Towards a Moldable Debugger

Andrei Chiş
Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch//staff/andreichis

Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
http://scg.unibe.ch/oscar

Tudor Gîrba
CompuGroup Medical
Schweiz AG
http://www.tudorgirba.com/

ABSTRACT

The debugger is an essential tool in any programming environment, as it helps developers understand the dynamic behaviour of software systems. However, traditional debuggers fail in answering domain-specific questions, as the semantics of what they show and do are fixed. In this paper we introduce our work towards a *moldable debugger* which, unlike traditional debuggers, both adapts itself and can be adapted to a particular debugging context. Thus, it allows developers to answer their questions by using concepts from their own application domains.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments, interactive environments*

General Terms

Languages, Design

Keywords

Debugging, Customization, Domain-specific tools, Smalltalk

1. INTRODUCTION

Answering questions about the runtime behavior of software is a prerequisite for maintaining and evolving software systems. Most of the time this is done by using the debugger, as it allows one to interact with a running system and inspect its state. This makes the debugger an essential tool in any programming environment.

Nevertheless, there is a mismatch between the way in which developers reason about their applications and the way in which they debug them. Developers encounter contextual problems and express them in terms specific to the domain models of their applications. On the other hand, when they use the debugger to solve these problems, they have to reason using a standard set of features. This happens as it is difficult to anticipate all types of problems developers will attempt to solve using the debugger.

In order to solve these contextual problems, one approach is to allow developers to adapt and customise their tools. For example, this idea is leveraged in query-based debugging [5, 4, 6] where instead of having a general-purpose tool showing a predefined set of information, developers write queries to extract exactly the information they need. In this paper we apply the same idea to the debugger. We envision a moldable debugger which, unlike a general-purpose debugger, both adapts itself and can be adapted to a debugging context. Thus, we delegate the customisation of the debugger to developers, as they are the ones that know the problem.

Another similar solution is the Debugger Canvas [3] which allows users to inspect long or complex control paths and control structures using the Code Bubbles [1] paradigm: instead of providing a predefined UI, users create "bubbles" containing only the information they are interested in (*e.g.*, call paths, variable values). In our approach users can create their own custom debugging actions.

In the remainder of this paper we present our vision of a moldable debugger. In Section 2 we investigate problems faced by developers when debugging the announcements framework from the Pharo system and show how a custom debugger can improve this process. In Section 3 we present our current work towards an infrastructure for supporting a moldable debugger. In Section 4 we discuss the implications of having a moldable debugger and we conclude in Section 5.

2. DEBUGGING ANNOUNCEMENTS

To motivate our work on custom debuggers, we consider the application domain of announcements. Since the control flow for announcements is event-based, it does not match well the stack-based paradigm follows by conventional debuggers.

The announcements framework from the Pharo[1] system is a framework providing a notification mechanism between objects based on a registration mechanism and first class announcements. It is the basis for implementing the Observer design pattern. Figure 1 presents an overview containing its structure and its usage.

The main component is the announcer, which allows objects to register interest in events and signals events. Objects

[1] http://www.pharo-project.org

Dyla '13, Montpellier, France

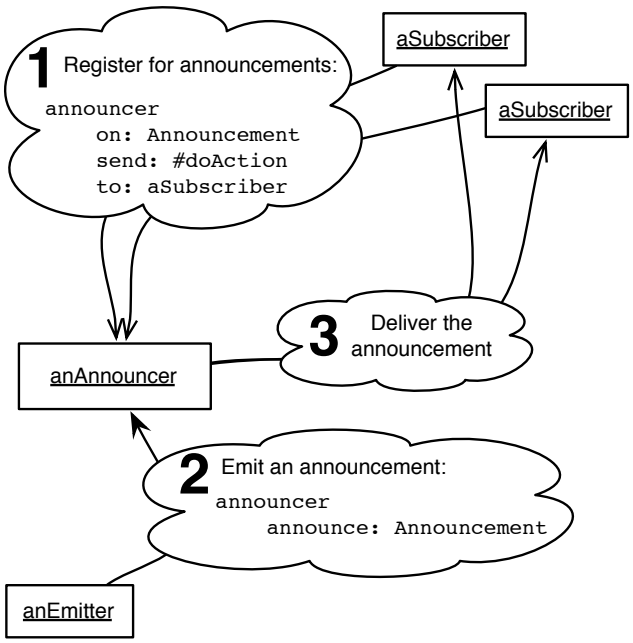


Figure 1: Basic usage of the announcements framework

register with an announcer by specifying the event types of interest and an action to perform when events of those types are signalled. The occurrence of an event is encapsulated in an announcement, *i.e.*, an object storing all information related to a particular occurrence of an event.

