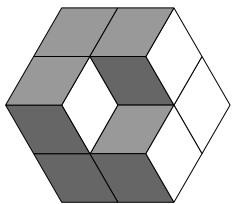


Università
della
Svizzera
italiana

Software
Institute



Reification as the Key to Augmenting Software Development

An Object is Worth a Thousand Words

Tommaso Dal Sasso

Prof. Dr. Michele Lanza
Research Advisor

Dissertation Committee

Mehdi Jazayeri Università della Svizzera Italiana, Switzerland
Cesare Pautasso Università della Svizzera Italiana, Switzerland
Rocco Oliveto University of Molise, Italy
Martin Pinzger Alpen-Adria-Universität, Austria
Andy Zaidman Delft University of Technology, The Netherlands

Dissertation accepted on 14 June 2018

Research Advisor
Prof. Dr. Michele Lanza

Ph.D. Program Co-Director
Prof. Walter Binder

Ph.D. Program Co-Director
Prof. Olaf Schenk

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Tommaso Dal Sasso
Lugano, 14 June 2018

*To my father, who taught me how to be a man.
To my son, who teaches me what it means.
To my mother, who taught me what love is.
To my wife, who teaches it to me every day.*



Alice: But I don't want to go among mad people!

Cheshire Cat: Oh, you can't help that. Most everyone's mad here. You may have noticed that I'm not all there myself.

Alice in Wonderland



Abstract

Software development has become more and more pervasive, with influence in almost every human activity. To be able to fit in so many different scenarios and constantly implement new features, software developers adopted methodologies with tight development cycles, sometimes with more than one release per day. With the constant growth of modern software projects and the consequent expansion of development teams, understanding all the components of a system becomes a task too big to handle.

In this context understanding the cause of an error or identifying its source is not an easy task, and correcting the erroneous behavior can lead to unexpected downtime of vital services. Being able to keep track of software defects, usually referred to as *bugs*, is crucial in the development of a project and in containing maintenance costs. For this purpose, the correctness and completeness of the information available has a great impact on the time required to understand and solve a problem.

In this thesis we present an overview of the current techniques commonly used to report software defects. We show why we believe that the state of the art needs to be improved, and present a set of approaches and tools to collect data from software failures, model it, and turn it into actionable knowledge. Our goal is to show that data generated from errors can have a great impact on daily software development, and how it can be employed to augment the development environment to assist software engineers to build and maintain software systems.

Acknowledgements

“Oh, I’ve had such a curious dream!” said Alice, and she told her sister, as well as she could remember them, all these strange Adventures of hers that you have just been reading about; and when she had finished, her sister kissed her, and said “It was a curious dream, dear, certainly: but now run in to your tea; it’s getting late.”

So Alice got up and ran off, thinking while she ran, as well she might, what a wonderful dream it had been.

The years during my PhD were, unsurprisingly, among the most intense in my life. During these years my world changed—several times. Most of all I became a father, the most astonishing change that I have experienced in my life.

Many crossed my path during these years and contributed in shaping this experience. I thank you all for the path you shared with me: I hope I was able to give you at least a fraction of what I received.

Thanks to Michele, advisor and friend. I know I have been an unconventional PhD student, if anything because no PhD student is a conventional one. Nonetheless, you had the patience to wait for me during weird times. Maybe not crazy-Rumanian weird, but still weird times. Yours was an example that left a mark in me. I was no Mike Ross, but I still I hope that my passing left a trace in the REVEAL family.

Speaking of which, a big thank goes to the REVEALers, in particular those who were, for me, its *hard core* members: Andrea, Luca, and Roberto, who shared with me this journey from the beginning. I loved you. And I hated you. Often both at the same time. I find amazing how much we changed together during these years. You are family to me.

Thanks to the new generation REVEALers who shared with me this last part of my journey: Bin, Emad, Jevgenija, Csaba, and Gabriele. You brought a fresh touch into the group and it was a lot of fun. I wish you the best of luck with your adventure.

Thanks to the Dean’s office, for accepting my madness and backing it up with even more madness. Oh, yes, and for the chocolate, too.

Thanks to the Tymchuk family, Yuriy, Natalia, and Sophia. You literally made us part of your family. I know that our bond will last in time, regardless of how far apart we will be.

Thanks to the Pharo community, who allowed me to bring ShoreLine to life. I had to almost implement my own rest server—several times—to do that but, hey, it was fun.

Thanks to Alberto Bacchelli, who brought me to Lugano in the first place and occasionally acted as ghostwriter. I owe you a lot for inviting me during my master’s thesis. We share so many inside jokes that I am not able to pick one. *Fottutorato*.

Thanks to Marco D’Ambros, who took part with me in a lot of funny adventures during my early days in Lugano, even if he remembers none of them.

Thanks to Giulia *Pizzinato*, for being on our side when we needed it. I’m happy that Samuele had the opportunity of knowing you and witnessing your sparks of crazy happiness. There will always be pizza for you at our home.

A special mention goes to the people in charge of making badges at conferences: Rarely I have seen such a creativity as the one that you used in writing my name. I have to admit that “Sasso, Tommaso Dal” is a masterpiece.

Finally, a deep thanks goes to my family who, directly or indirectly, lived this adventure with me. Mom and Dad, you supported me in ways that I just started to understand, now that I am a father. I am proud of the name I bring: You made me part of two families who share a silent, powerful bond.

Samuele, my son, you brought the light in our lives. Looking at you exploring the world reminds me of how fulfilling it is to improve a little every day. I will always be proud of you.

Elena, you have been a constant presence at my side. We have been through a lot and I am proud of what we have done. I know that we have wonderful things waiting for us. It will be amazing.

So Long, and Thanks for All the Fish
Tommaso

Contents

Contents	xi
List of Figures	xv
List of Tables	xvii
1 Introduction	1
1.1 Dealing With Software	2
1.2 Thesis Statement	2
1.3 Validation	3
1.4 Roadmap	4
2 State of the Art	5
2.1 Issue Tracking Systems	6
2.2 Visualizing Bug Reports	8
2.2.1 Visualizations of an Issue Tracker	8
2.2.2 Visual Storytelling	8
2.3 Bug Reports	9
2.3.1 Quality of a Bug Report	9
2.3.2 Bug Prediction	10
2.3.3 Bug Reports and Social Interactions	10
2.4 Data Collection	10
2.4.1 Collecting Runtime Errors	11
2.4.2 Errors as First-Class Citizens	11
2.4.3 Aiding Bug Fixing	12
2.4.4 Aiding System Comprehension	12
2.5 Engaging Developers and Users	12
2.6 Outline	13
3 Visual Analytics of Bug Repositories	15
3.1 Accessing Bug Repositories	16
3.2 IN*BUG in Detail	18
3.2.1 Main view	18
3.2.2 Details of a bug	18
3.2.3 Implementation & Current Dataset	20
3.3 More Than Meets the Eye	21
3.4 Outline	22
4 CrowdStacking Traces to Aid Problem Detection	23
4.1 Collecting Runtime Errors	24
4.2 On the Nature of Stack Traces	25
4.2.1 Interpreting Stacktraces	27

4.2.2	A Practical Use Case	27
4.3	CrowdStacking Traces	28
4.3.1	Data Collection	28
4.3.2	Data Representation	29
4.3.3	Analysis on the Collected Data	29
4.3.4	Extracting Information	31
4.4	Preliminary Results	34
4.5	Discussion	37
4.5.1	The Data	38
4.5.2	The Approach	38
4.5.3	Applicability of the Method	39
4.5.4	Next Steps	39
4.6	Outline	40
5	Reified Collection of Runtime Errors	43
5.1	The Tools We Use To Develop	44
5.2	A Domain-Specific Reporting Engine	45
5.2.1	Who Needs Models?	45
5.2.2	Design of the Framework	46
5.3	Implementing The Framework	48
5.3.1	Implementation Details	49
5.3.2	Using the Data	50
5.4	The Framework at Work	51
5.4.1	The Announcer Story	51
5.4.2	The Testing Story	56
5.4.3	Debugging Third Party Libraries	57
5.5	Discussion	58
5.5.1	Next Steps	58
5.6	Outline	59
6	Multi-concern Visualization of Large Software Systems	61
6.1	Exploring a System	62
6.2	The Ingredients	63
6.2.1	Source Code Changes	63
6.2.2	SHORELINE REPORTER and Stack Traces	64
6.2.3	DFLOW and IDE Interaction Data	65
6.2.4	Blended, Not Stirred	65
6.3	Visualization Principles	66
6.3.1	In Practice	66
6.3.2	The City Metaphor: Layout and Metrics	67
6.3.3	Color Harmonies and Blends	67
6.3.4	Under the Hood	69
6.4	Telling Evolutionary Stories	69
6.4.1	Those Awkward Neighbors	70
6.4.2	Market Districts	72
6.4.3	New in Town	73
6.4.4	The Purple Buildings	74
6.5	Discussion	75

6.6	Outline	76
7	What Makes a Satisficing Bug Report?	77
7.1	Good Bug Reports vs. Real Bug Reports	78
7.2	Research Method	79
7.2.1	Research Questions	80
7.2.2	Online Questionnaire	80
7.2.3	Data Collection	81
7.2.4	Data Analysis Techniques	82
7.3	Results	83
7.4	Discussion	91
7.4.1	Threats to Validity	91
7.4.2	Next Steps	92
7.5	Outline	92
8	How to Gamify Software Engineering	93
8.1	The Rise of Gamification	94
8.2	Games and Gamification	96
8.2.1	Why Do We Play Games	96
8.2.2	Gamification: Principles, Promises & Perils	97
8.3	Gamifying Software Engineering: (Not) An Easy Game?	99
8.4	Software Engineering Gamification Framework	100
8.4.1	Gamification Building Blocks	101
8.4.2	Example I: The Myth and De-Bug	104
8.4.3	Example II: The Empire of Gemstones	106
8.5	Evaluating Gamified Systems	108
8.6	Discussion	110
8.6.1	Reflections	110
8.6.2	Next Steps	110
8.7	Outline	110
9	Conclusion	113
9.1	Visualization of Bug Data	113
9.1.1	Reading Between the Lines	113
9.1.2	Narrating the Evolution of a System	114
9.2	Collecting Failure Information	114
9.2.1	Collecting Stack Traces	114
9.2.2	Reifying Bug Reports	114
9.3	Modeling an Issue Tracking System	115
9.3.1	The Model of a Bug Report	115
9.3.2	Gamification	115
9.4	Limitations and Future Work	115
9.5	Closing Words	116
	Bibliography	117

A Pharo	127
A.1 Smalltalk in the 21st Century	127
A.1.1 Runtime Errors in Pharo	127
A.2 The Pharo Community	128
B List of Publications	129

Figures

2.1	The first "bug" report.	5
2.2	An old Bugzilla submission form	6
2.3	GitHub bug report submission form.	7
3.1	Example bug report in the FogBugz bug tracking system.	16
3.2	Main user interface of In*Bug	17
3.3	In*Bug details page showing the properties of a bug report	19
3.4	List of events in a bug report	20
3.5	The interactions of In*Bug with the <i>FogBugz</i> and <i>SmalltalkHub</i> services	21
4.1	Example of a stack trace collected from a runtime exception	25
4.2	Distribution of the stack traces on the Pharo system using a city like visualization	26
4.3	The interactive interface of ShoreLine Reporter	29
4.4	Distribution of the stack traces on the methods of Pharo using a city like visualization	30
4.5	Force graph representing the stack traces and their neighbors	33
4.6	The bug report 12973	36
5.1	The workflow to collect data using collectors, showing the architecture of ShoreLine	49
5.2	The reporting window for ShoreLine	51
5.3	A bug report describing a duplicated behavior	53
5.4	The Smalltalk code implementing the extraction strategies for the Announcer collector	55
6.1	The Blended City – visualization principles and proportions	66
6.2	The view depicted in Figure 6.1	67
6.3	Color wheel and triadic color scheme	68
6.4	Linear color blend on triadic color scheme	69
6.5	Aging process: example in the timeline	69
6.6	View of the city with all the activities	70
6.7	The architecture of the Blended City	71
6.8	Details of the packages <i>Graphics-Files</i> and <i>Compiler</i>	71
6.9	<i>Spec</i> and <i>Morphic</i> market districts	72
6.10	Changes in the Pharo system	73
6.11	The changes of <i>GT-Tools</i> packages	74
6.12	A view of the system highlighting stack traces and developer interactions only	75
7.1	Survey results	83
7.2	Conceptual diagram of the model of a new bug report	84
7.3	Transition graph of all the states in JIRA	86
7.4	Transition diagram of all the states in Bugzilla	88
8.1	Gamification Activity Template	100
8.2	Concrete Gamification Activity	105

Tables

3.1	Bug report event color codes	18
3.2	Summary data of the <i>Pharo</i> bug tracker	21
4.1	Summary of the stack traces collected from June to November 2014.	30
4.2	The 10 most called methods in the collected stack traces.	31
4.3	Summary of the most popular stack traces, with the popularity metrics.	34
5.1	Summary of the collected stack trace data	54
6.1	Source Code Changes	64
6.2	Stack Traces Data	65
6.3	IDE Interaction Data	65
7.1	Expertise of the participants of the survey (average)	81
7.2	Overview of the projects in the dataset	81
7.3	Contents of the dataset	82
7.4	Different states of bug reports in Bugzilla and JIRA, with the count of the reports currently in each state and the total sum of all the times a bug report reached a state.	82
7.5	Number of custom fields per project	89
7.6	Prediction results: Proportion of bug reports classified in the correct time bucket, with increment over random classification (25% correctly classified bug reports).	90
7.7	Prediction results for bug reports of last year.	90
8.1	Points acquired per 1 gemstone.	107
8.2	Required number and type of gemstones to obtain noble title.	108



Introduction

Everybody uses software. It may be the control system for a production environment of a large factory, a *Customer Relationship Management* system, a social network, an application on a smartphone, or a website to book a flight.

Regardless of its use, software plays a central role in modern society: It is used in almost every human activity to automate trivial tasks and to simplify complex ones. Its usage became massively pervasive, to the point that virtually any structured activity and process relies on software to regulate its workflow, minimize errors, and reduce costs. Moreover, the surge of popularity of mobile computing pushed even further the momentum of this scenario, bringing the influence of software into every aspect of our lives.

As a consequence, development teams are required to write code and push new features at a sustained pace, producing changes in the codebase that cause a system to constantly change and evolve its behavior, sometimes more than once a day. Often changes are performed by different developers, or even by different teams working on different parts of the system: Therefore, it quickly becomes impossible for a single person to have comprehension of the whole system. It is easy to understand how a software system can gradually start to resemble an unknown black box, rather than an organized entity composed of deterministic processes.

To complicate things even further, it is often not sufficient to have a deep knowledge of the whole set of components that form a system to predict its final behavior. Given the large number of requirements that modern software systems have to satisfy, it is usually necessary to rely on external libraries to provide the desired features. Using external code is a good practice, as it allows for reuse and reduces the probability of a bug being present in a library used by more people. However, these benefits come at the cost of yielding control of the system to external code, leading to the consequence that, if an error arises from a library, it becomes virtually impossible to track down its origin unless through debugging the library itself.

The main focus during a development cycle is writing new code to develop new features: However, in such a complex and ever-changing scenario, a huge part of the resources put in a project are spent in maintenance and debugging [Cor89, FH82, ZSG79, MML15]. We can break down the constituent components of maintenance in: finding and collecting problems, understanding them, locating the source of the error, and fixing the issue. Each one of these phases comes with its own set of problems and complexities.

In such a scenario, one would imagine that the efforts for assisting developers would focus on refined tools to navigate, understand, and inspect the code. While this is partly true, many of the modern editors and IDEs put the biggest accent on how developers write code, leaving program comprehension as a secondary task, despite it being intrinsically more complicated.

1.1 Dealing With Software

It is easy to sense why understanding software is hard: Reading code means reading text containing 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 [VMV95]. 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 practitioners built a plethora of tools, like debuggers and code inspectors, to allow developers to run a program in a controlled environment while checking 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 are satisfied.

All these tools however 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. The main cause for 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.

In this thesis we propose a different approach for thinking about runtime errors and software defects. The data generated from runtime errors contains large amounts of information that is usually ignored, or stored in a textual format, that loses its original relations with the entities of the system. We propose to promote this class of data —such as bug reports and log files— to full-fledged entities and store them using objects, in order to maintain the relation with the original entities living in the system. The effect of treating entities such as bug reports or log files as first-class citizens of the development cycle, is that it enables us to deal with objects instead of plain text. This, in turn, provides the language to turn the data into information in the correct context. Such a tight integration allows us to enrich the tools we use to develop a system and creates a feedback loop with the developer that gives her a broader view on the evolution of a system.

We believe that discarding this data, usually considered as a disposable byproduct of the development process, wastes the enormous potential of an important data source. We want to leverage the usefulness of this data and turn it into actionable information that can effectively impact the development process and reduce maintenance costs.

1.2 Thesis Statement

The goal of our dissertation is to rethink how we deal we bug reports: We seek to increase the reliability of the data they contain and reduce the time required to read and understand them. We want to lay the foundations for a new generation of issue tracking systems, that leverages the power of the community and implements automated approaches to provide relevant information to developers [ZPB⁺10] and alleviate the problem of noise (*i.e.*, duplicate or abandoned bug reports) inside existing issue trackers [WZX⁺08].

We formulate our thesis as follows:

Reifying bug reports, promoting them to first-class citizens of the development process, enables a conversation with a software system that reduces debugging time and enables automated and reliable usage analyses.

We analyze existing bug repositories to understand the kind of data that is collected and how developers exploit this information to fix software defects. We implement and test a set of exploratory tools to investigate the usage of bug reports in existing open source projects, and we design our process to collect data in a structured and reliable way.

1.3 Validation

Redesigning the concept of bug tracking is a task that encompasses a number of different sub-problems. Many of these problems come from the specifics of each development community and arise from the way developers approach their work.

Given the large number of variables in play, we feel that a canonical approach of selecting an idea, conducting a controlled experiment, and produce a conclusion would not be the best approach to narrate the problem we are studying. Instead, since we had the opportunity to propose our work to the PHARO community, collect data from their day-to-day development work, and given the practical nature of the problem we considered, we decided to embrace a tool-driven approach and seek for the feedback of developers. We believe that conducting a full-blown evaluation process on a new issue tracking system would have been an impossible task, as its effects would begin to be observable only after years from its adoption. Therefore we decided to present a set of exploratory studies using the fresh development data we collected, and evaluate our approaches based on the feedback from developers.

1.4 Roadmap

This dissertation is structured in the following chapters and with the following contributions:

Chapter 2 presents the history and evolution of tracking bugs, the current trends and best practices.

Chapter 3 proposes a visual approach to explore the content of existing issue trackers, showing how a simple textual representation sometimes hides useful information in a bug database. We present IN*BUG, a tool for visually inspecting the contents of existing bug repositories, find hidden properties, and recurring patterns. [DSL13, DSL14, DS14].

Chapter 4 presents our approach for runtime errors retrieval, where we collect stack traces generated by the community during the development process to learn about the life of a system. We implemented our approach into ShoreLine, a platform that we deployed for the collection and reporting of runtime errors [DSML15].

Chapter 5 extends our stack traces collection approach, to log entities in a reified fashion and to capture the information implicitly stored in the relation among the objects. We extend ShoreLine to allow the reified collection of runtime data and allow a better representation of the status of a system during a failure [DSCM⁺17]

Chapter 6 shows how we can employ the data we collect to build tools that combine heterogeneous data sources for browsing the evolution of a system from different perspectives. We show BLEND, a tool that merges different data sources to browse the evolution of PHARO [DSMML15].

Chapter 7 discusses the general model used by issue trackers and how it falls short in helping users when they have an issue to report. We present a survey we performed to investigate what users deem easy to provide in a bug report. We distill a meta-model for a minimal bug report, establishing a basic layer of core features. We propose an improved model to represent and store a bug report and the related data without losing information about its context. [DSML16].

Chapter 8 considers the problem of the engagement of developers during a tedious activity such as reporting and fixing bugs. We discuss how we can improve the user experience inside an issue tracking system by employing *gamification* [DSMLM17].

Chapter 9 concludes our work by summarizing the proposed approaches and how these can show the direction for the development of integrated issue tracking systems with smarter and deeper bug reports.

Appendix A presents a brief background on PHARO, the main platform that we targeted to develop and test our approaches and tools.

Appendix B presents a list of the publications produced during our work.

The screenshot shows the Bugzilla 'Enter Bug' form for 'LJL Test Product'. At the top, there is a navigation bar with links for Home, New, Browse, Search, Reports, My Requests, Preferences, and Help. Below this is a notice about a recent data disclosure and a link to 'Forgot Password'. The form includes a 'Show Advanced Fields' link and a note that asterisks denote required fields. The form fields are: Product (LJL Test Product), Reporter (h696478@trbvm.com), Component (a dropdown menu with 'Component 1' and 'Component 2'), Version (1.0), Severity (normal), Hardware (PC), and OS (Mac OS X 10.1). A green message states: 'We've made a guess at your operating system and platform. Please check them and make any corrections if necessary.' The form also has a Summary field, a large Description text area, an Attachment section with an 'Add an attachment' button, and a 'Submit Bug' button at the bottom.

Figure 2.2. An old Bugzilla submission form

Correctness is arguably an essential property for a computer program to work. However, it is nearly impossible to determine whether a non-trivial program is really correct: the huge amount of different variables and environments that a program usually gets exposed to makes predicting all the potentially harmful situations an unmanageable task. Even if we could determine the absolute correctness of a program in a given moment in time, this does not guarantee that it will keep behaving correctly in the future: There are a number of external factors that influence the execution of a program that can trigger problems that could not be observed before. Such external factors could for example include changes in the underlying technology, like the operating system, or different usage conditions like a change of the input format. For this reason —since we cannot get rid of bugs— dealing efficiently with defects is a crucial aspect in the success of a software system.

In this chapter we illustrate the state of the art in issue tracking, providing an overview on the current tools and practices and the approaches they propose, to identify the useful elements in issue tracking.

2.1 Issue Tracking Systems

In 1998, the Mozilla Foundation released the first version of BUGZILLA, which would soon become the reference issue tracking system. During the years different alternative tools emerged, providing their own set of customizations and personalizations. In this section we present four platforms, selected by importance and overall adoption, showing their salient features: BUGZILLA, JIRA, the GITHUB issue tracker and FOGBUGZ.

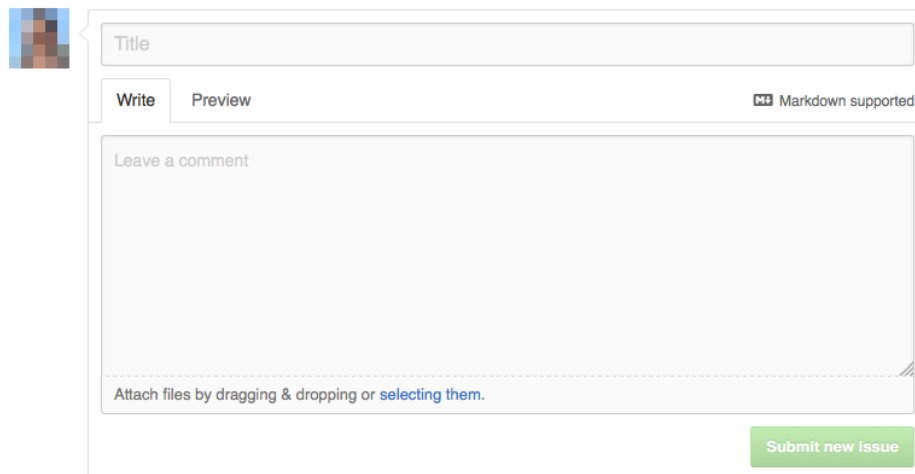
The image shows a screenshot of the GitHub bug report submission form. At the top left, there is a small profile picture icon. Below it is a text input field labeled 'Title'. Underneath the title field are two tabs: 'Write' (which is active) and 'Preview'. To the right of these tabs is a small icon and the text 'Markdown supported'. Below the tabs is a large text area with the placeholder text 'Leave a comment'. At the bottom of this text area, there is a dashed line and the text 'Attach files by dragging & dropping or selecting them.' In the bottom right corner of the form, there is a green button labeled 'Submit new issue'.

Figure 2.3. GitHub bug report submission form.

Bugzilla

BUGZILLA² is one of the oldest and most popular issue tracking systems, that inspired many existing issue trackers. Developed by the Mozilla Foundation, it is used by several open source projects, as well as industrial customers. BUGZILLA allows its users to obtain a great level of detail in specifying an issue at the price of a complex interface. Figure 2.2 shows the interface that BUGZILLA used to have in one of its previous versions.

Jira

JIRA³ by Atlassian is one of the most famous commercial issue trackers, used by Twitter, LinkedIn, and Ebay. It provides a polished interface and strong integration with the tools developed by the company. It uses a model similar to BUGZILLA.

GitHub

GITHUB⁴ is a popular Git repository hosting service, used to develop several popular open source projects, that offers a simple issue tracker. The submission form depicted in Figure 2.3 shows that GITHUB adopts a simplified model of a bug report, reducing but offers a strong integration with the versioned source code, by linking issues with specific commits.

FogBugz

FOGBUGZ⁵ is an issue tracker developed by FogCreek. It uses a bug model similar to the one of BUGZILLA, slightly more polished and user-friendly, due to its clean user interface and advanced filtering capabilities. It poses a strong accent on customization, by letting users define custom filters and views.

²<https://www.bugzilla.org>

³<https://www.atlassian.com/software/jira>

⁴<https://www.github.com>

⁵<https://www.fogcreek.com/fogbugz>

Together with these platforms, the open source and commercial scenes provide other popular solutions, like Redmine⁶ or Trac.⁷ It is however interesting to observe that, while these systems propose different degrees of integration with the tools in their ecosystem (e.g., the versioning system), the fundamental approach they adopt follows the same paradigm made popular by BUGZILLA: a textual description with additional customizable metadata. Further improvements to issue trackers, and the research around them, are built on top of this paradigm. In the following sections, we present the efforts of researchers to improve issue trackers and the model of a bug report in support of the bug fixing activity.

2.2 Visualizing Bug Reports

Many researchers showed how using data generated during the programming activity can provide valuable information about the evolution of a project. For example, Bacchelli et al. proposed an Eclipse plugin to integrate email communication in the IDE [BLH11]. They showed that having the email data produced during the development of a software system at one's disposal helps supporting program comprehension tasks, such as finding entry points in a system and recovering additional documentation. Another example has been given by Zimmermann *et al.*, who applied data mining techniques on version histories to detect changes and build prediction models to suggest future changes to developers [ZWDZ04].

2.2.1 Visualizations of an Issue Tracker

Just having the raw data, however, is often not enough: to turn it into actionable knowledge it is important to build tools to make use of this information. Reading a bug report is a difficult step in the debugging process: Browsing the large amount of information and deciding whether how much to rely on the data reported by the users, consumes a substantial amount of developers' time. To alleviate this burden, researchers devised a number of approaches based on the visualization of the data inside issue trackers. For example, D'Ambrosi *et al.* performed an analysis of the BUGZILLA bug repository: They summarized the diagram of the state transitions of a report and proposed a set of visualizations to support the analysis of a bug database at different levels of granularity. Their approach allows the user to navigate the history of a single issue tracker and inspect selected parts of the system with customized filters and a synthesized state transitions diagram of a report [DLP07]. They built visualizations to support the analysis of a bug database at different levels of granularity, depicting bug reports as independent entities. Their approach allows users to browse the history of an issue tracker and inspect parts of the system with custom filters. Knab *et al.* proposed visualizations to ease the understanding of the data in an issue tracker and find hidden patterns [KFGP09, KPG10]. Hora *et al.* proposed a visual exploration of the bug repository, creating interactive maps of the bugs in a system. To link bug reports to other software artifacts they argue the need for considering bugs as first class entities [HAD⁺12].

2.2.2 Visual Storytelling

Even with the means to efficiently inspect single bug reports, getting a big picture of the data contained in an issue tracker, and how it relates to the system, is often a completely different

⁶<https://redmine.org/>

⁷<http://trac.edgewall.org/>

task, as the data has to be interpreted at a different level. To effectively present and understand such an amount of data, many researchers and practitioners adopt a *Visual Storytelling* approach. The ability of contextualizing the information in a story that explains the meaning of the data is becoming more and more central to the skills required for data scientists [SH10].

Among the different visualizations that researchers used to represent a software system, the city metaphor has proven to be effective in giving a high level picture of a group of entities, allowing the user to navigate, zoom and inspect the various components and refine the view [WLR11]. This approach has been adopted in different scenarios, depicting different kinds of information pertaining to several steps of the development activity, such as changes in the system, the defects involving different components in the system, issues in quality checking rules or the exceptions in the system [PBG03].

Other visualization approaches tried to focus on the evolution of software systems, specifically the version repositories, the dependencies or the structures. For example, Fischer *et al.* [FG06] proposed EVOGRAPH, an approach based on data extracted from a system release history, that visualizes the evolution of structural dependencies through 2D visual representations. Girba *et al.* [GLD05] focused on the visualization of the evolution of class hierarchies, correlating the history of classes and their relationships, *e.g.*, inheritance. The approach by Voinea *et al.* [VT06] uses a combination of color and texture to represent as many attributes as possible to display information extracted from software configuration management systems. Another important approach is the one by Ratzinger *et al.* [RFG05], that represents systems as nested, zoomable graphs.

2.3 Bug Reports

Dealing with bug reports is a non-trivial task, that poses a number of communication problems among users and developers. Such a large, noisy, and sometimes redundant corpus of information, impacts the debugging time and the maintenance costs. To minimize this impact, researchers focused on improving several aspects of this process. In this section we present the efforts in automating the essential aspects of dealing with bug reports, such as its quality, its relevance, and predictions about the future behavior of the system.

2.3.1 Quality of a Bug Report

The reliability and completeness of bug reports is crucial to quickly solve a defect. Bissyande *et al.* showed that most reporters that contribute to a project are not developers [BLJ⁺13], posing a problem on the quality of the data. To understand how developers perceive the quality of a bug report, researchers conducted a survey, asking which elements help understanding a problem. They found that stack traces are the most useful item and often contribute to a faster resolution of a defect, suggesting that they should be collected and included in issue trackers [ZPB⁺10, BJS⁺07, SBP10]. Even when reliable, though, the amount of information in an issue tracker can hide the relevant information: To alleviate the information overload, Sun devised a technique to detect bug reports without useful information [Sun11]. Besides incorrect information, bug repositories often contain duplicate entries for the same defect. However, developers do not consider this harmful, but instead find the additional information useful to better understand the problem [BPZK08].

Managing a large bug repository is often a burden that adds a new layer of complexity on top of the bug fixing problem. To alleviate this burden, researchers proposed to use automated approaches [Wei06]. For example, Anvik *et al.* observed that large open source projects are often

overwhelmed by the rate of new bug reports and proposed a machine learning based approach to aid bug *triaging* decisions [AHM06], the process of selecting the right person to take care of an issue.

Guo *et al.* conducted a study to predict what impacts the resolution time of *MS Windows* bug reports [GZNM10], finding that a high number of reassignment of a report usually increases the issue lifetime, and that the reputation of the submitter also impacts the fixing time. Given the expensive nature of the bug fixing activity, a number of approaches exist to estimate the cost of a bug fix in person-hours [WPZZ07], predict bug fixing time [GPG10], locate features from bug reports [DRGP13], and perform traceability linking [BTW⁺13].

2.3.2 Bug Prediction

Solving defects does not represent the end of life of the information inside issue trackers: Yin *et al.* show the danger of hidden complexity behind a bug report, finding that 4.8% to 24.4% of sampled fixes for post-release bugs introduced new defects [YYZ⁺11]. They also noted that “Developers and reviewers for incorrect fixes usually do not have enough knowledge about the involved code”, and that “27% of the incorrect fixes are made by developers who have never touched the source code files associated with the fix”. Once a bug report gets closed, the data inside issue trackers can still contain valuable information, and has been exploited to predict the evolution of the code. D’Ambros *et al.* presented several approaches devised by researchers to predict future defects [DLR12]. For example, Zimmermann *et al.* proposed an approach based on network analysis on dependency graphs among components, to allow managers to identify central program units that are more likely to face defects [ZN08]. Kim *et al.* suggested that defects tend to show in places previously affected by other defects, proposing a caching method to prioritize the elements in the code to inspect [KZWJZ07]. It is the source code that contains the defects, but these defects are introduced through changes: As such, Hassan *et al.* proposed metrics for bug prediction that consider the changes in the code, rather than the code itself [Has09]. Despite the efforts in improving the accuracy of the bug prediction approaches, Bhattacharya and Neamtiu showed the low correlation of current prediction techniques and underlined the need to find additional features to increase the confidence of the time estimates [BN11].

2.3.3 Bug Reports and Social Interactions

One of the core aspects of an issue tracker is that it collects social interactions in a community: Users can give feedback to the developers and obtain information on the system. Breu *et al.* analyzed a sample of 600 bug reports, finding that interacting with developers helps solving an issue faster [BPSZ10]. Zhou and Mockus showed that users involved in the development activity, like bug reporting and participating in the community, are more likely to become stable, long-term contributors [ZM15]. Therefore, improving issue trackers to foster the relations between developers and users could result in faster resolution of defects.

2.4 Data Collection

As we already mentioned, bug fixing is well known to be a tedious activity, and identifying the source of a problem—even with a bug report—represents a non trivial task. Reproducing and understanding the error is usually cumbersome, as the developer does not have access to the original environment where the error occurred. As a consequence, she cannot fully rely on the information in the report, as it might contain incomplete, or even incorrect information.

2.4.1 Collecting Runtime Errors

To ease the process of resolving defects, researchers have devised a number of approaches to complement the information reported by the user with additional information, for example by automatically collecting data about the environment where the error occurred. Zimmermann *et al.* showed that the bug reports containing stack traces improve the general quality of the report, and result in a faster resolution of the report [ZPB⁺10]. Schröter *et al.* provided empirical evidence analyzing the Eclipse project that the use of stack traces in defect resolution provides value in the debugging activity, and suggested that software projects should provide means to include them in defect reporting [SBP10].

The idea of collecting runtime exceptions to analyze software errors has been adopted by different authors in different contexts. Glerum *et al.* used an automated approach to collect errors generated and submitted by WER, the *Windows Error Reporting* tool. They analyzed data collected from users of Microsoft's operating systems worldwide: In their approach they grouped the reports into buckets by looking for specific properties of the trace, and used this information to prioritize debugging and build a knowledge base where system administrators could check common problems of the system [GKG⁺09]. Inspired by this work, Han Shi *et al.* applied the same principle to performance debugging [HDG⁺12]: They proposed an approach called *STACKMINE*, designed to detect and report highly impacting performance bugs and address defects that cause long delays in the user experience. We believe that a similar approach to the one that they applied to an operating system, can be a valuable support for developers in building a programming environment. Mozilla adopts a similar approach to collect stack traces and runtime execution for debugging purposes [McL04].

The information of stack traces contained in bug reports represents a valuable support in debugging: as such, many researchers devised different methods to aid bug fixing and management of reports using stack traces. These works provided evidence that stack traces are a useful tool and a precious source of information [DR13, WKZ13, BMRC05, WPZZ07]: they provide precise information that are generally more reliable and useful than the descriptions produced by the submitter of the reports [KMC06]. Brodie *et al.* proposed an automated approach to group similar bug reports using stack trace [BMRC05]. Moreno *et al.* applied Text Retrieval techniques to compute similarity between bug reports using the stack traces contained in the report description, focusing on reducing the overhead to analyze large amounts of data [MTMS14]. Again, this was done in a localized post mortem way.

Managing bug reports is expensive and represents an open problem: Many studies proposed approaches to automatically manage them, by finding the right developer to fix the defect, predict the cost of fixing a bug and reduce maintenance costs [MKN09, AHM06, ŚZZ05, DLR10, LVZ10].

2.4.2 Errors as First-Class Citizens

Several tools in both academic and industrial contexts use the vast amount of data generated during the development and debugging process to enable a number of different analyses. However, any analysis of such kind sooner or later has to deal with the fact that the data collected is not in its original form: It is a representation of the original entities, serialized in a textual format. This, however, gives birth to a number of problems, as de-serializing is prone to interpretation and correctness errors, for example due to bad formatting. In this section we present an overview of the efforts to alleviate this class of issues.

2.4.3 Aiding Bug Fixing

The first major development activity that benefits from accessing clean runtime data is bug fixing. The purpose of the research in this area is to support and automate the identification of the portions of code that contain 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 [HDG⁺12, AADS⁺07]. The already mentioned survey performed by Zimmermann *et al.* finds that one of the biggest problems comes from the reliability of the reported data [ZPB⁺10], 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 [Aye11]. In the attempt of reducing the effort in crash debugging Soltani *et al.* devised *EvoCrash*, an approach to reproduce a reliable environment describing the context of a failure using genetic algorithms [SPvD17].

2.4.4 Aiding System Comprehension

Researchers 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, resulting in a large number of different studies and approaches [CZVD⁺09]. For example, Koike proposed a tool to visualize log files of the Snort⁸ intrusion detector and assist system administrators to identify intrusion attempts in a system [KO04]. 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 [MT07]. 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 [OJH03]. 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 [DPJM⁺02].

Finally, researchers also tried approaches to improve the textual representation of software artifacts by augmenting their description with a markup language [Bad00, MCM02].

2.5 Engaging Developers and Users

Despite all the tools that researchers and practitioners produced, fixing bugs remains a tedious activity. To support developers deal with an issue, there have been a few efforts in introducing *gamification* —the use of game elements in a non gaming context— into software engineering. Passos *et al.* [PMNC11] proposed to gamify the phases of software lifecycle by splitting the whole process into tasks, and setting achievements for their completion. While this is an interesting approach, it is essentially a *pointsification*, and as such puts too much emphasis on the rewards, thus being ineffective on the long run. Singer and Schneider performed a study on the gamification of commit messages [SS12]: they managed to influence the workflow of the students in the experiment, improving the workflow. They however received both positive and negative comments.

⁸<https://www.snort.org/>

Dubois and Tamburelli [DT13] pointed out that software projects often produce mediocre quality artifacts, do not respect the terms for milestones, or exceed the financial budget. They claimed that gamification could represent a solution to the issue, but only outlined a possible approach to gamification based on the three steps analysis, integration, and evaluation. Probably still being in the inception phase they did not provide concrete suggestions or a systematic set of recommendations.

