Sympathy for the Devil

Reified Collection of Runtime Errors

Tommaso Dal Sasso
Andrea Mocci
Michele Lanza
REVEAL @ Faculty of Informatics
University of Lugano Switzerland

Andrei Chiş
Tudor Girba
Feenk GmbH, Switzerland

Abstract

Software development involves iterations of writing, running, testing, and debugging code. When fixing a defect, developers construct a mental model of the system that explains the defect and eventually identifies its cause. However, filtering complete, coherent, and reliable information from a running system is not an easy task: Using a simple approach, like generic logging, is often ineffective because it deconstructs and flattens the state into textual data, thus requiring ad-hoc understanding and processing. On the other hand, collecting structured information in form of objects to observe and understand a precise property of the system requires specialized ad-hoc code, decoupled from the system's domain, and usually not reusable.

We present *ShoreLine*, a domain-specific data collection framework that enables the developers to extract selected information about a running system. The developer is able to take a snapshot of all the information deemed relevant about a piece of code by writing few lines of code, thus enabling structured and effective logging and reporting of errors. We detail our framework in the context of a bug reporting platform, and illustrate how such an approach can be used to create in-depth and reliable domain-specific bug reports.

CCS Concepts • **Software and its engineering → Software maintenance tools**; **Software testing and debugging**;

1 Introduction

Computer systems have become pervasive in many human activities, where the high penetration of machine-controlled devices led to a tremendous increase in the complexity of the involved software. This phenomenon turned modern software development into a multifaceted activity, where the key elements are collaboration and communication. Writing code is only a small part of the process: Several phases such as design, testing, and maintenance, play a role as fundamental in the success of a project. In fact, maintenance often represents a significative percentage of a developer's time: Researchers showed that the effort put in reading and understanding code outweighs the effort needed to write it [5, 9, 15, 19]. In such a scenario, one would imagine that the effort to provide means to aid developers would focus on refined tools to navigate, understand, and inspect the code. Instead, many of the modern editors and IDEs put the biggest accent on how developers write code, leaving program comprehension as a secondary task.

PLATEAU'17 Workshop on Evaluation and Usability of Programming Languages and Tools, October 23, 2017, Vancouver, CA

The Curse of Text. It is easy to see why understanding software is hard: Reading code requires reading text that contains structured information in a language that does not follow the same logic of natural language. To understand a fragment of code, a developer has to mentally parse a source file, identify and extract the necessary information, and build a *mental model* of the (intended) behavior of the software. The same process happens when printing log messages to expose the state of the system: Log messages embody fragments of information that the developer has to fit into her mental model, and use it to reverse engineer the source of an error by trial and error. To ease this process, both researchers and industry built a plethora of tools like debuggers and code inspectors, that allow developers to run a program in a controlled environment, and to check the internal status of its variables. Other tools, like code browsers, support fast linking between the entities in the code, while loggers allow to print and store useful runtime information. Finally, test suites allow to define a set of expected behaviors, and to constantly check if any of these rules is satisfied.

However, these tools do not change the fundamental way we interact with the code: Eventually, the developer needs to read the code, and therefore undergo the process of building its mental model. This is because all these tools rely on the same, strong, underlying assumption: Source code is text, therefore the tools we are using to interact with it are shaped around text editing tools. This assumption reflects the way we use to store our programs, *i.e.,* plain-text files containing the declaration of our models.

We propose a novel approach for runtime data collection as an extension for ShoreLine, a platform for automatic collection of runtime exceptions [6]. We advocate the use of reified entities to store information about an exception, in order to preserve the multidimensional nature of the information, and leverage the implicit properties that can be obtained by the data structure. Describing errors as first-class citizens of a system without flattening the information into text, allows us to leverage the expressive power of logs to support a number of development activities. Using a structured data source allows developers to build a set of specialized tools to browse the data in an incremental fashion and discover its implicit structure. It also enables the use automated analysis, by mitigating the need of a data cleaning phase. Finally, these entities can later be sent for storage, thus creating bug reports with a much higher level of detail and reliability than simple plain text.

2 Related Work

Several tools in both academic and industrial contexts use the data generated during development and debugging to enable a number of different analyses. Many aspects of development can benefit from leveraging this data, but among them it is interesting to consider two main areas especially oriented toward supporting software development: bug *fixing*, and *visualization* for program comprehension.