Debugging problems with announcements using the standard debugger from Pharo Smalltalk is not a straightforward process. We will exemplify this using two examples: *Glamour* [2], an engine for scripting browsers and *SystemAnnouncer*, the class that plays the role of announcer for events raised by the Pharo system.

2.1 Difficulties in debugging announcements

Glamour. In Glamour users specify a browser in terms of components and connectors with the help of an embedded domain specific language. The created model is platform-independent and can be displayed with various renderers. Announcements are used to synchronize the model and the UI.

Next, we will look at an example where due to a user action the model of the browser is changed. This raises an announcement delivered to the UI in order to update itself. To start we put a breakpoint in the UI method called when the announcement is delivered.

First, the standard debugger focuses on the stack as its key abstraction. This is natural since debuggers are intended to give developers to possibility to explore the run-time state, *i.e.*, the stack and the heap. However the stack is not necessarily an appropriate way to understand the application domain at run time. In the case of announcements, the stack is not useful for identifying the sender of an announcement. We can see this in Figure 2a: to locate the model component that raised the announcement we have to manually go through the stack. Second, the structure of the debugger only allows for one stack frame to be inspected at a time. When dealing with an announcement it is helpful to see both its sender and its receiver at the same time, instead of permanently switching between the two.

SystemAnnouncer. To highlight other types of problems that arise when debugging announcements we will look at the class *SystemAnnouncer*. This class is an announcer responsible for propagating all events raised by the Pharo system, like: *MethodModified*, *ClassAdded*, *ClassCommented*, *CategoryRemoved*, *etc.* As these events are only generated in well-defined conditions we are not interested in reasoning about their senders. The focus lies in understanding how they are propagated through the system: what are the objects being notified.

However, there is no straightforward way to determine all the objects that will receive an announcement using the standard debugger. To see this first we have to set a breakpoint in a method that is called when an announcement it is delivered to an object. We will put a breakpoint in the method *AbstractNautilusUI>>classAdded:* that is called every time an announcement of type *ClassAdded* is delivered to an instance of class *NautilusUI*. Figure 2b shows how we can then extract the required information: a certain stack frame has to be located and then the variable *interestedSubscriptions* has to be inspected.

We further notice there is a large number of subscriptions interested in this event. If from the debugger we want to know which have already been executed and which are awaiting execution we have to write a script that finds the position of the current subscription and then splits the array of subscriptions accordingly. The debugger does not offer direct support for obtaining these data.