2.6 Outline

In this chapter we have presented the efforts of practitioners and developers to support the activities of tracking and solving bugs. We believe that these efforts represent the desire of building smarter issue tracking system, that integrate with the IDE and interact with the existing development tools to reduce the time that developers spend dealing with boring and repetitive tasks. In the following, we present our contributions with respect to the state of the art.

Visualizations and analyses

The approaches that we saw focus on retrospective analyses. We believe that while conceptually interesting, there is little practical utility in daily development, since after all the goal of an issue tracking system is not to look at defects, but to actually fix them. This implies that even the most elaborated techniques are of limited actionability, since the bug fixing process takes place in a different space, namely the integrated development environment (IDE). We believe that the use for a visualization is not to simply display the data, but to establish a first-class link to the development environment.

In Chapter 3 we present IN*BUG, our tool for browsing issue trackers in a visual fashion, highlighting properties that are normally hidden when using plain text.

Collecting Runtime Data

We saw different approaches for program comprehension, focused on describing source code, that are still relevant to support bug fixing, as they explicitly render the properties that are hidden in the textual form of the source code.

In Chapter 4 we present our approach to data collection. In this context, we think that associating new stack traces to existing bug reports could provide immediate feedback to the users of the system and to assist development and bug fixing in a live fashion. In Chapter 5 we then propose a tool for collecting domain-specific data about failures in a system, to enable a reliable and conversational data source.

Visual Storytelling

Researchers presented several approaches to effectively visualize data about a single aspect that impacts or involves a system. However, these approaches fall short in correlating this information with knowledge coming from diverse data sources and impacting diverse concerns. Such additional information could effectively integrate the existing data to uncover further relations between the elements of the system. We think that an approach that considers more than one kind of data and presents the information in a unified, uniform view, normalizing and balancing each source, could provide a greater value in understanding a software project and the activities happening in its ecosystem.

In Chapter 6 we present a case study of visualizing and combining multiple data sources about a software system. From this visualization we are able to learn interesting stories about the evolution of the system.

Modes for Bug Reports

We saw that different bug tracking systems propose different strategies to describe a bug report. In Chapter 7 we investigate what developers believe to be *difficult* to provide when writing a bug report, to deepen our understanding of the problem of data reliability inside issue tracking systems. We then examine the data contained inside big issue tracking systems and try to define the *minimal* model to describe a software error.

Gamification

The engagement of users is a problem that is bringing more and more people interested in the use of gamification applied to software engineering. In Chapter 8 we present a framework to support developers during the design of a gamification engine applied to their systems and communities, highlighting the perils that an inconsiderate use of points and badges can bring.

3

Visual Analytics of Bug Repositories

Bug tracking systems are used to track and store the defects reported during the life of software projects. The underlying repositories represent a valuable source of information used for example for defect prediction and program comprehension [SFM99]. However, as we mentioned in Chapter 1, bug tracking systems store and present the actual bugs essentially in textual form, which is not only cumbersome to navigate, but also flattens the information, hindering the understanding of the intricate and multifaceted pieces of information that revolve around software bugs. We begin our journey towards an improved representation for a bug report by analyzing existing bug repositories, to investigate where the information that they contain can actually be leveraged to improve the knowledge that we have on the system.

In this chapter we present IN*BUG, a web-based visual analytics platform to navigate and inspect bug repositories. IN*BUG provides several interactive views to understand detailed information about the bugs and the people that report them.

Structure of the Chapter

Section 3.1 introduces the issues of dealing with data in issue trackers. Section 3.2 presents in detail the views that compose IN*BUG. Section 3.4 concludes the chapter summarizing what we learned visualizing issue tracking systems.

3.1 Accessing Bug Repositories

Due to the complexity and size of non-trivial software projects, the development of a system is always accompanied by software defects, or bugs. To manage these defects, modern software projects use bug tracking systems (also known as bug trackers or issue trackers), such as Jira or Bugzilla. With bug trackers, end users and developers can report bugs they encountered while using the system, usually by means of custom web interfaces, where one can enter details about a specific bug, creating a so-called *bug report*. A typical bug report, such as the one depicted in Figure 3.1, contains information about (1) the title and id of the bug, (2) the user who reported the bug and the people involved in its history, (3) its current status, (4) its opening and closing date, (5) its last modification date, (6) the project to which the bug report pertains, (7) events (such as changes of the people assigned to the bug report, etc.) during the life cycle of the bug, etc. The example bug report depicted in Figure 3.1 is from a specific bug tracker, FogBugz¹, but it does not differ significantly from the reports recorded with other bug trackers.

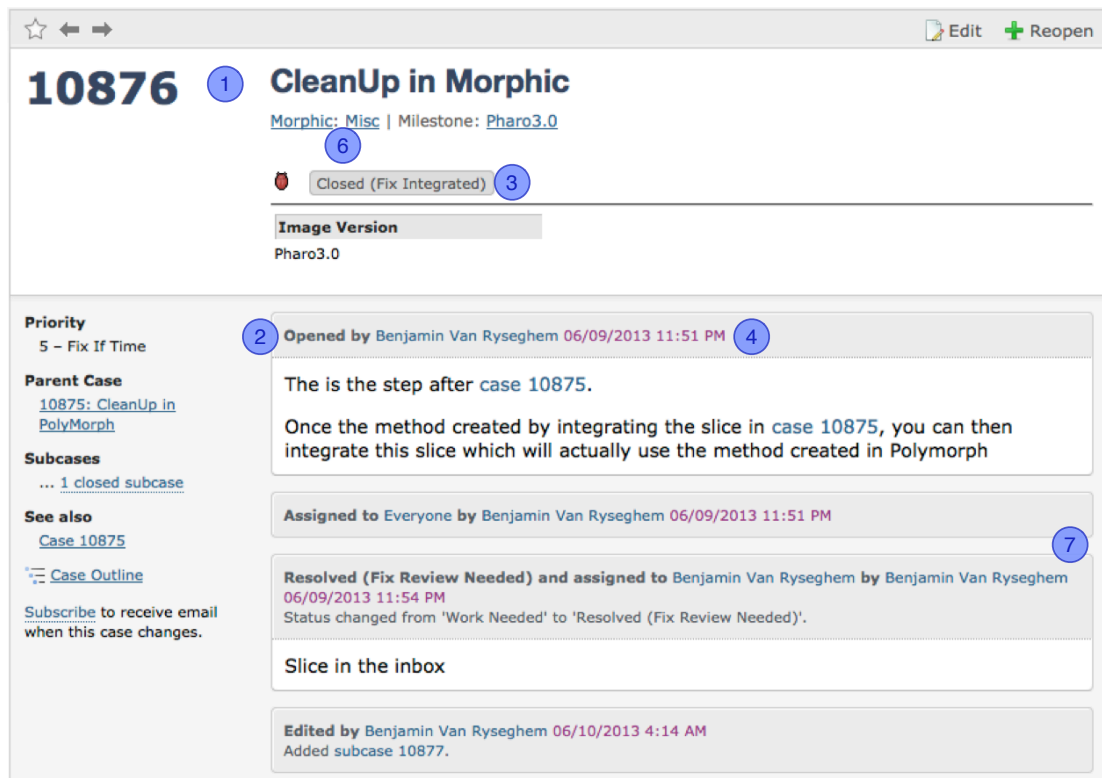


Figure 3.1. Example bug report in the FogBugz bug tracking system.

Various researchers have mined and used the information stored by bug trackers to perform several types of analyses, such as identifying duplicate bug reports [WZX⁺08], measuring the quality of a report [ZPB⁺10], predicting future defects [DLR12], performing traceability linking [BTW⁺13], locating features [DRGP13], ameliorating bug triaging decisions [AHM06], etc. The actual goal however is to ease the life of developers in the handling of bug reports, as part of the development process.

One problem is that bug reports are disconnected from the software system they pertain to, and it is up to the developers to restore the link between a bug report and the relevant components

¹<http://www.fogcreek.com/fogbugz/>

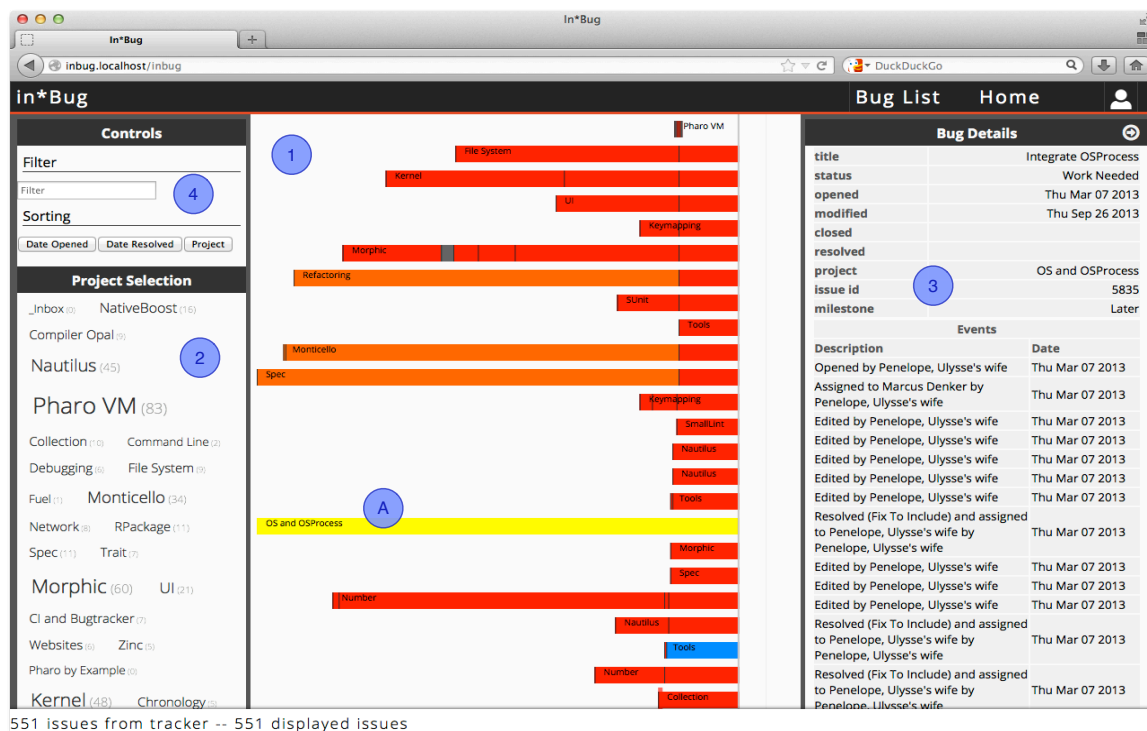


Figure 3.2. Main user interface of In*Bug

of a system. Another problem is that bug reports, such as the one depicted in Figure 3.1, are displayed on individual web pages that list their properties, making them cumbersome to handle and making it also difficult to obtain a “big picture” of the existing open bug reports and how they affect the system they pertain to. Moreover, this information is stored and presented as text, which makes it hard to understand the properties of a bug report.

We present IN*BUG, a web-based bug analytics platform, that eases the inspection, navigation, and comprehension of bug repositories, mostly by means of interactive visualizations. IN*BUG provides an entry-level big picture overview to browse the content of a repository, and a detailed, complementary, interactive, and finer-grained view to understand detailed information about the bugs and the people that report them.

Other researchers have produced custom visualization of bugs such as D’Ambros *et al.*, who proposed visualizations that tried to depict the complex information revolving around bugs, which are de facto independent entities when it comes to program comprehension, and not mere side effects of the evolutionary process that software systems are subjected to [DL07, DLP07]. While D’Ambros *et al.* only created standalone and static depictions of information taken from BugZilla, our goal with IN*BUG is to depict live data from a bug tracker, namely FogBugz, as it is the issue tracking system used by the PHARO community. The goal of IN*BUG is to offer a complementary view to inspect and analyze information pertaining to bugs reported in the context of the many projects that make up a software ecosystem. We built IN*BUG around the issue tracking system of the PHARO open-source community.

We now present the features of IN*BUG, discuss its current implementation, and illustrate its usage.

3.2 In*Bug in Detail









In this section we present the interface of IN*BUG, detailing its panels and how we used the data in the issue tracking system to build the visualizations.

3.2.1 Main view

Figure 3.2 depicts the main user interface, composed of the following panels:

Bug lifetime panel (1). This view depicts the bug reports contained in the bug repository, showing their duration (as a horizontal stacked bar chart) and status (using different colors, listed in Table 3.1).

Table 3.1. Bug report event color codes

Active		orange	Work Needed		red
Closed		gray	Resolved		dark gray
Working On		blue	On Hold		cyan
Unknown		light grey	Selected		yellow

In Figure 3.2 one specific bug (marked as A) is under focus. The vertical line to the right indicates the current date, making it also clear whether a bug report is still active or not (if it is, it will touch that line). This view also helps the developer to evaluate the complexity of a bug report by summarizing the events that occurred during its lifetime.

Project selection panel (2)

In this panel the user can pick the projects whose bugs she is interested in. All projects are shown as a tag cloud, where the tag size indicates the number of bugs reported for the project, also indicated with numbers between parentheses close to the name of the projects.

Details panel (3)

This panel provides all the information reported about the bug report under focus in the bug lifetime panel: It presents both the metadata and the list of events that happened during the lifetime of a bug, including description and date of each event. The metadata is presented as extracted from the bug repository, e.g., the opening date, the status, the last modification date, etc.

Filter and options panel (4)

This panel allows the user to sort and filter bugs. The three default sorting criteria order the issues by project, opening date, or date in which the bug has been resolved. The *filter* field offers the possibility to enter either regular expressions or pieces of *Smalltalk* code as queries, allowing the users to submit custom made queries to filter bugs.

3.2.2 Details of a bug

This view (see Figure 3.3) presents a detailed representation of a specific bug report. Each section provides a description of the elements that compose a bug report.

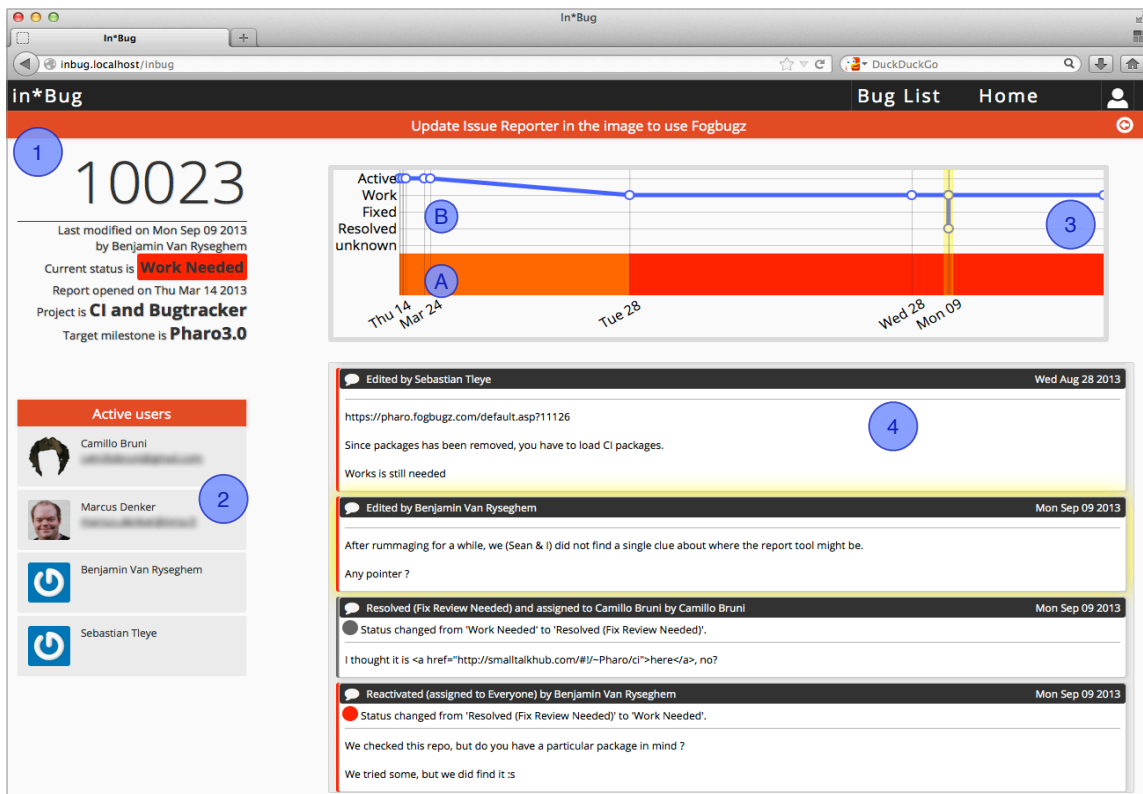


Figure 3.3. In*Bug details page showing the properties of a bug report

Bug Report Metadata (1)

The first panel summarizes the important metadata of the bug report: the id, the last modification, the current status, the opening date and possible closing date, the project and the target milestone for the issue resolution.

Users List (2)

This panel gives an overview of the people involved in the evolution of the bug. In particular, the list displays the information of each user that performed an action on the issue, that was stored as an event. The details include the picture of the user, the user name and the user's email address², to contact the people working on an issue.

Bug Report Life Visualization (3)

This panel shows a visualization of the life of a bug report during time. The left border represents the date the issue was opened, the right border represents the moment the bug was closed, or the current date if the bug report is still active. The (A) section proposes the same visualization of the list view in the main view (3.2.1), emphasizing the status changes during time. The (B) section shows a line diagram where the height represents the criticality of the status (*i.e.*, fixed is the lowest and active is the highest) and highlighting each event with a circle.

²We obfuscated the email addresses in the figure for privacy reasons.



Figure 3.4. List of events in a bug report

Event Interactive View (4)

This is a list of all the events that compose a bug report. It shows the metadata of the event and whether it is an automatic event or an event generated by a user. It also detects and highlights the patches of code submitted to the tracker for the issue resolution, and provides a link to download and inspect the patch. The user can click on an event to highlight it both in the events list and in the bug report lifetime visualization. Figure 3.4 shows an example of the event list, where we can observe the three types of events: (A) shows a comment by a user; (B) shows a submitted patch. The upper left icon offers a link to the repository page of the patch; (C) indicates events automatically generated from bots in the tracker. Inspired by more semantically rich and elaborated views, such as the storylines by Ogawa *et al.* [OM10] or Kuhn and Stocker's storytelling timelines [KS12], the left border of each event is colored according to the status of the event, to help the user to keep track of the evolution of the bug while inspecting the list of events.

3.2.3 Implementation & Current Dataset

IN*BUG is a web application built on top of the PHARO environment. It uses the SEASIDE web framework [DRSZ10] to provide the data stored in a *MongoDB* database and implements a *RESTful API* to communicate with the client. The client interface is implemented in *JavaScript* using the

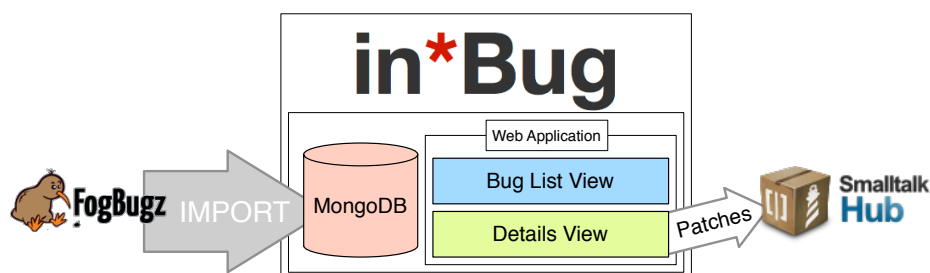


Figure 3.5. The interactions of In*Bug with the *FogBugz* and *SmalltalkHub* services

data manipulation and visualization library *D3.js*.³

IN*BUG is built to visualize the FOGBUGZ repository of the Pharo ecosystem. In Table 3.2 we provide a summary of the data we investigated.

Table 3.2. Summary data of the *Pharo* bug tracker

Number of projects	46
Number of bug reports	8,666
Number of open bug reports	613
Total number of events	79,437
Average events per issue	9

IN*BUG also provides links to patches on *SmalltalkHub*, a source code repository to store versioned Smalltalk code. In Figure 3.5 we can see how these three services interact.

The bug reports data is imported from FogBugz and stored in the MongoDB repository. The web application then loads the data and presents it in the list view of the main interface. The details of a single report are presented in the details view, where the user can follow a link that leads to a patch submitted to SmalltalkHub.

3.3 More Than Meets the Eye

While developing IN*BUG we verified that providing a visual feedback using data that is usually consumed through plain text is an effective method to observe and investigate properties that would normally be hard to isolate. For example, in Figure 3.2 we can easily separate the bug reports that are older from the newer ones, observe the amount of activity happening on an issue, spot cases where a bug report was reopened several times, or identify abandoned reports. The experience of serving textual data using a visual representation convinced us that a textual representation we employ in issue tracking systems is limiting the amount of information that a user can absorb.

³<http://d3js.org/>

3.4 Outline

In this chapter we presented IN*BUG, a web-based visual analytics platform to explore the content of a bug repository. IN*BUG allows to get a complete overview of a whole repository, as well as detailed and meaningful information on a single bug report, either through visualizations that allow to interact with the data, or with the query engine embedded in IN*BUG that allows the user to submit queries and dialog directly with the bug reports.

Since we designed IN*BUG as a tool for practical development, we focused on the Pharo platform and we targeted its community. However, the approach of IN*BUG is general and therefore it can be applied to any bug tracking system.

We saw that existing bug repositories contain information that can be highlighted using means other than plain text: This is a first important step in promoting bug reports as independent entities. The underlying information we display, however, is still in textual form, with the consequence that we have to rely on the expertise of the user to obtain useful information. We therefore decided to tackle the problem of the reliability of the data by means of automatic data collection during a software failure. In the next chapter we present our approach to crowd-driven data collection: We collect and investigate a large amount of stack traces generated during development activities, to provide automated and reliable feedback to users and developers.

4

CrowdStacking Traces to Aid Problem Detection

During software development, exceptions are by no means exceptional: Programmers repeatedly try and test their code to ensure that it works as expected. While doing so, runtime exceptions are raised, pointing out various issues, such as inappropriate usage of an API, convoluted code, as well as defects. Such failures result in *stack traces*, lists composed of the sequence of method invocations that led to the interruption of the program. Stack traces are useful to debug source code, and if shared also enhance the quality of bug reports. However, they are handled manually and individually, stored in a bug report as a copy-and-paste chunk of text that complicates the automatic processing of the information and raises questions about their reliability.

In the previous chapter we saw how treating bug reports as flat entities can hide properties that would be useful in the development process. In this chapter we argue that stack traces can be leveraged automatically and collectively to enable what we call *crowdstacking*, the automated collection of stack traces on the scale of a whole development community. We present our crowdstacking approach, supported by SHORELINE REPORTER, a tool which seamlessly collects stack traces during program development and execution and stores them on a central repository. We illustrate how thousands of stack traces stemming from the IDEs of several developers can be leveraged to identify common *hot spots* in the code that are involved in failures, using this knowledge to retrieve relevant and related bug reports and to provide an effective, instant context of the problem to the developer.

Structure of the Chapter

Section 4.1 outlines the elements involved in collecting runtime information about software errors. Section 4.2 illustrates the nature of stack traces and describes the data generated during the development process. Section 4.3 introduces SHORELINE REPORTER and the methodology we used to collect stack traces, as well as the process to link them to relevant bug reports. Section 4.4 evaluates our approach. Section 4.5 discusses our results and presents the possible extensions to our approach. Section 4.6 concludes the chapter.

4.1 Collecting Runtime Errors

Software development is an iterative refinement process: developers write code and then test it to increase their confidence that the program behaves as desired. This continuous process of running small, localized tests generates many errors that developers exploit to locate and correct the defects in the code. Some paradigms, like *Test Driven Development* [Bec02], adopt an inverse point of view: Developers define tests first, and then write the code that complies with the tests until they all pass. As a consequence, this process results in an even larger number of runtime exceptions, each of which may contain useful information about the context of failures.

The knowledge enclosed in such exceptions potentially provides useful insights that can be exploited to better understand the underlying system, its functioning, and its quality. For example, the number and the nature of errors related to the incorrect use of an API is correlated with the difficulty to approach it for a beginner, can suggest a lack of documentation, or a bad architectural design. Information from exceptions can also be exploited to get a deeper understanding of the runtime behavior of a complex fragment of code, and it is crucial to identify possible defects hidden in the program.

Programming environments generally deal with exceptions by means of *stack traces*, a textual description that depicts the execution of the steps that led to the error. In object-oriented programming languages this is the sequence of method invocations that led to the exception.

The information in a stack trace is useful to understand where the failure originated and which entities of the system are involved. Research has shown that it is also valuable to determine the cause of a defect: Including a stack trace in a bug report increases its quality by providing reliable and relevant information. Indeed, researchers showed that bug reports containing stack traces are closed sooner than bug reports containing only a generic description of the error [ZPB⁺10, SBP10]. However, a stack trace is generally checked manually by a developer to spot and fix single defects and its usefulness terminates once the bug gets resolved. As a result, a considerable amount of information is discarded and the knowledge it contains is lost. Researchers already used automated approaches to collect generated stack traces and identify bugs and performance issues [GKG⁺09, HDG⁺12]. However, these approaches remain *post mortem*, and largely focus on the properties of a running system by recording the behavior of users of operating systems.

We believe that the knowledge contained in stack traces should not be limited to the mere fixing of a single case, and that its use can be extended and in a *live* fashion by automatically and collectively gathering this information, using it to provide instant feedback to the developer. By establishing such a tight cycle between a failure and the feedback, we want to enable what we call *crowdstacking*, a collective process that involves a whole development community in gathering information automatically collected from stack traces, to boost the debugging process.

We present SHORELINE REPORTER,¹ a tool implementing crowdstacking by seamlessly and silently collecting stack traces from development sessions, and storing them on a shared, central repository. We used the collected data to perform various analyses, such as identifying the entities that are more prone to be involved in a failure, and searching and retrieving relevant knowledge already present in the community ecosystem. This additional information can be used to prompt a developer during the development process, for example by recommending a set of bug reports contained in the bug tracker that are related to the current exception, thus providing a more complete picture of the context of the error.

¹<http://shoreline.inf.usi.ch>

4.2 On the Nature of Stack Traces

Exceptions are a common mechanism in modern programming languages to represent errors and signal unexpected behavior in general; they are the standard error management technique in any modern object-oriented programming language. When they are left unmanaged, and thus they remain uncaught, exceptions ultimately result in the interruption of the executed program. Normally, an error message gets printed together with a stack trace, which represents the status of the dynamic call stack when the uncaught exception was thrown. Essentially, it represents a summary of the path that the program followed through the code, showing the entities that were involved before the failure. We collected large volumes of data from development sessions of users of PHARO. In Section 4.3 we detail the approach we used to collect the stack trace data in the PHARO system.

Anatomy of a Stack Trace

In PHARO, a stack trace is a list of pairs *Class>>selector*, where *Class* is the name of the class containing the method, and *selector* is the name of the method invoked. Figure 4.1 shows a concrete example of a stack trace that we collected with our tool.

```

PluggableButtonMorph(Morph)>>handleKeyDown:
KeyboardEvent>>sentTo:
PluggableButtonMorph(Morph)>>handleEvent:
PluggableButtonMorph(Morph)>>handleFocusEvent:
[
ActiveHand := self.
ActiveEvent := anEvent.
result := focusHolder handleFocusEvent: (anEvent
transformedBy: (focusHolder transformedFrom: self)) ] in
HandMorph>>sendFocusEvent:to:clear:
BlockClosure>>on:do:
WorldMorph(PasteUpMorph)>>becomeActiveDuring:
HandMorph>>sendFocusEvent:to:clear:
HandMorph>>sendEvent:focus:clear:
HandMorph>>sendKeyboardEvent:
HandMorph>>handleEvent:
HandMorph>>processEvents
[:h |
ActiveHand := h.
h processEvents.
ActiveHand := nil ] in WorldState>>doOneCycleNowFor:
Array(SequenceableCollection)>>do:
WorldState>>handsDo:
WorldState>>doOneCycleNowFor:
WorldState>>doOneCycleFor:
WorldMorph>>doOneCycle
[
World doOneCycle.
Processor yield.
false ] in MorphicUIManager>>spawnNewProcess
[
self value.
Processor terminateActive ] in BlockClosure>>newProcess

```

Figure 4.1. Example of a stack trace collected from a runtime exception. The most recent call is at the top, the oldest call is at the bottom. The snippets of code inside blocks are highlighted in blue.

As we can see from the listing, the stack trace occasionally contains small snippets of code included in blocks (between square brackets, highlighted in blue). This happens when a method executes a block. Since in Smalltalk a block is equivalent to a closure, it represents a pluggable behavior that can change the flow of the program and, as such, it is reported into the stack trace.

Some class names are complemented with the name of a superclass between parenthesis. This happens when the called method is not defined in the class itself, but it has been inherited from the specified superclass. This notation maintains the link between the class involved in the exception and its superclass: It is important to keep track of this information, since the cause of an error can be rooted in the superclass chain and suggest a possible defect in the original method, as well as being a consequence of the interaction with the code of the subclass.

Stack traces and dynamic typing

An interesting property of PHARO comes from its dynamic nature: the whole system is polymorphic, and polymorphism is obtained through the so-called *duck typing* [CRJ12]: every object can be used in place of other objects, as long as it is able to respond to the same messages. This entails that—as in other dynamic programming languages—there is no static type system and, as such, no static type checking: every type error happens at runtime, resulting in a *Message Not Understood* kind of exception. This peculiarity is important when considering the nature of exceptions in PHARO, because the vast majority of the exceptions is caused in this context: In our dataset an exception is thrown as a result of a message not understood in more than 72% of the cases. Among those cases, 68% are generated from a message sent to *UndefinedObject*. These are the equivalent of a *NullPointerException* in Java.

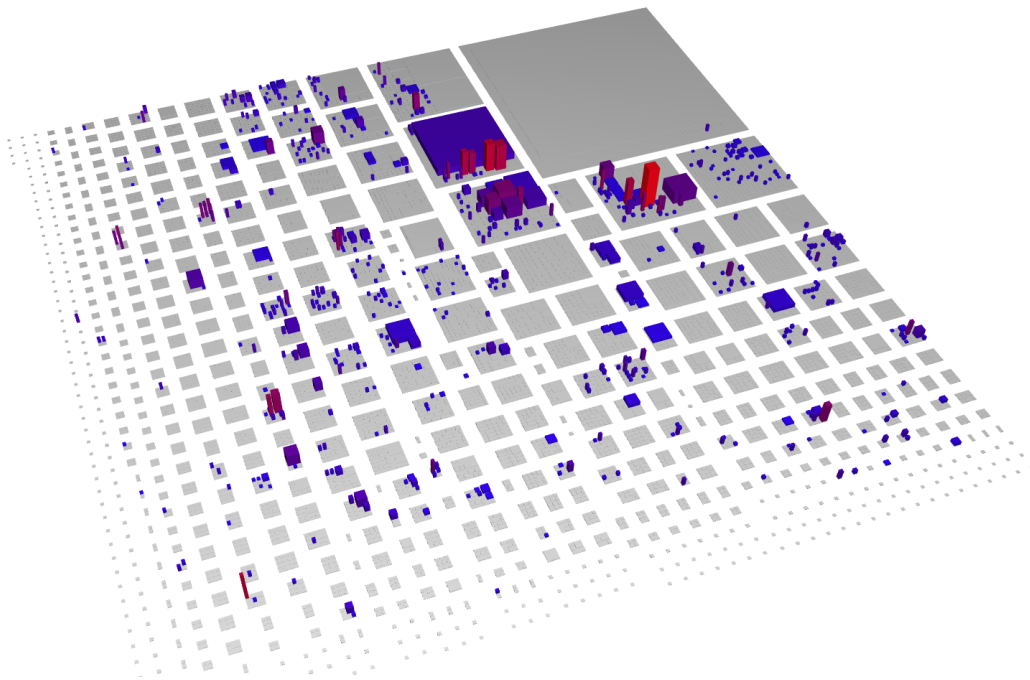


Figure 4.2. Distribution of the stack traces on the Pharo system using a city like visualization, where each building is a class. Pharo is composed of 14,045 classes distributed among 557 packages. We highlight the system with data from 7,532 stack traces that we collected. The height and the color of each building is determined by the number of traces the class appears in, while classes that are not involved in an exception are collapsed and depicted in gray. The gray squares enclosing the buildings represent the package containing the classes.

4.2.1 Interpreting Stacktraces

The amount of information represented by type errors is ambivalent: On the one hand, it may be an effect of trivial errors from the user, such as typos, and may represent noise among the useful exceptions. On the other hand, it contains a large amount of usage knowledge: being able to discriminate the actual failures from the occasional use errors can aid the early identification of defects and speed up debugging. Also, what at first may look as a “false positive”, can still be of great help in understanding how users and developers operate the system. A recurrent and consistent misuse of a method of an API may represent a flaw in the design of the public interface of a library. The maintainer of the library can then determine how to refactor the interface to improve the documentation. Moreover, using further data collected after the changes, she would also be able to measure the impact of her intervention on the workflow of the developers.

Another usage example can employ data showing a frequent pattern inside core classes of the system to identify the nodes in the system that manage the largest part of the computation. By identifying these spots, a developer could be able to prioritize her development activity and to perform targeted optimization.

So far we have considered the *horizontal* dimension of stack traces, that is, we considered the information of a group of traces based on their occurrence. However, an interesting property that we should also consider is represented by the *vertical* dimension of a stack trace. Since the order of the elements inside the trace is determined by the call sequence, the depth of a class can give us a hint of the role of the class in the computation: classes near the top of the trace provide an overview of the context where the exception originated, and can therefore be used to provide immediate feedback on the nature of the error and help debugging. Instead, classes towards the mid part and the bottom of the trace are more related to the mechanics of the system and could be usefully aggregated to identify anomalies in the core parts of the system.

Figure 4.2 shows the impact that runtime exceptions have on the PHARO system, adopting a city visualization [WLR11]. We aggregated the stack traces in one set of stack calls and counted the number of times a class would appear in a stack trace. Each class is depicted as a building, where the height and the color represent how often the class is involved in an exception: the more the class appears in a stack trace, the more the color tends to red and the higher the building. Classes that are not touched by any stack trace are depicted in gray and collapsed. From this figure we can see how the number of classes involved in exceptions is much lower than the total number of classes in the system, therefore suggesting some hot spots in the system that could be investigated for further development activities.

4.2.2 A Practical Use Case

To be able to deal with a potentially large volume of information, we need an effective approach to classify the stack traces. In Section 4.3 we present an approach based on clustering stack traces by similarity, and then stratifying horizontally the clusters using the number of members in each cluster to represent the frequency of the similar exceptions.

The most immediate advantage of using clustered stack traces is to leverage them for bug fixing. We developed an approach to analyze the contents of a stack trace and use the mined information to retrieve bug reports from the PHARO bug tracker that discuss the classes and methods in the trace. For example, by retrieving the reports related to the trace in Figure 4.1, which involves key events, we can find the bug report #12973, that discusses an issue related to keyboard shortcuts. By further reading the report, we can learn about the nature of the issue, and by checking the last events we can learn about the current status of the defect. In this case, we can see that the issue has already been resolved, a patch has been committed and is waiting to be

integrated. Thus, we can ignore the problem, knowing that it will be solved soon. Moreover, by checking future stack traces, we are able to determine if the problem has been completely solved or if it may appear again in some particular, missed corner cases.

Overall, by being able to access bug reports related to an exception while she is experiencing it, a developer can get an overview of a problem sooner and cut the overhead time spent searching for relevant information, thus supporting a more efficient debugging process. In the next section we show how to match bug reports and stack traces.

4.3 CrowdStacking Traces

Stack traces are a frequent and recurrent side product of the daily workflow of developers. Such data represent a significant amount of information that is usually not collected and thus lost. To benefit from this data we built SHORELINE REPORTER, a tool to intercept exceptions, the corresponding generated stack traces, collect the resulting data and submit it to a central server.

4.3.1 Data Collection

SHORELINE REPORTER is a plugin designed and built to integrate seamlessly into the PHARO development environment. We wanted to collect unbiased and uniform data, so we paid particular attention in building a tool that could be unobtrusive and that required minimal interaction with the user. For this purpose, SHORELINE REPORTER is highly configurable through a dedicated settings menu, and can work in two different main modes: an *interactive mode*, and a *shadow mode*.

Interactive Mode

It is designed to allow the developer to keep full control of her data and decide which are the traces to submit and which ones to discard. Figure 4.3 shows the main elements of the interactive user interface.

SHORELINE REPORTER activates when the user runs code that triggers an exception (A). The PHARO IDE generally pops up a *pre-debug window* (B), that illustrates a preview of the exception and the options that she can undertake. Here SHORELINE REPORTER shows up, proposing a *Report* button that allows the user to send the trace to the ShoreLine server. If she chooses to do so, she is presented with a window (C) that allows her to review the data that is being submitted to verify that it does not leak undesired information. Once the user presses the *Send* button, the reporter serializes the stack trace and submits it.

Shadow Mode

By acting on the system configuration, the user can reduce the level of interaction with the tool at the point of making it become completely transparent: She can decide to submit every stack trace without confirmation and also disable the intermediate check for the data she is sending. In short, SHORELINE REPORTER can become completely silent and gather all the stack traces from each exception. This is particularly important to avoid continuous prompts to the user asking for a confirmation and allow SHORELINE REPORTER to gather a significant number of stack traces without breaking the workflow of the developer.

