Handling Error-Handling Errors: dealing with debugger bugs in Pharo

Steven Costiou, Thomas Dupriez, Damien Pollet

Handling Error-Handling Errors: dealing with debugger bugs in Pharo

Steven Costiou
steven.costiou@inria.fr
Inria, Univ. Lille, CNRS, Centrale Lille
UMR 9189 – CRIStAL
Lille, France

Thomas Dupriez
tdupriez@ens-paris-saclay.fr
Univ. Lille, CNRS, Centrale Lille, Inria
UMR 9189 – CRIStAL
Lille, France

Damien Pollet
damien.pollet@inria.fr
Univ. Lille, CNRS, Centrale Lille, Inria
UMR 9189 – CRIStAL
Lille, France

Abstract

In Pharo, errors happening during the opening of a debugger provoke *error-handling errors*. The Pharo system then drops into a rudimentary emergency evaluator, which provides extremely limited debugging features. This is a real problem while developing debuggers, when debuggers are more subject to bugs. In addition, the Pharo debugging infrastructure exposes an heterogeneous, obscure interface with various usages and users. Therefore, trying to extend this infrastructure to cope with debuggers bugs is tedious.

In this technical paper, we present *Oups*[1], an improved debugger infrastructure for Pharo. Oups provides a unified interface as a single entry point to request the opening of debuggers. Upon a debugger opening request, Oups uses interchangeable debugger opening strategies to select which debugger to open. We implemented a strategy that allows for the debugging of a failing debugger by other debuggers instead of the emergency evaluator. Oups improves the resilience of the Pharo system for specific cases of error-handling errors that we analyse.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Error handling and recovery**.

Keywords: debugging, debugger, error handling, Pharo

ACM Reference Format:
Steven Costiou, Thomas Dupriez, and Damien Pollet. 2020. Handling Error-Handling Errors: dealing with debugger bugs in Pharo. In *IWST20: International Workshop on Smalltalk Technologies, September 29th and 30th, 2020, Novi Sad, Serbia.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] *Oups*, french for *oops*, as in *"oops, my debugger failed again!"*

IWST20, September 29th and 30th, 2020, Novi Sad, Serbia

1 Introduction

In Pharo, an *error-handling error* is an error that happens while opening a debugger to handle another error. For example, an error occurring while opening the default debugger cannot be handled by the system. It would result in the opening of the same debugger, which would also encounter that error, and so on. This typically happens when modifying the debugger itself. Indeed, debuggers are applications which need development and maintenance and can contain bugs. This also happens while working on tools and libraries upon which debuggers depend. For example, Pharo 9 uses the Spec framework[2] [4, 15] to describe user interfaces. Errors in the Spec framework impact the Spec debugger, which in turn encounters errors. If they occur while the debugger is opening, these errors cannot be handled. They become error-handling errors which need to be handled without using the failing debugger.

In Smalltalk systems such as Pharo, Squeak, or Cuis, error-handling errors provoke the opening of an emergency evaluator that blocks the system. This evaluator is rudimentary and provides a limited set of features. Basically, developers can revert only the last method change, or evaluate hand-written Smalltalk code without any tool support (no feedback, no code completion). It is also hard to understand why the system crashed because the emergency evaluator only provides a limited stack trace.

Developers need more flexibility to cope with error-handling errors. For example, in Pharo, there is more than one tool that is capable to debug errors. However, only the default system debugger is considered for debugging. If this debugger cannot open because of an internal error, it is not possible to open another debugger seamlessly. To use another debugger, developers have to manually set this debugger as the default system-debugger and reproduce their bug.

However, it is critical for debugger developers to be able to debug their debugger. When their debugger encounters an error, they need to use the first available debugger that can help them debug that error. On the other hand, end-users might not care about debuggers errors. They want to debug their domain code using the first available debugger that can help, and discard debuggers errors.

[2] https://github.com/pharo-spec/Spec

Such mechanism does not exist in Pharo, and it is hard to implement in the current debugging infrastructure. There are multiple entry points in the system to request the opening of a debugger. These requests go through heterogeneous and obscure debugging interfaces. Control of debuggers openings and errors is rigid, scattered in classes of different nature and responsibilities, within a monolithic infrastructure. In consequence, it is hard to improve that infrastructure to better cope with debuggers errors.