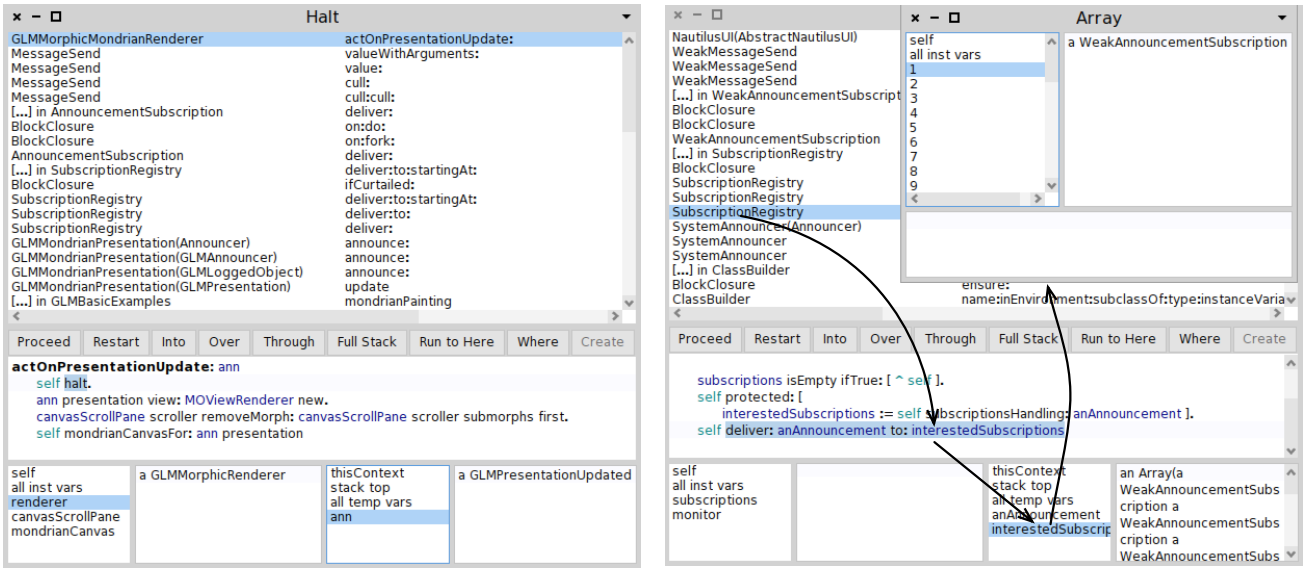
Once we understand the current subscription the next step is to move to the next object that receives the announcement. This is not a trivial action. As the standard debugger only provides a predefined set of general-purpose stack-oriented actions, it involves manually stepping into and stepping over a high number of instructions

2.2 An Announcer-Centric Debugger

In the previous paragraphs we showed that using the standard debugger to work with the announcements framework raises a series of difficulties. To address them we propose a custom debugger *molded* for the announcements framework, *i.e.*, an announcer-centric debugger, illustrated in Figure 3 and 5.

The debugger has a UI that only displays information relevant when dealing with announcements: the sender of the announcement, the receiver and the other subscriptions of the announcer. This can be seen in Figure 3 showing the same debugging situation involving Glamour as before, using the new debugger. We can easily spot both the model component that raised the announcement and the UI element that received it. Thus, the first two aforementioned problems are solved, as developers do not have to search the stack for the necessary information. We choose to also display the stack as it useful to know how the program reached its current state.

Apart from a different UI, this debugger also provides two categories of debugging actions for working with announcements: *announcer-centric stack-based actions* and *announcer-centric breakpoints*. An announcer-centric stack-based action steps through the execution of the program until the current stack frame satisfies a certain condition, after which it updates the current debugger. We have implemented two



(a) Debugging announcements in *Glamour*

(b) Locating subscriptions interested in the event *ClassAdded*.

Figure 2: Debugging announcements using the standard debugger from Pharo.

such actions:

- *stepToNextSubscription*
- *stepToSubscription: aSubscription*

The first steps through the execution until the announcer delivers the current announcement to the next subscription, while the second only stops when a certain subscription is reached. Using them we can navigate through the subscriptions registered with the *SystemAnnouncer* class, without having to perform multiple basic actions.

Announcer-centric breakpoints on the other hand, allow us to sets breakpoints when objects interact with announcements. This can be done while the software system is running without having to modify the source code. Unlike the previous type of actions, they close the debugger and resume the execution of the program. They are based on the concept of object-centric debugging [7] that allows one to perform operations directly on the objects involved in a computation. We added the following breakpoints:

- *halt when object receives announcement*
- *halt when object sends announcement*
- *halt when announcer delivers announcement*
- *halt when announcer receives announcement*

The first two can be applied on any object while that last two can only be applied on announcers. This way one can open a debugger when announcements are involved without having to search through the source code for locations that use announcements. For example, when debugging Glamour, we can set a breakpoint when the same UI element receives an update request.