Fixing bugs. The first major development activity that benefits from runtime data is bug fixing. The purpose of the research in this area is to support and automate the localization of the code that contains an error, thus alleviating the developer from the burden of walking through the whole execution path to localize the cause of a bug.

Several approaches use techniques to gather system information and detect errors in an automated fashion. For example, researchers collected large volumes of stack traces to identify patterns in the errors of a system, to assist the early detection of new problems or regressions, and to build a knowledge base of common problems [2, 6, 12]. Zimmermann *et al.* performed a survey asking developers about the challenges they have to undergo while dealing with bug reports, finding that one of the biggest problem comes from the reliability of the reported data [20], hinting at the need for an automated approach that collects meaningful data.

Cleaning the data in log files is also an issue when inspecting the data, or while performing analyses. For example, Aye proposed a preprocessing stage to overcome the problem of huge log files in web applications, with the purpose of cleaning the data to allow a subsequent mining step [3].

Comprehension and Visualization. Researchers also used the massive amount of data produced by the execution of a system to create a view of the system at a global level, to detect hidden interactions or unexpected patterns and give an overview of the system.

For example, Koike proposed a tool to visualize log files of the Snort[1] intrusion detector and assist system administrators to identify intrusion attempts in a system [13]. Moreta and Telea visualized log files using hierarchical clustering to uncover patterns of interest, with the purpose of monitoring dynamic allocation of memory and support the analysis of software repositories [16]. Orso *et al.* proposed a tool to monitor the logs of deployed software by means of visualizations generated by data mining techniques applied on runtime execution data [17]. De Pauw *et al.* built a tool to visualize the execution of Java programs, with the purpose of aiding the developer to understand the execution of the program and identify problems like performance bugs [8].

The approach by Dal Sasso *et al.* collected data from different data sources, and combined them to create a high level view of the usage of the system [6]. They showed that large amount of logging data and user interaction data can show hidden paths of usage of the entities of the system.

Finally, researchers also tried approaches to improve the textual representation of software artifacts by augmenting their description with a markup language [4, 14]. While these approaches are focused on describing source code, they are still relevant for program comprehension to support bug fixing, as they explicitly render the properties that are hidden in the textual form of the source code.

3 A Domain Specific Reporting Engine

In this section we outline our approach for collecting information about runtime errors. We explain the benefits of collecting this runtime data and show how and why development can benefit from modeling this information. The final goal is to integrate the resulting framework into a modern development environment, to enable smooth and descriptive fruition of debugging information, and to provide the groundwork for building interactive tools that present the data in a meaningful context.

3.1 Who Needs Models?

The purpose of a programming language is to equip the developers with the means to communicate, both to a machine and to other people, the intended behavior of a program. Therefore, we can view a program as the crossroads between the high level intent of the developer and the machine language that details the steps needed to accomplish it.

Clinging to the idea of a language that feels natural to describe algorithms, developers kept using the tools used for text editing to also manage source code. The large number of specialized tools that usually enrich the development experience in a text editor never evolved beyond its underlying representation, making writing source code mainly a string manipulation process.

Plain text has numerous advantages, but it has one major drawback: It employs a *flat* format do describe structured data, thus losing many properties. This means that the information that is contained in the data is not directly accessible by the developer, but hidden in the underlying implicit structure that has to be rebuilt, for example with a parser. Paraphrasing the allegory of the Cave of Plato, we are trying to

[1] https://www.snort.org/

learn the behavior of the entities in our system by looking at the shadows they project on the wall, represented by the textual representations [18]. While the goal of this work is not to criticize how we represent source code, it still helps us to comprehend how developers perceive software development, since the very beginning of their training. Unsurprisingly, if we write and think about source code in terms of text, the natural consequence is to treat as text also the product of the execution of such code. As a result, the majority of logging frameworks and bug reporting systems collect messages in a text-like format. We think we can improve how we deal with runtime errors by preserving their structure (*e.g.,* the structure of the involved objects at runtime), thus retaining the relations among the entities of the system, allowing fine grained analyses.