In this paper, we present *Oups*, an improved debugger infrastructure for Pharo. Oups provides a single, unified entry point interface to interact with the debugging infrastructure. This eases the comprehension and the extensibility of the infrastructure. To control the opening of debuggers, Oups uses debugger opening strategies. Using these strategies, developers control and customize how debuggers are opened and how their errors are handled. We implemented a debugger opening strategy resilient to error-handling errors. When a debugger fails to open, we look for another debugger in the system and try to open that debugger instead. In such cases, domain errors and debuggers errors are debuggable with real debuggers instead of the limited emergency evaluator.

We first review the Pharo debugging infrastructure and its limitations to deal with error-handling errors, which motivates our work (Section 2). Then, we describe the Oups infrastructure (Section 3) and discuss how it improves the Pharo debugging experience (Section 4). Finally, we discuss similar work (Section 5) and conclude (Section 6).

2 Background: debuggers openings and error handling errors in Pharo

In the following, we describe how debuggers are opened in Pharo and how their errors are handled. Debugger opening and error-handling errors management blend into the same infrastructure, that exposes heterogeneous interfaces (Section 2.1). In addition, means to debug error-handling errors are extremely limited (Section 2.2). This makes it difficult to develop and to debug debuggers. This drives our motivation to build a uniform way of opening debuggers and to provide tools to debug their errors (Section 2.3).

2.1 Opening debuggers: a blurry infrastructure

The current Pharo debugging infrastructure is depicted by Figure 1. The central point of the infrastructure is the UIManager class. It is a singleton responsible of managing user interaction with the user interface. This class exposes an interface to open a debugger that is used by system classes: Process, Warning (a particular exception class), ad-hoc clients (generally other system tools) and the UIManager itself.

Each Process instance also exposes an interface to start debugging. This interface is used by system objects, such as unhandled exceptions and ad-hoc clients, by user interrupts and by processes themselves.

Both these interfaces are heterogeneous and provide different methods with different input parameters. Objects requesting the opening of a debugger use different methods from this interface. Moreover, they do not always provide all the necessary parameters. This leads to intermediate method calls within the interface, adding nil values to fill the missing parameters. As a consequence, understanding the flow of debuggers openings is complex and tedious.

Finally, text editors directly access the system debugger and disregard interfaces exposed by UIManager and Process. This is a special case where users select custom pieces of code and ask the system to debug it. Text editors apply special control to the debugged process before opening debuggers. Because this control behavior is not centralized and exposed through an API, this leads to code duplication between tools.

Building and contributing system-level debugging tools is tedious because it requires to interact with this heterogeneous infrastructure. The infrastructure is intertwined with sensible system classes (Process and UIManager) and user interface classes (text editors). And all these classes apply a strong control over debugger openings and error handling. Improving debuggers errors handling requires to modify code in these classes and sometimes duplicate part of this code. It leads to the multiplication and the complexification of interface methods that control debugger openings.

2.2 Error handling errors and their recovery

Error-handling errors happen when the debugger cannot be opened after an exception is raised. For example, a bug in the debugger opening code prevents the debugger to open. Consequently, the debugger cannot open to debug its own error. The system signals a debugger error then raises a *primitive error*, which opens an emergency evaluator (Figure 1).

The emergency evaluator starts by giving a printed representation of the last 20 elements of the call stack (Figure 2). All other debugging information is dumped. Because it is only printed, a lot of contextual information is lost. It is hard, and sometimes impossible, to understand clearly why the system crashed. This is the only information developers have to devise what to do in the emergency evaluator.

The emergency evaluator (Figure 3) is a trivial *read-eval-print-loop* system with only three features. First, it can simply discard the error-handling error, providing the system can continue to work. Second, it allows developers to revert the last method change in the system. The idea is to recover from a fatal method change that provoked the error-handling error. But it only reverts the last modified method in the system and it cannot go beyond that. If the buggy method is not the last modified one, but another one modified much before, the revert command is of no help. Finally, it is possible to execute custom code, for example to manually recompile the buggy method. This is tedious and error-prone, as developers have to guess the problematic method, remember its original class name, the methods source code and they have to rewrite it