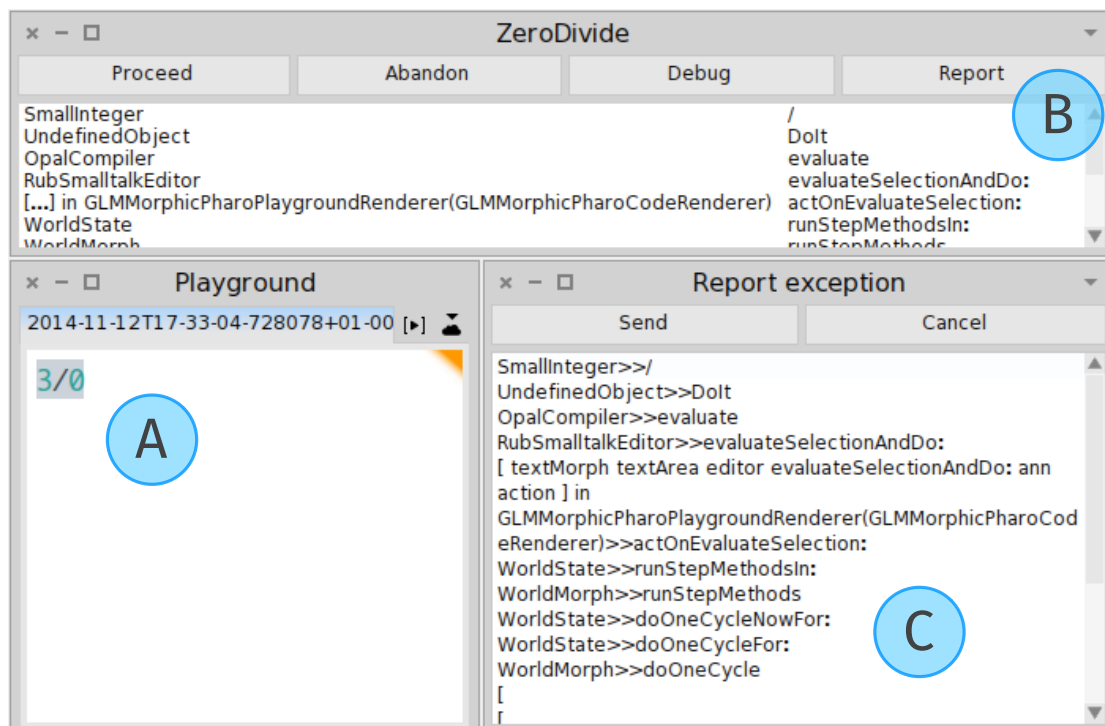


Figure 4.3. The interactive interface of ShoreLine Reporter

4.3.2 Data Representation

In PHARO, everything is modeled as an object. As such, a stack trace is a complex object containing a reference to the debugger, the full context of the exception and the sequence of method calls that constitutes the trace. However, to value privacy and to avoid our tool from being intrusive, we decided to serialize the whole stack trace as a list of strings, each one containing just the signature of methods, formatted as *ClassName>>methodSelector*. Thus, we discard all the elements that contain private data, such as the contents of instance variables. Encoding a stack trace using strings also guarantees compatibility and portability of the collected data, even when imported from different versions of PHARO.

Besides collecting the stack traces, we also added to the report additional metadata to allow a better categorization of the error. We collect the name of the author, which is the tag she uses to sign her commits, the date of the exception and the version of the PHARO build for which the exception happened. The version of the build can be useful to analyze the evolution of the system while it is developed. The PHARO development cycle is structured in two main branches: a stable version and a development version. In the PHARO community, the development version is actively developed and constantly improved by a large number of users and developers, and exception data from developed software can provide insights about the evolution of the system over time, as well as help spotting defects as soon as they arise, ultimately reducing the time required to fix a new defect after it is introduced.

4.3.3 Analysis on the Collected Data

We collected stack traces during a time period of five months, from June to November 2014. Table 4.1 shows a summary of the data we collected during that time span.

Table 4.1. Summary of the stack traces collected from June to November 2014

# of stack traces	7,532
# of lines in all stack traces	252,668
# of developers	8
average lines of a stack trace	34
size of the shortest stack trace	1
size of the longest stack trace	314

We visualized the data to highlight the parts of the PHARO system that were involved in the collected exceptions: Figure 4.4 shows a city visualization of the stack trace data mapped on the whole PHARO system. Using the same convention used in Figure 4.2, each building represents a class and each square enclosing a building is a package. Each building is composed of blocks, each one representing a method. The color of each method is determined by the number of times a method appears in a stack trace: it tends to red when the number is higher and to blue when the number is lower. Methods, classes and packages that do not appear in a stack trace are collapsed and depicted in gray.

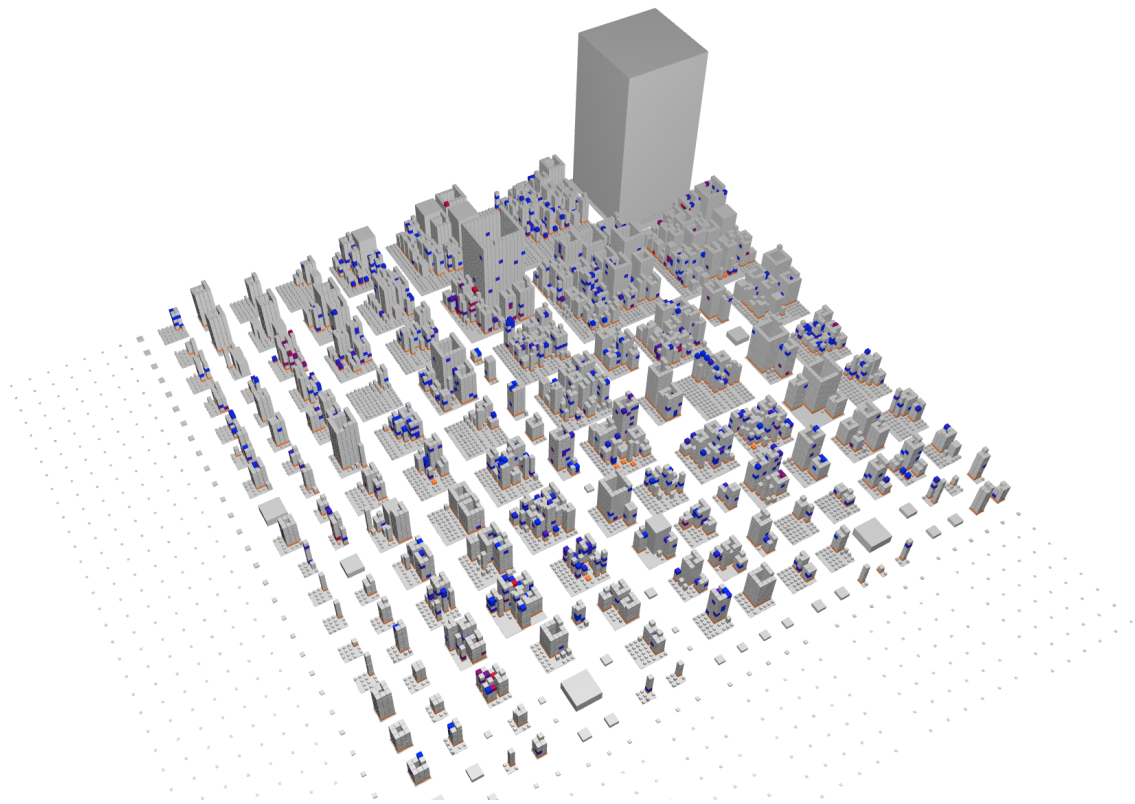


Figure 4.4. Distribution of the stack traces on the methods of Pharo using a city like visualization, where each building is a class composed by blocks representing methods. All the classes contain 112,558 methods, the color of each building is determined by the number of traces the class appears in, while the packages, classes and methods that are not involved in an exception are collapsed and depicted in gray.

The figure suggests that only a small part of the system is actually involved in the collected exceptions, and the vast majority of methods and classes is not impacted by them. By knowing these methods that work as entry points to the classes, a developer can view the impact that

Table 4.2. The 10 most called methods in the collected stack traces.

Class>>Method	Occurrences
BlockClosure>>on:do:	9,265
UndefinedObject>>doesNotUnderstand:	8,549
BlockClosure>>cull:cull:	6,268
Message>>sentTo:	4,980
PragmaMenuBuilder>>collectRegistrations	4,776
WorldState>>doOneCycleNowFor:	4,714
BlockClosure>>on:fork:	4,554
Array>>do:	4,497
BlockClosure>>ensure:	4,495
BlockClosure>>cull:	3,642

each class and method have in case of failures: This information can be useful in having a first prioritization to decide which methods she has to inspect first to search for a bug.

Table 4.2 shows a summary of the most active methods in all the stack traces. As expected, the most recurrent exceptions are related to some core elements of the language: *BlockClosure* is a core element used when passing code as argument, while *UndefinedObject>>doesNotUnderstand:* and *Message>>sentTo:* are part of the message sending infrastructure that is the foundation of Smalltalk. Despite being expected, the fact that the most common exceptions involve the dynamic nature of the language shows how the freedom provided by the absence of static type checking comes with the price of incurring in runtime exceptions even for experienced programmers.

Once verified that the most recurring exceptions are caused by common usage patterns of the language, we can consider these elements as outliers for the specific purpose of this work: The information they carry can still be useful in identifying other issues like API usage problems, but it is likely negligible to be connected to bug reports. Moreover, methods that appear in very few stack traces are also outliers, since there is an intrinsic lack of confidence that they can be significant to represent any pattern in the system.

4.3.4 Extracting Information

We saw that many stack traces are channelled mainly through few crucial points in the system. To inspect whether it was possible to group them, we applied a clustering approach to detect the stack traces that could be generated by similar errors. Clustering stack traces can give us the advantage of reducing the number of elements that we have to inspect to determine whether a given error is caused by a defect, by bad usage or simply by a behavior of the developer (e.g., in the case of Test Driven Development). Inspired by a technique used in information retrieval, we mapped our stack traces to a *vector space model* [SWY75]. A vector space model is a data structure to index documents and perform efficient comparisons between each document. In information retrieval it is built by splitting a document in a sequence of terms, and turning the document in a vector counting the number of occurrences of each word in the document. We build our vector space model by building a vector for each stack trace: we use the pair *ClassName>>MethodName* to identify the features (the terms) of the vector. We collected all the features in a dictionary and used it to build each vector, where the components of the vector contain the number of times that a method invocation appears in the stack trace.

For example, consider the two stack traces containing the method calls:

<u>Trace 1</u>	<u>Trace 2</u>
UndefinedObject>>DoIt	TabManager>>setTabContentFrom:
BlockClosure>>valueAfterWaiting:	Tab>>retrieveMorph:
BlockClosure>>newProcess	BlockClosure>>newProcess

We collect all the terms and build a dictionary composed of the features:

Dictionary
 BlockClosure>>newProcess
 BlockClosure>>valueAfterWaiting:
 Tab>>retrieveMorph:
 TabManager>>setTabContentFrom:
 UndefinedObject>>DoIt

Using the dictionary we can then build the vectors for the two stack traces:

Trace 1	$\langle 1, 1, 0, 0, 1 \rangle$
Trace 2	$\langle 1, 0, 1, 1, 0 \rangle$

Once we have our vector space model, we can define the distance between each stack trace. For this, we need to define a similarity measure, that indicates how two stack traces are different according to our metrics. Having a vector space model allows us to calculate distances by means of the *cosine similarity*, which for two vectors can be calculated from the definition of the Euclidean dot product, that is:

$$\cos \theta = \frac{A \cdot B}{\|A\| \|B\|}$$

In the case of documents, where the vectors have all positive components, the similarity ranges from 0 to 1. In the previous example, the distance for the two vectors representing Trace 1 and Trace 2 is 0.58. Using the cosine similarity we calculated the first nearest neighbor for each stack trace. With this data we were able to construct a visualization to understand the topology of the stack traces in our vector space model.

Figure 4.5 shows a force graph where each dot is a stack trace and every edge represents the connection between each trace and its nearest neighbor.

The figure shows evidence that there are groups of related stack traces, gathered around a pivotal point. In particular, few large groups gather the majority of stack traces, and the remaining form smaller groups. To represent each cluster, we chose the *medoids* [KR87]. A medoid is the element of the dataset that is nearest to the centroid of the cluster. The advantage of using medoids instead of centroids is that they are an element of the dataset, and thus they represent a stack trace that actually occurred. Moreover, centroids tend to be much more sparse than medoids, thus being more suitable for efficient computation of operations between vectors. We considered the medoids as *archetypes*, that represent the summary of each cluster. The number of incoming edges represents the measure of the popularity of the archetype and, as such, of the whole group.

From Table 4.3 we can see that the largest groups of stack traces are generated by exceptions related to the dynamic nature of the language, and as such probably caused by the style of programming of the developer. We still believe that this information can provide deep knowledge over the status of the system, but we think that their analysis represents a different set of problems that could be tackled with statistical analysis of big volume of stack traces during the evolution of the system. Therefore, at this stage we removed the most popular groups, and focused our

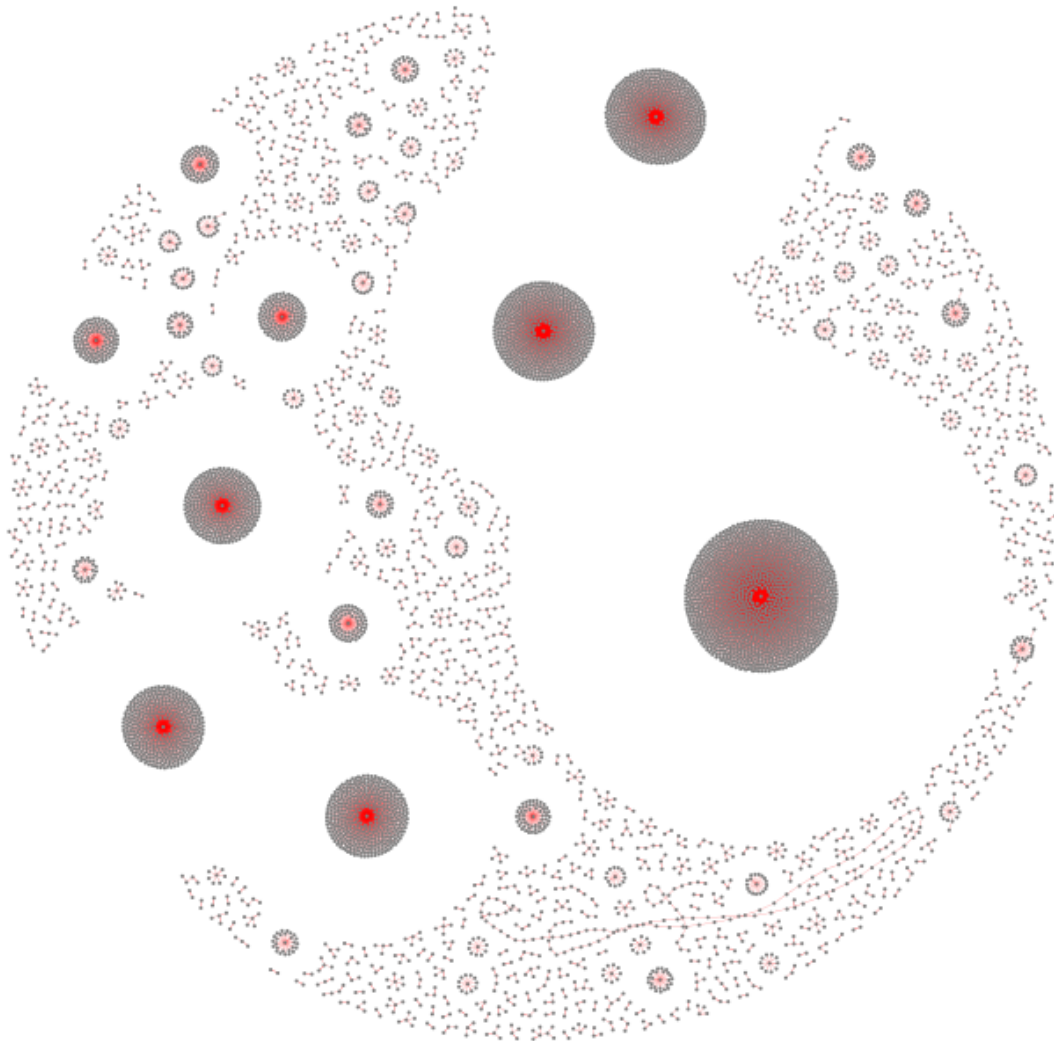


Figure 4.5. Force graph representing the stack traces and their neighbors. Each dot is a trace, each edge connects a stack trace with its nearest neighbor.

inspection on the traces positioned in the central part of the ranking. We used these samples to determine a possible correlation with existing defects.

We mined the PHARO bug tracker to collect the bug reports produced during the development of the platform. To focus our research on actual and relevant problems, but without risk of losing valid examples, we considered the reports opened between January and November 2014.

We extracted 1,910 bug reports, with 17,747 different events, including comments, patches and changes of status. During this period, 1,591 reports have been closed or are waiting for integration, and 319 are still active.

We then extracted from each archetype of stack trace a list of methods invocation. We used this list to search through the data extracted from the bug tracker using a full text search of the

Table 4.3. Summary of the most popular stack traces, with the popularity metrics.

Archetype (first line)	Popularity
UndefinedObject»doesNotUnderstand:	1,585
UndefinedObject»doesNotUnderstand:	647
UndefinedObject»DoIt	619
UndefinedObject»doesNotUnderstand:	428
UndefinedObject»doesNotUnderstand:	427
RGFactory»doesNotUnderstand:	363
BlockClosure»doesNotUnderstand:	127
UndefinedObject»doesNotUnderstand:	111
SystemDictionary»errorKeyNotFound:	71
MouseEvent»doesNotUnderstand:	69
UndefinedObject»doesNotUnderstand:	57
NBGLFrameBuffer»error:	41
RTDraggable»initializeElement:	29
UndefinedObject»DoIt	29
UndefinedObject»DoIt	28

pair *ClassName*>>*MethodName* into the contents of each comment that compose a bug report. After this operation, we obtained a list of the bug reports that are associated to each method invocation. Not every bug report can have the same relation with the stack trace, therefore we applied a heuristic approach to define a ranking to sort the reports in order of likelihood of relevance.

We discussed earlier how the lines of a stack trace that are closer to the top are more likely to be related to the current error, while the lines closer to the bottom are more likely to touch the core parts of the system and thus more generic. We leveraged this principle to give a higher ranking to the reports retrieved using lines closer to the top of the stack trace and lower ranking to those retrieved by lines close to the bottom. In the scenario of a context-aware tool that suggests interesting reports to developers while they encounter exceptions, we observed that the interesting bug reports are likely to be in the first three positions. After these, the link between a stack trace and the information in a bug report quickly becomes too general and likely related to internal mechanics of the system.

4.4 Preliminary Results

We obtained a list of bug reports connected to each archetype stack trace, that we called *topic*. We performed a qualitative analysis on the topics to determine if the retrieved reports could actually provide valuable information about the nature of the exception. We removed from the list the topics triggered by a *doesNotUnderstand:* and *UndefinedObject*, because they are mostly generic and less likely to contain specific bug reports in the tracker. After the filtering, we reduced the list to 629 elements. We then eliminated the elements with the lowest popularity, to exclude the groups that had not enough components to be significant. We set the threshold to be the 0.5% of the maximum popularity, which gave us a list of 23 elements, with a significantly diverse popularity ranging from 1,585 to 9.

We manually inspected the bug reports related to each topic, to determine if they contained information relevant to original exception that would therefore be useful in supporting the development and debugging process. Of the 23 topics, 15 of them had no bug reports connected in the top of the trace and had only reports in the bottom, related to generic mechanics of the

system, unrelated to the specific exception.

The fact that some stack traces had no connected bug reports could have different meaning, and may carry interesting information that may be used in the debugging activity:

1. It may represent an exception that occurs in code that is specific to the project of a developer and therefore not discussed in the system bug tracker;
2. It may be due to a misuse of an API that leads to type errors;
3. It may be caused by a new defect, not yet reported.

Each of the three cases can represent an interesting scenario that can be addressed with a different practical action.

In case (1) the information about runtime error of a developer's code can be of interest for the developer itself and, if collected for further usage during debugging, it can be used to signal possible defects in the code and prioritize the classes and methods to inspect.

Case (2) can take place when many developers use the same API in an incorrect way. Such a case may suggest an area of code or a class interface that requires refactoring.

Case (3) is the most interesting for the developers of the system: It means that the system is raising a lot of exceptions in an area of code that is not known yet for having unexpected behaviors. This could—by definition—represent a new defect, not yet known to the community, or not precisely defined. In this case, grouping the stack traces and highlighting them as a problem to investigate, may provide a valuable support for the community, for example by proposing to automatically open a new bug report containing the collected data to start the debugging activity when the popularity of the group reaches a critical mass, and help to severely cut down the latency between the introduction of a defect and its resolution.

For the remaining 8 stack traces we found related bug reports in the bug tracker. After assigning every report a priority depending on the distance of the top of the stack, we inspected them in order of priority. We observed that, given the structure of the stack trace, only the bug reports in the first two or three lines of the trace are relevant to define the context of the precise problem: the trace of calls then quickly dives into the system core classes becoming thus too generic to pertain to specific scenarios. We found the information of the reports to be relevant to the debugging activity, or to get information on the status of the malfunction: Either the identified reports were addressing the specific issue raised by the stack trace, or were not depicting the same specific context of the exception, but still discussing a related problem.

We now present two example bug reports and show how they are relevant in understanding an exception during development.

ChangeSorter CMD-W with menu open causes error

Morphic: 1. Pharo Image | Milestone: Pharo4.0

A

Closed (Fix Integrated)

Image Version
Pharo3.0

Opened by Camillo Bruni 22/02/2014 18:36

Steps to reproduce
- open change sorter
- right click on the top panel to get the context-menu
- press CMD-W (used to close the window)
- error:

```

1: NewListRenderer (Object)>>doesNotUnderstand: #commandKeyTypedIntoMenu:
2: MenuMorph>>keyStroke:
3: MenuMorph (Morph)>>handleKeystroke:
4: KeyboardEvent>>sentTo:
5: MenuMorph (Morph)>>handleEvent:
6: ...

```

Assigned to Everyone by Camillo Bruni 22/02/2014 18:36

Edited by Nicolai Hess 25/02/2014 14:17
Notified Benjamin Van Ryseghem.
Project changed from 'Tools' to 'Spec'.

NewListRenderer registers itself as a commandKeyHandler
but it does not implements commandKeyTypedIntoMenu:

Changed the project to spec.
(NewList was made for spec, right?)

Edited by Benjamin Van Ryseghem 25/02/2014 14:35
Project changed from 'Spec' to 'Morphic'.

No it was not :)

Edited by Marcus Denker 16/04/2014 09:48
Milestone changed from 'Pharo3.0: 31/03/2014' to 'Later'.

I would say this is not that important for Pharo3... as it happens in the change sorter

Resolved (Fix Review Needed) and assigned to Camillo Bruni by Marcus Denker 30/09/2014 17:22
Status changed from 'Work Needed' to 'Resolved (Fix Review Needed)'.
Milestone changed from 'Later' to 'Pharo4.0: 31/03/2015'.

Name: SLICE-issue-12973-ChangeSorter-CMD-W-with-menu-open-causes-error-MarcusDenker.1
Author: MarcusDenker
Time: 30 September 2014, 5:21:54.680954 pm
UUID: 11b9b8d3-788c-4a00-91fd-d00346f40779
Ancestors:
Dependencies: Morphic-Widgets-NewList-MarcusDenker.4

12973
ChangeSorter CMD-W with menu open causes error

Resolved (Monkey is checking) and assigned to Camillo Bruni by Ulysse The Galactic Monkey From Outer Space 30/09/2014 17:31
Status changed from 'Resolved (Fix Review Needed)' to 'Resolved (Monkey is checking)'.

Resolved (Fix Reviewed by the Monkey) and assigned to Camillo Bruni by Ulysse The Galactic Monkey From Outer Space 30/09/2014 18:02
Status changed from 'Resolved (Monkey is checking)' to 'Resolved (Fix Reviewed by the Monkey)'.
Added tag Validated in 40272.

Issue Validation Succeeded: <https://ci.inria.fr/pharo/job/Pharo-4.0-Issue-Validator/12111/artifact/validationReport.html>

Edited by Nicolai Hess 01/10/2014 09:11

ok, no dnu anymore.

(Although the whole application closes on cmd+w.
I would expect only the menu to close. Anyway if
this is an issue we can open another bug report)

Resolved (Fix To Include) and assigned to Camillo Bruni by Marcus Denker 01/10/2014 09:38
Status changed from 'Resolved (Fix Reviewed by the Monkey)' to 'Resolved (Fix To Include)'.

Resolved (Fix Integrated) and assigned to Camillo Bruni by Ulysse The Galactic Monkey From Outer Space 01/10/2014 13:41
Status changed from 'Resolved (Fix To Include)' to 'Resolved (Fix Integrated)'.

In 40274

Closed by Ulysse The Galactic Monkey From Outer Space 01/10/2014 13:41

B

C

D

E

Figure 4.6. The bug report 12973, related to the stack trace depicted in Figure 4.1. We can see the metadata (A), the initial description that opened the bug report (B), the discussion that followed (C), the submission of a slice and its validation (D), and the bug resolution (E).

Example 1

In Section 4.2 we already presented the stack trace shown in Figure 4.1. The example is particularly interesting because it shows a practical use case for a user or a developer that encounters the exception while she is writing code. Figure 4.6 shows the bug report retrieved for this stack trace. As the report shows, there is a known error caused by a defect in the system, and the community is already working to address it. In particular, since the stack trace that we considered was generated on date 7/7/2014, the user encountered the problem before its resolution and, at the time, the report was stuck in a low priority status. This information could have been exploited by a developer to report more precise information or to ask for an increase of the priority for a quicker defect resolution, while a user encountering the exception could know that there is work in progress, or if there is an estimated time to have an updated and fixed version.

By continuing to read the report, we can see that the problem has been further investigated, and that a *slice* (a piece of submitted code, that in PHARO works in a similar way of a patch) has been proposed and is being tested on the continuous integration server of the project. Finally we can see that the report was closed, the fix was accepted and it is waiting to be integrated in a later version.

Other than simply useful, this information can improve awareness among developers. For example, by depicting a lively and active community, it may disseminate and reward the contributions targeted at improving the general quality of the whole system.

Example 2

Another example is represented by the stack trace starting with:

```
SmalllintManifestChecker>>runRules:onPackage:withoutTestCase:  
RBPackageEnvironment>>classesDo:  
Set>>do:  
RBPackageEnvironment>>classesDo:  
Set>>do:  
RBPackageEnvironment>>classesDo:  
SmalllintManifestChecker>>runRules:onPackage:withoutTestCase:  
CriticBrowser>>reapplyRule:
```

The lines containing *CriticBrowser>>reapplyRule:* are related to bug report 14230. On closer inspection we can see that the bug report contains only three comments, but the last one points to report 110473, where a long discussion (40 comments) is ongoing regarding the relation of the method in the stack trace and the application of rules for the *CriticBrowser*. At the end of the discussion the report gets closed, but as a result of the fix, another bug report is issued to address further weird behavior of the *CodeCriticBrowser*. To add even more correlation to the trace and the report, we noticed that the author of the stack trace is active in the discussion of the report, and actively contributed to its resolution. This reinforces our belief that providing a bug report context when a user encounters an exception can provide great value in debugging software.

4.5 Discussion

We discussed our approach and its preliminary results in investigating the stack trace data that we collected. We now take a critical stance towards our approach, discussing the data, the approach itself and the actual impact that it can have on a development community.

4.5.1 The Data

During this experiment we collected novel data generated from actual daily development activity. The biggest threat that we see in our work is given by the nature of our dataset. Despite having a considerable amount of stack traces, the fact that they were produced by just eight developers may introduce hidden patterns caused by the specific style of programming of each developer, or by the codebase the developers were working on during the experiment. This could lead to a latent bias in our results, that may prove to be too tailored for our users. We are expanding the number of developers using SHORELINE REPORTER, and we will therefore be able to verify the generality and scalability of our approach.

Despite this threat, we believe that the data we collected contains valuable and unexploited information, that can lead to the discovery of hidden patterns in developers' activity. Analyzing this information can produce knowledge that can be helpful in supporting the developers during the bug fixing activities, and can support the work of the community.

4.5.2 The Approach

We designed our approach to find immediate use of the stack traces, and confirm that the data we collected contained information that was both significant and interpretable. However, there are many improvements that can be done to refine the way that we process stack traces and link them with bug reports.

One can argue that the use of clustering is not really necessary in finding a correlation between a stack trace and a report, and that a simple direct search of the elements of a stack trace is sufficient to find the relevant matches. However, we believe that the use of clustering carries some advantages that can be valuable in building a tool to provide feedback on actual data.

Generalization

First of all, the use of clustering allows to identify, group and “average” similar stack traces, having the effect of making the whole process more robust and noise resistant by considering only the most popular stack trace in the group. In this way, even changes in the system that would generate different, but still similar stack traces would have no immediate negative impact in the search result.

Scalability

Even more important, the use of clustering brings the crucial advantage of drastically reducing the size of the problem. While this is not an impossible problem to overcome with the size of the dataset that we considered, in a real world scenario with thousands of developer constantly providing stack traces from errors, the volume of the data would quickly become impossible to process. As such, building clusters that can be used as index and provide a quick lookup for the existing categories of stack traces is a necessary step in building a tool that provides real-time feedback to the user in an acceptable time.

Metrics

The final advantage of building clusters is that it eases further analysis on the dataset. Clustering provides an immediate measure of the popularity of the cluster, it can help in profiling the types of errors on the system during time and ease further investigation on specific groups of stack traces,

allowing deeper inspection of other unexpected behaviors on the system, such as the distribution of Message Not Understood or the distribution of the invoked classes and methods. To develop this approach, we used a very simple, yet effective clustering method based on the connected components of the graph formed by the nearest neighbor. Despite its simplicity, this method already provided useful results in identifying the main groups of stack traces in the dataset, as shown in Figure 4.5. The approach can be further refined with more specific algorithms, such as k-means [M⁺67] or k-medoids [KR87, PJ09], which could provide more precise results. However, the cost for such improvement could be represented by a drop of performance, since these algorithms are computationally expensive. Therefore, the nearest-neighbor clustering represents a good tradeoff between results and efficiency. Also, the problem of a clustering algorithm such as k-means, is that it requires to determine *a priori* the number of clusters to separate our dataset, but in the context of stack traces this information is indeed impossible from the beginning, and it can invalidate the notion of similarity, degrading the approach. Instead, our approach allows a to define a partition without previous knowledge, and that can be easily and quickly adapted as the number of instances increases, and different classes of exceptions and stack traces are discovered.

4.5.3 Applicability of the Method

We think that the ability to immediately link stack traces to bug reports can be effectively exploited to provide on-line help to a developer. We foresee additional benefits that require additional investigation and tool support. Our approach provides quick evidence of the problems in a system and helps finding the immediate context of the error: Therefore, it can be exploited to speed the debugging process, or it can provide information on the current status of a bug in the system. Moreover, since the information presented to the developer depends on the context she is working on, it may also work as additional documentation, and support the understanding of some parts of the code which are poorly documented.

Besides the pragmatic aspects of assisting developers, we think that having a way to access live information on the status of the system may result in a more integrated and open development process. A normal user can be reassured by knowing that the core development team is already dealing with a problem, while other developers may be encouraged to step in and help the resolution of the defect, either by providing additional information or by actually start working on the defect. In an open source project, this set of conditions could bolster the interactions among the community members, focus the attention to current problems and reinforce the whole community.

4.5.4 Next Steps

We see this work as first step towards a new way of dealing with information from error contexts. Current debugging workflows include a number of time consuming activities that could be automated, to reduce the time spent on fixing problems, speed up the development, and foster the improvement of the software project. The approach that we presented in this chapter is only one of the many possible ways that we see possible to adopt in employing this data: leveraging this information can lead to a number of tools that can deeply impact the way communities and developers deal with debugging.

- *Context aware debugging*: as we suggested during the chapter, we want to extend SHORELINE REPORTER to propose the possible bug reports to the developer whenever he triggers an exception, to provide a quicker access to the information needed to deal with the problem.

- *Bug triaging*: Having access to stack trace information can help identifying the types of defects that caused an exception, thus easing the process of triaging the bug [AHM06]. Also, we can use the data submitted by each user to create and update profiles of the developers and determine their area of expertise in a quick and reliable way.

There still is a significant amount of data that we did not consider during our analysis. The information regarding the dynamic nature of the language can still be exploited to get insights on the internals of the system.

- *System core exceptions*: We mainly focused on the top part of the trace, because it contains the part of information closer to the user. The bottom of the stack, which involves the deeper parts of the system, can be used to find bugs hidden in the core classes of the system.
- *Stack trace patterns*: We saw from Figure 4.2 and Figure 4.4 that many stack traces actually touch only a small part of the system. This is an interesting behavior that we want to investigate further, by looking for patterns in the call stack and detect how to deal with “hot areas” of the system.
- *Optimization*: Knowing the main areas of the system that are executed during an exception can also show the frequency of execution during the daily activity of the users. This information can be combined with code profiling techniques to determine where and how to perform optimizations on the existing code, and improve execution performances.

We envision a future where debugging, but also development activities are supported by means of context-aware tools that use automatically extracted information produced by a whole development community, to aid the tasks of developers and support debugging, with the support of the whole community.

4.6 Outline

Fixing defects is an expensive, tedious and time consuming activity: It costs money in industry and it consumes contributors’ time—and energy—in open source communities. The debugging process requires to deeply understand the system, and to gather information to shed light on the nature of the defect. As a result, the debugging process has the side effect of producing a lot of information describing the context of the error. This information is however usually discarded after solving the problem.

We presented SHORELINE REPORTER, a tool that seamlessly integrates into the PHARO system to collect stack traces produced during the arise of runtime executions in the system. The goal of SHORELINE REPORTER is to collect and store information, and reuse it to extract deeper knowledge of the underlying code, assist and boost the whole debugging process. Given the volume of the data produced by the collection approach, it is crucial to have a way to browse the stored information in an efficient and useful way, that allows fast access to the obtained knowledge. We presented a study on the data we collected, proposing an approach to group the stack traces into clusters and use those clusters to retrieve useful information for the developers. We generated the clusters by stack traces similarity, and selected the medoid of each group to represent the archetype of the collection: Each archetype represents a different type of error that happens in the system. We calculated the popularity of each group, that is determined by the number of stack traces that it contains, and used this metric to rank the clusters.

We showed a possible application to exploit the data contained in stack traces by mining the PHARO bug tracker to retrieve the bug reports associated with each archetype of stack trace. We found a connection with bug reports related to the exception and we showed that the information can be used to aid the debugging activity. In the cases where the clusters do not have a clear connection with existing bug reports, the system should highlight the anomaly and propose to open a bug report displaying the information gathered until then.

We have seen how the automatic collection of stack trace generated during a failure can complement the information contained in a bug report with knowledge about the system, to support a number of different debugging and development tasks. However, we are still collecting data that we flatten into a textual format. While the automation process helps increasing the reliability of the information that we provide to developers, we are still forced to turn to text mining techniques to extract the knowledge from the raw data. In the next chapter we further push our approach, and generalize it to enable the collection of arbitrary, domain specific data. By collecting data that has a shape, instead of plaintext, we can finally promote a bug report to an independent entity with its own structure and language.

5

Reified Collection of Runtime Errors

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 identify 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 is usually not reusable.

In Chapter 4 we saw that collecting stack traces can teach us stories about a system that we hardly would have noted otherwise. We were, however, still using a textual representation that limited the descriptive power of our model. In this chapter we generalize the approach by presenting a domain-specific data collection framework that enables the developers to extract selected information about a system, and store them as entities. 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.

Structure of the Chapter

Section 5.2 describes the approach from a conceptual point of view together with its requirements, while Section 5.3 illustrates the architecture and implementation of the proposed approach. In Section 5.4 we assess the technique through three stories that illustrate how our approach can support development and debugging. Section 5.5 discusses the generalizability of our approach how it could be extended. Finally, Section 5.6 concludes the chapter and outlines directions for future work.

5.1 The Tools We Use To Develop

Computer systems have become pervasive in many human activities. The high penetration of machine-controlled devices, that have to deal with an increasing number of different tasks, led to an increase in the complexity of the involved software. This phenomenon turned modern software development into a multifaceted activity, where a one-developer team is no longer a viable option: Building software is above all a collaboration and communication activity. Writing code is only a small part of the process: Several phases such as design, testing, and maintenance, play a fundamental role in the success of a project. In fact, maintenance often represents a significant percentage of a developer's time: Researchers showed that the effort put in reading and understanding code outweighs the effort needed to write it [Cor89, FH82, ZSG79, MML15]. In such a scenario, one would imagine that the effort for providing means to aid developers would focus on refined tools to navigate, understand, and inspect the code. While this is partly true, many of the modern editors and IDEs put the biggest accent on how developers write code, leaving program comprehension as a secondary task.

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 [SFM99]. 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.

All these tools however 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: We advocate the use of objects to store information about an exception, in order to preserve the multidimensional nature of the information and leverage the implicit properties—like the interactions among entities—that can be obtained by the data structure. By describing errors as first-class citizens of a system, and using a storage format that does not flatten the information, we can reify logs and leverage their expressive power to support a number of development activities, such as reproducibility of the error and automated generation of bug reports. A structured data source allows to build specialized tools to browse the data in an incremental fashion and discover its implicit structure. Collecting structured data also enables the use of automated analysis, mitigating the need of a data cleaning phase usually necessary when dealing with unstructured or semi-structured data [BDSDL12]. It can also be stored and sent for debugging purposes, thus creating bug reports

with a much higher level of detail and reliability than simple plain text.

5.2 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, therefore enabling a smoother access to debugging information and increase the descriptive power of a bug report, to provide the groundwork for building interactive tools that present the data in a meaningful context.

5.2.1 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. Using the widely understood format of plain text has numerous advantages, but it has one major drawback: It employs a *flat* format to describe structured data, thus losing track of several properties that have to be inferred. It is easy, by manipulating text, to observe the state of one or more entities: It becomes much harder, however, to describe how these entities are connected and interact. In other words, we can see where the data is but we cannot tell how it flows in our system. This means that the information carried by the structure of the objects in a system is lost or hard to observe. To rebuild the complete information that is hidden in the underlying implicit structure one has to rely to approximated approaches, for example by parsing the text to extrapolate the entities. Paraphrasing the allegory of the Cave of Plato, we are trying to learn the behavior of the entities in our system by looking at the shadows they project on the wall, represented by the textual representations [Plabc].

While the goal of this work is not to criticize how we represent source code, it still helps us to comprehend how a developer perceives software development, since the very beginning of her training. Unsurprisingly, if we 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 and bug reporting systems collect messages as lines of text, stored in a text file.