Researchers explored different representations for source code, like srcML [14] or JavaML [4]. In a similar fashion, the Smalltalk programming language proposes a system to store and access its source code that differs significantly from the usual text file approach. Smalltalk proposes an approach where the whole system is contained in a single file named *image*. This file contains a serialized version of the core system, its libraries, its IDE and tools, the code that the user writes inside the system, and the entire execution state composed of the existing objects when the image is saved. Therefore, the user does not write a program through a normal text editor, but uses the internal browser of the system to navigate its code, and can use *inspectors* to examine objects at any time. Using this approach allows Smalltalk to achieve full *liveliness*, as the whole system (both the source code and the runtime) can be manipulated programmatically. Inspired by the Smalltalk image example, there is no reason that prevents us to apply this approach to runtime generated data, to increase the capabilities of the development environment.

3.2 Design of the Framework

We want to record the behavior of a program in a structured and customizable way, creating a logging system to talk with objects and extract targeted information. Our first step is to define a model that describes the data we want to observe. The goal is to collect reified debug messages with a level of detail as near as possible to the original objects in the running system.

The model. A running system is a complex entity with several unknown variables: It is not possible to provide a complete and manageable description of its state. The usual approach to trace the source of an error is to verify the state of the program by means of log messages or of an object inspector. Both cases have a fundamental problem: To isolate the error, the developer has to identify an unexpected behavior, select the entities to observe, change the program to output these properties, and finally correct the program. This workflow implies that every change requires a new run of the system. While this might not be a problem in simpler development scenarios, it might become one when dealing with nondeterministic code, like in concurrent systems, or in programs depending on external input. Unfortunately, these scenarios are also the hardest to manage, suggesting that they require particular support from debugging tools. For example, different executions of a multithreaded application would result in different internal states: An error like unprotected access to a resource would appear only under certain conditions, resulting in what is called an *Heisenbug* [11].

Another important aspect during debugging is the reproducibility of the error. Understanding the condition under which a specific error occurred and reproducing is a complicated and time consuming part of the debugging process: Developers do not have access to the environment that generated the error, but they have to infer it from the data reported by the user. Users, however, cannot be expected to have the technical background to report a bug, which leads to a problem in the reliability of the information available to developers [20].

Our goal is to have an explicit and flexible model to allow developers observe the state of the system with a low effort. Such model would alleviate the developers from building a mental model of the system they are debugging, thus reducing the cognitive cost for fixing a bug.

We set the guidelines for designing such a framework for reified data collection as:

1. whenever possible, we collect the original entity that is involved in the event that we are observing;
2. when collecting the whole entity is not possible, we create and store a simpler representation;
3. we want a framework easy to extend and customize;
4. we should not collect data we are not interested in;
5. we should be careful in handling possibly sensible data.

The guideline (1) defines the core of our approach: We want information from an entity in the system. Therefore, we avoid to prematurely flatten the colleceted information: We store the whole entity, and delay its serialization, waiting for future instructions on how to use the information.

There are, however, some cases where the whole entity is not suitable for reporting. This is especially true with entities that might change their status for external causes or entities that might expire, for example in the case of database connections, or short lived sessions in a multiuser system.

In some other cases, we might not want our collection to expose sensible data, like passwords or private source code, as detailed in guideline (5). In this case, we apply the guideline (2): if it does not make sense to collect a piece of information, we anticipate the simplification process to create a safe copy of the original entity, and collect the cleaned version. Since it is not possible to generalize all the possible cases where we do not want to collect specific data, we rely on guideline (3): Our framework must allow easy customization of its details.

As an example, consider the case where we want to log the errors that users get while accessing a resource. Usually, we would write a line into a log file, to record the user and the action, leading to potentially huge log files. Using the approach we defined, we can setup a rule that activates only when the system generates an error involving the user, and store the entity of the user (guideline 1). In this way we can access the actual entity related to the user, query it about its associated session and the action that the user was performing. We can also avoid to collect sensitive data like passwords (guideline 2). This enables a conversation with the entities in the system, allowing the developers to customize interactive tools that empower the user with the ability of browsing the entities related to the error.

Collectors. We need a strategy to let the user of our framework describe its own custom data collection, to implement the flexibility required by our approach (guideline 3). To address this aspect, we define the concept of *collectors*: Small entities that describe how and when to observe a part of the system. A collector has three main purposes: (1) define how to collect some data; (2) define when to collect some data; (3) describe itself. The main goal of the collector is to define relevant data. For example, in the logging example, the system will pass some context to the collector, that will copy the user entity and remove the sensitive data, like the password, or mask the username, if the purpose of the collection is to be sent remotely and published in a bug report. When to collect the data is the other crucial aspect of the framework: We are defining an approach that defines a domain specific data collection. Collecting data about a user action is meaningless if we are dealing with an error generated by a string. Each collector has to know when to activate itself by analyzing the context of the error and checking its internal activation rule to decide whether to trigger the collection or not. Finally, we need a description to tag the collected data and present it to the user in an informative manner.