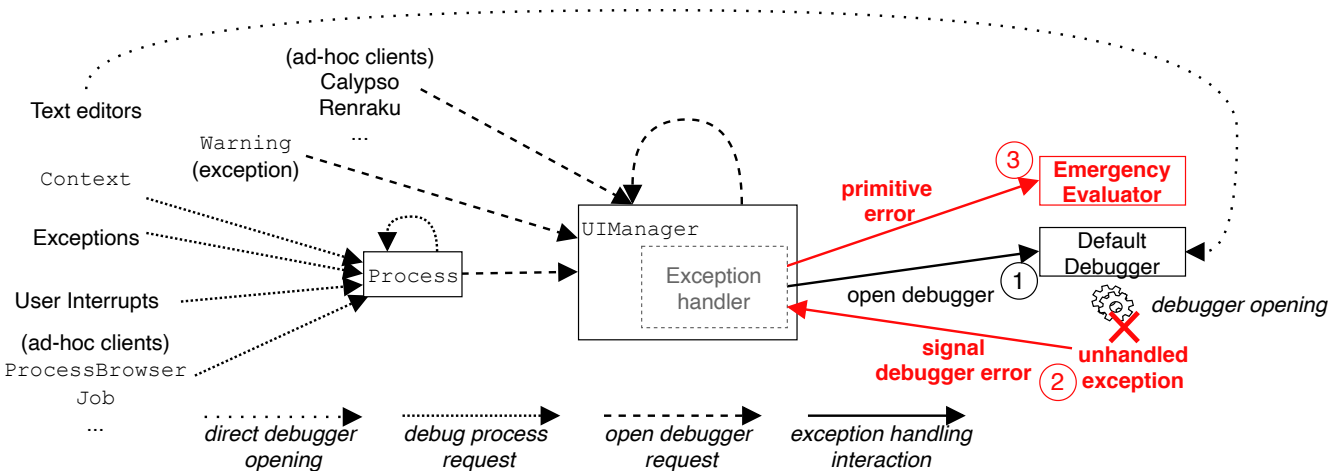

Figure 1. The Pharo debugger opening process, handled by the user interface manager: multiple clients, different APIs. If the opening of a debugger (1) encounters an error (2), the system falls back to the emergency evaluator (3).

without making any mistake. Although this is possible for simple methods, it does not scale to complex or subtile code. In addition, the emergency evaluator does not provide any support to write or to execute code (no code-completion and no feedback after an execution).

```
*** System error handling failed ***
Original error: ZeroDivide.
 Smalltalk tools debugger error: MessageNotUnderstood: GTGenericStackDebugger>> #a:
MorphicUIManager>>onPrimitiveError:
DebugSession(Object)>>primitiveError:
DebugSession>>signalDebuggerError:
[ :ex | debugSession signalDebuggerError: ex ] in [ "schedule debugger in deferred
   redraw problems after opening a debugger e.g. from
   the testrunner."
[ Smalltalk tools debugger
 openOn: debugSession
 withFullView: bool
 andNotification: notificationString ]
 on: Error
 do: [ :ex | debugSession signalDebuggerError: ex ] ] in MorphicUIManager>>debugPro
BlockClosure>>cull:
Context>>evaluateSignal:
Context>>handleSignal:
MessageNotUnderstood(Exception)>>signal
GTGenericStackDebugger(Object)>>doesNotUnderstand: #a
GTGenericStackDebugger(GTMoldableDebugger)>>compose
GTGenericStackDebugger(GLMCompositePresentation)>>initialize
GTGenericStackDebugger class(Behavior)>>new
GTGenericStackDebugger class(GTMoldableDebugger class)>>on:
```

Figure 2. Emergency evaluator with a debugger error's stack trace. It does not fit the screen and part of the trace is lost.

```
Type revert (without quote) to revert your last method change.
Type exit (without quote) to exit the emergency evaluator.
> revert
Revert: nil ?
Please type Yes or No followed by return
> yes
reverted: nil
>
```

Figure 3. The emergency evaluator. Sometimes it is unable to revert the last method change and is not helpful.

2.3 Motivation: debugging debuggers errors

As debugger developers, we need to debug our debuggers errors. However, the emergency evaluator is too limited to cope with those errors. Due to the limited infrastructure, we cannot simply debug our debugging tools using another tool, even if other debuggers are present in the system.