We can improve the way we deal with the information produced by a software system by employing a different approach, that captures the runtime data preserving its structure (e.g., the structure of the involved objects at runtime) and enables fine grained analyses. Changing the representation we use for bug reports means rethinking the underlying model, in particular with a structure able to capture the connected essence of the entities of a system.

To overcome the difficulties of dealing with plain text to analyze programs, researchers tried to employ different models to represent source code, like SRCML [MCM02] or JAVAML [Bad00].

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.

5.2.2 Design of the Framework

We want to devise an approach for recording the behavior of an executing program in a structured and customizable way, thus creating a powerful logging system that can talk with objects to extract specific and structured information.

To extend the behavior of the logging system, the first step is to define a model to describe the data we want to observe and collect. The goal of this model is to reify debug messages and store them to obtain a level of detail as near as possible to the original objects they are derived from, and to preserve all the information that describes the actual running system. We are reproducing the persistent capabilities of the PHARO image for a smaller subset of entities, without the overhead of saving and serializing the entire system every time an exception occurs.

Shape of the model

The state of a running system is normally a significantly complex entity, with several possible combinations of its variables: Providing a description that is both complete and easily understandable is not an easy task.

Usually, during a debugging session, the easiest approach to quickly understand an unexpected behavior is to verify the state of a program by retrieving data about one or more objects at runtime. Developers usually perform this operation by either printing a log message, or by using an object inspector.

In both cases there is a fundamental problem: To isolate the error, the developer has to (1) identify an unexpected behavior, (2) select a set of properties to observe, (3) change the program to output these properties, and (4) correct the program accordingly. Using this process implies that every change requires a new run of the system.

While this would not pose any hard consequence in trivial development scenarios, it might become a problem if the flow of the program is not fully deterministic, like in case of concurrency, or if it relies on user input. Unfortunately, these cases are also the hardest to identify and correct, which would therefore require the most support by the debugging tools. For example, in the case of a multithreaded application, different runs would result in different internal states: An error in the execution logic, like an unprotected access to a resource, would make an error to appear only under certain conditions, resulting in what is called a *Heisenbug* [GT05].

Another issue comes with the necessity of understanding and correcting the problems experienced by the users of a program. Understanding the condition under which a specific error occurred, and reproducing it for fixing, is one of the hardest things in the debugging process, that consumes a considerable amount of time. Since the developers do not have access to the original environment where the error originally occurred, they have to infer it from the description of the user, or from the textual logs generated by the system, if any. Users, however, cannot be expected to have the necessary technical background to effectively report a bug, which leads to a problem in the reliability of the information that they report to the developers [ZPB⁺10].

Our goal is to have an explicit, flexible, and specific out-of-the-box model that allows developers to observe the state of the system quickly, reliably, and with the lowest effort. The cognitive cost for building a mental model of a system takes a significant toll on the energy of developers. This method however brings a constant cost inside the iterative process of understanding-and-correcting code, as building a mental model is a task that must be performed every time, and does not scale. As such, we want to remove this cost.

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 our framework to be easily extendable and customizable;
4. we should not collect data we are not interested in;
5. we should be careful in handling possibly sensitive data.

The guideline (1) defines the main scaffolding of our approach: We are interested in collecting information from an entity in the system. Therefore, there is no need to prematurely flatten the information that the entity conveys: We rather 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 the information. 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 sensitive data, like passwords or private source code, as detailed in guideline (5). In this case, we apply 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. Of course, this is strictly connected to the application domain, and it is not possible to generalize to all the possible cases where we do not want to collect specific data. Guideline (3) specifies that, to be effective, our framework must allow easy customization of its details.

As an example, consider the case where we want to monitor, from a logging perspective, the errors that users get while accessing a resource. Usually, we would add a command to write a text line into a log file, to store the user and the action. The more information we want to extract, the more text we have to print, with the effect of cluttering the log file. Using the approach we defined, we can set up a rule that activates when the system generates an error that involves the user, and store the entity of the user (guideline 1). In this way, we can directly access the actual entity related to the user, query it about its associated session, and the information about the action that the user was trying to perform. We can also avoid to collect data that we do not want to share, like the password (guideline 2).

This allows us to have a conversation with the entities we are dealing with, rather than collecting some passive text, thus enabling the development and customization of interactive tools that empower the user with the ability of browsing the entities related to the error, and get a quicker and more reliable grasp of the status of the system during an 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 purpose of the collector is to define the data of interest: It must describe what to select and what to discard. For example, going back to 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.

Defining when to collect the data is the other crucial aspect of the framework. We are defining an approach that customizes the data need for certain types of data, therefore defining a domain-specific data collection. Collecting data pertaining to a user is meaningless if we are dealing with an error generated for example from a string. That is why each collector has to know when to activate itself: the system passes the context of the error to the collector, which checks its internal activation rule to decide whether to trigger the collection or not.

Finally, in a scenario where several collectors are involved and activated automatically depending on the entities involved with the error, we need a description to tag the collected data and present it to the user in an informative manner.

By employing the mechanics of the collectors we can build a data collection framework that is fully customizable and that collects first-class, reified entities, to enable a domain-specific system monitoring framework. Such a framework can be employed to replace normal 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 and 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 deep inspection of the state of a system, while preserving the privacy of its users. Being able to define the context of the error is crucial for the success of the whole approach, as we can only operate on the data we can manipulate.

5.3 Implementing The Framework

We now present the implementation details of the framework for the Smalltalk programming language, while in Section 5.6 we will also discuss the challenges of generalizing such an approach to other programming languages.

In implementing our framework, we need to consider a number of aspects pertaining to the control of the system, first of all the ability to directly catch errors and manipulate its context to extract the relevant data in a usable state. Accomplishing such a task is strongly dependent on the programming language of choice with the result that the amount of technical details that can be considered is humongous, and out of the scope of this work. We implemented and test the effective feasibility of our approach using PHARO. The fact that the whole system is described

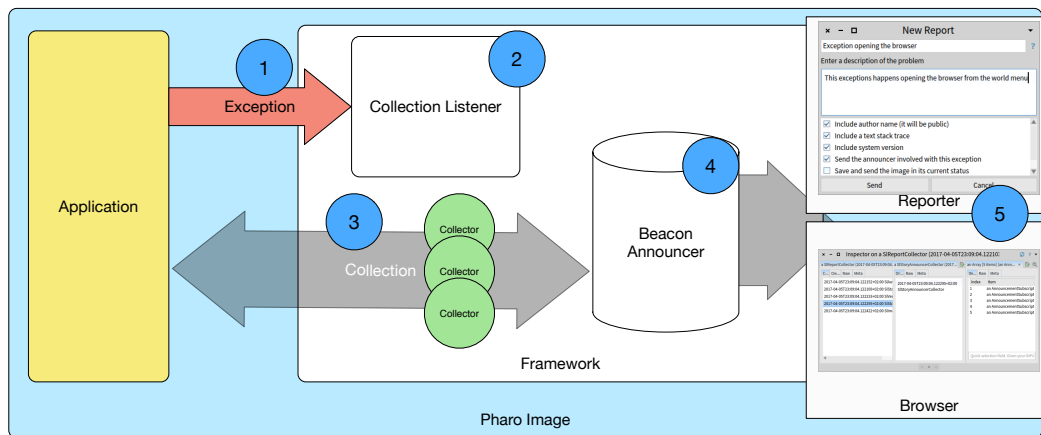


Figure 5.1. The workflow to collect data using collectors, showing the architecture of ShoreLine

using objects enables 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 5.6 contains a deeper discussion about the generalizability of our approach.

5.3.1 Implementation Details

The abstractions in the PHARO environment concern the whole runtime of the system, allowing to inspect and manipulate it by querying objects. The main benefit of PHARO is that we can freeze the execution of a program, and easily access the whole system status at the moment of the interruption.

The principal element we are interested in is the `THISCONTEXT` variable. This special variable stores an instance of `CONTEXT`, an object that mimics the behavior of an *activation record* that contains all the information about the current execution of the program, such as the list of the variables in the scope, the method that is currently executing, the class that owns the method, the program counter of the line of code we are executing, and other information useful in describing the running session.

Implementing Collectors

We designed our framework around the idea of *collectors*. In Section 5.2.2 we defined a collector as an entity that knows how and when to collect data, and that provides a description for this data. By following the object-oriented nature of PHARO we can implement a collector as a class. While having a class for each collector might seem overkill, that might eventually bloat the system rather than supporting maintenance, it has the advantage of providing full control to the user of our framework, granting the flexibility to select the data she wants to observe. The whole strategy can be encapsulated in a single class, thus decoupling the collector from the source code it is observing and providing a behavior that can be plugged and un-plugged seamlessly.

We define the class `DATALECTOR`, that defines the template for implementing a collector. A user can create a new collector by subclassing this class, and implementing 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 `CONTEXT` as parameter, that contains the references to the current execution environment with all the variable in scope and the invocation stack. While the *#when:* method determines if the context is interesting and needs to be collected, it is its responsibility of *#initializeFrom:* to construct what needs to be collected, *e.g.*, an object representing an abstraction of the system state.

In Section 5.4 we show three use cases for a collector, with an example implementation that presents the source code for implementing a strategy.

Triggering the Collection

The collection approach needs an entry point that signals the system that we might want to record the current context and extract the status of an application. We decided to trigger the 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 5.1 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`,¹ 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 the form of objects, rather than text, our approach allows to initiate a conversation with the system and a systematic and progressive exploration of the errors, rather than just providing a partial report of the exception. This enables us to observe the properties of the objects and deal with them in a customized way.

5.3.2 Using the Data

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

The two applications proposed by default by our approach consist of a local data browser, and a customized reporter. If a user is interested in browsing the data locally, for example during the

¹[www.smalltalkhub.com/#!/\\$sim\\$Pharo/Beacon](http://www.smalltalkhub.com/#!/simPharo/Beacon)

development phase of a project, she can inspect the contents of the report objects. Moreover, she can exploit the tools provided by the PHARO ecosystem, like the *Glamour Toolkit* [GBC⁺13] to create custom visualizations of the data to support the browsing session.

On the other hand, if the user of the system is not the developer of the original code, the system can serialize the report and send it to the issue tracker with a comment of the user explaining how she encountered the error. Figure 5.2 shows the interface for submitting a report: to protect the privacy of the users, as described by guideline (5), a user can read the description of the collected data and decide whether she wants to send it or to exclude it from the report.

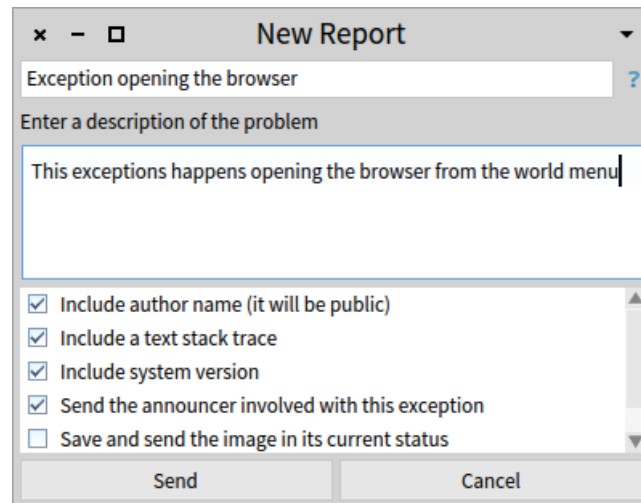


Figure 5.2. The reporting window for ShoreLine, where the user can select the data that she wishes to report and add a description

In the next section we show how to implement a collector to solve common development problems.

5.4 The Framework at Work

In this section we show how to employ our framework to support debugging and program maintenance. We present three scenarios and show how to collect domain-specific information from different environments. 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.

In Section 5.4.1 we 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, a messaging framework that reduces coupling but that might introduce non deterministic behavior; In Section 5.4.2 we outline how to collect data when running a test suite; Section 5.4.3 presents a brief discussion on how using SHORELINE can benefit debugging third party libraries in the context of complex entities.

5.4.1 The Announcer Story

The continuous evolution of the requirements of a software system results in a codebase that grows constantly, 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, with a defined operational context and limited to few specific responsibilities. In such a scenario, dispatching messages is fundamental to exchange information among the components and orchestrate the behavior of the modular system. While the gained modularity is invaluable in developing, testing, and maintaining a system, it comes at a cost: Integrating different modules can show errors generated by the interaction between components and, depending on how the communication is performed, might present non-deterministic behavior. 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. When an entity in the system wants to communicate a message to other entities, it instantiates an *announcer*. The entities interested in receiving a notifications can then register to the relevant announcer, specifying the type of events they want to observe. To broadcast a message, a component can create a new *announcement* and send it to the announcer, that dispatches it to the subscribed entities.

The strength of the Announcer framework is that announcements are treated as first class entities: once it occurs, an event is represented by an object that can contain arbitrary data and the subscribers that registered for that event can interact with it using its public interface. 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 and affects its status, requires accessing information that is not usually accessible by simply inspecting 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, hence not exploitable to understand and fix the problem. In particular, to understand an error generated during the broadcasting of an announcement it is not enough to observe the current stack of method calls: As such a structure would only contain information about the subscriber that generated the error, thus hiding the information about the behavior of the other subscribers. While this might still be sufficient to support debugging of simple problems, it lacks all the collateral information to create the big picture of the status of the announcer and its subscribers.

Usage of Announcements

Figure 5.3 shows a bug report submitted to the PHARO issue tracker, describing a problem with a tool where selecting an item from a menu would trigger the opening of two duplicate windows, instead of one.

During the discussion it becomes soon clear that the incident is compatible with the case of an entity registered twice in the announcer responsible for opening the window. The debugging process consists in hunting down the entity that contains the double registration and removing 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

GTDebugger stack actions are executed twice

GTools: Misc | Milestone: Pharo5.0


 Closed (Fixed upstream)

Image Version
Pharo5.0

Opened by [redacted] 01/20/2016 10:10 PM

call senders of / implementors of from the stack list opens a MessageBrowser - twice. (ConfigurationOfGToolkitCore 3.9)

Edited by [redacted] 01/27/2016 9:21 AM
Priority changed from '3 - Must Fix' to '2 - Really Important'.

Thanks for spotting. This is annoying.

Edited by [redacted] 01/31/2016 7:49 AM

Name: Glamour-FastTable-[redacted].56

Author: [redacted]

Time: 31 January 2016, 7:49:18.44604 am
UUID: 3a2f0bf6-e03a-4f68-9cc1-478200b91893
Ancestors: Glamour-FastTable-[redacted].55

This is a fix for the bug of triggering the actions twice in FastTable.

The problem was that the announcements were installed both during rendering (the correct place) and during handling dataSourceUpdated: (incorrect). I now removed the call from the dataSourceUpdated: and wrote a test for it.

<https://pharo.fogbugz.com/f/cases/17440/GTDebugger-stack-actions-are-executed-twice>

Figure 5.3. A bug report describing a duplicated behavior caused by an entity being registered twice to an announcer

can be tricky, and that further support from the development tools would probably be preferred. We therefore conducted a brief experiment to investigate how common are problems involving announcements in the exceptions that developers usually trigger while writing code.

For this purpose, we inspected the data collected through SHORELINE, the tool we presented in Chapter 4 to intercept stack traces from development exceptions and report them to a central server to support debugging [DSML15]. We considered the stack traces collected from 10 June 2014 to 28 February 2017. Table 5.1 shows a summary of the collected data.

The collecting tool can be set to submit every exception automatically, or to ask the developer whether she wants to explicitly submit it. The stack traces collected come from exceptions generated by users of the PHARO platform from their daily development activities. 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, and we found that 4,840 stack trace contain at least one reference to this class.

Table 5.1. Summary of the collected stack trace data

Oldest stack trace	10 June 2014
Newest stack trace	28 February 2017
# of days	994
# of developers	257
average # of traces per day	~41
# of stack traces	41,129
# of traces involving announcements	4,840
% of traces involving announcements	~12%

Such a result means that almost 12% of the exceptions that were collected by our tool as a result of a system exception, involved the usage of the Announcement framework. While this result does not imply that the framework is directly responsible or involved with the error, it shows that more than one exception in ten has in its source a relation with an announcement. This hints that the scenario is frequent enough to require a dedicated support by the debugging tools, not only to correct errors, but also to help understanding the status of the system during its execution.

Implementation of the collector

Our goal is to use a model able 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 bug reports capable of representing the exception with further details that are not representable using the stack trace generated by the exception.

We can accomplish the task using a custom extension of SHORELINE to collect runtime data that describes the environment of an announcement. To implement the collector we need to define the strategies for its activation and to describe the data extraction. There are two main scenarios that can lead to errors using an announcer: Figure 5.3 showed how an entity being registered twice to an announcer can lead to weird bugs, while the potentially non-deterministic nature of an announcer, given by the fact that messages are dispatched in no specific order, can cause bugs that are hard to reproduce. We therefore decide to observe 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 5.4 shows the implementation of *AnnouncerCollector* the class responsible for gathering data about an announcement.