Using the approach of collectors we can build a system monitoring framework that is fully customizable and that collects first-class, reified entities. Such a framework can be employed in place of logging messages with a detailed snapshot of the state of a program, that can be then browsed with interactive tools, or that can be employed to collect failure data an pack it for remotely reporting an unexpected behavior. This remote reporting mechanism can be the first step towards a smarter bug reporting system, that allows a deeper inspection of the state of a system, while preserving the privacy of its users.

4 Implementing The Framework

We now present the implementation details of the framework for the Smalltalk programming language. We chose to implement and test the effective feasibility of our approach using *Pharo*[2], an Object Oriented programming language inspired by Smalltalk. Pharo inherits a number of powerful properties from its Smalltalk origins, that can support our task of implementing the data collection framework. In particular, it is a *live* programming environment, with full reflectivity capabilities, and a control over the whole system that allows to access and manipulate programmatically the complete state of the program. The fact that the entities in the system are abstracted by means of objects allows us to easily inspect faulty states of the system by interacting through the `Context` object, and simplifies the reification process of the interesting entities.

While the use of Pharo enables full flexibility and control over the execution of a program, one may wonder whether this hinders the applicability of the approach to more general examples. We believe that this does not affect the possibility to implement an analogous framework for a different programming language. Section 6 contains a deeper discussion about the generalizability of our approach.

4.1 Implementation Details

The main benefit of Pharo is that its abstractions describe the whole system runtime, allowing us to inspect and manipulate it by querying objects: We can stop the execution of a program, and access the whole system status at the moment of the interruption. The principal element we are interested in is thisContext, a special variable that stores an instance of Context. This object mimics the behavior of an *activation record* and contains all the information about the current execution of the program.

Implementing Collectors. We based our framework on the idea of *collectors*. In Section 3.2 we defined a collector as an entity that knows how and when to collect data. The strategy of a collector can be implemented with a class, thus decoupling the collector from the source code it is observing and providing a behavior that can be plugged and un-plugged seamlessly. While having a class for each collector might seem overkill, it has the advantage of providing full control to the user of our framework and the flexibility to select the data she wants to observe. A user can create a new collector by subclassing the class DataCollector, and implementing the four methods that define its behavior: *#tag* — the name of the method, used to reference the collected data by means of an automated approach; *#description* — a short description of the data collected by the class, displayed to the user when presenting the data or when asking for permission to send the data to the issue tracking system; *#when:* — an expression that evaluates the state of the system to decide whether or not the collector is interested in observing the current context; *#initializeFrom:* — the main method that implements the strategy for extracting the data. Both the *#when:* and *#initializeFrom:* method receive an object of type

[2] https://pharo.org

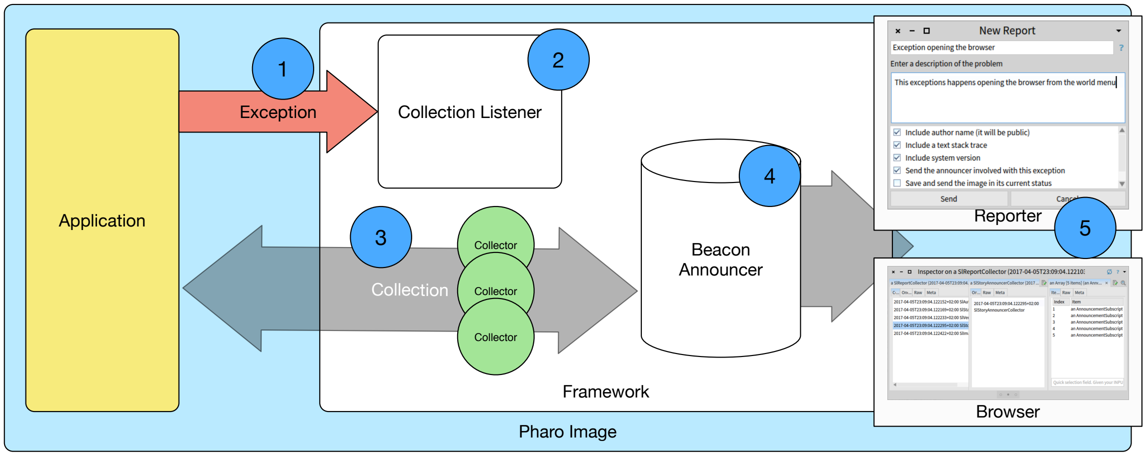

Figure 1. The workflow to collect data using collectors, showing the architecture of ShoreLine

Context as parameter, that contains the execution environment. The *#when:* method determines if the context is relevant to the collector, while *#initializeFrom:* performs the actual collection.