Changing the default debugger in the system settings after a debugger error is not sufficient. Once we get in the emergency evaluator, that debugger error and its context are lost. We need to quit everything, to change the default debugger, and try to reproduce the debugger problem to debug it with the other debugger. This is a show-stopper, as it is often precious to debug an error at the moment it happens [2] instead of trying to reproduce it later (which may be hard [5, 8, 9, 16]). This impacts end-user developers, who cannot debug their domain errors if the default debugger encounter errors. They also have to change the default debugger in the settings, and try to reproduce their domain error. In addition, debugger developers cannot introduce safely new debugging tools without risking to block users.

Trying to solve these problems in the current infrastructure is hard. To insert a simple variation in how errors-handling errors are dealt with, we need to deeply modify multiple debugger opening methods and their interactions in sensible system classes. Therefore, we aim at:

1. A clean, simplified and unified debugger opening infrastructure to facilitate debugger development, extension and experimentation,
2. as much as possible, neutralize debuggers errors by using other working debuggers to:
   - allow debugger developers to seamlessly debug debuggers errors when they occur,
   - allow end-user developers to seamlessly focus on the debugging of their domain errors.

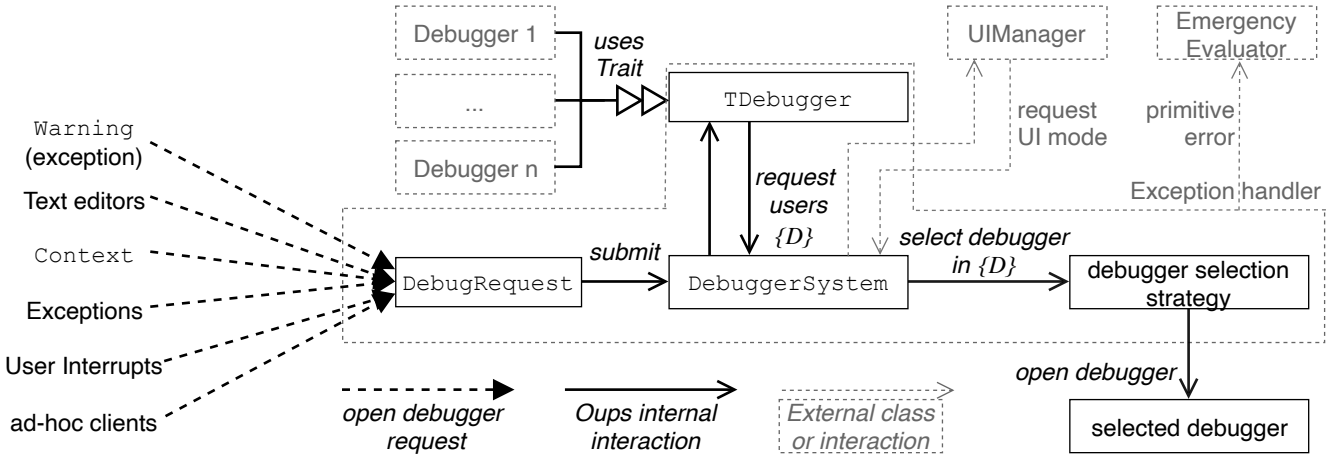

Figure 4. The Oups debugger infrastructure.

3 The Oups debugger infrastructure

Oups is a debugger infrastructure for Pharo, unifying means to request the opening of debuggers and allowing for the debugging of debugger errors. The infrastructure and how it works is described in Figure 4. In this section, we describe the Oups infrastructure. We detail the debugger selection strategy, *i.e.* how a debugger is chosen over another to debug a particular error. We finally explains how it handles error-handling errors, also called *meta-errors*, and which types of meta-errors Oups is resilient to.

3.1 Infrastructure overview

The Oups infrastructure mediates between Pharo's program execution and development environment layers. At the execution level, it acts as the ultimate error handler. At the tooling level, it ensures that any error that bubbles up that far is gracefully passed on to a debugger.

The Oups infrastructure is composed of five elements: *debug requests*, the *TDebugger* trait, *debugger system*, *debugger selection strategies* and *debugger failures*. Figure 5 shows the flow of an error through these elements, under ordinary circumstances (*i.e.*, without error-handling errors).

