

Debugging Integrated Systems: An Ethnographic Study of Debugging Practice

Thomas Østerlie, Alf Inge Wang
Norwegian University of Science and Technology
{thomas.osterlie, alf.inge.wang}@idi.ntnu.no

Abstract

This paper explores how software developers debug integrated systems, where they have little or no access to the source code of the third-party software the system is composed of. We analyze the practice of debugging integrated systems, identifying five characteristics that set it apart from existing research on debugging: it spans a variety of operating environments, it is collective, social, heterogeneous, and ongoing. We draw implications of this for software maintenance research and debugging practice. The results presented in this paper are based on observations from an ethnographic study of the Gentoo OSS community, a geographically distributed community of over 320 developers developing and maintaining a software system for distributing and integrating third-party software packages with different Unix versions.

1. Introduction

Software maintenance constitutes a significant factor (between 50 and 80 percent) in the total life-cycle costs of software systems [1]. Research suggests that software developers spend much of the maintenance effort simply trying to understand the software [2]. Current research is based on the premise that source code is the primary data source for understanding the software during debugging. Models of software errors proposed in the software engineering literature are based on the premise that software failures can be traced back to faults in the source code [3].

However, with increased attention on systems integration these are problematic premises. In component-based development [4], Web services and service-oriented architecture, along with information and enterprise systems integration [5], systems integrators have limited, if any, access to the source code of the integrated software. Even when integrating with open source software (OSS) components, research

suggests that few systems integrators actually access the source code [6]. As such, systems integrators face the situation of having to debug systems without the source code to build an understanding of the problem upon. We therefore ask: without the source code, *how do systems integrators make sense of problems when debugging integrated systems?*

Debugging integrated systems is largely unexplored in the research literature. The debugging process must be understood before it can be improved upon. This motivates a shift of focus from improving the debugging process, towards exploring how software developers debug integrated systems in practice. To this end, we have explored the practice of debugging integrated systems through an ethnographic study of the Gentoo OSS community. Gentoo is a geographically distributed community of volunteer systems integrators maintaining and operating a software distribution system for distributing and integrating third-party OSS with various Unix operating systems. Similar to existing studies of community-based OSS development [7], debugging is a core activity in the Gentoo community's software development process, too. The community is therefore well suited for studying the practice of debugging integrated systems.

The shift of focus towards debugging practice requires that we draw upon research on practice. In this study we therefore draw upon research on practice and problems in organization science. This research shows that in real-world practice problems do not present themselves to practitioners as given [8]. Rather, problems have to be constructed from the materials of problematic situations that are puzzling, troubling, and uncertain. The process of constructing well-defined problems out of problem situations is often called sensemaking [9]. We will use sensemaking as a theoretical lens for exploring the practice of debugging integrated systems.

This paper contributes to debugging research by identifying five characteristics that sets the practice of debugging integrated systems apart from existing research on debugging: it spans a variety of operating

environments, it is collective, social, heterogeneous, and ongoing.

The remainder of the paper is organized as follows. Section 2 presents existing work on debugging, illustrating the central role of source code as data source for debugging. The theoretical lens of sensemaking is also presented here. Section 3 describes the research methods employed and materials collected during the ethnographic fieldwork. Section 4 describes the overall debugging process in the Gentoo community. Section 5 is an analysis of debugging in Gentoo applying the theoretical lens of sensemaking. We conclude the paper by discussing the implications of our findings for software maintenance research as well as debugging practice in Section 6.

2. Related work

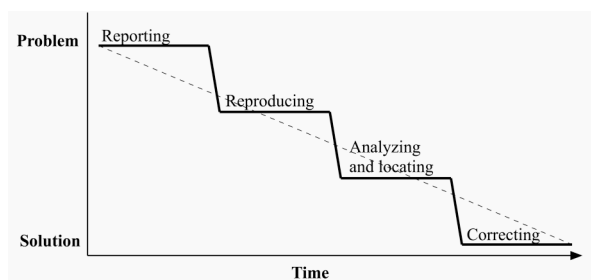
In this section we will illustrate how debugging can be understood as a linear process from problem to its solution. Such a linear model requires that the debugging developer can trace a causal chain from the software failure to its corresponding fault. To this end, we argue, source code is critical. Without the source code, tracing such causal chains becomes harder. This motivates our use of sensemaking as theoretical lens for analyzing how software developers understand problems when debugging integrated systems.

2.1. Debugging

Debugging is the process of locating and correcting the cause of an externally visible error in the program behavior [3]. Araki et al. [10] proposes a model for systematic debugging where debugging is viewed as a process of developing hypotheses about the cause of errors, expected program behavior, and how to modify the program to correct errors, and to refute or verify these hypotheses. Zeller [3] proposes a similar model. These models may be summarized as a stepwise process from a well-defined problem to its solution (as illustrated in Figure 1).