Triggering the Collection. We decided to trigger the data collection in two cases: For the handling of errors, or arbitrarily triggered by the user. The former is invoked automatically whenever an unhandled exception occurs, while the latter needs to be explicitly invoked using the ShoreLine public APIs. Figure 1 shows a diagram of the flow of the data from the collection to its usage. The collectors evaluate whether they should activate, and potentially perform the data collection. Once the collection is complete, the framework composes a Report object and announces its creation using *Beacon*,[3] an announcement-based (*i.e.,* publish/subscribe) logging framework for Pharo. *Beacon* broadcasts messages to the system to inform the interested tools of the presence of a report.

By collecting complex entities in form of objects, rather that text, we can initiate a conversation with the system and allow a systematic and progressive exploration of the errors.

4.2 Using the Data

Once a report is broadcast, every interested tool receives the data. This is intended to further improve the customizability of the framework, allowing the developer of a system to refine their tools for quickly inspect the data collected about their code, as proposed by guideline (3).

The two applications proposed by default by our approach consist in a local data browser, and a customized reporter. If users are interested in browsing the data locally, for example during the development phase of a project, they can inspect the contents of the report objects. Moreover, they can exploit the tools provided by the Pharo ecosystem, like the *Glamour Toolkit* [10] to create custom visualizations of the data to support the browsing session. If needed, the system can also serialize the report and send it to the issue tracker with a comment of the user explaining how she encountered the error. In the next section we show how to implement a collector to solve common development problems.

5 The Framework in Action

In this section we show how accessing specific information can support developers in quickly understanding the cause of a defect and the behavior of a piece of code. We first present an in-depth case study together with a possible implementation showing how our approach can support debugging errors in applications using the Announcer framework. We then outline a scenario about how our framework can support debugging third party libraries with complex entities.

5.1 The Announcer Story

The continuous evolution of software requirements results in a codebase that is constantly growing, both in size and complexity. To tame this problem, developers design software systems using a modular architecture, where large tasks are split into smaller functionalities, so that complex operations can be managed by composition of small entities. In such a scenario, communication between the modules is fundamental in defining the behavior of the system. Modularity is invaluable in developing, testing, and maintaining a system, but it comes at a cost: Integrating different modules can cause errors generated by the interaction between components and might also trigger nondeterministic behaviors. Since the flow of the execution is distributed into different locations, tracking the source of a defect can become a complicated and time consuming task. To enable communication among system components, Pharo offers the *Announcer* framework: a tool that implements an improved version of the Observer pattern and reduces coupling. The strength of the Announcer framework is that announcements are first class entities:

[3] See www.smalltalkhub.com/#!/~Pharo/Beacon

Table 1. Summary of the collected stack trace data

# of developers	257
# of stack traces	41,129
# of traces involving announcements	4,840
% of traces involving announcements	~12%

Once it occurs, an event is represented by an object that can contain arbitrary data. The use of announcements to manage the communications between different applications has numerous advantages, like loosing the coupling between the publisher of an event and its subscribers, and is a recommended best practice in developing an application in Pharo.

However, as discussed earlier, fragmenting the control flow of the program into a set of disjoint components carries the drawbacks of event-based programming with the consequence that finding the right fragment of code that is responsible for an error becomes a convoluted process of navigating through the callbacks to find the correct location in the system. This complicates the debugging process, as understanding how an announcement propagates through the system requires accessing information that is usually not expressed in the stack trace generated by an exception. The problem with debugging an announcement is that it follows a different logic than the usual sequential style of the rest of the system. Therefore, while all the information necessary to understand an error is available during an exception, this is usually not exposed by the tools used to catch and report the errors, like the system logger.