The methods *#tag* and *#description*, from the class side of *AnnouncerCollector* describe the collector, for indexing and user interaction purposes. The method *#when*: checks the current context object and 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. The other three methods are invoked during the initialization of the collector, once the *#when*: returned a positive response: *#initializeAnnouncementFrom*: extracts the information of the announcement that triggered the broadcasting process, together with possible data that it was supposed to deliver; *#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. Moreover, the method also extracts the

```

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 5.4. The Smalltalk code implementing the extraction strategies for the Announcer collector

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 easing the detection of possible conflicts between different subscribers.

The data collected by the Announcer collector provides a detailed picture of the status of a program, in an environment where it is usually hard to understand what causes the code to fail, thus giving the developer an immediate overview on the dispatching of the messages. Moreover, the data collected are still first-class entities of the system, that can be further queried to extract data: this process allows to retain the maximum amount of information for the longest time needed, so that it can be inspected during an exception or flattened into a textual format for the submission to an issue tracking system. If the system allows it, the entities can also be serialized and sent to the issue tracker, so that the maintainers of a software can navigate the errors generated by the users with a much higher degree of flexibility and introspection on the data than just plain text log files.

Stepping aside from the specific case of the announcer, this example shows how our framework can help developers to easily extend the behavior of the logging mechanism to collect domain-specific data that pertains to the software they develop. By distributing a software together with

some custom collectors, a developer can effortlessly select the exceptions that are caused by her software, collect relevant data—with the consent of the user—and browse it to debug the error.

5.4.2 The Testing Story

Bug fixing is an essential part of software development: Therefore, part of the development effort consists in ensuring that a piece of code behaves in the intended way as the codebase changes. The main tool for this purpose is represented by unit and integration tests, that help the developers to immediately spot unexpected behaviors and locate the responsible components. The use of test cases plays a central role in developing software, especially since the advent of agile techniques like *Test Driven Development* (TDD) and continuous integration [BBVB⁺01]. The former expects the developer to write a test mapping the expected outcome before writing the code to perform the task, while the latter consists in automating the testing process by integrating it in the development workflow.

We analyze the possible role of our approach based on collectors within the combination of these two specific contexts. While in the case of TDD the developer has immediate access to the error and its context, running a test suite to check the whole codebase can generate errors that are complex to frame in the correct context. Moreover, in the case of continuous integration, where the tests do not run on the machine of the developer but are executed on a remote server, developers only have access to the output of the tests, which usually consist of textual reports. This happens for example if a project adopts a continuous integration service like Travis CI,² that performs a full run of the tests every time the project is updated, and then sends a report of the outcome to the developers.

When a test fails, however, the trace of the error is flooded into the log messages generated during the build process, requiring time to identify and filter the relevant information. Once a developer finds information about the failed test, she has to read the text output to understand what was the possible cause of the failure, return to her development environment, reproduce the failed case, locate the source of the defect and fix it. This process might still be lightweight in the case of simple defects in small projects with one developer, but as the organization of the project grows in complexity, reproducing and locating the issue can become a burden that costs a significant amount of time. We can improve this scenario by introducing SHORELINE collectors. By writing a dedicated class, developers are able to extract the specific information they are interested in about a failing test. Writing a collector for a test case is simple: Smalltalk uses the testing suite called *SUnit* [Bec94], where each test class is a subclass of *TestCase*. We can therefore define the *#when:* method to activate the collector if there is an instance of such class in the invocation stack of the exception. We can then write the extraction strategies to gather the status of the part of the system that is subject of the testing, or simply extract the whole context storing all the objects in the stack.

Using collectors, our framework allows developers to generate automatic reports that can be sent and managed in an automated fashion, guaranteeing the reliability and quality of the data. This data can then be serialized using a framework like *STON*,³ so that a developer can still access the parts of the objects involved with the exception. The use of objects allows to directly communicate with the status, without having to revert to text mining techniques to infer the involved entities from textual logs.

For example, when a test fails, *SUNIT* generates an *ASSERTIONFAILURE* exception. Catching this exception allows to access the context of the test, containing the erroneous result and the local

²<https://travis-ci.org/>

³<https://github.com/svenvc/ston>

variables, especially the class that is object of the test. By serializing and sending this information, we can collect and group this data from all the failed tests to generate a report containing all the objects in a wrong state, and navigate it to identify the cause for the failing tests.

Sometimes, however, observing just the tested object is not enough: The cause of the failure might be hidden in the underlying system, or the object would reference volatile resources, like database connections or remote sockets. In this case we can adopt a different approach and offer a default option to cover the most difficult cases. Leveraging the Smalltalk approach of the *image*, that can be frozen and saved in a particular state and later reopened with the exact same state, we can write a collector that, when an exception occurs, saves the image and sends it to a central server with a report containing metadata about the exception. Using this approach, when a build fails to pass the tests, the developer can browse the repository of images, download the relevant one and open it on its local machine. The concept is similar to using *Docker*⁴ to deploy applications, where the system runs in an isolated controlled environment, independent from the host operating system.

Either using the automatic report generated by the failing tests, or downloading directly the image containing the error, developers can observe details of the system at the moment of the failure, thus shortening the time required to gather information to reproduce the error.

5.4.3 Debugging Third Party Libraries

The last example on the usage of the framework shows how a developer of a third party library can benefit from implementing a collector observing for specific data about her project. In the Smalltalk ecosystem, *Roassal*⁵ is a large visualization engine widely adopted in the SmallTalk community, made to “visualize and interact with arbitrary data, defined in terms of objects and their relationships” [ABC⁺13]. *Roassal* codebase consists of more than 800 classes and almost 6,000 methods, and is constantly evolving and improving using the feedback of the community. Maintaining such a large project is a complicated task, as tracing and addressing the errors experienced by the users can become like hitting a moving target, especially since *Roassal* integrates with other tools and because the community is usually split among stable, legacy, and development releases.

Understanding what triggered an error and rebuilding the environment to reproduce it can become quite painful, as the developers need information that might not be easy to provide. Using a coarse-grained approach as the one we described in Section 5.4.2 would not fit this scenario, as asking the user to submit the whole image would generate a lot of traffic that would be hard to manage, without considering that the image could contain sensitive data, like private source code, that a user might not want to share.

The maintainers of *Roassal* can improve this situation by observing data specifically related to the model of the engine. By detecting either when an exception occurs involving an object that is a subclass of *RTOject*, or by checking if it happens in the context of a builder—a *Roassal* object responsible for generating a visualization from a collection of data—a collector can determine if there is some information about *Roassal* available, that can be collected and reported.

By collecting domain-specific information, the maintainers can get a much more detailed picture of the error, and restrict the possible causes to the ones compatible with the collected data without the need to access the actual 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. Instead, knowing the settings that were used to configure

⁴<https://www.docker.com/>

⁵<http://agilevisualization.com/>

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.

By 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 lowering the cost for maintenance.

5.5 Discussion

In this section we discuss how our approach can be generalized to work with other programming languages, and how we think it could be further improved.

Generalizability of the approach

As we explained in Chapter A, we developed our approach using PHARO. The strong reflection capabilities of the platform allowed us to inspect the whole status of the system, retrieving valuable information about the whole 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 chapter, the data collection mechanism can be integrated with the issue tracking system of a project, allowing the developers of a project to integrate SHORELINE in their workflow, supporting debugging and maintenance tasks. About question (2), we believe that it is possible to implement such an approach also in other programming languages. The PHARO system makes the perfect candidate to test 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 5.4 can be implemented with any language with reflective capabilities.

There is also an interesting aspect to consider pertaining to how we used a collector in Section 5.4.2 to save and submit the whole PHARO image. Given the current interest in DevOps technologies like Docker⁶, it is possible to execute an application in a Docker container, stop it and save its status during an exception, and submit the image of the application 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.

5.5.1 Next Steps

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 the users use a software, hence helping developers not only to debug a system, but also to optimize the existing code and improve the API.

⁶<https://www.docker.com/>

We envision a future where development activities are supported by the system using the language of the system, not resolving to flat and bloated chunks of plain text that represent the side effects of the code, but rather first-class entities that narrate the exact status of a program. Starting a conversation with the entities we can develop a paradigm of programming that focuses on the models and their interactions, rather than manipulating strings, and achieve a programming environment that is really live and responsive.

5.6 Outline

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 domain-specific knowledge that follows the model of the application she wants to observe. By preserving the structured, object-oriented nature of the collected data, rather than serializing the information to text, we are able to query the state of a program and observe it by filtering the data we are interested in, resulting in both more expressive and less bloated reports. Such capabilities of observing a system enable to create flexible inspection tools, that are able to get a deeper representation of the execution context of a piece of software.

By giving the possibility to report and collect specific information from the system, SHORELINE offers data that is more reliable than a stack trace submitted by a user, and allows to deal with the collected data in an automated fashion, giving the possibility to programmatically perform tasks that would otherwise consume time of the developers and weight on the cost of maintenance. Moreover, providing a defined structure for execution data, our framework allows us to perform a number of analyses without having to resort to information retrieval and text mining techniques to clean the data, but enables us to talk directly with the collected entities that map the original 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 to select the data needed by the developer fixing a bug.

In this chapter we put in place a platform to collect detailed domain-specific information about components of a system. At last, our model is no longer text-based, but represents entities using objects. As a result, we can start a conversation with the system and define a language that allows us to integrate this information with the development tools and to create smarter and conversational environments. In the next chapter we present a use case where we employ the information collected about a system in a tool for browsing the history and the evolution of a software system.

6

Multi-concern Visualization of Large Software Systems

While constructing and evolving software systems, developers generate directly and indirectly a large amount of data of diverse nature, such as source code changes, bug tracking information, IDE interactions, stack traces, etc. Often these diverse data sources are processed and visualized in isolation, leading to a partial view of systems.

In the previous chapters we saw that we can collect data from runtime errors and use this data to get insights into the development process. In this chapter we present a *blended* approach to visualize several data sources. We combine these “ingredients” at once, to give as complete an answer as possible to the question “*What happened to the system in the last few days?*”. The goal is to enable a quick and comprehensive assessment of what happened to a software system in any given time frame.

Structure of the Chapter

In Section 6.2 we describe the ingredients of our blended visualization, which is presented in Section 6.3. In Section 6.4 we use the visualization to tell interesting evolutionary stories. Section 6.5 discusses possible extension to our approach. Finally, Section 6.6 summarizes and concludes the chapters.

6.1 Exploring a System

Software development involves a variety of activities carried out with a number of tools, components and environments, that relate to many different aspects of a system. The increasing size of software projects, the increasing popularity [GZSVD15] of distributed development platforms like GitHub¹, and the amount of tools and frameworks available for every language, turned a significant part of modern software development into an integration process, where the developer can define a behavior by orchestrating and specializing library components and third-party entities. This has turned the engineering of any software system into an information-heavy process, which is ultimately distilled into (hopefully functional) source code. The vast majority of the corollary information (such as discussions, design decisions, email communication between developers, bug reports, etc.) is either discarded or ignored. This is in part due to its often only semi-structured nature, where structured fragments are interleaved with natural language. The mining of such unstructured data has become a research field of its own in the past few years, creating also a workshop of its own [BA10].

When it comes to the understanding of any system, the natural focus is the source code, and indeed it –and the overarching structure and architecture– has been the primary subject of study of program comprehension research. In the context of software visualization many approaches have been developed to visualize the (evolving) structure of software systems, which range from static visualizations to historical or dynamic ones. What strikes in this context is that many approaches consider only single concerns, such as the architecture, the structure, the evolution, the relationships, etc., but there is little in terms of visualizing multiple concerns at once.

We present here an approach to visualize multiple concerns concurrently. The concerns we tackle are interaction data, failure information, and evolution. Interaction data stems from how developers interact with the integrated development environment (IDE) while developing and maintaining a system. In essence, it provides evidence of where and how people have been active while developing [MML15]. Failure information is generated each time the debugger is triggered because an exception has been raised. In our previous work we have shown that such data can be leveraged to understand where the particularly tricky spots in a software system are located [DSML15]. Both interaction data and failure data are more fine-grained than their respective counterparts, namely versioning information and bug reports. We complement these two types of data with a third one, the evolution of the system.

Although we focus on these types of information, our approach can be extended to feature any kind of information artifact related to a large software system under development. In essence, our goal is to answer one of the most often asked questions raised by developers and managers alike, namely “what happened to our system recently”? [SMDV08]

We present a visual approach to *blend* development data originating from different sources, e.g., by different tools that record and persist code changes, interaction data and stack traces. We propose an interactive map that summarizes the relevant events that involved the system in a given period of time, using the city metaphor to represent a software system [WL07], and coloring each entity according to the combination of data gathered around it. We allow to explore the evolution of the system by navigating the information during time, and refining the search of interesting events to specific moments. We then present some stories obtained through our visualization that illustrate interesting properties of an existing software project and its community.

The contributions we present in this chapter are:

- A novel approach to visualize multiple concerns concurrently in large scale software systems.

¹<https://github.com>

- The supporting tool infrastructure to mine and integrate the data stemming from various sources of information.
- Initial anecdotal evidence that our approach indeed allows us to discover and investigate facts that would otherwise remain hidden in the literal “sea of data” that surrounds any large and long-lived software system.

6.2 The Ingredients

In this section we briefly describe the three main ingredients together with the tools that enable the data collection process.

To get a tractable subset of meaningful data, we decided to focus on a timespan ranging from January 1st 2015 to May 1st 2015. In this section we present the context of our analysis, then we briefly describe the three main ingredients together with the tools that enable the data collection process. Our visualization presents a composition of different information, obtained by blending together different data sources and enabling visual analytics from heterogeneous and multidimensional perspectives. In the rest of the section we describe the three main ingredients together with the tools that enable the data collection process. For further details about the PHARO platform see Appendix A.

6.2.1 Source Code Changes

A typical metric that is often considered in evaluating the growth and evolution of a system is the number of changes that it goes through during its development. In the case of PHARO, the whole system is self-contained and distributed as an *image*, a single file that works as a virtual environment where new code is installed inside the default system. The PHARO system is released once a year, and during this period it goes through an intense phase of improvement, debugging and polishing. The test and release process is managed by a continuous integration server,² that stores the previous builds of the system. In our analyses we modeled and extracted all the source code changes between subsequent releases of the PHARO system.

Retrieving the different version

We focused on the release of PHARO 4, which just finished its release cycle. We downloaded all the development versions from the file server,³ that we also used to retrieve the exact release date of each version. The full cycle of development images ranges from version 40,000 to the image 40,613, from May 26th 2014 to May, 5th 2015. The last release in date May 1th 2015 was version 40,611.

Extracting a system model

We extracted from each image a model representation of the system. Such a model is composed of the names of all packages, classes, instance and class methods, and instance and class attributes.

²<https://ci.inria.fr/pharo/>

³<http://files.pharo.org/image/40>

Generating an incremental change model

We leveraged each system model to obtain an incremental diff model that describes each change. We considered as change a variation in the names of the collected entities. Since we had no way to precisely determine when an entity was renamed, we considered every event in terms of creation and deletion. Table 6.1 summarizes the available source code changes data.

PHARO, the target IDE of our study, is an open-source system maintained by an active community. During its evolution it undergoes a series of minor and major releases, managed by a continuous integration server⁴. In our analyses we modeled and extracted all the source code changes between subsequent releases of the system. Table 6.1 summarizes the available source code changes data.

Table 6.1. Source Code Changes

Metric	Value
Number of considered versions	611
Number of changes	4,928
Average changes per version	8
Max number of changes per version	527
Min number of changes per version	0

6.2.2 ShoreLine Reporter and Stack Traces

A consistent part of the time spent by developers consists in finding and solving defects. The debugging activity involves tests to reproduce a problem or verify that a defect has been solved. This process generates many stack traces, that contain valuable information about the failures in a system. Such information is normally used by a developer to identify a faulty status in her program. Moreover, if collected and stacked together, stack traces can also give a hint of what parts of the system are the most active, or which ones are causing more troubles. To exploit this source of information, we developed SHORELINE REPORTER [DSML15], a platform to collect and store stack traces generated by the whole PHARO community. The data we collect contains the signature of every method invocation, to keep track of each entity involved in the failure, though excluding the method parameters, to avoid privacy issues for the single developer.

In enabling the reporter, each developer can decide to inspect each stack trace and choose the ones to submit, or enable the automatic reporting feature and submit all the traces produced by its activity. While this option produces many duplicates and non relevant data, it is still interesting to see where the activity of the developers focuses during different periods of time. The collected data can then be used to aid the debugging activity, for example detecting if a large volume of new stack traces coming from different developers involve a specific class, or by looking for existing bug reports in the bug tracker to provide a contextual help when a user encounters an exception and ease the understanding of a piece of code. The presence of many different stack traces for a specific component might also suggest that an API has a problematic design, and that the users struggle in understanding its usage, thus highlighting the need for documentation or refactoring.

Table 6.2 summarizes the collected and available data for stack traces.

⁴<https://ci.inria.fr/pharo/>

Table 6.2. Stack Traces Data

Metric	Value
Number of traces	14884
Number of submitters	43
Total number of stack trace lines	714,420
Average stack trace size (in lines)	48
Longest stack trace	1,086
Shortest stack trace	1

6.2.3 DFlow and IDE Interaction Data

During the process of software construction and evolution, supported by integrated development environments (IDEs), developers generate a large amount of data known as “*IDE Interaction data*” [KM05, MKF06]. Examples of such data include i) *IDE meta events*, like adding a method to a class, saving some edited code, or inspecting a variable in the debugger, ii) *UI events*, like moving a window or a tab in the IDE, or resizing them, and *low-level events*, like keystrokes, mouse clicks, drags and simple movements.

Since current IDEs do not record these data, we used the data collected by Minelli *et al.* who developed DFLOW, a silent interaction profiler for the PHARO IDE [MML15]. DFLOW records 32 different types of events at different levels of abstraction. For this work we only focused on a subset of meta events that involve code entities. Some meta events have an associated program entity: A browse event, for example, where the user opens a new code browser, can be performed on a method or on a class. For this work we aggregated all meta events to the class-level: An event performed on method `foo` of class `Bar` counts as an event involving directly the class `Bar`. In total we have ca. 239,000 interaction data events covering a timespan of 4 months (*i.e.*, from January to April 2015).

The IDE interactions impact 2,988 different classes, of which 965 are part of the standard PHARO distribution. The remaining 2,023 classes are user defined classes that are outside the scope of our study. Out of the 32 types of meta events recorded with DFLOW [MML15], only 13 types of events appear in the dataset. This is because some of the recorded meta events do not carry any information related to program entities. For example, the meta event that represents the opening of a `Finder`, a user interface used in PHARO to search for pieces of code, has no associated program entity. Table 6.3 summarizes the dataset and provides additional details.

Table 6.3. IDE Interaction Data

Metric	Value
Number of Interaction Events	238,741
Number of Developers	18
Number of Interested Classes (in the PHARO distribution)	2,988 (965)
Number of Different Event Types (total)	13 (32)

6.2.4 Blended, Not Stirred

Our goal is to develop a visualization approach which can represent diverse data sources, such as the ones we just presented. The approach is not geared towards the specific types of sources, and also not limited to depicting just those, but is in principle extensible to feature any number and any data source.

6.3 Visualization Principles

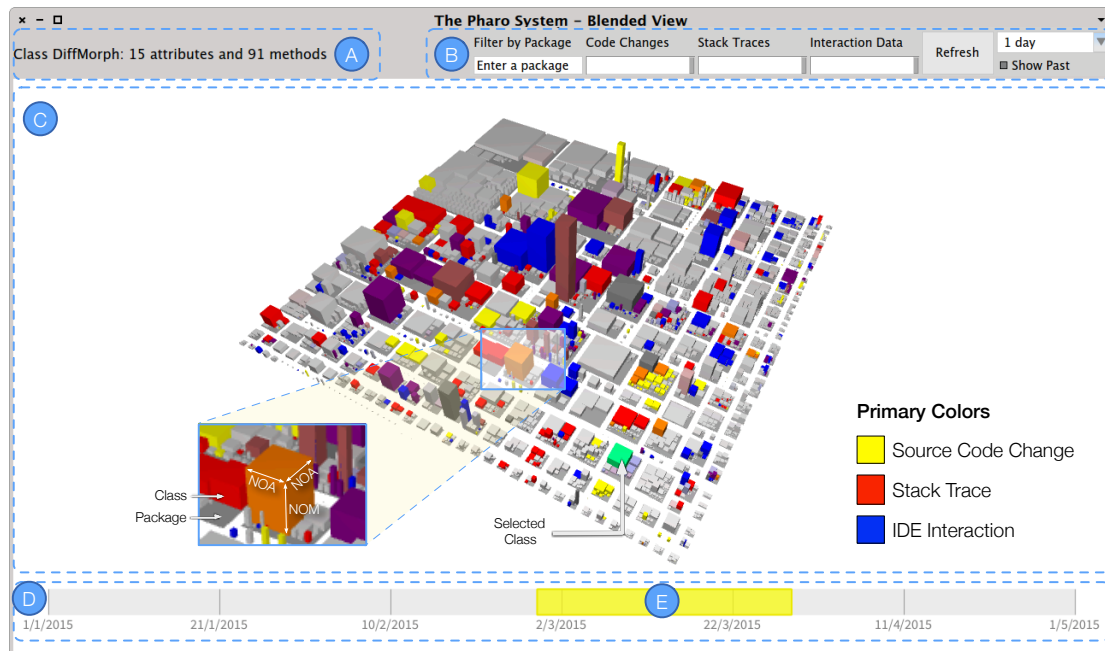


Figure 6.1. The Blended City – visualization principles and proportions

Section 6.2 introduced the three “ingredients” of the visualization: source code changes, stack traces, and IDE interaction data. Until now these diverse data sources are processed and visualized in isolation, leading to an incomplete view of the system. Our goal is to visualize all these ingredients to enable a quick and comprehensive assessment of what happened to a software system in a given time frame. To do so, we propose the “*Blended City*”, a visualization that uses the *City Metaphor* to depict all the ingredients of a software system. Wettel and Lanza initially used this metaphor in *CodeCity*, a tool that depicts software systems as cities [WL07]. In addition to the structural source code information presented by *CodeCity*, our *Blended City* uses a mixture of colors to depict different aspects of the software system itself. Figure 6.1 shows an example of our visualization.

6.3.1 In Practice

Figure 6.1 shows the tool that we implemented to visualize the *Blended City*. It is composed of four main parts: A status bar to display additional information on the selected entity (Fig. 6.1.A), a toolbar to customize the visualization (Fig. 6.1.B), the view canvas (Fig. 6.1.C), and a timeline slider (Fig. 6.1.D). With the timeline slider the user chooses the visualized data timespan. The width (*i.e.*, granularity) of this slider can be adapted using the dropdown menu on the right part of the toolbar. In the example of Figure 6.1 the user selected one month of data, starting from March 1st. The toolbar (Fig. 6.1.B) also features a text-input and a set of sliders. The former enables simple queries to highlight particular packages in the system while the latter let the user choose the visual weight of each of the three ingredients of our visualization. These weights affect the intensity of the color associated to each of the ingredients. In the example of Figure 6.1, all the sliders are at 100%, thus all the ingredients have the same importance. Figure 6.2, instead, shows

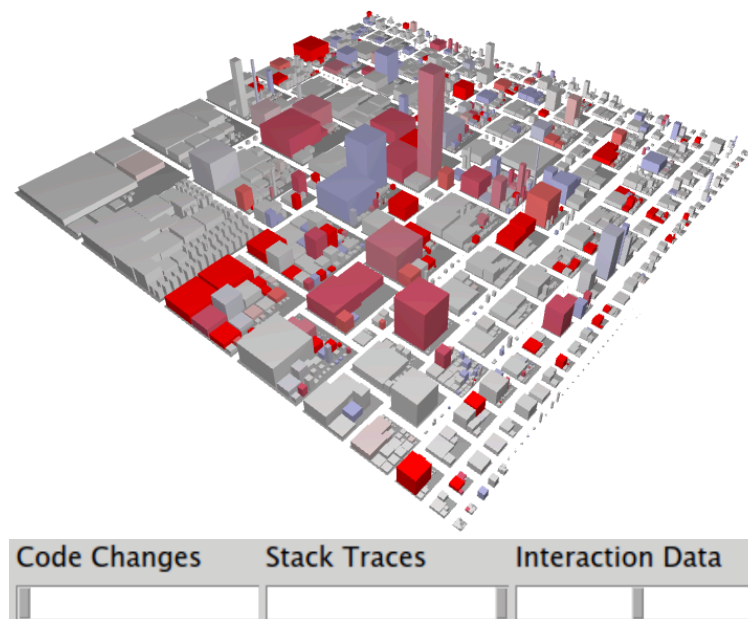


Figure 6.2. The same view depicted in Figure 6.1 with the following weights: 0% source code changes, 100% stack traces, and 50% interaction data

the data presented in Figure 6.1 giving high importance to stack traces (100%), little importance on interaction data (50%), and no importance to source code changes.

In addition to changing the weights of the three components and the granularity of the visualized timespan, the view also features standard interactions such as panning and rotation in the 3D space. Moreover, the user can click on an entity and get additional information on the status bar. In Figure 6.1 the user selected the class `DiffMorph` and the tool shows that this class has 15 attributes and 91 methods (see Figure 6.1.A). Selected entities are colored with a bright green.

6.3.2 The City Metaphor: Layout and Metrics

In the city metaphor every district of the city is a package and the buildings, contained inside the districts, represent the classes [WL07]. The view uses a rectangle-packing algorithm to create the layout and it is *polymetric*, *i.e.*, each dimension of the visual entity is proportional to a particular metric of the program entity being represented [LD03]. Since the visualization is 3D, classes are cuboids and have 3 dimensions that correspond to three metrics. Our visualization, similar to the original `CodeCity`, uses the same metric for both width and depth and a different measure for the height. In particular, we use number of attributes (*i.e.*, NOA) for both width and depth of a class and number of methods (*i.e.*, NOM) for the height of the cuboid representing a class. The magnification in Figure 6.1 exemplifies these mappings.

6.3.3 Color Harmonies and Blends

Our Blended City presents different types of data, from structural properties of source code to stack traces and interaction data. Structural source code relationships (*i.e.*, nesting of the package and software metrics) are the foundations for the layout while colors present the remaining information.

We use a triadic color scheme made of primary colors to present this information: Yellow for source code changes, red for stack traces, and blue for interaction data. Figure 6.3 shows a the color wheel with an emphasis on the triadic color scheme, where colors are evenly spaced around the color wheel.

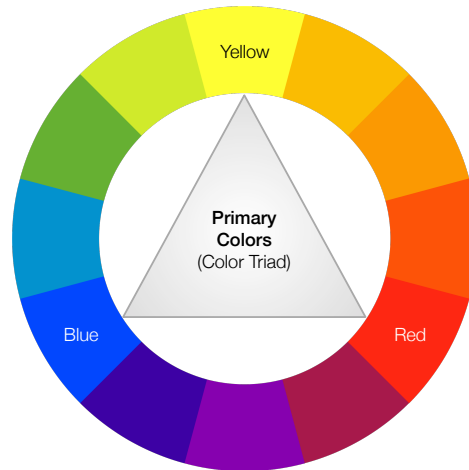


Figure 6.3. Color wheel and triadic color scheme

This offers strong visual contrast while retaining balance, and color richness. Using colors equally spaced around the color wheel facilitate the addition of extra sources of information, *i.e.*, when we need to display n sources of information, we can create a new color harmony composed of n colors evenly spaced around the color wheel.

Color Blends

The three primary colors can only depict entities which are affected by a single of the three information sources. However, in a given timespan a class might be affected by both IDE interactions and stack traces, for example when a developer is adding new functionalities to a class and testing them. To depict this information, we use linear color blends between the different sources of information. A class with both IDE interactions and stack traces is depicted in purple, the linear blend between the color of IDE interactions (*i.e.*, blue) and stack traces (*i.e.*, red). Figure 6.4 shows examples of the different linear color blends on the triadic color scheme adopted by our visualization. In this work we only considered the linear blending of colors. It is part of our future work the investigation of different techniques to combine the colors, *i.e.*, color-weaving.

Aging Mechanism

When the user selects a timespan to visualize, the tool pre-loads and displays also the data happening in the immediately preceding interval (of the same length). This enables the user to draw conclusions from the visualization having also in mind what happened immediately before. To show this data, the tool uses an *aging mechanism* that linearly reduces the color saturation as the age of the datapoint grows, *i.e.*, the older the more intense fading towards the default color of nodes (*i.e.*, gray). Figure 6.5 shows how colors fade with such mechanism in a timeline.

In the “present” interval (*i.e.*, the one selected by the user), colors are at their default saturation. In the “past” interval, instead, the color saturation fades. At the end of this interval, the nodes have the default color, *i.e.*, light gray.

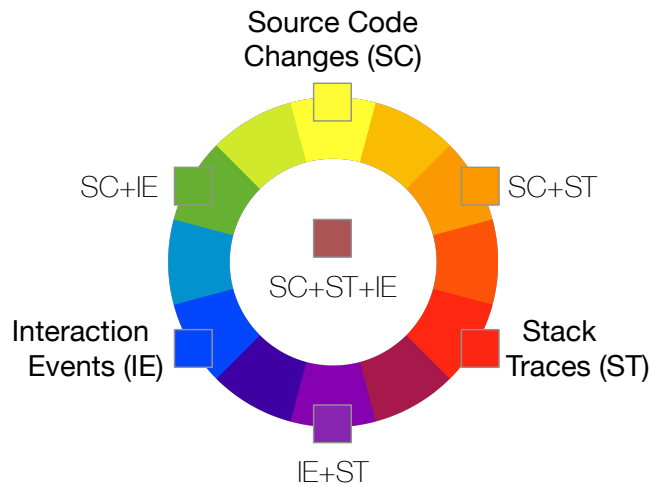


Figure 6.4. Linear color blend on triadic color scheme

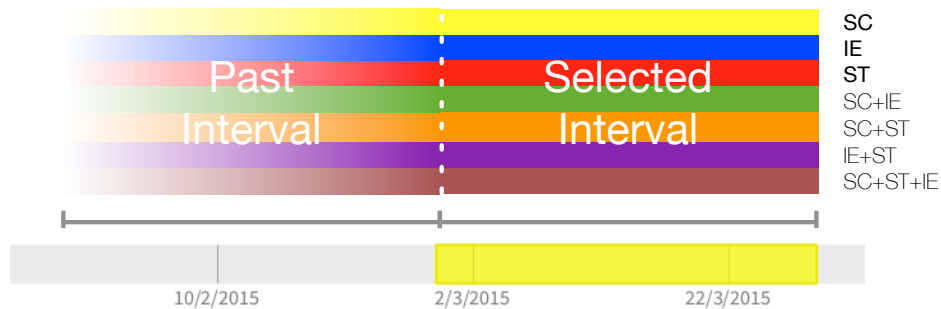


Figure 6.5. Aging process: example in the timeline

6.3.4 Under the Hood

The tool deals with a large volume of entries coming from heterogeneous data sources. To conveniently manage them we standardized their format, using different data pre-processors, and store them in a central place. We use *MongoDB*⁵ databases to conveniently store the data.

When the user selects a timespan to visualize (Fig. 6.7.1), the tool loads the data through optimized *MongoDB* queries (Fig. 6.7.2) and builds the blended model of the data (Fig. 6.7.3). Later it computes the city layout, applies the blended color scheme, and presents the view to the user (Fig. 6.7.4). The user can then use the toolbar to refine the visualization (Fig. 6.7.5).

6.4 Telling Evolutionary Stories

This section presents four stories, supported by our blended view, that narrate the evolution of the PHARO system.

⁵<http://mongodb.org/>

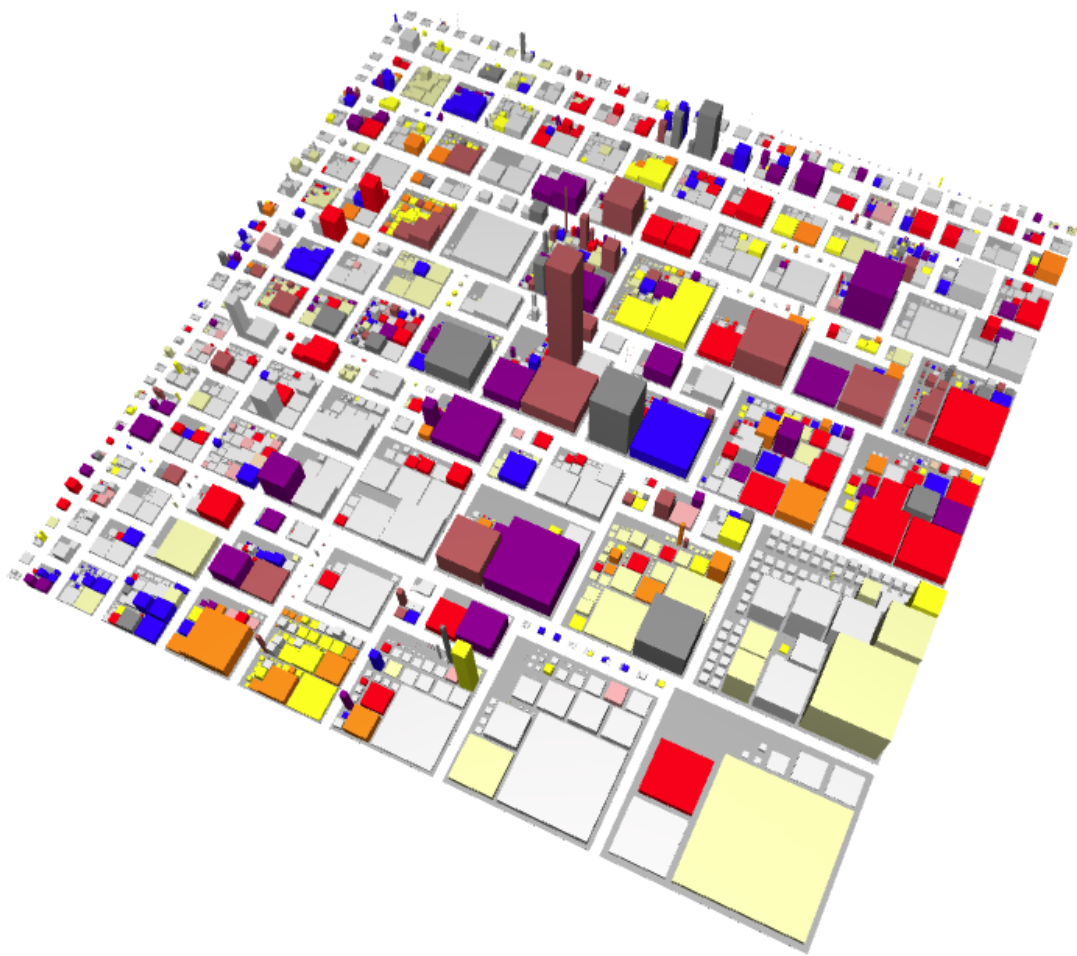


Figure 6.6. View of the city with all the activities

6.4.1 Those Awkward Neighbors

By selecting the full available timespan of the data we obtain a visualization that displays all the activities that involved the PHARO system over a period of five months. This enables to obtain a comprehensive view of the system evolution and derive long-term considerations and properties. Figure 6.6 shows the overall view of the available data. One interesting example is represented by what we call the *awkward neighbors*, i.e., big but silent packages that have little or no activity.

In the lower part of Figure 6.6, we can spot two big packages that contain entities that are mostly colored with grey, meaning that they had almost no activity in the whole timeframe. Moreover, they present almost no change in the entities they are composed of, and since the color of the changes is blended, those are all antecedent to the selected start date. This means that in the last release they have been mostly ignored. These two districts are the packages *Graphics-Files* and *Compiler*, whose details are shown in Figure 6.8.

A further investigation of the package *Graphics-Files* reveals that it contains 10 classes. These classes are dedicated to exporting graphics and writing them in different file formats. Since PHARO stores the dates of the changes of a method, we can determine when the changes took place. We can see that there are three main batches of changes: A small update in 2014, regarding a small

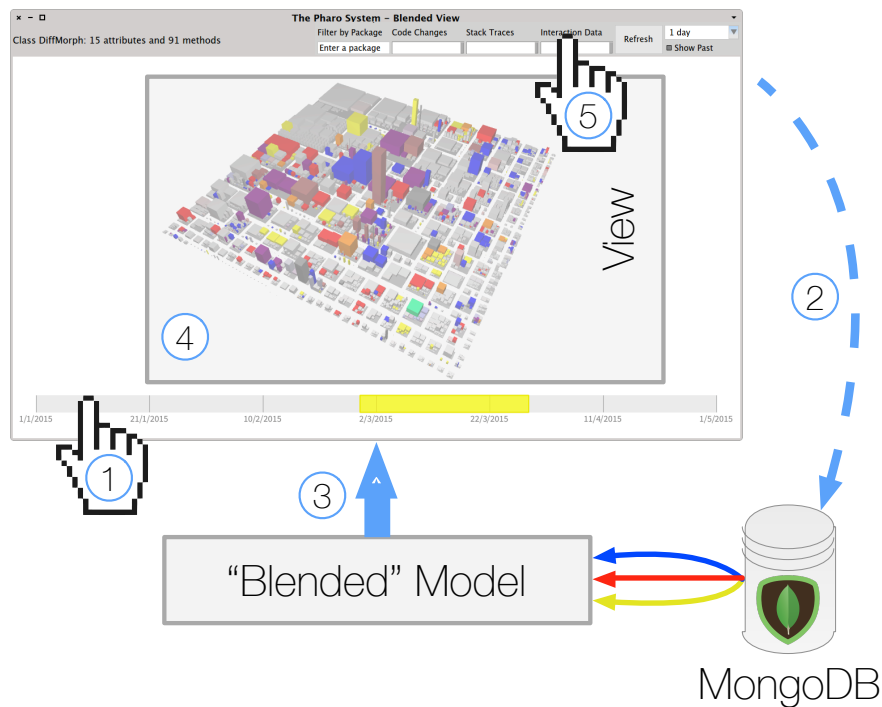


Figure 6.7. The architecture of the Blended City

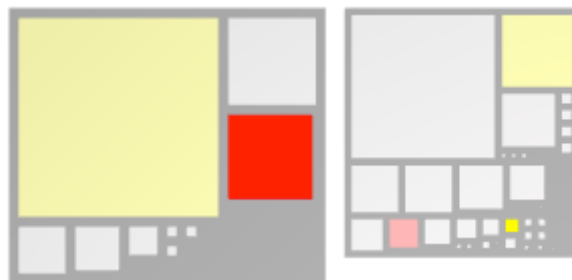


Figure 6.8. Details of the packages *Graphics-Files* and *Compiler*

refactoring of an error message, one in 2001 and one in 1997. This is interesting, because it indicates that the package has been part of the system for a long time, it had little changes and is by now a solid foundation of the system. Similarly, the package *Compiler* contains 46 classes, and apart from some recent modification in 2013 to the structure of the compiler, many of the methods are unmodified since 2006, 2003, or 1998.

One might wonder how it is possible that some parts are older than the PHARO project itself. The reason is that PHARO was born as a fork of the SQUEAK project⁶, which in turn is a reimplementation of the original SMALLTALK-80 system, which was evolved from the SMALLTALK-72 system. This means that some of the methods and classes in these packages might very well be 40+ years old.

⁶<http://www.squeak.org>

6.4.2 Market Districts

While examining some of the packages with the most activities, we found districts with many interactions from all three data sources, and we call them *market districts*. Figure 6.9 shows an example of market districts corresponding to the packages of *Spec* and *Morphic*. *Morphic* is the core graphic library of PHARO, while *Spec* is a framework to build user interfaces, built on top of *Morphic*.

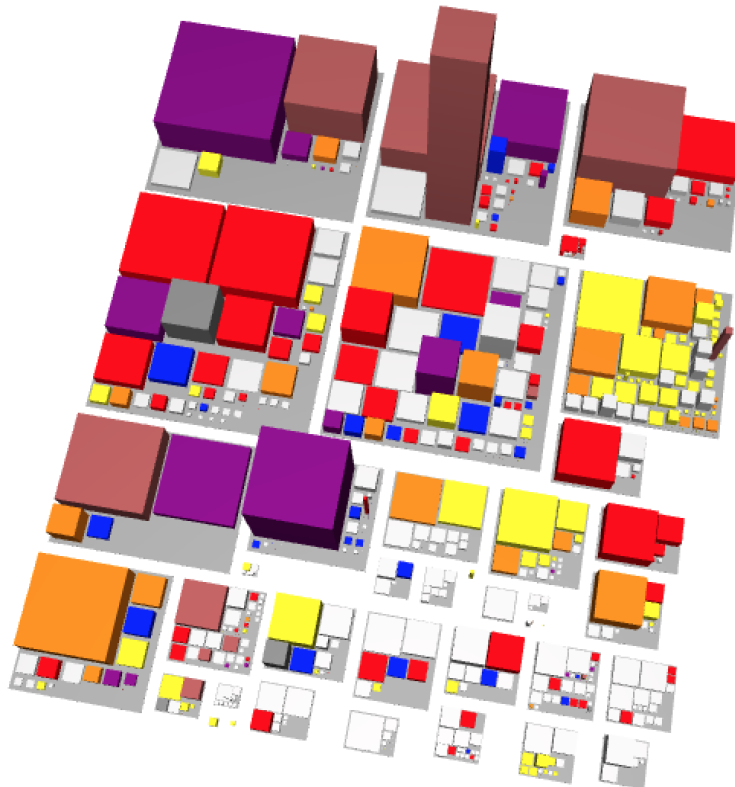


Figure 6.9. *Spec* and *Morphic* market districts

Many classes are involved in exceptions, they were recently changed or they were subject to developer interactions. This reveals a long known problem in the community, that is, the fact that the code of *Morphic* is old and has been ported through various platforms. The case of *Spec* is similar: since *Spec* is a framework built on top of *Morphic*, it shares its weakness and part of its complexities.

Differently from the awkward neighbors (shown in Figure 6.8), the market districts for *Morphic* and *Spec* are not settled and solid: Instead, they are often causes of bugs and issues. The view also shows that many classes that act as entry points received frequent developer interactions, meaning that they likely have an unclear public interface.

Moreover, we can see that the *Morphic* packages are still frequently changed, showing that the community is constantly trying to fix the codebase. Finally, the high number of classes involved in the stack traces suggests that the code modification, together with the difficulty of understanding the API, is likely a cause of many programming errors. In particular, there are some *hotspots*, i.e., packages where classes are mostly colored in red only. These classes are involved in failures, but

they are rarely modified or involved in interaction data.

These theses are confirmed by the fact that the community is trying to replace the code of Morphic with a new, polished and easy-to-use replacement called *Bloc*, to address issues that we can spot in Figure 6.9. However, as the complexity of the picture suggests, replacing this code is not an easy task, and has been work-in-progress for more than a year now.

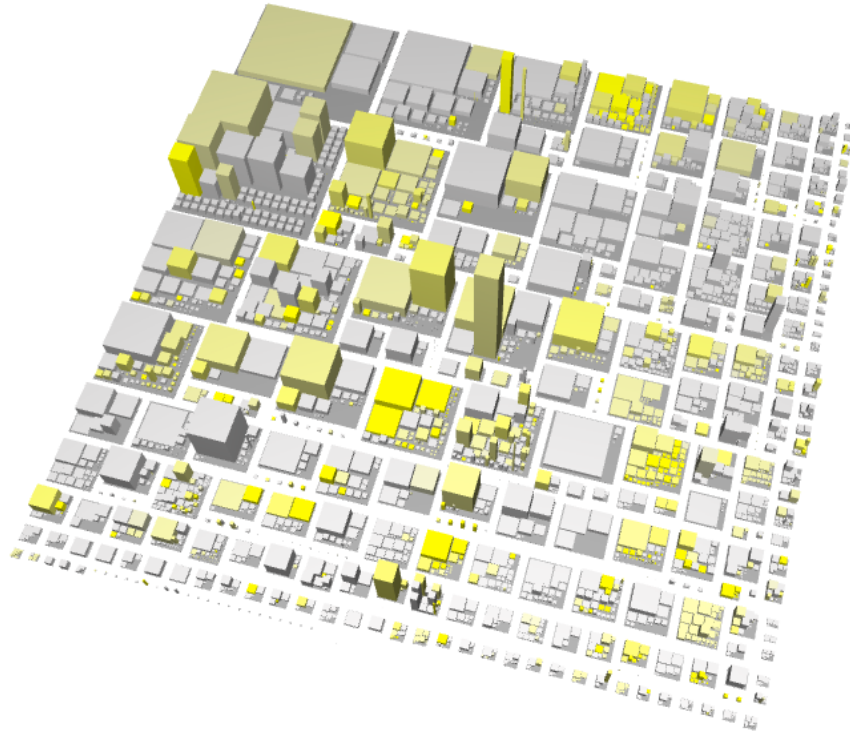


Figure 6.10. Changes in the Pharo system

6.4.3 New in Town

During the development of PHARO 4, many classes got updated and some new components were added. We want to analyze the progressive introduction of these changes, and how they impacted the system after the integration. We then use the sliders in Fig. 6.1.B to remove all the data sources, except for the changes. Figure 6.10 shows in full yellow the entities touched by a change in the last five months, and in blended yellow the changes in the previous five months. We can verify that there are elements that remained untouched, while some others were subject to intense development.

By moving the slider we can select a timespan to restrict the changes to a given moment of the story of the components and inspect the status of the system during time. We can notice that from a certain point on there was the appearance of packages related to the *GT-Tools*, a set of tools to improve the interaction with the objects in the system. By restricting the timespan to the beginning of January (*i.e.*, the first appearance of activities), to determine the moment of integration.

Figure 6.11 visualizes the blended city for the *GT-Tools* packages. Some classes are involved in all three data sources, *i.e.*, they are colored in dark brown. This can be explained by the fact that the first phases of integration usually require adaptation, refinement, and debugging, thus

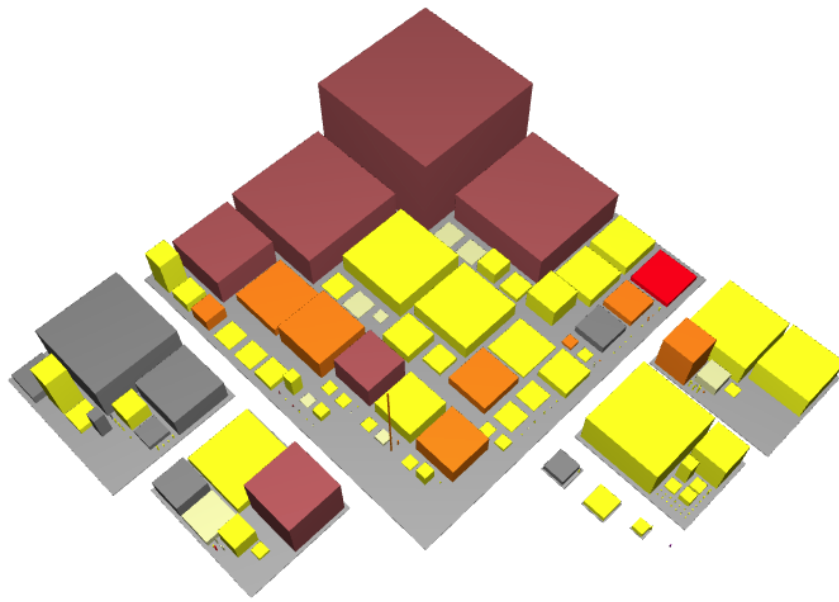


Figure 6.11. The changes of *GT-Tools* packages

generating (other than changes) frequent exceptions and developer interactions.

The other interesting observation that we can derive from the visualization is that the classes involved in user activities are also the biggest. This can be explained by considering that those classes act as main entry points to the package, a starting point for developers who want to use or inspect the code.

6.4.4 The Purple Buildings

A benefit of our blended city approach is that a data source can be removed to spot behaviors that are independent from it. Figure 6.12 shows the blended city for PHARO with only stack traces and developer interactions. While the stories we presented so far try to consider the code entities at a package level, the blended city without changes reveals the interesting role of some classes.

Scattered across the system, there are some big and medium classes colored of purple without apparent correlation with the color of its neighbors. By inspecting their names, we find examples like *DateAndTime*, *Float*, *Job*, *SmalltalkImage*, *Socket*, *SocketStream*, *SystemWindow*, *TestRunner*, and many others related to rendering of graphics, that we covered in the previous stories. These classes are not problematic *per se*, but represent an interesting area of the system that we could define as *Advanced APIs*. These classes appear in many stack traces and in many development interactions, an information that suggests that they occur near the source of the exceptions, when these exceptions are not directly generated from them. This context could signify that the user is trying to understand a class that has a name suggesting a behavior, but that she needs some further understanding to learn how to use the objects of the class by trying the various methods.

The use of this information could be used by the maintainer of the system to prioritize the areas of that could need more public documentation, to ease the learning process of those entities and their API.

Note that the same information, blended with the addition of code changes and applied to

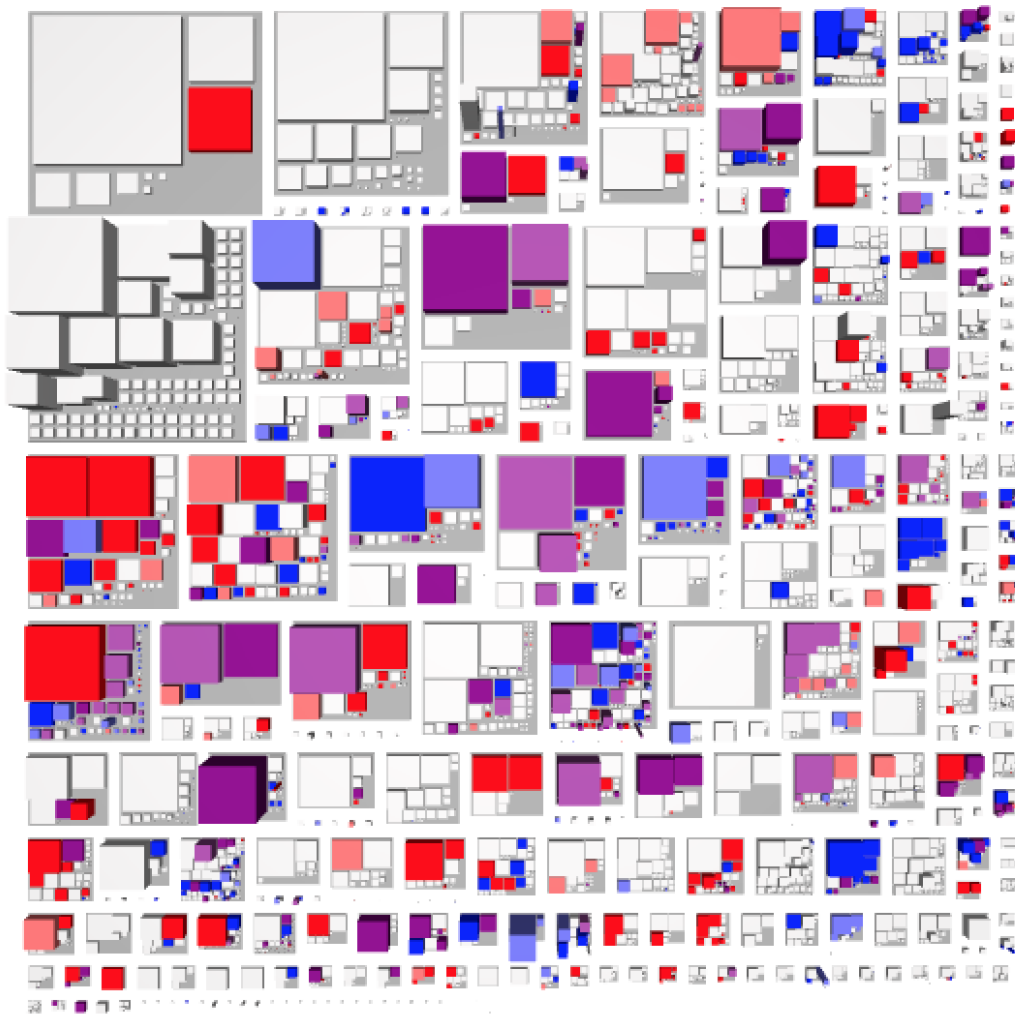


Figure 6.12. A view of the system highlighting stack traces and developer interactions only

classes that are not part of the core system, could signify that a developer is applying a *Test Driven Development* approach, by implementing incomplete methods and completing them whenever the system tries to execute a method that is not yet implemented, a common practice in the Smalltalk community [BDN⁺09].

6.5 Discussion

Developing our approach we became aware of many details that are hard to grasp in terms of how the users interact with the code entities. We believe that an important next step for this analysis would be to improve the system by providing updated information on fresh data mined in real-time.

Our visualization considers activity data, but maps this information on the static entities of the system. However, in Object Oriented Programming, the main focus lays on how these entities communicate among them, rather than how these objects are structured internally. We think that this approach could be further improved by also considering the interactions caused by the

messages sent as a result of the interactions.

Finally, from the user stories we saw how to retrieve information about the evolution of the system by looking at the way users interact with it. We think that a similar approach of combining data can be effectively put into use when analyzing old code, to understand and maintain legacy systems and support software archaeology [HT02].

6.6 Outline

Software visualization and analysis usually focus on giving a detailed representation of a single aspect of the examined entities. We presented an approach where we visualize data from three different data sources and contexts, blending them to produce multi-dimensional information about a system, its code and how developers interact with it. We considered a combination of system changes during the development phase of a system, the interaction data generated by users and the stack traces of the exceptions triggered during the daily usage of the platform.