Looking back at the aforementioned examples, using the new debugger simplifies the process of working with announcements: we do not have to search the stack to see from what code location announcements were raised. Also,

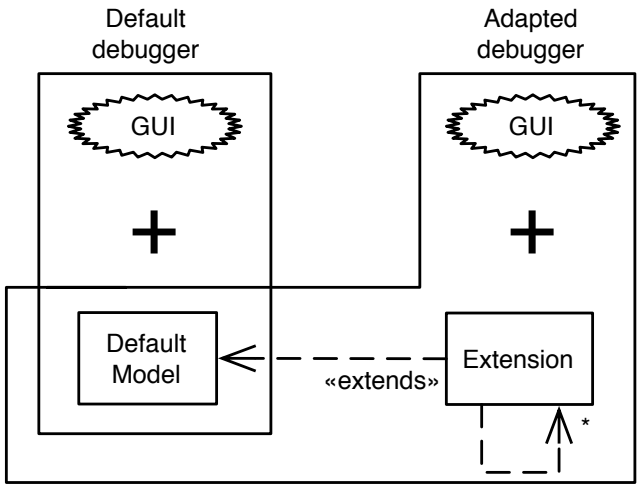


Figure 4: Adapting debuggers

we do not need any knowledge about the internals of the announcements framework in order to determine what other subscriptions are executed. If we want to set a breakpoint when an object receives or sends announcements, we can do it from the debugger. We do not have to modify the source code and restart the debugger. Thus, using an announcer-centric debugger improves the way in which the announcements framework is debugged.

3. TOWARDS A MOLDABLE DEBUGGER

In the previous section we saw that using the standard debugger to work with the announcements framework is difficult, and that by using a custom debugger it becomes easier. This also applies to other situations in which there is a gap between the semantics of the application one is working with, and the semantics offered by the standard debugger.