Usage of Announcements. During the development of Pharo 6 a problem emerged, where selecting an item from a menu would trigger the opening of two duplicate windows, instead of one. During the discussion on the bug report it became clear that the incident was compatible with the case of an entity registered twice in the announcer responsible for opening the window. Debugging such a problem consists in locating the entity that contains the double registration and remove one of the two snippets of code that perform the subscription. While this case is not directly a consequence of an exception, it shows how debugging the behavior of code using announcements can be tricky, and that development tools could be improved to support similar cases. We therefore conducted a brief experiment to investigate how common are problems involving announcements in the exceptions that developers usually trigger while writing code. We inspected the data collected through ShoreLine, a tool to intercept stack traces from development exceptions and report them to a central server to support debugging [7]. We considered the stack traces collected from 10 June 2014 to 28 February 2017. Table 1 shows a summary of the collected data. The collecting tool can be set to submit every exception automatically, or to ask the developer for explicit submission. The stack traces come from exceptions generated by users and developers in the Pharo community during their daily development. We collected 41,129 stack traces from 257 different developers, on a time period of almost three years. We queried the collected data looking for references to *AnnouncementSubscription*, the class responsible to dispatch the announcement to the registered entities, finding that 4,840 stack trace contain at least one reference to this class. This means that almost 12% of the exceptions that were collected by our tool as a result of a system exception, involve the usage of the Announcement framework. While this result does not imply that the Announcer framework is directly responsible or involved with the error, it shows that more than one exception every ten has in its source a relation with an announcement, hinting that the scenario is frequent enough to require a dedicated support by the debugging tools.

Implementation of the collector. Our goal is to collect and present domain-specific information about the message dispatching. We can use this information to refine the inspection tools used to investigate the system, or to create interactive bug reports that allow to inspect the objects of the original exception. To implement the collector we need to define its activation conditions and the data extraction. The two main causes that can generate errors using an announcer are multiple registration and the potentially nondeterministic behavior, given by the fact that messages are dispatched in no specific order, causing bugs that are hard to reproduce. We therefore focus on four features: (1) the subscribers of the announcer listening for the specific announcement, (2) the announcement being dispatched, (3) the subscriber that generated the exception, and (4) the list of subscribers that already received the announcement compared to the list of subscribers that did not receive it yet. Figure 2 shows the implementation of *AnnouncerCollector* the class responsible for gathering data about an announcement.

The class methods *#tag* and *#description* describe the collector, for indexing and user interaction purposes. The method *#when:* verifies if there is a reference to a AnnouncementSubscription object in the first 10 lines of the method invocation stack, to ensure that announcements are involved in the exceptions in the immediate surroundings of the current context. *#initializeAnnouncementFrom:* extracts the announcement that triggered the broadcasting process and the data it contains; *#initializeSubscribersFrom:* extracts all the entities registered to the announcer, regardless of the kind of announcement they are listening to; *#initializeInterestedSubscribersFrom:* extracts the distribution list of the announcer. Since this data is collected during the execution time, this list contains the order in which this announcement is being distributed, thus removing the nondeterminism. Moreover, the method also extracts the index of the current entity: In this way the developer can access the list of the entities that already received the announcement and the one of the entities that did not receive the message, thus helping the detection of conflicts between different subscribers.

```
AnnouncerCollector class>>
tag
      ^ #'story-announcer-collector'
```

```
AnnouncerCollector class>>
description
      ^ 'Story Announcer collector'
```

```
AnnouncerCollector>>
when: aContext
      | stackSelectSize |
      stackSelectSize := aContext stack size min: 10.
      ^ (aContext stack first: stackSelectSize) anySatisfy: [ :e |
            e receiver class = AnnouncementSubscription ]
```

```
AnnouncerCollector>>
initializeAnnouncementFrom: aContext
      announcement := aContext stack second arguments first
```

```
AnnouncerCollector>>
initializeSubscribersFrom: aContext
      | announcer |
      announcer := (aContext stack detect: [ :e |
            e receiver class = AnnouncementSubscription ])
                  receiver announcer.
      subscribers := announcer subscriptions subscriptions
            collect: #subscriber
```

```
AnnouncerCollector>>
initializeInterestedSubscribersFrom: aContext
      | arguments |
      arguments := (aContext stack detect: [ :e | e method =
            (SubscriptionRegistry>>#deliver:to:startingAt:) ])
                  outerContext arguments.
      interestedSubscribers := arguments at: 2.
      index := arguments at: 3
```

Figure 2. The Smalltalk code implementing the extraction strategies for the Announcer collector

The data collected is still composed of objects, that can be further queried: This process allows to retain the maximum amount of information for the longest time needed, while still allowing for a later flattening or serialization. The entities could also be sent to the issue tracker as serialized objects, so that the maintainers of a software can navigate the errors generated by the users with a higher degree of flexibility and introspection than just plain text.