We presented a tool that visualizes our blended information on a city view of PHARO, a dynamic, flexible and active programming ecosystem. We showed how our tool allows to select different timespans and weigh the diverse components, to enable a fine grained inspection of each entity during its recent evolution. We think that our approach has a real potential to be successfully applied in a development context to allow for multi-dimensional incremental and interactive analysis of a system, supporting a deeper understanding of the code entities by highlighting the synergies among its recorded activities, and the relations and interesting behaviors otherwise hidden or harder to detect. We illustrated four stories where we extract and analyze some real-world issues by looking at the blending of the data and identifying some existing problems, or finding suggestions for problems that could be addressed by the maintainers of the platform to improve the system. We believe that the knowledge highlighted by our approach can help in presenting and tackling existing problems and provide a deeper understanding of a system.

In the last chapters we saw how we can collect information about the errors on a system and exploit this data to support debugging, understand how a system is used, and navigate its evolution. In this chapter we presented a use case where we combined the data we collected together with additional data sources, to create a wide-ranging representation of the history of a system. By using our tool, a developer can improve her comprehension of a system and understand how a system is used by its users. We saw that we can collect and store data in an effective way and with a powerful representation. However, being able to collect any kind of data does not tell us *what* we should collect. In other words, we still have to understand *how* to improve the representation of a bug report. In the next chapter we investigate the data contained in several in issue tracking systems, and how it impacts the resolution of a defect. By looking at this data, we try to distill the model for a minimal bug report.

7

What Makes a Satisficing Bug Report?

To ensure quality of software systems, developers use bug reports to track defects. It is in the interest of users and developers that bug reports provide the necessary information to ease the fixing process. Past research found that users do not provide the information that developers deem *ideally useful* to fix a bug [ZPB⁺10]. This raises an interesting question: What is the *satisficing*¹ information to speed up the bug fixing process?

We conducted an observational study on the relation between provided report information and its lifetime, considering more than 650,000 reports from open-source systems using popular bug trackers. We distilled a meta-model for a *minimal* bug report, establishing a basic layer of core features. We found that few fields influence the resolution time and that customized fields have little impact on it. We performed a survey to investigate what users deem easy to provide in a bug report.

Structure of the Chapter

In Section 7.1 we discuss what we should expect from a bug report. Section 7.2 presents our research method and introduces our research questions, that we answer in Section 7.3. In Section 7.4 we discuss the meaning of our findings and how our work can be extended, while in Section 7.5 we summarize and conclude the chapter.

¹Satisficing is a neologism coined by Simon [Sim57, Sim01] combining the verbs *to satisfy* and *to suffice*, and it is used to describe a solution that is roughly satisfactory and meets some criteria of sufficiency and is better than an optimal solution that would be too complex or would imply too strong constraints.

7.1 Good Bug Reports vs. Real Bug Reports

When users file a bug report for a software project, their main hope is that developers will fix it quickly, to minimize its impact. But what information should they provide to make this happen? There is a stark mismatch between what developers perceive as *optimally* useful in this respect (*i.e.*, steps to reproduce, stack traces, and test cases) and what users are effectively able, or sometimes just willing to provide, when filing a bug report [ZPB⁺10].

Bug tracking systems and software projects should define a reasonable common ground for information to be provided in bug reports, so that it is not too demanding for users, yet provides enough information to developers. Nevertheless, considering what popular issue trackers and projects (*e.g.*, *Bugzilla*, *JIRA*, *FogBugz*, and the issue tracker provided with *GitHub*) demand from users, we see that it is quite diverse and specialized. In particular, each bug tracking system provides a core set of common fields, which are often complemented with additional fields. Such fields may reflect requirements for specific domains, or represent additional data customizable by project owners.

For example, the commercial issue tracker *JIRA* defines several fields that describe in detail aspects pertaining to the time management of issues, while the *GitHub* issue tracker provides a minimal (and sometimes criticized²) model that, together with the integration with the *Git* versioning system, conceives a bug report as a conversation among developers, fostering the philosophy of collaborative development proposed by *GitHub* [TBLJ13].

Overall, there is currently no consensus among software projects and creators of bug reporting systems on essential mandatory fields to be filled by users in each report, optional fields that give useful additional information, and free space for users willing to provide more detailed descriptions.

Our vision is to define the minimum set of information needed to describe a software defect, to clarify what should be required by each bug reporting system. In this chapter, we make a step in this direction: We investigate what makes a *satisficing* bug report. We move from defining a good or more precisely an *optimal* bug report and adopt a more pragmatic view on what users should provide.

We conduct our investigation in three steps: (1) We investigate what users and developers perceive as *difficult* in writing a report, by means of an online questionnaire; (2) we investigate the usage and evolution of issue tracker data, by means of a large-scale quantitative analysis of the status changes in submitted bug reports and the impact that customized fields have on the resolution of a defect; (3) we study which fields developers use to describe defects, by means of a further quantitative analysis on the lifetime of reports in relation to the evolution of report's state and its *completeness* in its core and customized fields.

Our results show that providing more specific fields in a report relates to the fixing time: In particular, the bug reports with longer descriptions tend to be solved quicker. While this might be intuitive, issue trackers still do not emphasize this aspect during the submission of a new bug report, putting the accent on the customization capabilities of the platform. At the same time, the project-specific bug report fields have little impact to the fixing time. This chapter provides insights on the current issue tracking practices and defines guidelines in building the foundations for a new model of an issue tracking system.

²<https://github.com/dear-github/dear-github>

The contributions presented in this chapter are:

- A survey that identifies the components of a bug report considered difficult to provide by users (Section 7.2.2).
- A dataset of 650,000 bug reports, collected from the issue trackers of BUGZILLA and JIRA (Section 7.2.3).
- The model for a minimal bug report, that puts the emphasis on the shared components of a bug report (Section 7.3)
- An analysis of the usage of the fields in an issue tracker, from active open source projects (Section 7.3).

Reflection

To understand the essential traits of bug reports, we analyze how the data included in bug reports influences their lifetime. We next analyze the features of a set of bug reporting systems, to distill a model of common/specific fields for their bug reports. This model serves as a basis for further empirical analysis, to determine how these commonalities and customizations influence the life of the reports.

7.2 Research Method

To determine what makes a satisficing bug report, we first need a way to rate the quality of a bug report, then we can conduct quantitative analysis to determine which features relate with higher quality. Measuring the quality of bug reports is hard to do in an automated and unbiased way. For this reason, researchers proposed different metrics to measure it [HW07], all with their limitations, but reasonable enough to be realistic. In this work, we decide to consider the *lifetime* of a bug report (*i.e.*, the time between the opening and resolution of a defect) as a viable proxy for its quality rating, as the time spent dealing fixing software defects is crucial in reducing the time the system contains a problem. Limitations of this proxy metric include the fact that the trivial bugs, or the non-issues, are the ones that require less time to fix, and that the severity can also have a not negligible impact on how quickly developers decide to fix a problem. Nevertheless, the information shared in the bug report has to be satisficing enough to let developers understand whether it is a trivial fix, an urgent matter, or something that can wait longer. For this reason, we find lifetime of a bug report a useful approximation in aggregate statistical analyses to provide a high-level view over bug repositories.

We investigate how users and developers use issue tracking systems and the impact that the provided information has on the lifetime of a bug report. According to Zimmermann *et al.* the information provided by submitters of bug reports can be partial or incorrect [ZPB⁺10]. To understand what is reasonable for a user to provide in a report, we conducted a survey asking developers what they think are the difficult elements to provide. We then focus on two of the main components that compose a bug report: (1) its state and (2) the core and optional attributes, to understand how the provided data is used.

7.2.1 Research Questions

When collecting information about software defects, it is important to know when the submitted data is reliable and accurate. Our goal is to investigate what users can easily provide and what is harder to obtain; we structure our investigation into the following question:

RQ1. What are the elements that are perceived as difficult to provide when reporting a defect?

To understand the relationship between what is described in a bug report and its lifetime, we have to consider the different kind of data that reports can provide. This is not trivial, because different platforms offer different fields to provide information, with different meaning and values. As a first step for our quantitative evaluation, we investigate how to define a meta-model to comprehensively describe information stored across different issue reporting systems:

RQ2. What is a comprehensive unified meta-model for describing data from different bug tracking systems?

After having defined the meta-model, we can quantitatively investigate several aspects of reports related to their lifetime and evolution. During development, a bug report changes its state, sometimes several times, ideally converging to a closed state. The changes in the state of a report are important to understand its evolution [DLP07]. We are interested in considering the evolution of the states and see whether the aggregate of these changes can provide knowledge on the inner logic of an issue tracker. This leads to the following question:

RQ3. What are the most frequent states and state transitions in bug reports?

Together with a state, a bug report comes with a set of attributes that describe the properties of a report. These attributes can also be defined by the users, to create project-specific customized fields. We investigate the completeness of core and custom fields with respect to the lifetime of a bug, considering the following research question:

RQ4. Does the completeness of standard and project-specific attributes in a bug report relate to its lifetime?

To answer our questions, we both run a survey (Section 7.2.2) and we collect, model, and analyze a large dataset of bug reports from open source projects (Section 7.2.3).

7.2.2 Online Questionnaire

Zimmerman *et al.* asked users and developers what they think are the useful elements in a bug report and how hard it is, in their opinion, to provide those elements [ZPB⁺10]. We proposed a similar questionnaire to the *Pharo* open source community to further understand what it is reasonable to expect from users submitting a bug report. The questionnaire is composed of two parts: (1) We collect demographic information inquiring about expertise with programming and with submitting, handling, and fixing bug reports; and (2) we collect information about

Table 7.1. Expertise of the participants of the survey (average)

Activity	Average
Experience with Object Oriented programming languages	1.5
Knowledge of Pharo	1.3
Have often bug reports assigned	-0.4
Often handle bug reports	0.6
Often participate in discussion in bug reports	0.3
Often submit bug reports	0.7

Table 7.2. Overview of the projects in the dataset

	Project	Issues				
		First	Last	Count	Age (days)	Frequency
Apache	Cassandra	Mar 7, 2009	Jul 8, 2015	9,723	2,314	5h 42m
	Hadoop	Jul 24, 2005	Jul 8, 2015	10,191	3,635	8h 33m
	Lucene	Oct 9, 2001	Jul 8, 2015	6,641	5,019	18h 8m
	Maven	Nov 20, 2002	Jul 23, 2015	4,663	4,628	23h 49m
	Mahout	Jan 30, 2008	Jun 25, 2015	1,752	2,702	37h 6m
	Pig	Nov 2, 2007	Jul 7, 2015	767	2,804	87h 44m
	Sorl	Jan 25, 2006	Jul 8, 2015	7,728	3,451	10h 43m
	Zookeeper	Jun 6, 2008	Jul 3, 2015	2,207	2,582	28h 4m
Mozilla	Air Mozilla	Apr 14, 2009	Jun 16, 2015	509	2,254	106h 16m
	Bugzilla	Apr 15, 1998	Jul 27, 2015	19,395	6,312	7h48m
	Core	Mar 28, 1997	Jul 17, 2015	292,358	6,684	33m
	Firefox	Jul 30, 1999	Jul 8, 2015	155,078	5,821	54m
	Firefox for Android	Sep 11, 2008	Jul 28, 2015	18,906	2,510	19m
	SeaMonkey	Nov 10, 1995	Jul 27, 2015	92,757	7,198	1h 51m
	Thunderbird	Jan 2, 2000	Jul 8, 2015	42,247	5,666	3h13m

respondents' perception of how difficult it is to provide different kinds of information when submitting a bug report. All the questions are formulated as statements (e.g., "It is easy to provide a description of the failure") and the respondents have to declare their agreement using a 5-level Likert-type scale. We map the results into an integer scale from -2 (i.e., "strongly disagree") to 2 (i.e., "strongly agree").

We advertised the survey through the development mailing list of Pharo and we received a total of 22 complete responses. Table 7.1 summarizes the respondents' expertise. The respondents are experienced with object-oriented programming and with the Pharo IDE. While they have experience in submitting and handling bug reports, their experience is lower in participating in discussions about bug reports and much lower in having reports assigned to them. For this reason, we deem the respondents' sample to be in line with the aim of our survey. In fact, we are especially interested in knowing the point of view of submitters of bug reports, rather than the view of the developers that "consume" these reports [ZPB⁺10].

7.2.3 Data Collection

To understand what users and developers collect and provide in bug reports, we mined the contents of the issue trackers of several software projects. To collect real development data for our study, we consider the Apache Foundation and the Mozilla Foundation: Both platforms contain a considerable number of popular and active open source projects, with years of development history. Moreover, both platforms host several projects tracked on public, dedicated bug trackers:

Mozilla uses BUGZILLA, Apache uses JIRA. They offer a public REST API to access their repositories in JSON format, allowing for a clean and reliable data collection.

We built a downloader and an importer to collect the data, serialize the contents of each report, and store the polished data in a PostgreSQL database. Table 7.2 describes our dataset.

The dataset contains more than 650,000 bug reports, 15% of which were still open during the data collection phase. Table 7.3 shows an aggregated summary of the dataset we collected. Each bug tracker has a different set of bug report states.

Table 7.3. Contents of the dataset

	Apache	Mozilla	Total
Open issues	7,545	91,336	98,881
Closed issues	36,127	529,914	566,041
Total Issues	43,672	621,250	664,922

Table 7.4 details them, for each tracker, with the counts of the bug reports for each state at the moment of the download.

Table 7.4. Different states of bug reports in Bugzilla and JIRA, with the count of the reports currently in each state and the total sum of all the times a bug report reached a state.

Tracker	State	Current	Total
JIRA	Closed	21,847	22,460
	Resolved	14,280	33,386
	Open	6,736	43,203
	Patch Available	471	18,944
	Reopened	235	3,042
	In Progress	84	2,175
	Awaiting Feedback	14	15
	Testing	4	86
	Ready to Commit	1	3
Bugzilla	RESOLVED	391,919	579,488
	VERIFIED	136,783	143,082
	NEW	65,816	353,264
	UNCONFIRMED	19,821	297,319
	ASSIGNED	3,701	129,057
	REOPENED	1,998	32,745
	CLOSED	1,212	1,537

7.2.4 Data Analysis Techniques

The large volume of data we collected enables us to explore the usage of issue trackers and to investigate the common practices of bug tracking. Understanding these aspects can help us to answer our questions and verify whether the usage of the properties of a tracker influences the life of a report.

To investigate our research questions, we adopt the following approach. To answer RQ3, we build a transition diagram of all the state changes for each issue tracker, to highlight the common patterns in the growth of a report, and we weight the nodes and edges of the diagram with the values from the dataset. To answer RQ4, we build a machine-learning-based prediction model to verify how completeness of fields of a bug report relates to its lifetime.

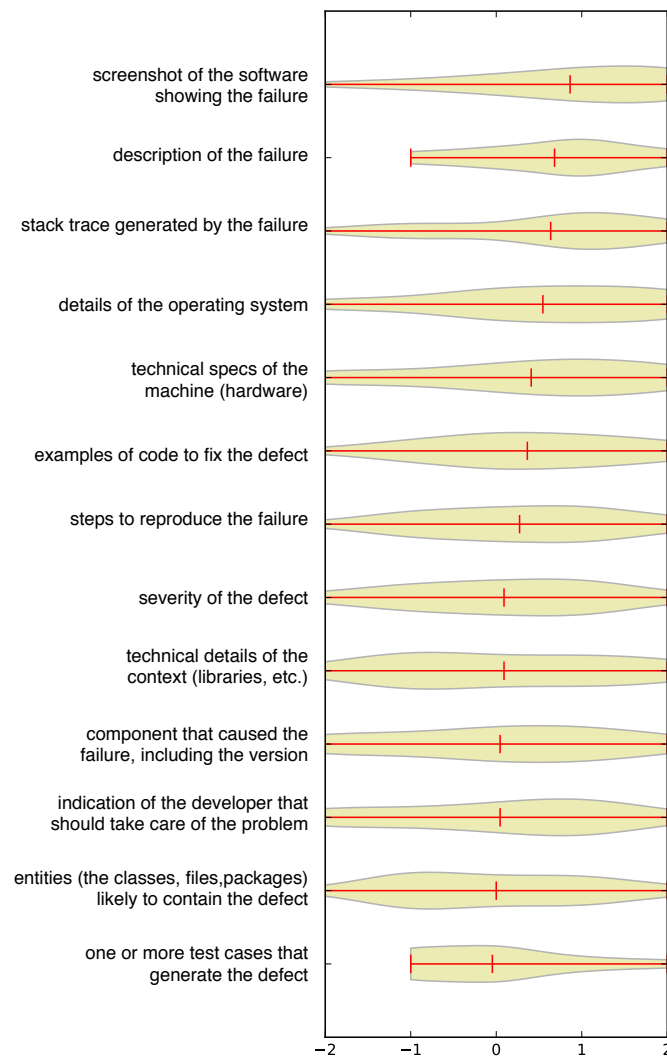


Figure 7.1. Survey results: The higher the values, the easier it is to provide the corresponding information, according to the respondents' perception.

7.3 Results

The data we collected allowed us to reply to the research questions in Section 7.2.1. We review them in order.

RQ1: What are the elements that are perceived as difficult to provide while reporting a defect?

We asked respondents how easy it is to provide 13 different elements in a report, using a 5-level Likert scale from -2 (“strongly disagree”) to 2 (“strongly agree”). Figure 7.1 shows a summary of their answers, sorted by increasing difficulty as reported by the respondents. The majority of the users does not find excessively hard to provide most of the elements. This is due to the fact that the Pharo community is composed of experienced programmers. Interestingly, finding the assignee is not considered excessively difficult: Again, this can relate to the community experience,

that has a strong core of well-known developers that work as hub when dealing with defects. The elements considered to be harder to provide are the entity (e.g., class, file) that likely contains the defect, the steps to reproduce the failure, and a test case showing the defect.

Conclusion

Figure 7.1 shows that some elements are perceived as more difficult to provide when submitting a bug report. There is a set of easier elements, like screenshots, descriptions of the failure, stack traces, and the details of the operating system and hardware. Those elements are useful in identifying the defect, but are less effective than other elements we identified to support its resolution.

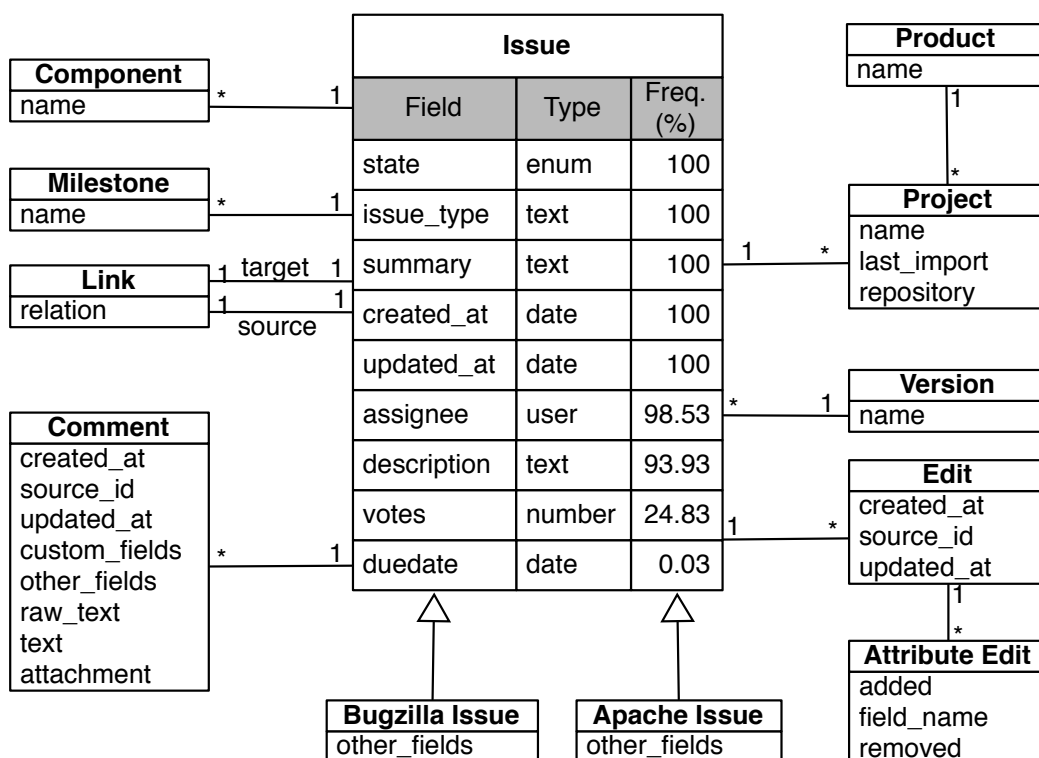


Figure 7.2. Conceptual diagram of the model of a new bug report

RQ2: What is a comprehensive unified meta-model for describing data from different bug tracking systems?

To devise a unified meta-model for the data we collected from the different issue trackers, we extract the model for each separate platform by reverse engineering the data and by using the documentation for the various trackers. We identify the entities that compose a bug report, the fields composing it, and the relation between the various entities. We intersect the list of each bug report and select the most common ones, to summarize the salient traits of a bug report.

Anatomy of a Report

Issue trackers are platform independent: They share a flexible common core structure to meet all the possible requirements of a software system's development process. A bug report is then built around a text description of an issue, where the user can specify the steps to reproduce the issue or include snippets of code that exemplify the context where the issue may happen.

The text description is complemented by additional metadata, used to improve the report and to track the evolution of the bug, and it can also contain attachments, like stack traces and patches. While the description of the issue and the possibility to attach files is common to all issue trackers, the metadata used to integrate the description differ in each platform. We can classify these attributes in three layers:

- *Common*: metadata in every report in each platform, *i.e.*, the core set of attributes that describes a bug report.
- *Platform specific*: metadata that are used throughout a single platform.
- *Project specific*: custom metadata set by the users, used in a single project.

The Model

From the list of entities in a tracker and their list of metadata, we built a model to access the data. Given our focus, we present a view of the model from the submitter's point of view. Figure 7.2 shows the conceptual diagram of the unified model for a typical bug tracking system with the frequencies of use for the common fields and trimmed of the post-report information.

- *Issue*: The main entity representing a bug report, with the text description and the metadata provided by the user.
- *Comment*: User-provided additional information on a report.
- *Edit*: A change in the existing report. It can group several changes.
- *AttributeEdit*: A change to a single element: It contains the modified attribute, the added, and removed text.
- *Link*: The relation (if any) to another report. A link maps the connection and defines the type of relation (*e.g.*, parent or duplicate).
- *Project*: The project the issue tracker refers to (*e.g.*, Firefox).
- *Product*: A single instance of an issue tracking platform (*e.g.*, Bugzilla or JIRA).
- *Component*: The area of the code affected by the defect.
- *Versions*: The software version(s) where the bug was observed.
- *Milestone*: The software version(s) targeted for a fix, for planning purposes.

There are additional attributes that are not present in every platform. To map these specific elements, there are entities that derive from `ISSUE` (*e.g.*, `BUGZILLA_ISSUE`).

These entities contain the fields `other_fields` and `custom_fields`. These are two *dictionary* fields that collect all the fields that are not represented in each model, in an unstructured fashion.

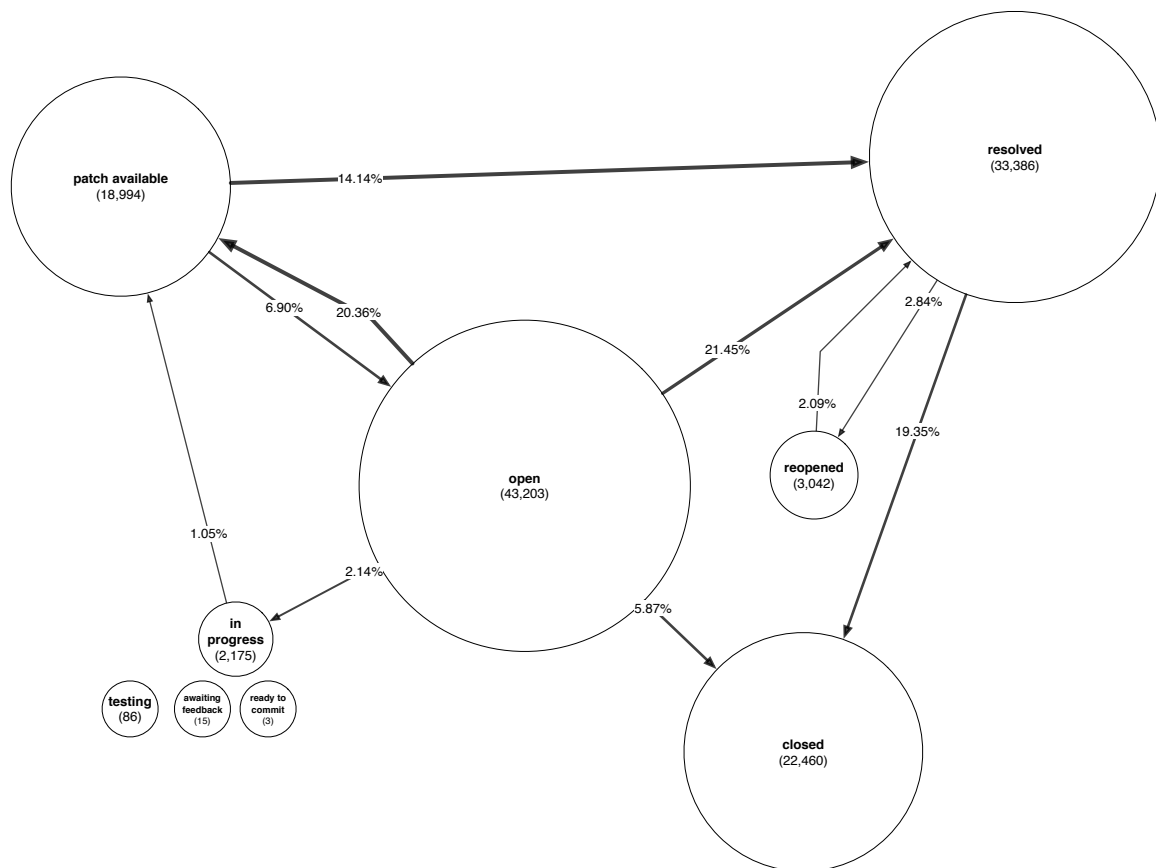


Figure 7.3. Transition graph of all the states in JIRA

The field `other_fields` contains the information from a specific bug tracker, shared in all the projects in that database (like the field `alias` in BUGZILLA). The field `custom_fields` contains non standard attributes that are customized by the maintainer of each project. For example, the attribute `cf_status_firefox41`, of the project FIREFOX in BUGZILLA. Some fields may seem redundant: For example, the field `updated_at` of ISSUE could be derived by the information contained in the EDITS; we tolerate a small degree of duplication of the data, in exchange for flexibility and completeness with different bug reporting systems.

RQ3: What are the recurrent states and transitions in reports?

We tracked the evolution of bug reports using the state attribute, which is an enumeration from a set of predefined states. Table 7.4 shows the states used in the two bug trackers we consider. Each platform proposes different conventions to map the state of a report. Often, different projects use the same states in a different context and a different distribution, *e.g.*, bug reports in JIRA converge toward the CLOSED state, while in BUGZILLA they converge toward a state called RESOLVED. We analyze the state changes by building a transition graph, with an approach similar to the one used by D'Ambros *et al.* [DLP07].

Figure 7.3 and Figure 7.4 show the transition diagrams for JIRA and BUGZILLA obtained by the collected data.

In the diagrams each node is a state, where the area grows with the number of reports that traverse that state, as presented in Table 7.4. Each arc between two states indicates a transition

from one state to another and its width represents the total number of transitions. The diagram excludes all edges that make up less than 1% of all the transitions. Given Figure 7.3 and Figure 7.4 we can classify the states in three groups:

- *Active states*: The first group contains the most active states (*i.e.*, touched by the majority of bug reports), that are often involved in loops between them.
- *Intermediate states*: These states (*e.g.*, TESTING, IN PROGRESS, REOPENED) indicate states where an action is taking place or expected (*e.g.*, a patch is waiting for review or the continuous integration server is running the tests).
- *Unused states*: Some states are rarely used: AWAITING FEEDBACK and READY TO COMMIT. They represent some corner cases that detail extremely specific aspects of the fixing activity. Their very low usage may hint at a little interest in tracking these aspects in this way.

The analysis highlights that some projects do adopt customized states to track the intermediate aspects of their projects' workflow, but they tend to be not used in practice.

Conclusion. The analysis on the usage of the states in Section 7.3 seems to suggest that:

- A simple model with a few states, as the one described by D'Ambros *et al.* [DLP07], satisfies the need of tracking the state of an issue;
- Adding customized values to describe additional specific and intermediate steps in the fixing process is not working to track a better evolution of the state of a report.

The latter aspect is strengthened by the fact that JIRA offers less states than BUGZILLA, but these additional states are rarely used in practice.

RQ4: Does the completeness of standard and project-specific attributes in a bug report relate to its lifetime?

To investigate the impact that the fields have on solving a defect, we considered the *lifetime* (defined as the time to the final fix) of the closed reports.

In addition to its standard set of attributes, each issue tracker we consider allows projects to define additional fields to customize the structure of a bug report. In our study, we group all the attributes in three *layers*:

- *Core Fields*: The fields that are common to all projects and all the issue trackers. They map the essential information to describe a software defect;
- *Tracker-Specific Fields*: The fields that are shared among all the projects in an issue tracker, but are not present in all the platforms;
- *Project-Specific Fields*: The fields that are customized by the user and appear only in a single project.

Each project in our dataset specifies its own set of custom fields. We also investigate whether these fields have a measurable impact on the lifetime of a bug report.

Table 7.5 shows a count of project-specific attributes in our dataset, including the average and maximum lifetimes of the corresponding bug reports, reported in days.

We now explore the relationship between the various attributes adopted by the different platforms and projects we considered and the effectiveness of a bug report, measured as its lifetime.

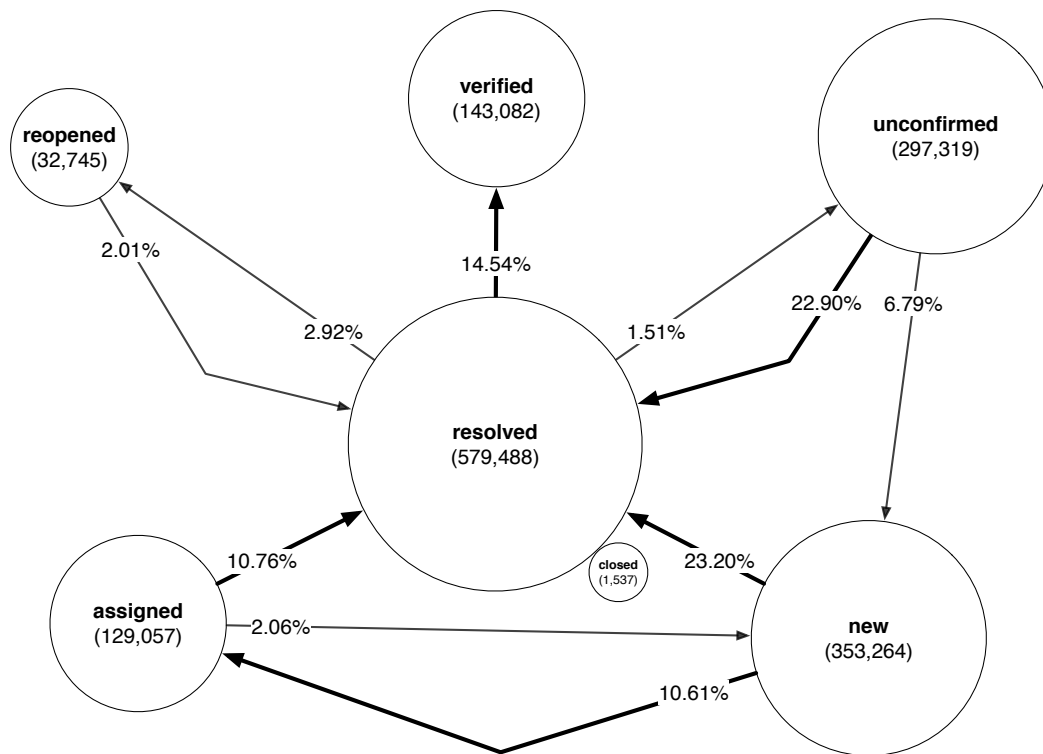


Figure 7.4. Transition diagram of all the states in Bugzilla

Preparing the data

To interact with the dataset, we created a *vector space model* to allow us to test statistical and machine learning approaches. Predicting the exact lifetime of a report would be unpractical and unnecessary: a timeframe for the resolution would provide a useful, human-understandable measure, while allowing more accurate predictions. To introduce such a degree of tolerance, we divide the reports into *buckets* according to their lifetime.

Using bucketing we can deal with discrete values and adopt a classification approach, as opposed to a regression to predict a continuous variable. We split the lifetime space into four buckets: less than one day, less than one week, less than one month, and more than one month. We chose these intervals because they reflect humane time periods and they describe increasing timespans, reflecting that the longer a bug report stays open, the less relevant its exact resolution time becomes. After bucketing the issues, we model each report as a vector of booleans (each field maps an attribute of the report and its value is 1 iff the user filled it) and associate it with its classification into a bucket of lifetime, which we can feed to different prediction algorithms.

Principal Component Analysis

To understand the relation between the completeness of a bug report and the fixing time of a defect, we want to inspect how much each field contributes to the lifetime of a bug report.

For this purpose, we use *Principal Component Analysis* (PCA) [AW10] to extract the variance between the different fields. PCA is a statistical procedure that aims to extract only the salient features from a data table. PCA transforms the existing data into variables called *principal components*, which are described as a linear combination of the existing features. The other features are then projected on the principal components.

Table 7.5. Number of custom fields per project

Project	# of fields	Avg. lifetime (d)	Max lifetime (d)
Air Mozilla	5	154	1,004
Bugzilla	5	343	5,650
Core	142	227	5,936
Firefox	112	235	5,314
Firefox for Android	89	76	2,176
SeaMonkey	90	278	5,437
Thunderbird	74	259	5,451
Cassandra	11	61	1,728
Hadoop	13	172	3,012
Lucene	13	182	3,787
Mahout	11	94	1,235
Maven	9	402	3,443
Pig	11	72	2,149
Sorl	7	148	2,858
Zookeeper	12	158	2,108

The components extracted by PCA represent the eigenvectors of the covariance matrix. Internally, PCA implements a *single value decomposition* to extract the scores of the factors. The number of components to extract is an open problem, but generally, when solving a correlation problem, PCA keeps the components that have an eigenvalue above the average.

We use PCA to determine which combination of fields carries the most information with respect to the lifetime of a defect, by observing which elements are selected to compose the principal components. To interpret the results and obtain a general set of fields that influence the lifetime of a bug report, we consider the core fields of the projects. This operation gives us the important fields that impact the lifetime of a bug report.

After running PCA, we obtain a set of new components that can be used to map the dataset. We are not interested in the new features per-se, but — since the features of the dataset are the fields of the bug reports — we investigate which original features were selected to describe the components.

We then inspect how the components are calculated, obtaining the following selected fields:

- *assignee_id*: the person the bug report is assigned to;
- *creator_id*: the person that submitted the bug report;
- *description*: the number of words in the description of a bug report;
- *duedate*: if the bug report has a due date;
- *reporter_id*: the person that initially reported the defect (can be different than the creator)
- *summary*: the number of words in the summary of the bug report.

These fields were extracted by the algorithm as the most relevant in impacting the lifetime of a bug report. Although they do not represent the whole amount of information that is needed to describe a software defect, the fact that they were selected by PCA indicates that their contribution in determining the lifetime of a report is significant. It follows that users and developers should take these elements into account when submitting a bug report and the issue tracker should ensure that these fields are exploited accordingly.

Predicting the Lifetime of a Defect

We studied the core fields that are the most relevant in impacting the lifetime. Now we investigate how the lifetime gets influenced by the different fields defined by each project. For such an analysis PCA is not suited, as the data is too sparse and the features would be discarded in the process. We therefore adopt a *machine learning* approach to estimate an approximate lifetime of a bug report given its “completeness,” *i.e.*, the number of completed fields when submitted.

We verify the impact on the prediction of the different levels of attributes using various machine learning algorithms on our model, by employing the `SCIKIT-LEARN` analysis tools [PVG⁺11]. In particular, we used *Naïve Bayes* [Mit97], *Decision Trees* [Mit97], *AdaBoost* [Bis06], and *Random Forest* [BS01] and validated our approach using *k*-fold cross-validation. We balance the training dataset to get homogeneous buckets containing 50,000 bug reports each, to prevent the different distribution of the sets to give a bias towards the biggest buckets [BPM04]. In this context, a random classifier would correctly classify 0.25 of the instances, so a classifier is better than random if it achieves a higher proportion.

Table 7.6 shows the prediction results: Each column represent a classifier, while each row represents each layer of attributes we add to the model.

Table 7.6. Prediction results: Proportion of bug reports classified in the correct time bucket, with increment over random classification (25% correctly classified bug reports).

	NB	DT	AdaBoost	RF
Common	0.27 (+0.02)	0.27 (+0.02)	0.27 (+0.02)	0.27 (+0.02)
+ words	0.28 (+0.03)	0.28 (+0.03)	0.29 (+0.04)	0.28 (+0.03)
+ tracker	0.36 (+0.11)	0.36 (+0.11)	0.42 (+0.17)	0.36 (+0.11)
+ project	0.37 (+0.12)	0.36 (+0.11)	0.42 (+0.17)	0.37 (+0.12)

In the first round we use the *common* attributes displayed in Figure 7.2; in the second, we add the number of words that compose the summary and the description of the report; in the third, we add the *tracker* features, *i.e.*, the attributes that appear in some issue trackers; in the last, we add the *project* features, *i.e.*, the non standard attributes that are customized by the users of the platform. We follow this order to increasingly add the more and more specific fields and evaluate the impact that the different customizations have on the overall model. We can see from Table 7.6 that the best results are achieved by AdaBoost [Bis06] using the tracker-specific fields, with an overall accuracy of 0.42. Differently from the shared and tracker-specific fields, the project-fields may vary over time. They are, in fact, constantly added: Firefox, for example, adds a new custom field specific for each release, which happens once every 6 weeks. This mutability can raise the question whether the contribution of these fields is diluted in such a long timespan. To mitigate this effect, we recompute our experiments on the subset of bug reports collected in the timeframe that starts exactly one year before the dataset collection. The new dataset is composed of 31,472 bug reports. Table 7.7 shows the results of our second batch of experiments.

Table 7.7. Prediction results for bug reports of last year.

	NB	DT	AdaBoost	RF
Common	0.27 (+0.02)	0.28 (+0.03)	0.28 (+0.03)	0.28 (+0.03)
+ words	0.30 (+0.05)	0.30 (+0.05)	0.33 (+0.08)	0.30 (+0.05)
+ tracker	0.34 (+0.09)	0.37 (+0.12)	0.46 (+0.21)	0.39 (+0.14)
+ project	0.35 (+0.10)	0.39 (+0.14)	0.46 (+0.21)	0.42 (+0.17)

Indeed, the results on the most recent dataset do not differ significantly from the results based

on much longer timespans.

7.4 Discussion

From our study using PCA, we observe that there exists a set of core elements of a bug report that impact and influence its lifetime. Comparing these result with the perceived difficulty presented in RQ1, we see that some of these elements, like a description of the problem, the screenshot or the stack trace, compose the description field that we saw impacting the resolution time. Another relevant element is the assignee of the report, but users find it hard to provide it. Interestingly, the elements that are the most useful in the resolution of a software defect are also harder to provide. From the experience studying the various issue trackers and their user interfaces, we believe that a user submitting a bug report should be offered a clean interface, that minimizes the amount of required information, highlights the most effective elements, and progressively requires the harder or less relevant ones. It is interesting to see how the recent issue tracker provided by GITHUB follows the same approach by providing a simple and clean interface. The AdaBoost machine learning model achieves the best results, yet it can only predict the lifetime of a limited number of bug reports. The increment over a random classifier prediction is particularly small for the *common* attributes. This can be explained by the terse nature of the core model. Moreover, the *tracker* fields improve prediction, showing a relation between more detailed bug reports and bug lifetime.

After calculating the lifetime of each bug report in the tracker, we compare data from the two considered platforms. By analyzing the average lifetime of the bug reports in each platform we note that they have a longer lifespan on BUGZILLA than on JIRA, with an average fixing time of 239 and 166 days, respectively. Even if the longer life of BUGZILLA projects may explain this phenomenon, we measure a gap between the lifetime of the reports in the two platforms (109 days for BUGZILLA and 93 days for JIRA), even when we restrict ourselves to consider reports submitted after 2009 (*i.e.*, when all the projects were active). There is an interesting, unexplained substantial difference in the way bug reports are processed in the two platforms. Studies can be designed and carried out to determine whether and how the bug reporting system itself leads to this behavior or there is a possibly unconscious self-selection of projects in using one or the other system. Concerning project-specific attributes, from the results of the test, depicted in Table 7.6 and Table 7.7, it emerges that they have the least weight in predicting the lifetime of a report. This suggests that they are not related to the fixing time. This may be a hint that these fields probably track collateral aspects of the evolution of a report that are not related to how quick a bug will be solved.

Last, we examine which fields impact the prediction the most: They are the number of words of the description and the summary, suggesting that an accurate description of the problem is important to engage the developers. The fields that connect the issue with other reports are also relevant, for example the dependent issues, as well as the fact that a bug report is already assigned at the time of submission.

7.4.1 Threats to Validity

Dealing with large amounts of data can pose some problems in creating an abstraction sufficiently broad to comprise all the aspects of the data, but still specific enough to capture its details. We spent a considerable amount of time dealing with the representation of the data, extracting its features and cleaning the unneeded parts. In particular, we carefully excluded from our prediction model all the fields that could yield a-posteriori information on the lifetime of a report. In a large

dataset it is hard, however, to guarantee the complete soundness of the whole corpus, that could contain hidden relations between some attributes.

There is the concern that the lifetime of a bug report, that we used as a measure of quality of a bug report, is not relevant for our task. However, this metric proved to be an interesting open problem in the field and it represents an interesting heuristic in determining the effectiveness of a report.

7.4.2 Next Steps

By the usage that we observed in our dataset, we gather that developers tend to use simple models in describing software defects. Even when provided with customization means, the additional information did not show a correlation with the lifetime of a report. Modern issue trackers like JIRA and BUGZILLA are complex interfaces over a set of tables in a relational database and the need for additional features over time makes those platforms grow over time, progressively turning them into inflexible *colossi*.

GitHub adopts the opposite approach, by providing a minimal structure of a bug report that is mostly a note attached to a commit or a piece of code. This interesting approach, however, lacks the descriptive power of the other two platforms. The need for a simpler model is hinted by the choices of the development team of BUGZILLA that on version 5.0, released in July 2015, proposes a simplified interface that asks the user for a summary and a description of the problem, polished of all the additional information.

We believe that the future of issue tracking systems lies in flexible structures that can dynamically adapt to different aspects of the development activity.

7.5 Outline

We conducted an investigation to identify the features that are relevant to obtain a *satisficing* bug report. In doing so, we provided the following contributions:

1. An overview of the perceived difficulty of submitting elements of a bug report for users;
2. A meta-model for bug reports that represents both the common and specific elements available in reports of different issue trackers;
3. A publicly available dataset of more than 650,000 bug reports, modeled according to our meta-model;
4. An analysis of the contents of the issue trackers, to identify features that are related to reports' lifecycle;
5. Evidence that increasing the number of fields provided when submitting a bug report has little relation on shortening the lifetime of a bug.

This chapter concludes our discussion about collecting information about bugs. We proposed different approaches to support developers in their bug fixing activity. We used automation to collect reliable data, we built visualizations to provide effective access to the data, and we discussed how we can improve the model of a bug report. We are still left, however, with the issue that bug fixing is intrinsically a boring and tedious activity. In the next chapter we try to mitigate this problem by exploring the field of *gamification*, investigating whether its use can help developers and communities in processing the large amount of unstructured information that populates issue tracking systems.



How to Gamify Software Engineering

All our efforts up to this moment were aimed at reducing the time spent dealing with bug reports by providing tools to access the data in a faster way than just plaintext. Providing faster and smarter tools solves the problem of reducing maintenance costs, but does little to improve the fundamental issue of engaging developers and contributors. Software development, like any prolonged and intellectually demanding activity, can negatively affect the motivation of developers. This is especially true in specific areas of software engineering, such as requirements engineering, test-driven development, bug reporting and fixing, where the creative aspects of programming fall short. The developers' engagement might progressively degrade, potentially impacting their work's quality.

Gamification, the use of game elements and game design techniques in non-game contexts, is hailed as a means to boost the motivation of people for a wide range of rote activities. Indeed, well-designed games deeply involve gamers in a positive loop of production, feedback, and reward, eliciting desirable feelings like happiness and collaboration.

The question we investigate is how the seemingly frivolous context of games and gamification can be ported to the technically challenging and sober domain of software engineering. Our investigation starts with a review of the state of the art of gamification, supported by a motivating scenario to expose how gamification elements can be integrated in software engineering. We provide a set of basic building blocks to apply gamification techniques, present a conceptual framework to do so, illustrated in two usage contexts, and critically discuss our findings.

Structure of the Chapter

In Section 8.1 we provide an in-depth introduction to gamification, while in Section 8.2 we present its principles. In Section 8.3 we discuss the applicability of gamification principles in a software engineering context, while in Section 8.4 we introduce a framework for applying gamification to a software project.

In Section 8.5 we propose some guidelines to evaluate the effectiveness of a gamified system, while in Section 8.6 we discuss the possible improvements of our work. Finally, in Section 8.7, we summarize and conclude the chapter.

8.1 The Rise of Gamification

Games have been a fundamental part of human civilization for thousands of years. In 440 BC Herodotus wrote about the Kingdom of Lydia in Asia Minor, where 3 millennia before his time the Lydians invented several games, such as the dice and the ball, to overcome an 18 year long famine. They would engage in games one day so entirely as not to feel any craving for food, and the next day to eat and abstain from games [HerBC]. While it is unclear whether the story is true, its moral truths reveal the essence of games, which is not escapism, but rather a purposeful and helpful activity to cope with the sometimes adverse or boring reality, which McGonigal goes even as far as to call it a “broken reality” [McG11].

Gamification is defined by Werbach and Hunter as “The use of game elements and game-design techniques in non-game contexts” [WH12]. The concept, not to be mistaken with Game Theory, was pioneered in the 1980s by Richard Bartle, the inventor of the first MUD (Multi-User Dungeon) game, who defined gamification as “turning something not a game into a game” [Bar03].