Debug requests: to open a debugger, the system creates a debug request object and submits it to the debugger system. Listing 1 shows an example of system code sending a debug request. The information passed to the debug request depends on the reason for debugging, as described in Section 2.1, but includes at least either an exception or a process. By filling up implicit information from the context, debug requests provide a single abstraction to the debugger system thereby unifying the multiple entry points of the legacy architecture.

```
Process>>#debugException: anException
  (DebugRequest newForException: anException)
    process: self; submit.
```

Listing 1. Example of sending a debug request.

The TDebugger trait: it is a stateful trait [14] providing a debugger opening interface methods and an instance variable for debugger priority configuration. Any class can use this trait to be recognized as a debugger by the system. The provided interface methods ensure that debugger classes are usable by the infrastructure.

Debugger system: it is the entry point into the debugging tool layer. Its instances handle debug requests according to the system state and configuration. First, before opening a debugger, the *Debugger System* object queries the *UI Manager*. If interactive debugging is not enabled, the debug request concludes there and the error is logged (*e.g.*, if the system is running headless). Second, the debugger system collects the list of users of TDebugger (*i.e.*, each debugger class) and sorts them based on their per-debugger user-defined priority. Then, a debugger selection strategy selects a debugger from that list to open the error. Finally, the debugger system ensures that the debug request is handled reliably. If an exception is raised within the debugger system itself, or if the debugger selection strategy fails to select a debugger, then the original error is handled by the emergency evaluator .

Debugger selection strategy: it is the object responsible for selecting a debugger from a given list of debuggers, and for opening the error with it. The debugger infrastructure implements a default strategy (Section 3.2) which is interchangeable with user-defined strategies.

Debugger failures: these objects are exceptions created when the infrastructure is configured to debug error-handling errors. When a debugger encounters an error

while opening, a DebuggerFailure exception is instantiated and signaled. A debugger failure references:

- the debugger that failed to open the original error: it is used by debugger selection strategies to avoid opening recursively the same faulty debugger,
- the original error with its interrupted context: if the debugger error is fixed, the debugger failure is resumed and the fixed debugger opens the original error.

The infrastructure is configurable either to focus on the debugging of user-level code, or to handle debugger failures to allow for the debugging of debugger errors. In addition, the debugger infrastructure interacts with two elements from the Pharo system:

The UI Manager: a system class responsible for defining how to communicate various UI events to the user. An interactive and graphical UI Manager will allow Oups to open a graphical debugger. A non-interactive command-line UI Manager will log the error instead.

The Emergency Evaluator: a rudimentary tool to recover from error-handling errors (described in section 2.2).

3.2 Debugger selection strategies

Debugger selection strategies are responsible of opening debuggers. Developers build these strategies to control and extend how debuggers are opened by the infrastructure. A strategy takes as input a list of sorted debuggers and a debugger opening request. The strategy determines which debugger in that list is eligible to the request. It then selects one of these debuggers and uses it to open the request. In this section, we describe the default debugger opening strategy in the case where there is no debugger error. The variation in case of debugger errors is described in Section 3.3.

Debugger Selector is the default debugger opening strategy of Oups. To determine which debugger is eligible, this strategy first considers that there is always an exception associated to a debug request. When that is not the case (*e.g.*, upon a debugIt action), an *ad-hoc* exception is created for the context to debug. Second, each debugger is asked if it can handle the request. For example, a debugger for unit test errors will decline handling requests that do not come from test executions. Third, the exception is asked if it can be handled by the debugger. By default, any exception can be handled by any debugger, except for debugger failures. The first debugger that is eligible to the request is selected and opened. If no debugger is eligible, the emergency evaluator is used as a last resort.

As an example, imagine we are handling an error in a system containing two debuggers A and B. Debugger A is a high priority debugger specialized in debugging unit test failures. Debugger B is a general purpose debugger. Debugger Selector will receive as input an array of two debuggers {A. B} sorted by priority. The strategy will check if the first debugger A is eligible. If the execution to debug is not that of a unit test failure, the specialized debugger A will declare itself unable to handle the request. Then debugger B will be examined and will declare itself able to handle the request, as general purpose debuggers are typically able to handle any requests. If the exception of the request is not a debugger failure, that exception will declare itself to be handle-able by debugger B. As B is the first eligible debugger found by the strategy, the strategy will open the debug request with B.