This scenario shows how our framework can help developers to extend the behavior of the logging mechanism and collect domain specific data about their code. By distributing software together with custom collectors, a developer can effortlessly obtain detailed information about the behavior of selected parts of her code.

5.2 Debugging Third Party Libraries

Library developers can benefit from collectors to observe for specific data about their project. For example, in the Smalltalk ecosystem *Roassal* is a popular visualization engine [1]. Roassal codebase consists of more than 800 classes and almost 6,000 methods, and it is constantly evolving using community feedback. Such a large project poses several maintainability challenges, especially since the community is split among stable, legacy, and development releases.

Understanding and reproducing the causes of an error can become complex, as the developers need information that might not be easy to provide. The maintainers of Roassal can improve this situation observing data specifically related their model: They can either set default collectors to detect problems that can arise, or they can react to existing bugs to detect specific errors. For example, when an exception occurs, a collector can verify if the source entity is a subclass of *RTObject*, or if the error happens inside a builder —a Roassal object to generate visualizations from a collection of data. By collecting domain specific information, the maintainers can get a detailed picture of the error, and restrict the possible causes without the need to access the whole data of the user. For example, by knowing the number of nodes that a visualization is rendering, one could tell if the error is due to a memory problem, or if the visualization has scalability issues. Also, knowing the settings that were used to configure a builder can tell if there is a bug in the builder's code, or if the public API is poorly designed and therefore often misused by the users. Finally, knowing the kind of data that a visualization received can help in finding if there is a bug in managing objects of different (specific) types.

Shipping their own collectors for observing their code, developers can support debugging in the context of the project, therefore reducing the time required to understand an error and the cost for maintenance.

6 Conclusion

We presented an approach to define ad-hoc collection of runtime data to support debugging. By extending our framework, a developer can define a custom strategy to gather reified domain-specific knowledge about an application. By preserving the structured, object-oriented nature of the collected data, rather than flattening it into text, we are able to query the state of a program and observe it by filtering the relevant data, providing more expressive reports. We can use this approach to create flexible inspection tools, that offer a deeper representation of the execution context of a program.

By giving the possibility to report and collect specific information from the system, our framework offers data that is more reliable than a stack trace submitted by a user, and allows us to deal with the collected data in an automated fashion, performing tasks that would otherwise weight on the maintenance cost. Moreover, providing a trusted structure of the data, our framework enables a number of analyses without the need of information retrieval and text mining techniques to clean the data. Finally, dealing with data that is not flattened allows to perform a progressive inspection of a report, enabling the discoverability of complex data and structuring the debugging session as a browsing process.

Generalizability of the approach. We developed our approach using Pharo, an object-oriented, live programming environment inspired by Smalltalk. The strong reflection and inspection capabilities of the platform allowed us to access the unmodified execution context of the software. Given these premises, one might wonder (1) why should this approach be relevant in the Pharo ecosystem, and (2) if it is still relevant outside Pharo, when trying to apply it to other programming languages. To answer question (1), this framework comes after a long collaboration with the Pharo community, to understand the types of errors that users get during the use and the development of the platform. As we discussed through the paper, the data collection mechanism can be integrated with the issue tracking system of a project, allowing the developers of a project to integrate our framework in their workflow, supporting debugging and maintenance tasks. About question (2), we believe that such an approach can also be employed in other programming languages. The Pharo system provided the perfect candidate to prototype such a framework, easing the implementation process by providing the APIs to talk with the system and the tools to navigate the collected data, but the use cases we have shown in Section 5 can be implemented with any language with reflective capabilities.

Given the current traction of DevOps technologies like Docker, it is also interesting to consider how it is possible to execute an application in a Docker container, stop it and save its status during an exception, and submit the container to a remote server. Analyzing the stored data would not be simple, as there is a lack of tools to access the state of applications in these circumstances, but our approach could be an interesting match for these similar scenarios.