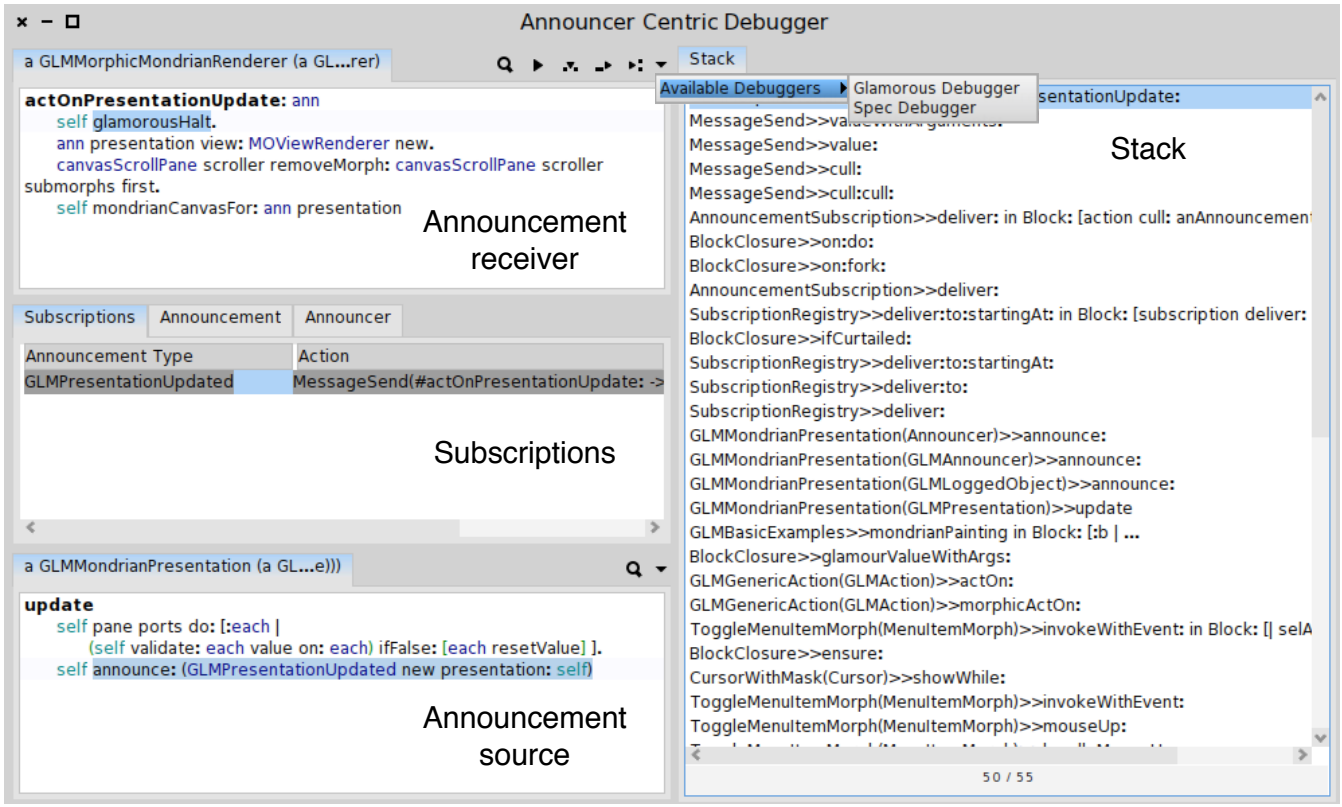


Figure 3: Debugging *Glamour* using the announcer-centric debugger.

A solution in these situations is to customise the debugger in order to debug using terms and concepts from the application domain, instead of working with generic features. However, when faced with the choice of either using the available debugger or adapting it to their needs, developers choose to keep using the one that is already available.

Developers choose not to invest effort in customising debuggers as they perceive this as a time-consuming and complicated activity. If we start from a monolithic debugger that is difficult to extend, this is indeed the case. However, if we start from a moldable debugger that can be easily *adapted* to a specific domain, the effort required to build new debuggers decreases significantly.

Furthermore, if this becomes feasible then a developer may adapt the debugger to fit multiple domains or scenarios. During a debugging session, a moldable debugger could at each step also adapt itself to the current debugging context. For example, we could start with a default debugger and reach a point where the announcer-centric debugger would be more useful. A moldable debugger would realise this and offer us the chance to continue using the announcer-centric debugger. Thus, the debugger *molds* to the debugging context.

In order to support this vision of a moldable debugger we identified three key features that an infrastructure must provide:

- mold debuggers to a domain/scenario;
- integrate the molded debuggers into the programming environment;
- switch between molded debuggers at runtime.

3.1 Molding debuggers

A moldable debugger must offer a way for developers to adapt it to new domains and scenarios. To support this we split a debugger into two components: (i) a model and (ii) a user interface. The *model* is the main component as it provides the debugging actions. The *user interface* integrates those actions and augments them with relevant information.

As a starting point for building new debuggers we provide a default model and a user interface. The default model encapsulates the logic of working with processes and stack frames and provides the basic stack-based debugging actions, like *step into* instruction, *step over* instruction, *resume process*, *restart execution*, *etc.* We aim to leverage this common base behavior for constructing new, custom debugging actions.

New debuggers can be created by adapting the default debugger. This is done by providing a new model and a new user interface relevant for the new model (Figure 4). New models are create by extending the class *DebugSession*, representing the default model.

To create the announcer-centric debugger we followed the process presented above. We extended the default model, implemented the new debugging actions and a new user interface. The new stack-based debugging actions were implemented by combing the available ones. As an example the *stepToNextSubscription* has the following implementation:

```
stepToNextSubscription
    | subscription|

    self stepOver: announcerModel loopContext.
    self stepIntoWhileTrue: [
        (self context selector ~= #deliver: or: [
            self context closure isNil]) and: [
                self context selector ~= #deliver:to: ]].
    (self context selector = #deliver:to:)
        ifTrue: [↑self].
    subscription := self context receiver.
    self stepIntoWhileFalse: [
        self context receiver = subscription subscriber
            and: [ self context method selector =
                    subscription action selector]].
```

First it does a *step over* action that skips the current subscription, followed by a set of *step into* actions to reach the delivery of the next subscription. It ends with skipping the internal details of actually delivery.

3.2 Integration of Molded Debuggers

As developers adapt the default debugger to fit their needs, several molded debuggers will eventually be present in the environment. Thus the following problem arises: when should a molded debugger be used? For example, the debugger for announcements is only relevant when an object receives an announcement. To address this concern we split it into two parts: (i) when do we need a molded debugger, (ii) which molded debugger should we use.