Much of the existing research on debugging focuses on the process of locating the cause of errors. Broadly speaking, three approaches have been suggested [11]. The *bottom-up* approach to debugging involves reading program statements in the source code and chunking these into higher-level abstractions. In the *top-down* approach, software developers reconstruct knowledge about the problem domain and map this to the source code. A mixed model approach has also been suggested.

Figure 1 Linear model of debugging



Several techniques, like delta debugging [3], and tools, like Eden [10], have been proposed to support the process of locating the cause of errors in the source code.

2.2. Sensemaking

The subsection above illustrates how existing research describes debugging as a linear problem solving process, progressing from a well-defined problem to its solution (illustrated by the dashed line in Figure 1). For systems integrators, however, problems do not present themselves as given. Rather, problem situations are ambiguous and open to multiple interpretations [12].

Research within organization science shows that interaction among actors increases in ambiguous situations. In a study of field service technicians repairing copying machines, Orr [13] shows that to make sense of a faulty machine the technicians engage in an ongoing dialogue about the machine with the customer. Similarly, in a study of modern professionals, Schön [8] finds that the daily work of practitioners is not about problem solving, but rather about problem setting; the kind of work professionals undertake to make a situation that is initially ambiguous, puzzling, troubling, and uncertain into something that makes sense.

Confused by ambiguity people engage in sensemaking [9]. The basic premise of sensemaking is that a person or a group's collective experiences of a problem situation are progressively clarified. Rather than starting with well-defined problems, sensemaking is a framework for analyzing how practitioners make sense of a situation that initially makes little sense. In contrast to problem solving starting with well-defined problem, the question driving sensemaking is not 'which of the available means are best suited to solve the problem?' but rather 'what is going on?'. To make sense of a problem situation, people act on basis of previous experience. By actively engaging with the problem situation, understanding emerges as people make retrospective sense of what occurs by enlarging small cues from the available data and forming a

structure to provide meaning. Another central premise of sensemaking is therefore that action precedes understanding.

3. Methods and materials

This research is based on the first author's ethnographic study of the Gentoo OSS community. This section briefly describes the research setting and the ethnographic fieldwork this study is based upon. A more detailed description of the research including a more thorough discussion on research validation can be found in [12].

3.1. Research setting: Gentoo

Gentoo is an OSS community of volunteers maintaining and operating a software distribution system for distributing and integrating third-party OSS with various Unix operating systems. In addition, the community provides a GNU/Linux distribution, Gentoo Linux, on top of the software distribution system. The community consists of 320 official developers distributed across 38 countries and 17 time zones¹. To the best of our knowledge, none of the developers are geographically co-located. As with most OSS communities, users are an important part of the Gentoo community, contributing with problem reports as well as source code. However, it is impossible to tell how many users are active in the community at any one time.

For the remainder of the paper we will use the term Gentoo about the Gentoo software distribution system, Gentoo Linux about the GNU/Linux distribution provided on top of Gentoo, and the Gentoo community when talking about the community of volunteer systems integrators. These volunteers call themselves Gentoo developers.

Gentoo distributes third-party OSS packages in the form of installation scripts. The installation scripts are stored in a central repository. One script exists for every version of each of the 8486 supported packages, for a total of 23911 installation scripts. The total SLOC of installation scripts in the repository is 671971². The installation scripts make up 90% of the source code in the repository. The rest are mainly patches and configuration files. The installation scripts are written and maintained by the Gentoo community. While some Gentoo developers may be quite familiar and knowledgeable of the source code of the components they integrate, most treat the software being integrated

as a black box. Up to six different Unix versions may be supported by a single installation script: GNU/Linux, FreeBSD, OpenBSD, NetBSD, MacOS X, and Dragonfly. For GNU/Linux, five different processor architectures may also be supported in the script.

The repository is mirrored on every Gentoo system. A Gentoo system is a computer system using Gentoo for integrating third-party OSS on the local system. The Portage package manager is the application that integrates third-party packages locally on Gentoo systems, calculating dependencies to other packages, downloading the source code, as well as configuring, compiling and integrating the package with the Gentoo system's live file system.

3.2. Ethnographic fieldwork

Data was collected during a ten months period of participant-observation. Participant-observation is the predominant method for ethnographic fieldwork [15]. In this study, participant-observation meant that the first author participated in the Gentoo community by submitting and resolving problem reports, interacting with the Gentoo users and developers on Internet Relay Chat (IRC) and e-mail, as well as participating in a major restructuring effort of the Portage package manager.

During the ten months period of participant-observation the first author wrote field notes at the end of each day of fieldwork [16]. In addition to the field notes, four of the Gentoo IRC channels were logged to file, one file per day for each channel, totaling 1027 files.

