorph}];
  format: [:morph | morph printString];
  children: [:morph | morph submorphs];
  when: [:morph | morph submorphs notEmpty]
```

5. Improving the inspection process

To show that the GTInspector improves the inspection process in this section we look at how it addresses the limitations of traditional object inspectors encountered in Section 2.

5.1 Multiple presentations for objects

Section 2 showed that depending on the development context one needs to see either the visual appearance of a morph or its structure. The GTInspector addresses this requirement as it can provide two basic presentations capturing these two aspects (Figure 7). It can further provide dedicated presentations for inspecting FileReference objects based on the type and content of the object. For example, a FileReference object representing a directory has a presentation showing list of files/directories within that directory (Figure 8). FileReference objects representing files have dedicated presentations that display the content of the file in a proper way (*e.g.*, a file storing a picture is displayed using a visualization – Figure 6, while a file representing a script using an editor with proper syntax highlighting – Figure 8.)

5.2 Flexible navigation

Visually searching for particular objects within lists becomes possible with the GTInspector: while iterating over the list elements one can obtain a moldable presentation showing each element. For example, to locate a file based on its content one can iterate over the files of a directory and view each file using a specialized presentation (Figure 8).

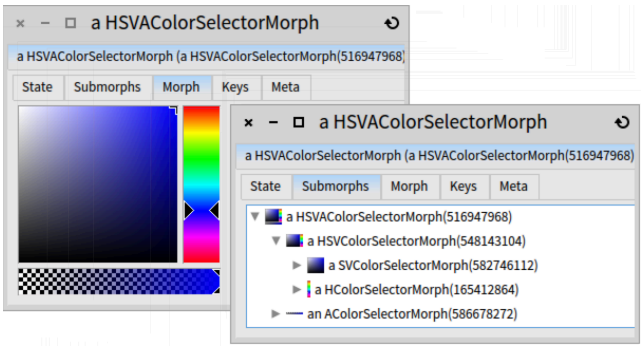


Figure 7: Two different ways to look at a morph for choosing colors.

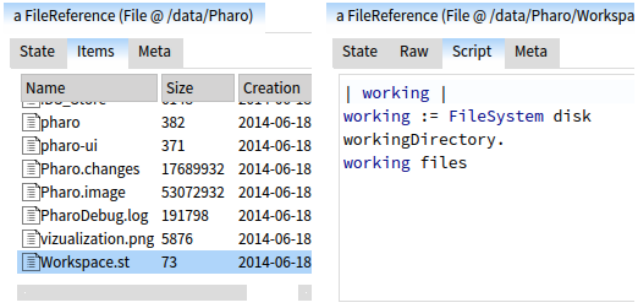


Figure 8: Exploring the content of a directory.

Using code to guide the navigation process makes it possible to reach objects not directly stored within instance variables: Figure 9 shows how one can obtain the context menu of a list morph showing file objects, execute an action from that menu and inspect the result. First the context menu is extracted using the code "self getMenu: false" (the false value indicates the shift key was not pressed). As the menu is a morph we can inspect it visually. Then we can execute the last action, "Copy", which copies a textual representation of the selected object to the clipboard, and finally inspect the current value from the clipboard to see if it is correct.

6. Further directions

While the GTInspector supports a fully functional object inspector there are a number of directions that can be explored in order to further improve it. These include, but are not limited to: identifying common types of recurring run-time scenarios and determining types of basic presentations useful in for addressing those scenarios, modeling the history of an inspection session as a first class entity, improving navigation through large inspection sessions.

6.1 Identifying recurring run-time scenarios

Currently the GTInspector allows objects to have different representations in different development contexts. However, the responsibility of deciding which presentation is relevant in the current development context falls solely on the devel-

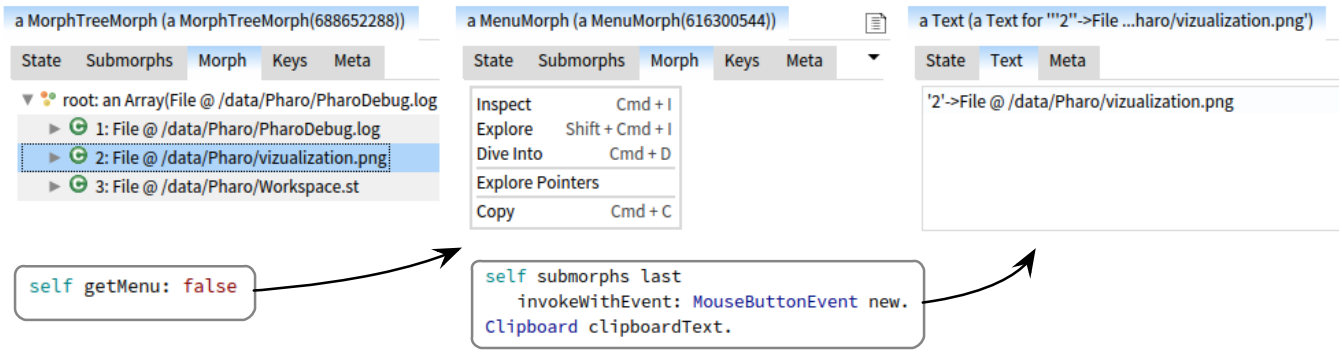


Figure 9: Performing a navigation scenario involving objects that are not directly linked through instance variables: extract the context menu of a list morph, run an action from the menu and verify the result. The first pane shows the tree morph, the second the context menu and the last the current value from the clipboard. At each step code is used to navigate to the next object.