To address (i) we will use *triggers*. A trigger encapsulates a context in which a debugger should be opened. The simplest kind of trigger is a breakpoint that opens a debugger when the execution flow reaches a certain code location. By using a breakpoint a developer only indicates that he would like to open a debugger. He does not indicate which debugger he wants. We also support triggers based on the idea of object-centric debugging. This way, a debugger can be opened on an already running system when several objects interact in a given way. Again, only the fact that a debugger should be opened is captured. The four aforementioned announcer-centric breakpoints are implemented as triggers using this approach.

Triggers only indicate when a debugger should be opened. They do not have to know which debugger to open. To solve (ii) we need to determine which debugger to open in a given situation. This is done by using a *dispatcher* that asks all available debuggers, by sending them the message *handlesContext:*, if they want to handle the current debugging context. The announcer-centric debugger, for example, will only want to handle debugging context where the stack-frame represents the delivery of an announcement:

```
handlesContext: aContext
    ↑(self stackFilterFor: aContext sender)
        locateAnnouncerEntryContext notNil
```

For this to be possible, when a molded debugger is added to the environment, it registers with the dispatcher. To solve situations where multiple debuggers might be appropriate, when a molded debugger registers with the dispatcher it also specifies a rank. A low rank indicates a general debugger while a high rank indicates a specific debugger. If more than one debugger is available the one with the highest rank is selected.

3.3 Runtime Adaptation of Molded Debuggers

In Section 3.2 we saw how to decide what debugger to open. However, even if we start with the right molded debugger, during a debugging session we might reach a point where a different molded debugger might be more suitable. To support this feature, at each step during debugging we can move to another molded debugger, if it can handle the current debugging context. This is determined as before by using the dispatcher. Thus, the debugger molds to the debugging context.

Therefore, if we are using the default debugger and reach a point where announcements are involved we can switch to the announcer-centric debugger. The reverse situation is also possible: while working with the announcer-centric debugger we can reach code that has nothing to do with announcements. As this debugger also supports all the standard debugging actions we can continue using it, even if switching to the default debugger would be more useful. In some other situation we might be forced to switch to another debugger when the current one is no longer suitable. For example, we can see in Figure 3 that from the current debugger the user can switch to two other debuggers.

4. DISCUSSION

We have implemented two debuggers using our infrastructure: (i) the classic debugger (the default debugger that is adapted), and (ii) the announcer-centric debugger. As an indicator of the cost associated with building these debuggers we use the number of lines of code (see Table 1).

Debugger	Model (LOC)	UI (LOC)	Total
Default	424	346	770
Announcement	380	666	1046

Table 1: Size of molded debuggers

The small size required for the implementation makes the cost affordable. The availability of such an infrastructure opens new possibilities: the developers of a library or framework can ship a dedicated debugger together with the code. For example, we can envisage the developers of the announcement framework to have built the announcement debugger themselves and ship it together with the framework.

We plan to explore more ideas to reduce the time and the effort needed to create a new debugger, like: offering a better mechanism for creating domain-specific actions, predefined widgets for displaying certain types of data (*e.g.*, the stack), a domain specific language for specifying triggers, better matches to propose the appropriate debugger *etc.*

5. CONCLUSIONS

Debuggers are an essential tool in any programming environment as they allow developers to interact with a running system. Despite their importance the semantics of what they show and do are fixed. This makes solving domain-specific problems difficult, as one has to reason using a standard set of features.