Current State and Future Work. Building collectors to observe specific parts of the system can improve the workflow of debuggers, and reduce maintenance costs. The regular collection of domain specific data can provide statistics on the frequency of errors in selected parts of the system, and hint how a software is used, hence helping developers not only to debug a system, but also to optimize existing code and improve its API. We plan to relase our framework for Pharo, proposing a set of generic collectors to support standard difficult debugging tasks, like the one that we proposed in Section 5.1 about the Announcer framework. By reaching a larger user base we can collect more specific data about the usage of the tool, refine further the collectors implementation and evaluate the impact of such an approach in daily development activities.

We envision a future where development activities are supported by the system using the language of the system, without flattening the information into chunks of plain text, but rather using first-class entities that narrate the precise status of a program. Starting a conversation with the entities we can develop a paradigm of programming that focuses on models, rather than manipulating strings, and achieve a programming environment that is really live and responsive.

Acknowledgements. We acknowledge the Swiss National Science foundation's support for project 146734 "HI-SEA".

References

[1] Vanessa Pena Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. 2013. Agile Visualization With Roassal. *Deep Into Pharo* (2013), 209–239.

[2] Dorian C Arnold, Dong H Ahn, Bronis R De Supinski, Gregory L Lee, Barton P Miller, and Martin Schulz. 2007. Stack trace analysis for large scale debugging. In *Proceedings of IPDPS 2007 (IEEE International Parallel and Distributed Processing Symposium)*. IEEE, 1–10.

[3] Theint Theint Aye. 2011. Web log cleaning for mining of web usage patterns. In *Proceedings of ICCRD 2011 (3rd IEEE International Conference on Computer Research and Development)*, Vol. 2. IEEE, 490–494.

[4] Greg J Badros. 2000. JavaML: a Markup Language for Java Source Code. *Computer Networks* 33, 1 (2000), 159–177.

[5] Thomas A Corbi. 1989. Program understanding: Challenge for the 1990s. *IBM Systems Journal* 28, 2 (1989), 294–306.

[6] Tommaso Dal Sasso, Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. Blended, Not Stirred: Multi-Concern Visualization of Large Software Systems. In *Proceedings of VISSOFT 2015 (3rd IEEE Working Conference on Software Visualization)*.

[7] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. 2015. Misery Loves Company - CrowdStacking Traces to Aid Problem Detection. In *Proceedings of SANER 2015 (22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering)*. IEEE CS Press, 131–140.

[8] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. 2002. Visualizing the execution of Java programs. In *Software Visualization*. Springer, 151–162.

[9] Richard K Fjeldstad and William T Hamlen. 1983. Application Program Maintenance Study: Report to Our Respondents. *Proceedings Guide* 48 (1983).

[10] Tudor Girba, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. 2013. Glamour. *Deep Into Pharo* (2013), 192–207.

[11] Michael Grottke and Kishor S Trivedi. 2005. A Classification of Software Faults. *Journal of Reliability Engineering Association of Japan* 27, 7 (2005), 425–438.

[12] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. 2012. Performance Debugging in the Large via Mining Millions of Stack Traces. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 145–155.

[13] Hideki Koike and Kazuhiro Ohno. 2004. SnortView: Visualization System of Snort Logs. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*. ACM, 143–147.

[14] Jonathan I Maletic, Michael L Collard, and Andrian Marcus. 2002. Source Code Files as Structured Documents. In *Program comprehension, 2002. proceedings. 10th international workshop on*. IEEE, 289–292.

[15] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer – An Investigation of How Developers Spend Their Time. In *Proceedings of ICPC 2015 (23rd IEEE International Conference on Program Comprehension)*. 25–35.

[16] Sergio Moreta and Alexandru Telea. 2007. Multiscale visualization of dynamic software logs. In *Proceedings of the 9th Joint Eurographics/IEEE VGTC conference on Visualization*. Eurographics Association, 11–18.

[17] Alessandro Orso, James Jones, and Mary Jean Harrold. 2003. Visualization of program-execution data for deployed software. In *Proceedings of the 2003 ACM symposium on Software visualization*. ACM, 67–ff.

[18] Plato. 380 b.c.. *De Res Publica*.

[19] Marvin V Zelkowitz, Alan C Shaw, and John D Gannon. 1979. *Principles of software engineering and design*. Prentice-Hall Englewood Cliffs.

[20] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering (TSE)* 36, 5 (2010), 618–643.