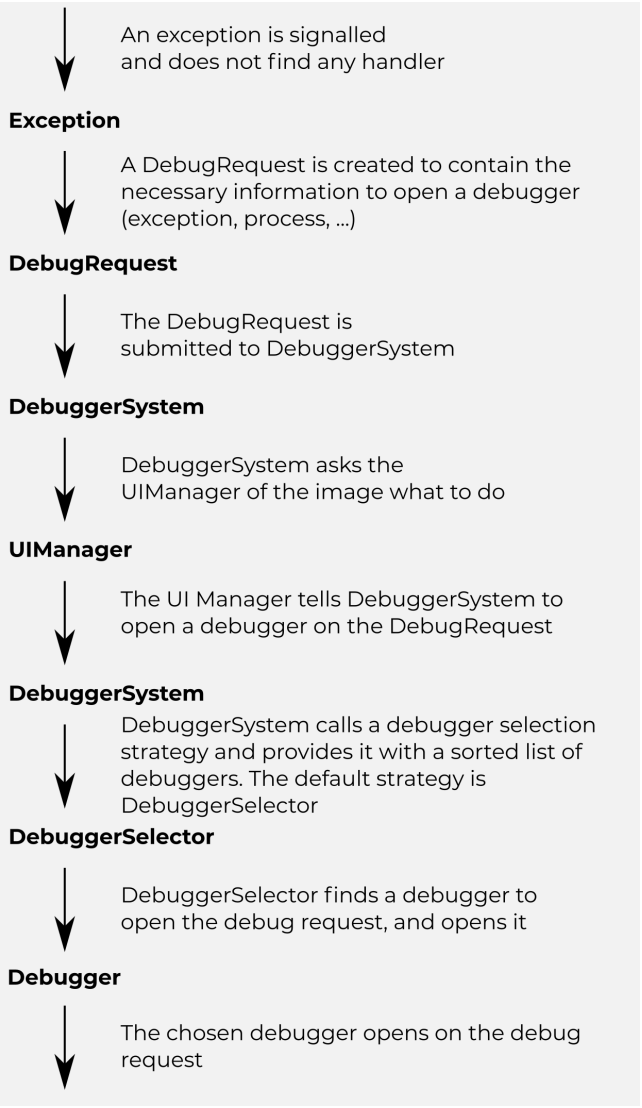

Figure 5. The sequence of actions from the signaling of an exception to the opening of a debugger, when no error occur.

3.3 Handling meta-errors in Oups

In this section, we describe how the infrastructure copes with error-handling errors. Because these errors are due to debuggers errors, we call them *meta-errors*. We call *original error* the error that the system was handling when the meta-error occurred. We identify eight types of meta-errors:

1. An error occurs in the exception mechanism,
2. an error occurs in the creation of a debug request,
3. an error occurs in the code of the debugger system,
4. an error occurs in the UI Manager,
5. an error occurs in the debugger opening strategy,
6. an error occurs in the opening of the debugger selected by the debugger opening strategy,
7. an error occurs in the emergency evaluator,
8. the debugger opening strategy does not find any debugger eligible to the debug request.

Oups is not resilient to meta-errors of type (1), (2) and (7). In other words, it assumes that a) no error occurs in the exception mechanism, b) no error occurs in the creation of a debug request and c) no error occurs in the emergency evaluator. Oups is resilient to all the other types of meta-errors, which are debuggable.

Meta-errors of type (3), (4) and (5) are handled by an exception handler installed at the entry point of the DebuggerSystem class. When this handler catches an exception, it means the debugger infrastructure is broken and cannot be used. In this case, the handler opens either the original error or the meta-error (depending on the infrastructure configuration) with the emergency evaluator.

Meta-errors of type (6) mean that the selected debugger to open the original error is broken. If the infrastructure is configured to focus on user-level errors, it simply ignores the meta-error. The Debugger Selector strategy then tries to open the original error with the next eligible debugger. If the infrastructure is configured to debug meta-errors, a DebuggerFailure is instantiated and signaled as an unhandled exception. The debugger system handles the resulting debug request like any other. However, because debugger failures reference the debugger from which their meta-error originated, Debugger Selector handles them differently. It considers failed debuggers referenced by debugger failures as non-eligible (see section 3.2). This prevents recursively trying to open the same failing debugger, while allowing the debugging of a failed debugger with another debugger.