But, what is a game? According to McGonigal [McG11] all games share four defining traits: a *goal*, *rules*, a *feedback system*, and *voluntary participation*. The goal gives a sense of purpose. The rules unleash creativity and foster strategic thinking. The feedback system provides motivation. The voluntary participation makes the experience safe and pleasurable. Suits sums it up with “playing a game is the voluntary attempt to overcome unnecessary obstacles” [Sui05].

McGonigal provides several examples of contexts where the performance of subjects has been boosted through gamification [McG11]. The contexts range from house holding chores to physical exercise. While this may seem remote from the software engineering domain, Werbach and Hunter provide an illuminating example closer to our discipline: Microsoft’s testing team in charge of the multi-language aspect of Windows 7 invented the Language Quality Game, recruiting thousands of participants who reviewed over half a million dialog boxes, logging 6,700 bug reports, resulting in hundreds of fixes [WH12]. Another example is StackOverflow, a popular Q&A website, where asking and answering technical questions is rewarded with points and badges. There is evidence that this gamification mechanism is in part responsible for StackOverflow’s success [VFS13].

Lured by this success, one could be tempted to spread a gamification layer on any kind of software engineering activity. The questions we answer in this chapter is not only how such a thing can be done in a systematic way, but also whether and when this can lead to a desirable outcome, *i.e.*, higher motivation in developers and increased productivity. First, let us make a small digression in the realm of psychology. Behaviorism is an approach to psychology that combines elements of philosophy, methodology, and theory. Its tenet, expressed in the writings of Skinner [Ski78], is that psychology should concern itself with the observable behavior of people and animals, not with unobservable events that take place in their minds. Skinner was a firm believer of the idea that human free will is an illusion and that any human action is the result of the consequences of that same action: If the consequences are bad, there is a high chance that the action is not repeated; however if the consequences are good, the actions that led to it will be reinforced. Put simply, this is the approach “if you do this, you’ll get that”.

Gamification is related to behaviorism, as it is built on the concept of rewards (points, badges, etc.) for specific actions. However, contrary to the intuition of many, there is substantial evidence that behaviorism does not work: Kohn describes several experiments (for diverse contexts, such as losing weight, quitting smoking, etc.) which revealed that “token programs show behavior change only while contingent token reinforcement is being delivered. Removal of token reinforcement results in a return to baseline performance” [Koh93]. In essence: When the goodies stop, people go back to acting the way they did before. Other studies done in schools and work places even brought forth evidence that subjects who were rewarded for doing certain things were performing

poorer than subjects who did not receive rewards.

A popular, almost archetypal example of a supposed failure of gamification is the recent removal of the badges and points from the localized search and discovery app Foursquare¹. While Foursquare's gamification layer has probably been the cause of its initial growth and success, it was so emphasized that users ended up considering Foursquare just as a game, and not as a business application. FourSquare's CEO declared that gamification was phased out because of a perception problem of the real purpose of the app itself².

How can the success stories mentioned previously be explained, then? Is gamification a lost cause? We believe the answer is no, for a number of reasons.

First, gamification is only partially connected to behaviorism. A key point is that games represent *voluntary* efforts of the subjects to do something, while behaviorism was conceived as a way to (sometimes forcefully) influence the behavior.

Second, "simple" behaviorism is built on fairly tight feedback loops (do this and you get that), while well implemented gamification, such as the one in StackOverflow, has a much longer running time. Moreover, taking StackOverflow as an example, the presence of an Avatar who is being assigned rewards represents a key ingredient of successful gamification, as we will later see.

Third, and most important, the rewards that come out of successful gamification are not of a venal nature, but according to McGonigal they fall into four categories, that in conjunction represent "the foundation for optimal human experience [...], they're the most powerful motivations we have other than our basic human needs (food, safety, and sex)" [McG11]. These four categories are *satisfying work*, the *experience/hope of being successful*, a *social connection*, and a deeper *meaning*. We will discuss these aspects in the coming sections.

Summing it up, gamification is not about rewarding people with trinkets and tokens, it is about enriching their activities with "gameful" aspects. As this represents a fairly novel field, we have performed an in-depth investigation of the topic [Mas14], which we distill here into a systematic approach for the gamification of software engineering.

With this chapter we make the following contributions:

- An in-depth discussion of the principles, promises, and perils of gamification (Section 8.2).
- A conceptual framework with which one can gamify software engineering activities (Section 8.4).
- A set of reusable building blocks that serve as a foundation for our gamification framework (Section 8.4.1).
- An illustration, through several concrete examples and scenarios, of how our gamification framework can be used for the gamification of diverse software engineering activities (Section 8.4.2 and Section 8.4.3).
- A critical discussion about our findings and a roadmap for future work in this area (Section 8.7).

¹<https://foursquare.com>

²See <http://www.gamification.co/2013/03/15/the-removal-of-foursquare-gamification/>. Interestingly, the phasing out backfired, leading to a sensible reduction of the user base growth.

8.2 Games and Gamification

First, we discuss the principles of game design (Section 8.2.1) and gamification (Section 8.2.2). This will help us to understand how the obtained background can be leveraged to apply gamification in software engineering.

8.2.1 Why Do We Play Games

The idea that games can be adapted to positively influence tasks and activities in other domains is older than the term *gamification*, which only gained popularity in the recent years.

In 1980, Malone [Mal80] studied what makes computer games captivating to extract the features that can be used to support teaching. He considered two types of motivation: *extrinsic motivation*, triggered by means of a *reward*, and *intrinsic motivation*, triggered by the *satisfaction* of performing an action. Malone identified three main elements that influence the engagement in a game:

- (a) **Challenge** introduces uncertainty through hidden information, randomness, cognitive limitation of players, and variable difficulty. Self-contained and small goals are better than long term ones at sustaining performance and interest in an activity.
- (b) **Fantasy** refers to the mental images of things and situations out of the actual experience of the player. Malone discerns two types of fantasies: Extrinsic fantasies that depend weakly on the skills used in a game, and intrinsic fantasies that the player feels while using a particular skill in the game.
- (c) **Curiosity** arises from incomplete or contradictory knowledge. Sensory curiosity regards the attraction toward changes in the environment, while cognitive curiosity concerns the expectation of reaching a higher level of cognitive structures.

Building on Malone's work, Gee [Gee03] identified 36 learning principles crucial in video games and learning contexts, which we present in summarized form to identify the salient traits:

- **Learning Process:** the learner creates a mental model of the domain, and probes it to test her knowledge. The cycle of creating hypotheses and testing them is a crucial element of games and learning processes, and is present in humans already at the infancy stage.
- **Sources of Knowledge:** Learners acquire knowledge through several modalities including images, words, sounds, symbols, interactions, abstractions, *etc.* All this leads to an enrichment of the person playing.
- **Path to Competence:** Learners reach some achievements for which they receive intrinsic rewards, which also works as feedback. The learning process is performed slightly outside the comfort zone of the learner, so that the learner perceives the activity as “challenging but not unfeasible”. This connects to the concept of “Flow”, defined by Csikszentmihalyi [Csi90] as the mental state of operation in which a person performing an activity is fully immersed in a feeling of energized focus, full involvement, and enjoyment in the process of the activity.
- **Safe Environment:** The environment where learners operate is designed to keep low risks for each action, to allow exploring without facing serious consequences. In essence, dying in a game is not a bad thing, because it usually leads to learning. Moreover, the environment is disclosed gradually, to let the learner discover new parts of the subject domain, thus also feeding curiosity.

- **Learning Progress:** The process of learning begins with a simplified image of the real domain. What the apprentice learns in earlier steps leads to abstractions of the concept that she can use again in similar situations. Learners build their knowledge “bottom-up”, starting from basic skills, and making up hypotheses when a more complex case shows up, exploiting what they previously found. This feeds again curiosity and reinforces self-confidence.

In “Reality is Broken” [McG11] McGonigal suggests that the use of game elements can help making daily life and reality more interesting and engaging. She defines games as a combination of a goal, rules, feedback and voluntary participation; this makes games perfect environments to (im)prove our own capabilities. Pushing our skills to their limit, and then some more, means “producing hard work”, and provide a sense of achievement that is the exact opposite of depression. The immersion created from voluntary work can improve the mood for hours or days, “because when the source of positive emotion is yourself, it is renewable” [McG11]. McGonigal identifies four crucial elements that should be craved to achieve happiness: satisfying work, hope of being successful, social connection, and meaning. The use of games elicit positive participation towards a common interest, thus helping the development of communities. To improve the engagement in reality, she proposes a *sustainable engagement economy* built around intrinsic rewards. She defines *collaboration* as the sum of three types of concerted effort: cooperation (acting voluntarily toward a common goal), coordination (synchronising activities and resources), and co-creation (producing a result together).

Massively multi-player online games are illuminating embodiments of this concept: Even when competing for resources, the players constantly collaborate in the definition of the game world. McGonigal also proposes the idea that different affinity groups can collaborate and give value to the different qualities of each community, to create a *superstructure* that is able to solve problems that each single group would not be able to tackle. “A superstructure brings together two or more different communities that do not already work together. A superstructure is designed to help solve a big, complex problem that no single existing organization can solve alone. A superstructure harnesses the unique resources, skills, and activities of each of its subgroups. Everyone contributes something different, and together they create a solution” [McG11].

In essence, games enrich gamers and provoke positive emotion, which, if leveraged, help to structure experience and provide a powerful tool for inspiring participation and motivating hard work.

8.2.2 Gamification: Principles, Promises & Perils

Werbach and Hunter summarized the positive effects of a well designed gamification system as [WH12]: i) *Inherent relatedness*, i.e., being part of something bigger than ourselves; ii) *Rewards for doing good*, i.e., doing activities that are self-rewarding; iii) *Behaviour change*, i.e., getting people doing something that they did not use to do or they did not engage in, changing their habits.

According to Huizinga [Hui71], there is a virtual line that separates the game world from the real world. When a person is in this *magic circle*, the game rules matter over the rules of the real world. The purpose of gamification is to put the user in the magic circle, emphasizing the attitudes of voluntariness, learning, problem solving and exploration.

The most common form of feedback used in games is the *PLB Triad*, where *PLB* stands for Points-Badges-Leaderboards. Points, Badges, and Leaderboards are also widely used in gamification systems, because they appear to work moderately well as extrinsic motivators. To introduce a gamification layer on a real or virtual system, the first step is to understand whether there are the right assumptions to make it successful, which Werbach and Hunter [WH12] identified as:

- **Motivation:** Where to derive value from to encourage a certain behaviour?
- **Meaningful Choices:** Are the target activities sufficiently interesting?
- **Structure:** Can the desired behaviors be modeled through algorithms?
- **Potential Conflicts:** Does the game avoid tension with other motivational structures?

This schema must be considered in every phase of the gamification of a system, and used to verify the ideas that survive the review process. Depending on which game dynamics and techniques the game designers exploit, a gamified system takes a particular shape, often in the following forms:

- **Inducement Prizes:** They define a competitive game environment concretized into a contest to motivate efficiency, creativity, and flexibility. Prizes can assume several forms, where the PLB Triad is most frequent.
- **Collective Action:** This is a collaborative game context where people come together and accomplish a task. The main requirement is that the tasks can be split up to exploit “crowd sourcing”.
- **Virtual Economies:** Small, complete and structured economies that arise in virtual worlds. A well-known example comes from loyalty programs (like the ones of supermarket chains). The risk of crossing the line between virtual and real economies is often underestimated.

Adopting a gamification system means modifying the behavior of people and influencing their routine, which, as we have seen in the introduction might actually backfire. As such, it represents a delicate matter that may negatively impact well functioning parts of the system. Put simply: Adding a reward to a boring task may help to motivate the user, but will not turn it into an engaging activity.

Similarly, gamifying an already interesting activity may move the focus from the activity itself to the reward system. For example, Grant and Betts [GB13] carried out a study on the behavior of Stack Overflow users, and showed that many new users work intensively to acquire the easiest badges as quickly as possible, with increased user activity immediately before the awarding of a badge and a strong activity decrease in the period afterwards.

In general, gamification succeeds at the workplace only when it is well designed and the employees truly consent to it. Also, it was discovered that the most reliable predictor of consent to Gamification comes from the fact that employees are used to play games in their free time or not: A person used to gameplay has less difficulties in embracing the experience of the game, catching its rules, and engaging it [MR13].

Alfie Kohn raised serious concerns about the use of reward systems and virtual economies in education and the workplace [Koh93]. He argues that rewarding a certain behavior educates the user towards obtaining the specific reward, hiding the actual goal of the task. It is also possible that the users perceives the rewards as a controlling mechanism, thus generating repulsion instead of engagement. While this is a crucial aspect to consider, we believe it is still possible to successfully use gamification to improve a system. If we consider the StackOverflow example, the points obtained by answering a question are used to build a reputation system that is used through the platform to identify experts. At the same time, the points awarded are subject to a quality review from the users, who concur in the evolution and the quality of the platform. As such, if gamification is used to enrich existing interactions, rather than to force users to perform boring actions, it can be a valuable tool in growing a successful community.

The last set of perils we discuss are of a legal and moral nature, but not necessarily connected to the professional world. First, there is the question of *privacy*, as gamified systems and contexts can be misused to collect a vast amount of information about the players. Second, as stated by Bogost³ in an essay entitled “Exploitationware”, gamification might induce people to do things that are not really in their interest, *i.e.*, proposing to “replace real incentives with fictional ones. Real incentives come at a cost but provide value for both parties based on a relationship of trust. By contrast, pretended incentives reduce or eliminate costs, but in so doing they strip away both value and trust.” Third, gamified systems can be easily tweaked to implement deceptive marketing and advertising. Last, but not least, since players spend vast amounts of time and effort in building up their avatars/personas, they conceptually “own” them, which in turn might lead to unforeseen issues about property and ownership. This constitutes a new area of law, further complicated by its borderless nature.

Overall, gamification is a double-edged sword, but it is a rising phenomenon, which must be better understood to leverage its great potential.

8.3 Gamifying Software Engineering: (Not) An Easy Game?

We use a concrete running example to explain why gamifying software engineering areas is far from trivial. The running example is the one of bugs, in terms of reporting, tracking, and fixing them. Bug tracking systems (also known as issue trackers) are being used to store and manage bug reports since decades now. In short, developers and users use them to report new bugs they encountered, by providing data about the encountered bug, the situation in which it came up, etc. They report those bugs using web-based systems, such as Bugzilla and Jira. Developers then take up the bug report, try to understand it also by reconstructing the context, and then provide fixes and patches that hopefully correct the reported bug. Despite their many benefits, modern bug trackers are far from perfect, and suffer from redundant reports, incorrect data, and in general a poor quality of the bug reports, as pointed out by a number of researchers [BBA⁺09, ZPB⁺10]. Moreover, open source communities suffer from lack of participation by the users in this context. For example, at the time of writing, the Mozilla Firefox⁴ bug tracker contains ca. 20,000 open bug reports of which over 90% have not been assigned to anyone.

Enter gamification. How can it be used to ameliorate the situation, and can it be used to increase participation from the community as well as lead to higher quality reports?

A seemingly simple approach is to spread over bug trackers a layer of points and badges, and every week post leaderboards with the most active reporters and fixers. We believe that such an endeavour would at the beginning be successful, and probably there would be an increased participation of people. However, soon enough what gamers call “pointsification” would kick in, which is the focus of players on the rewards (the points, the badges) and not on the actual (technical and intellectual) achievement that led to the rewards. Put simply, soon enough there would be users who would start reporting non-existent bugs just to notch up their leaderboard ranking. This would lead to a situation, similar to the one observed in StackOverflow by Grant and Betts, where people would stop reporting/fixing certain bugs as soon as they obtain the corresponding achievement. The pun being intended, it would be “game over” for such a gamification approach.

The real goal of gamification has to be a different one, namely to improve the organization of the community, by helping and stimulating experts, by highlighting important bug reports, by making visible important achievements such as the closing of a difficult bug report, and in general

³See http://www.gamasutra.com/view/feature/134735/persuasive_games_exploitationware.php

⁴<https://bugzilla.mozilla.org/>

by fostering and maintaining motivation over a longer period of time.

We need an approach which supports what McGonigal [McG11] identified as the 4 key aspects of gamification: *Satisfying work*, the *experience/hope of being successful*, a *social connection*, and a deeper *meaning*. Next, we present our framework for the gamification of software engineering, which we distilled from a vast literature review [Mas14]. Due to space constraints we discuss and present only the salient underlying theory.

8.4 Software Engineering Gamification Framework

Our framework is an extension of Taje’s layered approach to game design⁵. Taje lists six layers, from lowest to highest, named Token, Properties, Dynamics, Goal, Meta, and Psycho. Game design elements can be mapped into the six layers and interact with each other by means of interactions. Our goal is not to describe Taje’s approach here, but to describe our framework. The reason is that Taje’s approach targets game design in general, while our framework targets gamification and in particular software engineering gamification. In essence, Taje’s layers are a subset of the components of our framework.

Activity	
Role & ID	
Description	
Building Blocks	
Analysis	
Rationale	
Emotional Goal	
Implementation	
Actors	
Dynamics	
Meta	
Hazards	
Testing	
Target	
Methodology	
Expected Results	
Actual Results	

Figure 8.1. Gamification Activity Template

Our framework is based on the concept of *Activity* (depicted in Figure 8.1), which is composed of *Analysis*, *Implementation*, and *Testing*. Each activity pertains to a specific user type (role), present in gamification systems, which can be *i) Observer*, who acts in read-only mode and does not contribute anything new, *ii) Writer*, who only interacts by modifying existing contents and *iii) Solver*, who accomplishes the objectives of the gamification system. People interacting with a gamification system dynamically switch between these roles.

An **Activity** consists of an *ID* formed by the initial letter of the role plus an incremental number (e.g., the first activity listed in *Writer* has the ID “W1”), a brief *description*, and a list of pertinent

⁵http://www.gamecareerguide.com/features/355/gameplay_deconstruction_elements_.php

gamification building blocks (which we describe later). Each activity is structured in the following way:

1. **Analysis:** Each activity within the gamification environment must come with an easily understandable *rationale* to connect to the global objectives of the environment, and the *emotional goal* we want to achieve in the people. Without this analysis step, a gamification effort risks turning into a random set of arbitrary decisions.
2. **Implementation:** To implement an activity the *actors* must be known and we need to understand which gamification *dynamics* they will be involved in, which represent the tactics to engage people in a specific activity. This is instantiated with game components we call *meta*, following Taje's nomenclature. Last, one must ponder the *hazards* that can arise from a game structure (algorithmic issues, misbehavior, hardware requirements, etc.).
3. **Testing:** The last component is devoted to testing the activities, where it must be understood which entities are the *target* of the testing, which *methodology* can be used to perform the testing, and lastly, which the *expected results* and the actual results, to facilitate an iterative approach to the development of a gamification environment.

This description of the framework is given from a conceptual point of view, obtained through several iterations and pilot tests we do not describe due to space constraints. Before we can provide concrete examples of how the framework is to be used, we need one last missing and fundamental piece: Each activity hinges on one or more **building blocks**, which also denote the particular categories of gamification it belongs to.

8.4.1 Gamification Building Blocks

The ten building blocks we present here have been identified during the construction of several software engineering gamification environments we have constructed, and which we briefly present in a later section. We do not claim the list of building blocks is exhaustive, but after constructing the aforementioned gamification environments we did not see other building blocks emerge from our efforts. The building blocks are denoted by a series of aspects recurrent in the literature: According to Werbach and Hunter players *go through a journey*, progressing through an environment, first by “on-boarding”, then by “scaffolding”, and later by achieving “mastery” [WH12]. Adopting Lazzaro's “keys to emotions” [Laz04], good emotions triggered by solving puzzles, accepting challenges, and designing strategies are elicited by *hard fun*. Moreover, the *people factor*, which stems from socializing and working with people and giving/receiving gratitude is fundamental in community-based gamification environments. Embracing Seligman's concept of *resource building* [SC00], it is beneficial to provide some form of avatar of the player which matures and grows as the gamification environment is being explored. This in turn is tied to the concept of “leveling up” described by McGonigal [McG11].

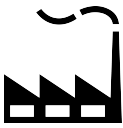
Before coming to the ten building blocks, one further consideration: As opposed to existing gamification environments, one which is tailored for software engineering must include the possibility of dynamically adapting itself. Since software systems are developed for very long periods of time, even decades, an environment should feature the possibility of removing existing rewards and adding new ones as the environment is being used.

Portal



When users cross the boundaries of the gamified platform, they register a profile and provide information that describes them to the virtual community. Despite being a trivial operation, it has a relevant feature: It is the very first action that users accomplish in entering the new world, and should be acknowledged with a reward. *Example:* Bob registers to the Bug Tracker and receives a “Welcome” badge.

Production



After registering, users must become immediately productive in the environment, because delays in starting using the platform may result in a drop of interest and cause users to quit. We split this block into three sub-blocks, according to the ways in which users have the possibility to produce content and receive rewards.

- **Symbiosis:** performing an activity that directly or indirectly helps someone else’s activity or state. Acting well in favour of others benefits both parties. *Example:* Bob provides useful comments to a bug being handled by someone else.
- **Narcissus⁶:** doing something to self-improve one’s position in the community. This action helps users to understand the structure and mechanisms of the community. *Example:* Bob provides his first bug fix.
- **Hive:** proposing an idea to improve the platform and community life. *Example:* Bob proposes to introduce a “Bug of the day” notification mechanism.

Bravery



In the production process, users may attempt hard tasks. The more skilled they become, the more confidently they will attempt to achieve bigger goals. Such bravery leads to important achievements and should be equally rewarded. *Example:* Bob fixes an old bug that made many people despair and is awarded by the community with an “Unstoppable” badge.

Scrum



In Rugby, Scrum is a way of restarting the game: Players bind together in order to make the other team collapse and take possession of the ball. The key is to rely on the strengths of everyone. Cooperating, collaborating, sharing useful tools, competing against, socializing with other community members is intrinsically motivating. The system should reward and promote teamwork. *Example:* Bob spends time assigning bug reports to users that he knows to be expert in the area, or tagging easy bugs for newbies.

Chameleon



While gaining skills and experience, the user may do something unique, spectacular, or never tried before. The environment should react by introducing a new achievement and release an ad-hoc reward, which becomes part of the gamification library of the system and achievable by other users. Conversely, if a specific reward has never been reached by any user for a long time, the reason might be its impracticability; the system should dynamically remove such an achievement from the library. We affiliate such a dynamism with the ability of chameleons to change their own skin colour according to the surrounding environment. *Example:* Bob closes five bug reports with a single fix. The system administrators create a special “Epic” badge, and add it to the possible badges users can achieve.

Thunderbolt



When users become experts, with many obtained rewards, they might fall into a state of boredom. The result is decreased motivation and productivity. To awake them from inactivity, the system should hit them like a thunderbolt with an announcement and direct them toward a new challenge, such as a one-week long quest where contestants can be awarded custom prizes. This should spur many users to participate. *Example:* Bob has not participated in any bug fixing activity for the last month. He and similar users are notified about a complex bug and a bounty for fixing that bug.

Phasing



Users may perform actions in the virtual world that, in reality, produce a permanent impact on the surrounding. Phasing suggests to mutate the environment according to the progression of each user’s expertise. Two users, at different stages of their progression see different representative phases of the same scenario and can interact with it in different ways. *Example:* Bob tags bugs that are old and inactive, but still interesting. The administrator then creates a new section highlighting such bugs, and acknowledges the contribution of Bob.

Beautification



Appearance, even if only virtual, is important to many. The users’ avatars change appearance over time and become more appealing as they progress in the environment. In the opposite case of inactivity, the appearance of the avatars starts to slowly degrade. *Example:* As Bob becomes an expert bug fixer, his avatar (for example depicted as a warrior) is decorated with better clothes and weapons. After a period of inactivity due to his (real) holidays, Bob’s avatar is depicted out of shape and with a broken sword.

Champagne



Since achievements inside the magic world are important to users, they want to celebrate their success not only within the virtual world, but also in the real one. *Example:* Bob is looking for a new job and on his curriculum puts a link to his profile in the bug tracker, as proof of his expertise⁷.

Ascension



A game usually has an end. It is intrinsically rewarding and fulfilling to see the words *The end* on a screen, even though the actual satisfaction comes by what was done along the way. This building block does not come with a reward, as otherwise inactivity might set in. If users collected vast amount of rewards and participated in the community, they should be rewarded in the real world as well. *Example:* Bob has been a productive bug hunter for many years, and is rewarded by the environment admins by being invited to become also an admin.

Putting everything together. In the following we provide two concrete examples of gamification environments we have been developing.

8.4.2 Example I: The Myth and De-Bug

The objective is to develop a gamification system for a bug tracking system. Fixing a bug is like struggling against a monster that threatens a village. This image inspired the overall theme of ancient Greece, full of heroes, gods, legends, and epic battles with mythological beasts. We set the following goals for a gamification system in bug reporting:

- (1) *improve the quality of bug reports:* we want to stimulate users to include meaningful information. Zimmermann *et al.* showed that some elements are crucial to ease the fixing process, such as stack traces [ZPB⁺10].
- (2) *stimulate the participation of the community:* we want to create a friendly environment for newbies and with engaging activities for experts.
- (3) *ease the fixing process:* we want to reduce the time spent dealing with cumbersome information, to allow developers to spend their time in fixing the defects. We want to encourage users to deal with unsorted data in the tracker, like assigning bug reports to the appropriate user, closing duplicate reports, or highlighting important bugs.

The Myth and De-bug reflects the journey of a player that begins with the on-boarding phase, continues with some scaffolding, and terminates with mastery. We produced a large set of activities, such as the one in Figure 8.2.

We present a selected list of the activities that we designed: the goal is to clarify that creating activities is a lengthy process which must be done in an iterative way.



As the player signs up for the game, she enters the magic world of ancient Greece and receives her first reward, the **Newbie badge** and a small amount of drachmas (the ancient Greek currency), with which the player can buy her avatar some equipment. The game awards different amounts of Drachmas according to the difficulty of the accomplished task. This first operation is trivially easy (just registering and give some personal information for the user profile), but it has a special feature: It is the first action that the user does to get into the platform and the first active contact with the community. Moreover, receiving immediately an unexpected reward works as a bait for the new player who is motivated to add another prize to her collection as soon as possible.

Activity	
Role & ID	S4
Description	Re-opening a closed bug
Building Blocks	Bravery, Scrum, Champagne
Analysis	
Rationale	A bug that was not solved properly has been re-opened because it needs additional work
Emotional Goal	Re-opening a closed bug is a brave feat. If someone closed thought to have solved it and did not, probably that bug is hard.
Implementation	
Actors	The system and the community
Dynamics	Recognizing that a bug is still unsolved is already an important point that needs a reward. Additional rewards come from being able to actually solve it.
Meta	<p>As the user re-opens the bug, she earns 10 Drachmas and the event is published on the homepage of the platform. If the user also expresses the interest in trying to solve the bug:</p> <ol style="list-style-type: none"> 1. The system sends the bug report to 10 expert users (having more than 150 accumulated Drachmas) and asks to estimate a time in days needed to solve it (considering not more than 3-4 hours of work per day). 2. The least and highest returned values are removed, and the system computes the average of the other eight values. This averaged value is communicated to the user so that he knows that the community expects her to solve the bug in that number of days. 3. When she solves the bug, her "Heracles" badge assumes a colour computed as the mathematical function of how many days totally other past programmers worked on that and how long it has been closed. Moreover, if she managed to solve it within the estimated time, she earns 50 Drachmas. If she employs 1 week more, she earns 40 Drachmas, and so on. 4. The user can share her success on her favourite social network. <p>After the 5th week beyond the estimated time, the user gets no Drachmas and her badge remains white. At that time, she must declare to the community whether she gives up, or wants to assign the bug to another user, or wants to ask an extension of the available time. If she decides for the last option, she needs to publish on the bug report exactly what she did and what she thinks should be still done to solve it. The evaluation with the 10 expert users is done again and the user now has that time to solve the bug. The user can ask consecutively an extension up to 3 times, then she must give up or assign it to another programmer.</p>
Hazards	The bug was solved and should not have been re-opened in the first place. An expert user should check first whether re-opening is the right thing to do.
Testing	
Target	Average Time period to fix re-opened bug before gamification, and after the introduction of the gamification layer.
Methodology	Compute the respective averages times and see whether the time decreases after gamification.
Expected Results	Average time to fix re-opened bugs has decreased.
Actual Results	to be determined

Figure 8.2. Concrete Gamification Activity



The second unexpected reward is quite easy to acquire too: Becoming conscious of the rules holding in the world of Ancient Greece, the player earns the **Briefed badge**. She does so by going through a tutorial which explains the bug tracker and the rules of the game.

The system assists the player along the whole path to mastery: It directly furnishes to the user practical tasks that she can afford with her current skills. While writing a bug report, the system supports the player by using mandatory **box fields** asking for specific information or suggesting where to look to find it. It helps reporters to not forget essential information and provides some scaffolding to boost a player to mastery. Motivation is a precious good that some techniques are able to elicit, but at the same time can be shut down easily. A single apparently insignificant demonstration of disapproval from some other member of the community can hurt a newbie. *The Myth and De-Bug* avoids such an effect by impeding questions and answers with scores smaller than 0 and avoiding the so-called "Dislike" system.



A point of strength of this gamification layer is that everyone, from the new user up to administrator, has the chance to propose improvements for the environment. The player whose proposal for an extension of the environment has been accepted by the

community, gains the **Phidias badge**.



An open problem of gamification communities (for example Stack Overflow), is keeping the motivation of users high or to recover it when it naturally decreases [GB13]. We designed badges that level up proportionally with the amount of work performed, e.g., **Tomb Raider** is a badge achievable when a developer explores old posted reports, finds something interesting, and sets the status of the report back to active.



Heracles⁸ is a badge of the same nature of Tomb Raider, but is awarded for closing re-opened bugs.

Our environment also deals with the issue of *balance*. If a gamification layer is too linear in terms of dispensing mere (and in a way meaningless) points, the danger of pointsification comes up: Users start to hunt for points by performing meaningless and contradictory actions, such as re-opening bugs that do not need to be reopened. The building block *Scrum* is crucial in this case, which means to rely on the community, for example by setting time limits for specific activities.

Also, our environment does not make large use of leaderboards because they are gamification elements that, in a number of cases, may demotivate players. We designed the leaderboard “Twenty Top Hoplitest of The Week” by relying on the fact that having a considerably high work rate is an occasional ability. Since a developer cannot be constantly productive, the leaderboard thus becomes dynamic.



When users are on the leaderboards for an extended period of time, they gain the **Achilles**⁹ badge and an amount of bonus Drachmas to refurbish the avatar.

To foster epicness, one of the traits identified by McGonigal as instrumental to gamification, our environment provides a number of places where players can acknowledge the feats of other players. This happens for instance when a developer closes a difficult bug, or the community reaches a landmark (e.g., closing the 1000th bug) collaborating as a team. The environment also features specific leaderboards, in the form of halls of fame, where important contributors are acknowledged or where productive former newbies are entered into the category “The New Greek Legends”.

The Myth and De-bug is an instance of inducement prizes: Its goals are efficiency, development of creativity, and stimulating collaboration among the community even while competing. It is also “cheap” because it only involves virtual goods, and pays a deep attention to balancing issues. We just described a possible instantiation of this gamification system. We could take exactly the same framework, substitute the name of the badges and imprint the game toward modern heroes (Spider Man, Batman, Superman, etc). They are just fancy names, and we can use the fantasy we like to shape the same gamification dynamics.

8.4.3 Example II: The Empire of Gemstones

The first example was developed in the context of a novel bug tracker we are implementing [DSL14]. We also devised a number of other software engineering gamification environments, which led to the distillation of the building blocks discussed previously. We now present another case study of a gamification layer for a software engineering context: Modern Code Reviews. Due

⁸Heracles was the greatest hero in Greek mythology. He had incredible courage, physical strength and ingenuity. Among the many ventures attributed to him, he defeated the Hydra monster, a sea serpent with nine heads. Every time someone cut one away, it grew anew. This is a conceptual parallelism with what happens with closed bugs that are reopened.

⁹We choose for this badge the figure of Achilles, the king of the Myrmidons, son of Zeus and Thetis. The parallelism with the badge comes from the fact that his most common epithet in Homeric works is “swift-footed” because Achilles was known to be very fast at running.

to space limitations we do not present the solution at the same level of detail as the previous example, but focus here mostly on a concept that was only sketched in the previous section: *leveling up*.

Code reviews are a software engineering practice that consists in manually reviewing source code written by other people, to verify and improve the quality of the code. While the effectiveness of this method has been proven during the years [SS08], this practice is often considered expensive, cumbersome, and, as such, difficult to adopt. Bacchelli and Bird proposed *Modern Code Reviews* [BB13], a code review approach that is informal, tool based, and performed on a frequent basis. They developed *CodeFlow*, a tool where the user can annotate the source code and interact with other users with a chat. A developer that wants to propose his code for review has to create a package with the changes, write a brief description and submit it to the *CodeFlow* service.

The area of code reviews still has many open questions, but the *CodeFlow* platform represents the ideal environment to develop a gamification layer to stimulate the amount of motivation necessary to turn code reviews into a habit.

In the context of a code review tool that we were building in the research group, we designed a gamification environment named *The Empire of Gemstones*, to exploit the parallelism between collecting gemstones and improving the quality of the code. We employ gems as a virtual currency to reward positive feedback while using proposed solutions. The number and the type of gems compose a reputation system based on noble titles, used by the system to rank users, which also facilitates the finding of experts in specific areas.

It has been shown that teams use code reviews for the following purposes: (1) finding defects in the code; (2) improving the code; (3) finding better implementations; (4) transferring knowledge in the group; (5) increasing the team awareness and transparency; and (6) sharing code ownership [BB13, BBZJ14]. Given the strong implicit collaborative nature of code review tools, we pose a strong accent on blocks that expect interaction with other users, like *Scrum* and *Champagne*. However, also the motivation of single users can be catalyzed, for example by rewarding quality code, thus suggesting the use of *Bravery* and *Thunderbolt* blocks.

In parallel with a set of badges devised with a similar procedure to the one used to build *The Myth and De-Bug*, we introduce a “leveling” mechanism to provide users with a feeling of progression and growth while reviewing the code: Leveling is one of the main drivers of gamification systems, as it fosters positive competition among the players.

By reviewing other’s code, a user gets a gem. The kind of gem depends on the number, size and difficulty of the reviews. Each gem has a different value according to its rarity, as we see in Table 8.1.

Table 8.1. Points acquired per 1 gemstone.

Gemstone	Points
Emerald	10
Sapphire	9
Tanzanite	8
Aquamarine	7
Ruby	6
Jade	5
Citrine	4
Topaz	3
Amethyst	2
Quartz	1