In this paper we presented our current work towards a moldable debugger which, unlike a general-purpose debug-

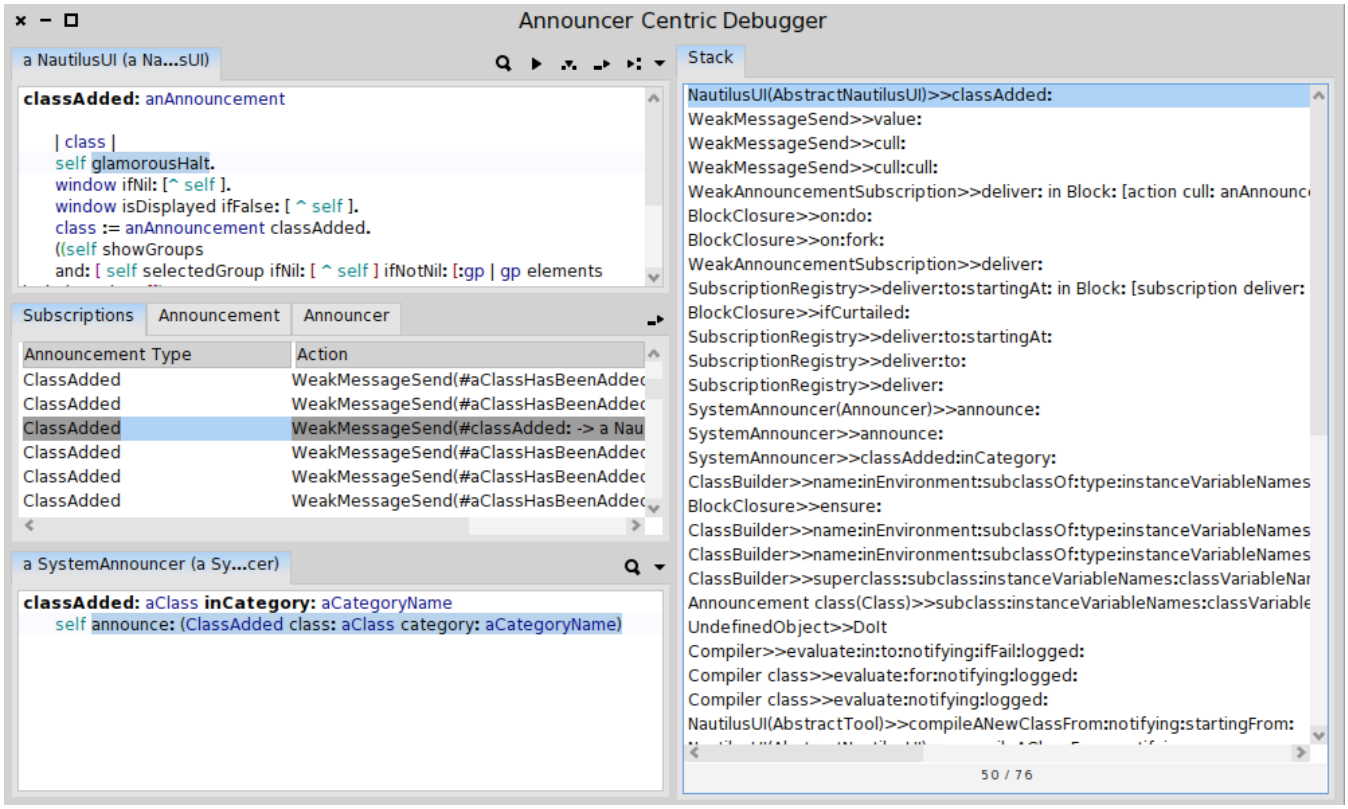


Figure 5: Debugging the *SystemAnnouncer* class using the announcer-centric debugger.

ger, it both adapts and can be adapted to a debugging context. Thus, it allows developers to solve their problems using a debugger that is closer to their domain. We plan to further investigate what a general framework for moldable debuggers could look like and how it can improve the debugging process.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions in improving this paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project Np. 200020-144126/1, Jan 1, 2013 - Dec. 30, 2015). We also thank CHOOSE, the special interest group for Object-Oriented Systems and Environments of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

6. REFERENCES

[1] 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 *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*, pages 2503–2512, New York, NY, USA, 2010. ACM.

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

[3] R. DeLine, A. Bragdon, K. Rowan, J. Jacobsen, and S. 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.

[4] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming (OOPSLA'97)*, pages 304–317, New York, NY, USA, 1997. ACM.

[5] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: a program query language. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 363–385, New York, NY, USA, 2005. ACM Press.

[6] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, page 251, Washington, DC, USA, 2004. IEEE Computer Society.

[7] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012.