The infrastructure only allows for the debugging of a single meta-error within the same process. The first debugger that breaks is debuggable, but not other debuggers that would fail while trying to debug that first debugger. *Meta-*-meta-errors*, if they do exist, are discarded by the infrastructure.

Finally, meta-errors of type (8), *i.e.*, when the debugger selection strategy does not find any eligible debugger to open, are handled by simply opening the original error with the emergency evaluator.

4 Evaluation

In this section, we compare Pharo with and without Oups in terms of which types of meta-errors they can debug. We then illustrate Oups through a few example scenarios.

Table 1. Improvements in meta-error debuggability

Meta-error in:	Pharo	Pharo+Oups
(1) the exception mechanism	✗	✗
(2) the creation of a debug request	**N.A.**	✗
(3) debugger system	✗*	**E.E.**
(4) the UI Manager	✗	**E.E.**
(5) the debugger selection strategy	**E.E.****	**E.E.**
(6) a debugger opening	**E.E.**	✓
(7) the emergency evaluator	✗	✗
(8) no debugger found	**E.E.****	**E.E.**

✓ Debuggable with a working debugger.
✗ Cannot be debugged.
E.E. Debugging falls back to the emergency evaluator.
N.A. Not applicable: Pharo does not implement debug requests.
* Part of the UI Manager analogous to the debugger system.
** Debugger lookup, analogous to the selection strategies.

4.1 Comparing the debuggability of meta-errors in Pharo with and without Oups

Self-supporting programming environments like Squeak or Pharo include many opportunities to *shoot yourself in the foot* [13]. Thanks to a few mechanisms and design choices, these environments are surprisingly reliable. While not foolproof in themselves, these mechanisms allow for the recovering from some of the meta-errors defined in Section 3.3.

Table 1 summarises how Pharo copes with meta-errors, with and without Oups. We observe that Oups cannot do much for errors that occur either too early during exception handling at the execution level (1, 2), or too late in the emergency evaluator (7) which is already a last ditch tool. However, Oups does improve the system's resilience to problems occurring at the platform level when opening debuggers. In cases 3 & 4, Oups drops into the emergency evaluator instead of the system entering an infinite error-handling recursion. We consider this as an incremental improvement. First, it concerns system code that is not likely to change often. Second, as an example of Kent Beck's advice[3] *"make the change easy (warning: this may be hard), then make the easy change"*, it could be ported back to Pharo without the rest of Oups.

Case 6 is the real benefit of Oups, because it makes the debugger less special to the system. Working on a debugger with Oups becomes more like application development.

4.2 Example scenarios

This section shows the Oups infrastructure in practice. In each case, the debugger selector picks between a high priority debugger A and a low priority one B. We describe what happens when user code divides by zero, depending on whether A fails or not and on the configuration of Oups.

No debugger crash. The high priority debugger A is selected and opens successfully. The user debugs the division by zero with debugger A.

[3] https://twitter.com/KentBeck/status/250733358307500032

Crash, but focus on domain errors. Debugger A fails while opening: Oups falls back to debugger B which opens successfully. The user debugs the division by zero with B.

Crash, and debugging meta-errors. Debugger A fails while opening, and Oups is configured to debug meta-errors. Oups raises a debugger failure exception referencing A and its error. That debugger failure triggers a new debug request, but for which debugger A is ineligible, so Oups selects B. The user debugs the debugger A error with debugger B. If the debugger A error is fixed and the execution proceeds, the original opening of debugger A resumes. The user now debugs the division by zero with debugger A.

5 Related Work

The moldable debugger [1] is the current Pharo debugger. It implements an algorithm to select the best suited debugger for a particular context (*e.g.*, unit tests). It not resilient to meta-errors, and works in parallel to the Pharo debugger infrastructure which makes it difficult to extend. Squeak has projects that contain separate system tools [13]. Upon a tool failure, it is possible to lookup in other projects to find another debugger. In Kansas [12] users develop in worlds. When a world breaks, another one is created from which the first world can be repaired. Kansas cannot be debugged if that ability to create new worlds breaks. Similarly, Sindarin [3] is a debugger scripting API for Pharo. If a debugging tool breaks, Sindarin operators allows developers to debug an execution without tooling support. However, if a Sindarin operator breaks then the debugging API is inoperable.