For example, a reviewer may check some code that includes changes for fifty lines of code over five different files, for which she receives a Jade. Another user reviews three small changes,

Table 8.2. Required number and type of gemstones to obtain noble title.

Family	Noble Titles						
	Prince	Duke	Marquis	Count	Viscount	Baron	Knight
Emerald	27	21	16	12	9	6	3
Sapphire	34	27	21	16	12	8	4
Tanzanite	41	33	26	20	15	10	5
Aquamarine	48	39	31	24	18	12	6
Ruby	55	45	36	28	21	14	7
Jade	62	51	41	32	24	16	8
Citrine	69	57	46	36	27	18	9
Topaz	76	63	51	40	30	20	10
Amethyst	83	69	56	44	33	22	11
Quartz	90	75	61	48	36	24	12

each one involving only one file, and she receives three Quartz. Reviews that spot bugs, that propose a better implementation of the reviewed code (goals 1, 2 and 3) get higher value gems, but since reviewing code also means knowledge transfer (goal 4 and 5), users get a reward even if the review causes no changes.

Submitting code for review implies willingness to collaborate and accept critics, an not being protective of her code (goal 6). We decided however not to assign gems depending on the outcome of the review to the submitter, to avoid pointsification and because that would suggest an idea of code reviews begin judgmental, which in the long run would discourage a user from submitting his code for review. A submitter can however still receive badges for particular behaviors, like the **Collector** badge for users that submit regularly their code, or the **Houskeeper** for users that submit large numbers of reviews in a short time.

By collecting gems, a user can grow his estate and obtain noble titles which reflect the expertise level in the community, as depicted in Table 8.2. For example, a new user in the team is reviewing many small changes to understand the project he is working on. He then collects many Quartz, slowly being promoted to Knight after 12 reviews, and Baron after 24.

The avatar of the player in this environment is then also depicted in a gameful way, such as a house which gets more beautiful as the player obtains more gems. In the code review tool, these avatars could then be shown to other reviewers when they log into the tool.

As we anticipated, the leveling mechanism is useful in stimulating positive competition among team members. Given the context of code reviews, which by definition happen inside the same team, company or community, we believe that the level system is particularly effective in leveraging the interpersonal bonds and endorse motivation in improving the quality of the code.

8.5 Evaluating Gamified Systems

Once a system is gamified, we need to be able to measure the impact of the gamification, and how much it contributed to reach the *business objectives*. It is crucial not to confuse business objectives with game objectives: with the coexistence of “serious” and “fun” layers, it is easy to exchange the goals of the two aspects, thus misjudging the effects.

In building our framework, we included a testing section, whose purpose is to design, together with an activity, the conditions to establish how successful the game elements applied to each single activity are. But, a system is more than the mere sum of all its parts: As such, testing all the single elements does not imply the success of the whole gamification system, exactly as in software development we need integration tests.

We propose five methods to assess the general performance of the gamification on top of a software engineering context: *success metrics*, *analytics*, *conflicts*, *jen ratio* and *survey*. The first three focus on technical aspects to consider the business objectives, while the last two consider emotional aspects. Due to their subjective nature, we cannot get a precise measure of emotional response of the users and compare the obtained values in a consistent way. The relative metrics have then to consider the imprecise nature of the data they deal with.

Success Metrics

The first approach is to define a set of goals at design time, and verify them after the system has been in production for a while. We recommend to make a list of the goals of the gamification system, define success metrics (number of new users in the last month, average activity increase per user, *etc.*) tailored to specific activities and verifiable with usage data. A long enough timeframe must be used to perceive a noticeable change: People's habits take a while to deal with novelties. A significative amount of data must be collected before and after the introduction of the gamification layer to enable before/after testing.

Analytics

A useful metric is represented by the measure of users interacting with the environment: *Daily Active Users (DAU)* is the number of unique users that interact with the software tool during a day, while *Monthly Active Users (MAU)* is the average number of unique users that interacted with the software tool in the previous 30 days. By computing the ratio $\frac{DAU}{MAU}$ we have the trend of usage of the software tool in a given moment. The result of such a ratio goes from 0 to 1: It is close to 1 when the tool is engaging, and it is close to 0 when its popularity is decreasing. $\frac{DAU}{MAU}$ is a relevant parameter to keep under observation because, if it increases the number of active users is growing; if it decreases they are decreasing.

Conflicts

Some gamification elements can create conflicts with existing elements on the system. Listing and prioritizing the conflicts, also by listening to the users through forums and mailing list, is helpful. If a conflict persists, the involved gamification elements should be pulled out of the environment, as user dissatisfaction can be harmful to the whole community.

Jen Ratio

Establish two sets of interactions in the user community: *positive interactions* (e.g., virtual gifts, acknowledgements), and *negative interactions* (e.g., misbehaviours, rude comments). Compute the *Jen Ratio*: total positive interactions among users over the total negative interactions, in a given period of time and context. The outcome is between 0 and 1. The jen ratio assesses how positive the attitude of the users is: the closer to 1, the better the social well-being of the community.

Survey

Selected users, of all expertise levels, should be periodically surveyed, where key questions should not only regard technical aspects, but also emotional aspects.

Beyond the use of these metrics, it is important to perform an evaluation on the effective gain of the system, to quantify how the use of gamification impacted the activity of the users and if it brought actual benefits. For example, in a bug tracking system we can measure the number of bug reports opened and closed every day, the average duration of a bug report and the number of bug reports that a user examines. However, it is clear that such metrics are domain specific, and have to be calibrated for each different gamification context.

8.6 Discussion

In this section we propose our reflections on the lesson we learned developing our work. We then discuss how we think our work can be improved.

8.6.1 Reflections

The examples in Section 8.4 hint at a fact that should not be disregarded about gamification: To create such environments is a far from trivial endeavor. The reasoning that goes into creating thematic environments, the way that leveling is handled, how and when awards and badges should be assigned, is a strongly iterative process. One might be tempted to bypass such a labor-intensive work by using the simplest solution, which is to award points and to base the leveling on such points. However, apart from the danger of pointsification, there is another risk, which we define as “stalling”: If the gamification layer is not constantly revisited, maintained, and evolved, it risks to quickly become obsolete, and therefore will not only be ignored by the users, but it might even cause decreased participation. Last, there is also the issue of adoption: Since many software engineering activities are done with tools that come from vendors or open-source communities, one would have to convince those to introduce the gamification layer on top of their tools. It is doubtful that this would happen if there is no substantial evidence that the gamification layer actually works, which brings us back to the concern of evaluating such environments.

8.6.2 Next Steps

We composed the gamification layers presented in this chapter as part of the process to understand the basic concepts of gamification and practically see what is meaningful or what should be highlighted as dangerous. The main result of our work were the gamification framework and the ten essential building blocks to use as a reference in building the system, but the presented scenarios are actual software engineering problems currently investigated by researchers.

The focus of our work revolves around the activities performed by the users. However, further insights can come from considering the different types of users in a community, to avoid the negative effect of marginalizing some users. For example, Vasilescu *et al.* studied the difference between men and women in approaching—and leaving—a community [VCS12], while Koivisto *et al.* showed how the ease of use of gamification tends to decline with age [KH14].

8.7 Outline

We presented a critical overview on the relevant literature on gamification, and proposed a framework to support the design of a gamification layer to support software engineering tasks. We showed how to implement practical actions to successfully gamify a system, and we distilled

ten essential *building blocks* that represent basic elements to be considered when designing gamification activities. We then discussed two example software gamification environments highlighting a number of challenges. Last, we outlined a proposed procedure to evaluate a gamified system.

Our hope is that integrating gamification elements in software engineering will allow developers to build tools where the potential of gamification is leveraged to foster collaboration and contributions by the community. In that sense: The game has just started.



Conclusion

Software development produces large amounts of raw data pertaining to the evolution of a system. The majority of this data is dismissed as a byproduct of the development process and lost. Even when fragments of this data are saved and used, they are flattened in textual format, complicating automated analyses and reducing its reliability. We are convinced that such data is an invaluable asset in supporting developers, to understand both how a system works and how users interact with it. We think that it can become the central component in the design of the next generation of issue tracking systems.

We introduced our work by showing an overview of the efforts by researchers and practitioners to improve the organization and fruition of bug reports. We presented a set of approaches and tools to propose an improved style for collecting bug reports. We implemented our core idea of automatic and reified data collection in SHORELINE, a platform to record runtime exceptions and to gather domain-specific information about specific parts of the system. We developed our tools in PHARO, a dynamic language with a tightly integrated IDE and a strong community. The use of PHARO helped us prototype our tools and quickly test different ideas, allowing us to easily access all the details of the system. Interacting with the PHARO community allowed us to get feedback on the tools we deployed, and to perform qualitative studies to get a preliminary evaluation on our approaches.

The data we collected showed us that failure data can be exploited to support program comprehension, debugging, and optimization of existing systems. This in turn can help reduce the time spent on maintenance, thus containing development costs.

9.1 Visualization of Bug Data

During the preliminary phase of our work we performed a visual inspection of existing bug repositories, looking for patterns and hidden properties that could help accessing the stored information. Later on, we employed again visualizations to navigate the data we collected. We believe that, given the amount of data generated during development, visualizations are an effective means to summarize the activity on a project and provide a selective and layered point of view on specific aspects of the system.

9.1.1 Reading Between the Lines

In Chapter 3, we presented IN*BUG, a web platform to visually inspect the contents of existing issue tracking systems. We ascertained that bug reports contain information that is not properly

conveyed with a mere textual representation. We exploited the structured parts of a bug report (e.g., its metadata) to build a view of the life and evolution of a bug report, highlighting its lifetime and the events of which it is composed. We were able to spot interesting cases of bug reports, like stale bugs that are opened but did not receive any recent activity, or bug reports reopened multiple times. This result suggested us that there is room for improvement in accessing the information that we store about software defects.

9.1.2 Narrating the Evolution of a System

In Chapter 6 we presented BLEND, a tool to display and merge multiple data sources about a system. We used the city metaphor [WLR11] to depict the entities in the PHARO system and their properties. We used the stack traces collected with SHORELINE, our tool to collect information about runtime errors, we extracted the changes in the PHARO system during one year of development, and we integrated the user interaction data dataset provided by Minelli *et al.* [Min17]. We then colored each entity combining different colors to represent the data collected about that entity. From the resulting visualization we could navigate the evolution of the system from a historical perspective, and tell stories about development that can help development decisions or highlight the need for maintenance.

9.2 Collecting Failure Information

The second step in our work consisted in augmenting the reliability of bug reports by augmenting them with automatically collecting failure data.

9.2.1 Collecting Stack Traces

In Chapter 4 we presented our crowdstacking approach: the collection of stack traces from the community to spot recurring errors and understand the usage of a system. We introduced SHORELINE REPORTER, our tool for implementing this approach in the PHARO system, and analyzed the 7,532 stack traces that we collected between June and November 2014. We used the data we collected to show the activity of the users on the system, thus showing the components that can be optimized or the ones that need improvement. We then searched the issue tracking system of PHARO looking for references to the entities in the stack traces. We found that for some stack traces we were able to find an existing bug report. We believe that the approach of automatically providing contextual feedback when an error occurs can greatly improve the experience of the user on a software system and save time.

9.2.2 Reifying Bug Reports

In Chapter 5 we extended the approach presented in Chapter 4 by allowing developers to collect not only stack traces, but also domain specific information about a software component. By employing *collectors*, a developer can specify when an error is interesting to collect and specify the rules to collect it. Collecting the information in its object form, rather than flattening it into a textual representation, allows us to start a conversation with the system that can unveil the hidden properties among the entities in the software.

9.3 Modeling an Issue Tracking System

In the final part of our dissertation we discussed how to improve the experience of users and developers in the issue tracking system. We observed the problem from two different points of view: how to model a bug report to ease the life of reporting users, and how to engage users and developers in participating in the debugging activity.

9.3.1 The Model of a Bug Report

In Chapter 7 we explored the usage of existing issue tracking systems for projects from the *Apache* and *Mozilla* foundations. We conducted a survey to understand what users perceive as difficult to provide when submitting a bug report. We then showed that an increasing number of fields in a bug report has little relation with the lifetime of a bug report. We believe that this study suggests us that a redesign of an issue tracking system should start from simplifying the existing one, rather than adding more textual information.

9.3.2 Gamification

In Chapter 8 we explored the possibility of boosting user engagement when using an issue tracking system by means of *gamification*, the use of game elements in non-gaming contexts. We presented an overview on the history of gamification and its evolution over time. We proposed a framework for systematically gamifying software engineering, posing particular care in highlighting the pitfalls that must be avoided when dealing with gamification. We think that gamification can become a valuable tool, if used to highlight and improve the interactions already existing on a community and not to enforce a specific behavior. It can support the management of software projects, help welcoming new users, and motivating the expert ones.

9.4 Limitations and Future Work

We believe that developing our research project we only scratched the surface of the possible improvements that we can apply to current development methodologies. We provide an overview of the directions that we would like to further investigate, while also discussing the limitation of the approaches we employed.

User Interface

We used the data we collected to generate knowledge on the system. We did not, however, consider the process from a user interface perspective. We are aware that presenting the information to the user in a meaningful and non-intrusive fashion is as crucial as providing correct information: We therefore think that investigating how to display such contextual information to the user is a crucial aspect that should be tackled.

Evaluation

Given the size of the task that we considered, we were able to evaluate our approaches in small contexts, mostly from a tool-driven, qualitative point of view. We believe that a full evaluation of a new issue tracking system, if possible, would require years to complete. Still, a deeper study of the interaction of the various improvement of the development process could shed light on further directions in rethinking issue tracking systems.

Privacy

During our research project we collected a large amount of stack traces from developers performing real development tasks. We were careful in allowing our users to avoid submitting sensible information, but we believe that further efforts in this direction could ease the adoption of such tools and allow the collection of more useful data while safeguarding the privacy and the intellectual property of developers.

Integration With the System

Data collection alone is not enough to provide a smoother development experience. By having access to structured data, we can integrate such information with development tools, for example by recreating the context where a bug occurred with a single click on a website.

9.5 Closing Words

In this dissertation we showed that data generated during software failures carries useful information in understanding and improving a system. We argued that this information should not be discarded, but rather promoted to first-class citizen in the development process by treating it with customized representation, rather than using plain text. This would allow the creation of contextual tools such as visual browsers, recommender systems, or automated build systems. To support software development further, however, it is essential that development tools (*i.e.*, the IDE) integrates such data to create a *holistic* experience.

We see our thesis as a first step in rethinking the idea of bug report, to build smarter issue tracking systems that support development in a deeper and integrated fashion.

Bibliography

- [AADS⁺07] Dorian C Arnold, Dong H Ahn, Bronis R De Supinski, Gregory L Lee, Barton P Miller, and Martin Schulz. Stack trace analysis for large scale debugging. In *Proceedings of IPDPS 2007 (IEEE International Parallel and Distributed Processing Symposium)*, pages 1–10. IEEE, 2007.
- [ABC⁺13] Vanessa Pena Araya, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Agile visualization with Roassal. *Deep Into Pharo*, pages 209–239, 2013.
- [AHM06] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of ICSE 2006 (28th International Conference on Software Engineering)*, ICSE 2006, pages 361–370, New York, NY, USA, 2006. ACM.
- [AW10] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):433–459, 2010.
- [Aye11] Theint Theint Aye. Web log cleaning for mining of web usage patterns. In *Proceedings of ICCRD 2011 (3rd IEEE International Conference on Computer Research and Development)*, volume 2, pages 490–494. IEEE, 2011.
- [BA10] Nicolas Bettenburg and Bram Adams. Workshop on mining unstructured data (mud) because" mining unstructured data is like fishing in muddy waters"! In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 277–278. IEEE, 2010.
- [Bad00] Greg J Badros. JavaML: a markup language for Java source code. *Computer Networks*, 33(1):159–177, 2000.
- [Bar03] Richard Bartle. *Designing Virtual Worlds*. New Riders, 2003.
- [BB13] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of ICSE 2013 (35th ACM/IEEE International Conference on Software Engineering)*, pages 712–721, 2013.
- [BBA⁺09] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced? bias in bug-fix datasets. In *Proceedings of ESEC/FSE (7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering)*, pages 121–130. ACM, 2009.
- [BBVB⁺01] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.
- [BBZJ14] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: Which problems do they fix? In *Proceedings of the 11th working conference on mining software repositories*, pages 202–211. ACM, 2014.

- [BDN⁺09] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, Marcus Denker, et al. *Pharo by example*. 2009.
- [BDSDL12] Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, and Michele Lanza. Content classification of development emails. In *In Proceedings of ICSE 2012 (34th ACM/IEEE International Conference on Software Engineering)*, pages 375–385, 2012.
- [Bec94] Kent Beck. Simple Smalltalk testing: With patterns. *The Smalltalk Report*, 4(2):16–18, 1994.
- [Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.
- [BJS⁺07] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiß, Rahul Premraj, and Thomas Zimmermann. Quality of bug reports in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 21–25. ACM, 2007.
- [BLH11] Alberto Bacchelli, Michele Lanza, and Vitezslav Humpa. RTFM (Read The Factual Mails) –augmenting program comprehension with remail. In *Proceedings of CSMR 2011*, pages 15–24, 2011.
- [BLJ⁺13] Tegawende F Bissyande, Daniel Lo, Lingxiao Jiang, Laurent Reveillere, John Klein, and Yves Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *Proceedings of ISSRE 2013 (24th International Symposium on Software Reliability Engineering)*, pages 188–197. IEEE, 2013.
- [BMRC05] Mark Brodie, Sheng Ma, Leonid Rachevsky, and Jon Champlin. Automated problem determination using call-stack matching. *Journal of Network and Systems Management*, 13(2):219–237, 2005.
- [BN11] Pamela Bhattacharya and Iulian Neamtiu. Bug-fix time prediction models: can we do better? In *Proceedings of MSR 2011 (8th Working Conference on Mining Software Repositories)*, pages 207–210. ACM, 2011.
- [BPM04] Gustavo EAPA Batista, Ronaldo C Prati, and Maria Carolina Monard. A study of the behavior of several methods for balancing machine learning training data. *ACM Sigkdd Explorations Newsletter*, 6(1):20–29, 2004.
- [BPSZ10] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 301–310. ACM, 2010.
- [BPZK08] Nicolas Bettenburg, Rahul Premraj, Thomas Zimmermann, and Sunghun Kim. Duplicate bug reports considered harmful... really? In *Proceedings of ICSM 2008 (24th International Conference on Software Maintenance)*, pages 337–345, Sept 2008.
- [BS01] Leo Breiman and E. Schapire. Random forests. In *Machine Learning*, pages 5–32, 2001.

- [BTW⁺13] Tegawendé F. Bissyandé, Ferdian Thung, Shaowei Wang, David Lo, Lingxiao Jiang, and Laurent Réveillère. Empirical evaluation of bug linking. In *Proceedings of CSMR 2013 (17th IEEE European Conference on Software Maintenance and Reengineering)*, pages 89–98, 2013.
- [Cor89] Thomas A Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.
- [CRJ12] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: A logic for duck typing. *SIGPLAN Not.*, 47(1):231–244, January 2012.
- [Csi90] Mihaly Csikszentmihalyi. *Flow - The Psychology of Optimal Experience*. Harper Perennial, 1990.
- [CZVD⁺09] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.
- [DL07] Marco D’Ambros and Michele Lanza. Bugcrawler: Visualizing evolving software systems. In *Proceedings of CSMR 2007 (11th IEEE European Conference on Software Maintenance and Reengineering)*, pages 333–334. IEEE CS Press, 2007.
- [DLP07] Marco D’Ambros, Michele Lanza, and Martin Pinzger. “a bug’s life” — visualizing a bug database. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 113–120. IEEE CS Press, 2007.
- [DLR10] Marco D’Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31–40. IEEE CS Press, 2010.
- [DLR12] Marco D’Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.
- [DPJM⁺02] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of Java programs. In *Software Visualization*, pages 151–162. Springer, 2002.
- [DR13] Steven Davies and Marc Roper. Bug localisation through diverse sources of information. In *Proceedings of ISSREW 2013 (IEEE International Symposium on Software Reliability Engineering Workshops)*, pages 126–131. IEEE, 2013.
- [DRGP13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.
- [DRSZ10] Stéphane Ducasse, Lukas Renggli, David Shaffer, and Rick Zaccane. *Dynamic web development with seaside*. Square Bracket Associates, 2010.
- [DS14] Tommaso Dal Sasso. Managing software defects. In *Proceedings of ICSME 2014 (30th International Conference on Software Maintenance and Evolution, Doctoral Symposium)*, page to be published, 2014.

- [DSCM⁺17] Tommaso Dal Sasso, Andrei Chis, Andrea Mocci, Tudor Girba, and Michele Lanza. Sympathy for the devil: Reified collection of runtime errors. In *Proceedings of PLATEAU 2017 (8th International Workshop on Evaluation and usability of Programming Languages and Tools)*, pages 1–8. ACM Press, 2017.
- [DSL13] Tommaso Dal Sasso and Michele Lanza. A closer look at bugs. In *Proceedings of VISSOFT 2013 (1st IEEE Working Conference on Software Visualization)*, pages 1–4, 2013.
- [DSL14] Tommaso Dal Sasso and Michele Lanza. in*Bug: Visual analytics of bug repositories. In *Proceedings of CSMR-WCRE 2014 (1st Joint Meeting of the European Conference on Software Maintenance and Reengineering and the Working Conference on Reverse Engineering)*, pages 415–419, 2014.
- [DSML15] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. Misery loves company - crowdstacking traces to aid problem detection. In *Proceedings of SANER 2015 (22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering)*, pages 131–140. IEEE CS Press, 2015.
- [DSML16] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. What makes a satisficing bug report? In *Proceedings of QRS 2016 (IEEE International Conference on Software Quality, Reliability and Security)*, pages 164–174. IEEE, 2016.
- [DSMLM17] Tommaso Dal Sasso, Andrea Mocci, Michele Lanza, and Ebrisa Mastrodicasa. How to gamify software engineering. In *Proceedings of SANER 2017 (24th IEEE International Conference on Software analysis, Evolution, and Reengineering)*, pages 261–271. IEEE CS Press, 2017.
- [DSMML15] Tommaso Dal Sasso, Roberto Minelli, Andrea Mocci, and Michele Lanza. Blended, not stirred: Multi-concern visualization of large software systems. In *Proceedings of VISSOFT 2015 (3rd IEEE Working Conference on Software Visualization)*, pages 106–115, 2015.
- [DT13] Daniel J. Dubois and Giordano Tamburrelli. Understanding gamification mechanisms for software development. In *Proceedings of ESEC/FSE 2013 (9th Joint Meeting on Foundations of Software Engineering)*, ESEC/FSE 2013, pages 659–662. ACM, 2013.
- [FG06] Michael Fischer and Harald C. Gall. EvoGraph: A lightweight approach to evolutionary and structural analysis of large software systems. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, pages 179–188. IEEE Computer Society, 2006.
- [FH82] R. K. Fjeldstad and W. T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. In Girish Parikh and Nicholas Zvegintzov, editors, *Tutorial on Software Maintenance*, pages 13–30. IEEE, 1982.
- [GB13] Scott Grant and Buddy Betts. Encouraging user behaviour with achievements: An empirical study. In *Proceedings of MSR 2013 (10th Working Conference on Mining Software Repositories)*, MSR 2013, pages 65–68, Piscataway, NJ, USA, 2013. IEEE Press.

- [GBC⁺13] Tudor Girba, Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Glamour. Deep Into Pharo*, pages 192–207, 2013.
- [Gee03] James Paul Gee. What video games have to teach us about learning and literacy. *Comput. Entertain.*, 1(1):20–20, October 2003.
- [GKG⁺09] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of SIGOPS 2009 (ACM 22Nd Symposium on Operating Systems Principles)*, SOSP '09, pages 103–116. ACM, 2009.
- [GLD05] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering)*, pages 2–11. IEEE CS Press, 2005.
- [GPG10] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of RSSE 2010 (2nd International Workshop on Recommendation Systems for Software Engineering)*, RSSE '10, pages 52–56, New York, NY, USA, 2010. ACM.
- [GT05] Michael Grottke and Kishor S Trivedi. A classification of software faults. *Journal of Reliability Engineering Association of Japan*, 27(7):425–438, 2005.
- [GZNM10] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 495–504. IEEE, 2010.
- [GZSVD15] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering—Volume 1*, pages 358–368. IEEE Press, 2015.
- [HAD⁺12] Andre Hora, Nicolas Anquetil, Stephane Ducasse, Muhammad Bhatti, Cesar Couto, Marco Tulio Valente, and Julio Martins. Bug maps: A tool for the visual exploration and analysis of bugs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 523–526. IEEE, 2012.
- [Has09] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.
- [HDG⁺12] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of ICSE 2012 (34th International Conference on Software Engineering)*, ICSE '12, pages 145–155. IEEE Press, 2012.
- [HerBC] Herodotus. *The Histories*. 440 BC.
- [HT02] Andy Hunt and Dave Thomas. Software archaeology. *IEEE Software*, 19(2):20–22, 2002.
- [Hui71] Johan Huizinga. *Homo Ludens*. Beacon Press, June 1971.

- [HW07] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.
- [KFGP09] Patrick Knab, Beat Fluri, Harald C Gall, and Martin Pinzger. Interactive views for analyzing problem reports. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 527–530. IEEE, 2009.
- [KH14] Jonna Koivisto and Juho Hamari. Demographic differences in perceived benefits from gamification. *Computers in Human Behavior*, 35:179–188, 2014.
- [KM05] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for IDEs. In *Proceedings of AOSD 2005 (4th International Conference on Aspect-Oriented Software Development)*, pages 159–168. IEEE, 2005.
- [KMC06] Andrew J Ko, Brad A Myers, and Duen Horng Chau. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 127–134. IEEE, 2006.
- [KO04] Hideki Koike and Kazuhiro Ohno. Snortview: Visualization system of snort logs. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 143–147. ACM, 2004.
- [Koh93] Alfie Kohn. *Punished by Rewards*. Houghton Mifflin, 1993.
- [KPG10] Patrick Knab, Martin Pinzger, and Harald C Gall. Visual patterns in issue tracking data. In *New Modeling Concepts for Today's Software Processes*, pages 222–233. Springer, 2010.
- [KR87] Leonard Kaufman and Peter Rousseeuw. *Clustering by means of medoids*. North-Holland, 1987.
- [KS12] Adrian Kuhn and Mirko Stocker. Codetimeline: Storytelling with versioning data. In *Proceedings of ICSE 2012 (34th International Conference on Software Engineering)*, pages 1333–1336, 2012.
- [KZWJZ07] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.
- [Laz04] Nicole Lazzaro. Why we play games: Four keys to more emotion without story. In *Game Developers Conference*, March 2004.
- [LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, September 2003.
- [LVZ10] Bart Luijten, Joost Visser, and Andy Zaidman. Assessment of issue handling efficiency. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 94–97. IEEE, 2010.
- [M⁺67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. California, USA, 1967.

- [Mal80] Thomas W. Malone. What makes things fun to learn? heuristics for designing instructional computer games. In *Proceedings of SIGSMALL 1980 (3rd ACM Symposium and the First SIGPC Symposium on Small Systems)*, SIGSMALL '80, pages 162–169. ACM, 1980.
- [Mas14] Ebrisa Mastrodicasa. Ludus opus proficit — a gamification framework for software engineering. Master's thesis, University of Lugano, 2014.
- [McG11] Jane McGonigal. *Reality is Broken*. Penguin, 2011.
- [McL04] L. McLaughlin. Automated bug tracking: the promise and the pitfalls. *Software, IEEE*, 21(1):100–103, Jan 2004.
- [MCM02] Jonathan I Maletic, Michael L Collard, and Andrian Marcus. Source code files as structured documents. In *Program comprehension, 2002. proceedings. 10th international workshop on*, pages 289–292. IEEE, 2002.
- [Min17] Roberto Minelli. *Interaction-aware development environments: recording, mining, and leveraging IDE interactions to analyze and support the development flow*. PhD thesis, Università della Svizzera italiana, Switzerland, November 2017.
- [Mit97] Tom M Mitchell. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45, 1997.
- [MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.
- [MKN09] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings of MSR 2009 (6th IEEE International Working Conference on Mining Software Repositories)*, pages 131–140, May 2009.
- [MML15] Roberto Minelli, Andrea Mocci, and Michele Lanza. 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)*, pages 25–35, 2015.
- [MR13] Ethan R. Mollick and Nancy Rothbard. Mandatory Fun: Gamification and the Impact of Games at Work. *SSRN eLibrary*, 2013.
- [MT07] Sergio Moreta and Alexandru Telea. Multiscale visualization of dynamic software logs. In *Proceedings of the 9th Joint Eurographics/IEEE VGTC conference on Visualization*, pages 11–18. Eurographics Association, 2007.
- [MTMS14] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval based bug localization. In *Proceedings of ICSME 2014 (30th International Conference on Software Maintenance and Evolution)*, 2014.
- [OJH03] Alessandro Orso, James Jones, and Mary Jean Harrold. Visualization of program-execution data for deployed software. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 67–ff. ACM, 2003.

- [OM10] Michael Ogawa and Kwan-Liu Ma. Software evolution storylines. In *Proceedings of Softvis 2010 (6th ACM International Symposium on Software Visualization)*, pages 35–42, 2010.
- [PBG03] Thomas Panas, Rebecca Berrigan, and John Grundy. A 3d metaphor for software production visualization. In *2013 17th International Conference on Information Visualisation*, pages 314–314. IEEE Computer Society, 2003.
- [PJ09] Hae-Sang Park and Chi-Hyuck Jun. A simple and fast algorithm for k-medoids clustering. *Expert Systems with Applications*, 36(2):3336–3341, 2009.
- [Plabc] Plato. *De Res Publica*. 380 b.c.
- [PMNC11] Erick Baptista Passos, Danilo Medeiros, Pedro A. S. Neto, and Esteban Walter Gonzalez Clua. Turning real-world software development into a game. In *SBGames*, pages 260–269. IEEE, 2011.
- [PVG⁺11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.
- [RFG05] Jacek Ratzinger, Michael Fischer, and Harald Gall. EvoLens: lens-view visualizations of evolution data. In *International Workshop on Principles of Software Evolution*, pages 103–112, 2005.
- [SBP10] Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. Do stack traces help developers fix bugs? In *Proceeding of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 118–121. lol, May 2010.
- [SC00] Martin EP Seligman and Mihaly Csikszentmihalyi. *Positive psychology: An introduction*. American Psychological Association, 2000.
- [SFM99] Margaret-Anne Storey, F. David Fracchia, and Hausi Muller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.
- [SH10] Edward Segel and Jeffrey Heer. Narrative visualization: Telling stories with data. *Visualization and Computer Graphics, IEEE Transactions on*, 16(6):1139–1148, 2010.
- [Sim57] Herbert A. Simon. *Models of Man: Social and Rational*. John Wiley & Sons, 1957.
- [Sim01] Herbert A. Simon. *The Sciences of the Artificial*. MIT Press, 3rd edition, 2001.
- [Ski78] Burrhus Skinner. *Reflections on Behaviorism and Society*. Prentice Hall, 1978.
- [SMDV08] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE TSE 2008 (Transactions on Software Engineering)*, 34(4):434–451, 2008.
- [SPvD17] Mozhan Soltani, Annibale Panichella, and Arie van Deursen. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering*, pages 209–220. IEEE Press, 2017.

- [SS08] Forrest Shull and Carolyn Seaman. Inspecting the history of inspections: An example of evidence-based technology diffusion. *IEEE Software*, 25(1):88–90, 2008.
- [SS12] Leif Singer and Kurt Schneider. It was a bit of a race: Gamification of version control. In *GAS 2012 (2nd International Workshop on Games and Software Engineering)*, pages 5–8. IEEE, 2012.
- [Sui05] Bernard Suits. *The Grasshopper: Games, Life and Utopia*. Broadview Press, 2005.
- [Sun11] Jian Sun. Why are bug reports invalid? In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 407–410. IEEE, 2011.
- [SWY75] Gerard Salton, Anita Wong, and Chung-Shu Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.
- [ŚZZ05] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):1–5, 2005.
- [TBLJ13] Ferdian Thung, Tegawende F Bissyande, David Lo, and Lingxiao Jiang. Network structure of social coding in github. In *Proceedings of CSMR 2013 (17th IEEE European Conference on Software Maintenance and Reengineering)*, pages 323–326, March 2013.
- [VCS12] Bogdan Vasilescu, Andrea Capiluppi, and Alexander Serebrenik. Gender, representation and online participation: A quantitative study of stackoverflow. In *Social Informatics (SocialInformatics), 2012 International Conference on*, pages 332–338. IEEE, 2012.
- [VFS13] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *Proceedings of SocialCom 2013 (International Conference on Social Computing)*, pages 188–195, Sept 2013.
- [VMV95] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.
- [VT06] Lucian Voinea and Alexandru Telea. Multiscale and multivariate visualizations of software evolution. In *Proceedings of the 2006 ACM symposium on Software Visualization*, pages 115–124. IEEE Computer Society, 2006.
- [Wei06] Westley Weimer. Patches as better bug reports. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 181–190. ACM, 2006.
- [WH12] Kevin Werbach and Dan Hunter. *For the Win*. Wharton Digital Press, 2012.
- [WKZ13] Shaohua Wang, Foutse Khomh, and Ying Zou. Improving bug localization using correlations in crash reports. In *Proceedings of MSR 2013 (IEEE 10th IEEE Working Conference on Mining Software Repositories)*, pages 247–256. IEEE, 2013.
- [WL07] Richard. Wettel and Michele Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th IEEE International Conference on Program Comprehension)*, pages 231–240, 2007.

- [WLR11] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of ICSE 2011 (33rd International Conference on Software Engineering)*, pages 551–560. ACM Press, 2011.
- [WPZZ07] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of MSR 2007 (4th International Workshop on Mining Software Repositories)*, MSR 2007, pages 1–, Washington, DC, USA, 2007. IEEE Computer Society.
- [WZX⁺08] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, ICSE '08, pages 461–470. ACM, 2008.
- [YYZ⁺11] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairava-sundaram. How do fixes become bugs? In *Proceedings of ESEC/FSE 2011 (19th Symposium and the 13th European Conference on Foundations of Software Engineering)*, ESEC/FSE 2011, pages 26–36. ACM, 2011.
- [ZM15] Minghui Zhou and Audris Mockus. Who will stay in the floss community? modeling participant’s initial behavior. *Software Engineering, IEEE Transactions on*, 41(1):82–99, 2015.
- [ZN08] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.
- [ZPB⁺10] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering (TSE)*, 36(5):618–643, 2010.
- [ZSG79] Marvin V Zelkowitz, Alan C Shaw, and John D Gannon. *Principles of software engineering and design*. Prentice-Hall Englewood Cliffs, 1979.
- [ZWDZ04] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of ICSE 2004 (26th International Conference on Software Engineering)*, pages 563–572. IEEE CS Press, 2004.



Pharo

In this chapter we briefly present an overview of the technologies that we used to develop our tools and test our approaches.

A.1 Smalltalk in the 21st Century

The majority of the tools presented in this dissertation are developed using PHARO [BDN⁺09]. PHARO is an object-oriented, *Smalltalk* inspired programming environment, composed of the PHARO programming language, an integrated development environment and a set of libraries covering the common needs for the daily programming tasks.

While such a choice might at first seem extreme and anachronistic, it carries a number of advantages and features especially useful when prototyping an idea or a tool. Pharo inherits a number of powerful properties from its Smalltalk origins, that can support our task of implementing the data collection framework. In particular, PHARO 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. By exploiting the characteristics of the platform, we were able to analyze and access programs as they were running and easily collect runtime data, without having to fight the system to extract the data.

In retrospective, we judge that using Pharo in our work and exploiting its abstractions over the entities of the system resulted in a technical advantage that alleviated us from the burden of fully instrument a virtual machine to reproduce and study the defective environment.

A.1.1 Runtime Errors in Pharo

An interesting property of Pharo comes from its dynamic nature: the whole system is polymorphic. This polymorphism is obtained through the so-called *duck typing* [CRJ12]: every object can be used in place of other objects, as long as it is able to respond to the same messages. This entails that—as in other dynamic programming languages—there is no static type system and, as such, no static type checking: every type error happens at runtime, resulting in a *Message Not Understood* kind of exception. This peculiarity is important when considering the nature of runtime errors in Pharo, because the vast majority of the exceptions is caused in this context: In Chapter 4 we show how we collected a dataset of development problems, where in more than 72% of the cases an exception is caused by a message not understood. Among those cases, 68% are generated from a message sent to *UndefinedObject*. These are the equivalent of a *NullPointerException* in Java.

A.2 The Pharo Community

Apart from a rich software collection, the PHARO ecosystem is composed of a vibrant and active community¹ that includes about 2,000 developers both from academia and industry. The community actively participates in the development of the system by building tools to improve the user experience, submitting bug reports and proposing patches to solve defects.

Such a small and active community was invaluable when deploying our tools and collecting real data from daily development.

¹<http://pharo.org/community>



List of Publications

- Content Classification of Development Emails** [BDSDL12]
Alberto Bacchelli, Tommaso Dal Sasso, Marco D'Ambros, Michele Lanza
In Proceedings of ICSE 2012 (34th International Conference on Software Engineering), pp. 375–385, IEEE CS Press, 2012.
- A closer look at bugs** [DSL13]
Tommaso Dal Sasso, Michele Lanza
In Proceedings of VISSOFT 2013 (1st IEEE Working Conference on Software Visualization), pp. 1–4, IEEE CS Press, 2013.
- in*Bug: Visual analytics of bug repositories** [DSL14]
Tommaso Dal Sasso, Michele Lanza
In Proceedings of CSMR-WCRE 2014 (1st Joint Meeting of the European Conference on Software Maintenance and Reengineering and the Working Conference on Reverse Engineering), pages 415–419, IEEE CS Press, 2014.
- Managing Software Defects** [DS14]
Tommaso Dal Sasso
In Proceedings of ICSME 2014 (30th International Conference on Software Maintenance and Evolution), page 669, Doctoral Symposium, 2014.
- Misery Loves Company: CrowdStacking Traces to Aid Problem Detection.** [DSML15]
Tommaso Dal Sasso, Andrea Mocci, Michele Lanza
In Proceedings of SANER 2015 (22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering), pp. 131–140, 2015.
- Blended, Not Stirred: Multiconcern visualization of large software systems** [DSMML15]
Tommaso Dal Sasso, Roberto Minelli, Andrea Mocci, Michele Lanza
In Proceedings of VISSOFT 2015 (3rd IEEE Working Conference on Software Visualization), pp. 106–115 2015.

What Makes a Satisficing Bug Report

[DSML16]

Tommaso Dal Sasso, Andrea Mocci, Michele Lanza

In Proceedings of QRS 2016 (The 2016 IEEE International Conference on Quality, Reliability, and Security), pp. 164–174, IEEE CS Press, 2016.

How to Gamify Software Engineering

[DSMLM17]

Tommaso Dal Sasso, Andrea Mocci, Michele Lanza, Ebrisa Mastrodicasa

In Proceedings of SANER 2017 (24th IEEE International Conference on Software Analysis, Evolution, and Reengineering), pp. 261–271, IEEE CS Press, 2017.

Sympathy for the Devil: Reified Collection of Runtime Errors[DSCM⁺17]*Tommaso Dal Sasso, Andrei Chis, Andrea Mocci, Tudor Girba, Michele Lanza*

In Proceedings of PLATEAU 2017 (8th International Workshop on Evaluation and Usability of Programming Languages and Tools), pp. 1–8, ACM Press, 2017