oper. Identifying a set of common scenarios and determining which presentations are useful in those scenarios would lift part of this burden from developers: with the common use cases already supported, developers would only need to focus on creating presentations for their specific situations.

6.2 The inspection session as a first-class entity

While the GTInspector provides support for moldable navigation one has to manually repeat inspection sessions. Modeling the history of an inspection session as a first-class entity would make it possible to store, find and reuse inspection sessions.

6.3 Navigation improvements

While Miller columns are intuitive to use, they have two main drawbacks: *(i)* they require horizontal scroll bars to show deep hierarchical structures (*e.g.*, a deep inspection session) and *(ii)* they do not indicate the relation between two columns (*e.g.*, what did the user do to navigate from one object to the other). We are currently investigating how to solve these problems by providing a new type of scrolling widget showing an overview of the entire inspection session and indicating the relation between columns.

7. Related work

There is a wide body of research looking at how to improve the effectiveness of comprehension and development tools by finding and highlighting contextual information and providing better support for exploring code and data.

Code Bubbles brings the idea of a session of inspection to code understanding and debugging [1, 4]. The approach shows the related entities next to one another and allows the developer to manipulate and store them in sessions. However, this approach still relies on single representations for each entity regardless of the context, and object inspection is particularly only offered through a classic tree like view.

While the focus of our paper is on the conceptual structure of an inspector, the actual implementation is still an interesting aspect that deserves a discussion. The rendering offered by Code Bubbles allows the developer to manipulate a tree, rather than only a list. On the one hand, this is a powerful tool to understand more complicated scenarios. On the other hand, it is a more complicated interface that relies on the developer to organize the bubbles. Our implementation relies on a Miller columns design that requires little space and little spatial maintenance effort from the developer.

jGRASP is an integrated development environment providing object viewers that like our approach allow objects to have multiple presentations [3]. However, *jGRASP* always shows the same objects through the same views as it does not take into account the development context in which those objects are encountered.

Eclipse[4] allows developers to create custom textual representation for objects using "Detail Formatters". Each class can have a Detail Formatter consisting of a snippet of code that constructs a custom string value used to display instances of that class. NetBeans[5] and IntelliJ[6] allow developers to attach multiple such formatters to a given class and switch between them at run time. Unlike the Moldable Inspector these approaches only allow objects to have text representations.

8. Conclusions

Different types of questions exercise different aspects of an object and require different kinds of interaction depending on the relationships between objects, the application domain and the differing developer needs. To support this we presented a novel approach, called the Moldable Inspector, for developing an extensible object inspector that can be adapted to both the objects of a domain and the questions at hand. The development context is reified and used to both select presentations and steer the navigation between objects using a workflow in which multiple levels of connecting objects

[4] `eclipse.org/ide`

[5] `netbeans.org`

[6] `jetbrains.com/idea`

can be seen together. To show that the Moldable Inspector has practical applicability we presented the GTInspector prototype, and discussed several scenarios in which it improves the inspection process.

The Moldable Inspector is part of a broader work on meta-tooling (*i.e.*, tools for building tools) that aims to enable developers to quickly and effectively customize the IDE to suite their development contexts [7]. The Moldable Inspector follows on the Moldable Debugger [2] work that proposed a new approach for developing domain-specific debuggers.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project No. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We also thank Erwann Wernli and Jorge Ressia for their comments.

References

[1] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. 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*, pages 2503–2512, New York, NY, USA, 2010. ACM. doi:10.1145/1753326.1753706.

[2] Andrei Chiş, Oscar Nierstrasz, and Tudor Gîrba. The Moldable Debugger: a framework for developing domain-specific debuggers. In *7th International Conference on Software Language Engineering (SLE)*, 2014. to appear.

[3] James H. Cross, II, T. Dean Hendrix, David A. Umphress, Larry A. Barowski, Jhilmil Jain, and Lacey N. Montgomery. Robust generation of dynamic data structure visualizations with multiple interaction approaches. *Trans. Comput. Educ.*, 9(2):13:1–13:32, June 2009. URL: http://doi.acm.org/10.1145/1538234.1538240, doi:10.1145/1538234.1538240.

[4] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. Debugger canvas: industrial experience with the code bubbles paradigm. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1064–1073, Piscataway, NJ, USA, 2012. IEEE Press. URL: http://dl.acm.org/citation.cfm?id=2337223.2337362.

[5] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[6] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper. doi:10.1145/1095430.1081707.

[7] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *Proceedings of International Conference on Program Comprehension (ICPC 2012)*, pages 3–10, 2012. doi:10.1109/ICPC.2012.6240507.

[8] Steven P. Reiss. The paradox of software visualization. *VISSOFT 2005. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, page 19, 2005. doi:10.1109/VISSOF.2005.1684306.