Other recovery tools could be envisioned. Remote debugging infrastructures [6, 7] deport tools in another system. If a remote tool breaks, it could theoretically be debugged from another remote debugger. In contrast, the ability to load different versions of the same code in the same image [11] could enable the debugging of a tool by another version of itself. Object-centric debugging [10] can scope breakpoints to a single instance of a debugger. Another instance of the same debugger can then debug the other. This enables putting breakpoints in debuggers but does not apply to meta-errors.

6 Conclusion and future work

In this paper we presented *Oups*, a debugger infrastructure for Pharo. Oups simplifies debugger openings by providing a unified interface to open debuggers on errors. It is extensible through a mechanism of interchangeable strategies to control how to select and open debuggers. It improves the debuggability of debugger errors by automatically selecting another debugger upon the failing of a first debugger.

We defined *meta-errors* as errors that occur while a previous error is being handled. Oups increases the number of meta-errors that can be debugged with regards to the current Pharo debugger infrastructure.

One limitation of Oups is its inability to recover from error-handling errors occurring outside Oups or after Oups itself failed. As future work, we plan to improve the debuggability of such errors. We also plan a survey-based empirical study on the impact of Oups on developers' daily work.

References

[1] Andrei Chiș, Tudor Gîrba, and Oscar Nierstrasz. 2014. The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. In *Software Language Engineering*. Springer, 102–121. https://doi.org/10.1007/978-3-319-11245-9_6

[2] Steven Costiou. 2018. *Unanticipated behavior adaptation : application to the debugging of running programs.* Theses. Université de Bretagne occidentale - Brest. https://tel.archives-ouvertes.fr/tel-02082447

[3] Thomas Dupriez, Guillermo Polito, Steven Costiou, Vincent Aranega, and Stéphane Ducasse. 2019. Sindarin: A Versatile Scripting API for the Pharo Debugger. In *DLS'19, Dynamic Language Symposium.*

[4] Johan Fabry and Stéphane Ducasse. 2017. *The Spec UI Framework.* Square Bracket Associates. 84 pages. http://books.pharo.org

[5] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *2013 ACM/IEEE international symposium on empirical software engineering and measurement.* IEEE, 383–392.

[6] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2018. Out-Of-Place debugging: a debugging architecture to reduce debugging interference. *The Art, Science, and Engineering of Programming* 3, 2 (Nov. 2018). https://doi.org/10.22152/programming-journal.org/2019/3/3

[7] Nick Papoulias, N. Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. 2015. Mercury: Properties and Design of a Remote Debugging Solution using Reflection. *Journal of Object Technology* (2015). https://hal.inria.fr/hal-01185730

[8] Michael Perscheid, Benjamin Siegmund, Marcel Taeumel, and Robert Hirschfeld. 2017. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal* 25, 1 (2017), 83–110. https://doi.org/10.1007/s11219-015-9294-2

[9] Eric S Raymond and Guy L Steele. 1996. *The new hacker's dictionary.* Mit Press.

[10] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-Centric Debugging. In *Proceeding of the 34rd international conference on Software engineering* (Zurich, Switzerland) *(ICSE '12).* https://doi.org/10.1109/ICSE.2012.6227167

[11] Théo Rogliano, Guillermo Polito, and Pablo Tesone. 2019. Towards easy program migration using language virtualization. In *International Workshop of Smalltalk Technology 2019.* Köln, Germany. https://hal.archives-ouvertes.fr/hal-02297756

[12] Randall B. Smith, Mario Wolczko, and David Ungar. 1997. From Kansas to Oz: collaborative debugging when a shared world breaks. *Commun. ACM* 40, 4 (April 1997), 72–78. https://doi.org/10.1145/248448.248461

[13] Marcel Taeumel and Robert Hirschfeld. 2016. Evolving User Interfaces From Within Self-supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop.* 43–59.

[14] Pablo Tesone, Stéphane Ducasse, Guillermo Polito, Luc Fabresse, and Noury Bouraqadi. 2020. A new modular implementation for Stateful Traits. *Science of Computer Programming* (2020).

[15] Benjamin van Ryseghem, Stéphane Ducasse, and Johan Fabry. 2012. Spec, a framework for the specification and reuse of UIs and their models. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2012)* (Ghent, Belgium) *(IWST '12).* ACM, Gent, Belgium, 2:1–2:14. https://doi.org/10.1145/2448963.2448965

[16] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging*. Elsevier.