Ethnographic research does not follow a step-wise process [17]. Rather, ethnographic data analysis is an ongoing process from the moment the field worker enters the field until the complete research report is written. During the field work the data analysis was informal. Upon withdrawing from the field, the first author spent a year working systematically through the collected data, looking for recurring patterns. This formal data analysis was a process of incrementally generalizing from a multitude of singular observations to increasingly more generalized descriptions of activities. Throughout the process, non-recurring details of the singular observations were omitted and recurring issues included, leading to the analysis presented in this paper.

4. The Gentoo debugging process

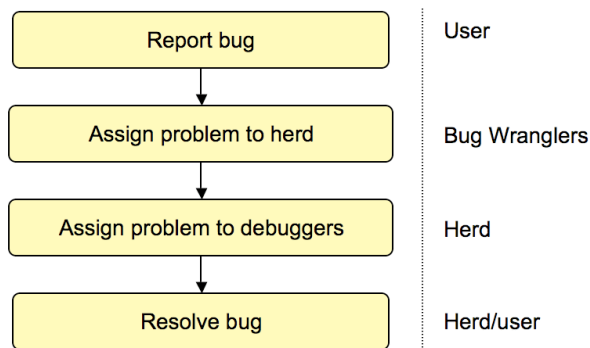
As reported in previous studies of community-based OSS development [7], debugging is a central activity

¹ Unless otherwise stated, all provided figures are of March 30, 2006, the day the fieldwork was concluded

² Data generated with SLOCCount [14]

for the Gentoo community, too. New installation scripts are made available in the repository after marginal quality assurance. Instead, users are expected to report problems. As such, debugging plays a central role as a quality assurance mechanism in Gentoo. The debugging process (illustrated in Figure 2) is managed through an installation of the Bugzilla defect tracking system [18]. While Bugzilla is the name of a product, unless otherwise noted we will use the term Bugzilla about Gentoo's installation of this system for the ease of reference.

Figure 2 Overview of the debugging process



4.1. Roles

The distribution of roles in the Gentoo community's debugging process is similar to that reported in existing research on community-based OSS development [7]. Users submit the majority of problem reports in the Gentoo community. The *Bugwranglers* is the name of the change control board responsible for assigning newly submitted problem reports to the relevant herd. A *herd* is a team of Gentoo developers responsible for a collection of third-party packages. There are 124 such herds, varying in size from a single Gentoo developer to over 20 developers. The herd is responsible for resolving problem reports.

4.2. Responsibilities

Gentoo integrates software from hundreds of different third-party providers. When debugging, the Gentoo developers are responsible for problems related to the way the third-party OSS packages are integrated. They are not responsible for resolving defects in the third-party software. Similarly, the Gentoo developers are not responsible for problems related to the configuration of a particular Gentoo system. In the latter case, user support is handled on dedicated IRC channels, mailing lists, and Web forums, not through Bugzilla.

4.3. Submitting and assigning reports

Users submit problem reports when they have run out of resources locally to resolve a problem. New problem reports are submitted through a standardized Web-based form. The form defines a number of fields to describe the problem, including a short description of the failure situation, the operating system and hardware platform of the failing Gentoo system, the component where the problem has occurred, the package's version number, as well a longer description of the problem situation including steps to reproduce, which software packages are affected, the reproducibility of the problem, any error messages generated when the software fails, as well as a standardized systems information of the user's system generated by running a Gentoo-specific tool.

When a new problem report is submitted to Bugzilla, an e-mail is sent to the Bugwranglers' mailing list. The Bugwranglers will assign newly submitted problem reports to the relevant herd.

4.4. Resolving problem reports

Once the Bugwranglers have assigned a problem report to a herd, an e-mail is sent to the herd's mailing list. Herds have different ways of distributing work. Many developers scan incoming problem reports to see if they immediately can resolve the report. Other herds formally distribute problem reports among themselves.

Resolving a problem report does not necessarily mean that the problem itself is resolved. This is one of the ways defect reports are resolved. The other ways are to mark the report as a duplicate, to mark it with the flag NEEDINFO meaning that the user has to provide additional information about the system or software failure itself, to reject the problem report as the problem is a user problem, or to mark the problem report as upstream. The latter option is used when the reported problem is caused by a defect in the third-party software itself.

Reaching the closure with one of the above five resolutions to problem reports requires an understanding of the system causing the software failure. In the next section we will analyze how this understanding is produced.

5. Results and analysis

With basis in the overview of the Gentoo debugging process above, we will now revisit the research question posed in the introduction: how do systems integrators make sense of problems when debugging integrated systems? We do so with the theoretical lens

of sensemaking. Our focus in this analysis is therefore on *what people do*, rather than prescribing what should have been done to improve the debugging process. We do so by identifying five characteristics of debugging an integrated system; (C1) it spans a variety of operating environments, (C2) it is collective, (C3) social, (C4) heterogeneous, and (C5) ongoing.

5.1. C1: Variety of operating environments

Zeller [3] states that to fix a problem, the developer must first be able to reproduce it. Although many reported problems are reproducible, the Gentoo developers often face problems they are unable to reproduce, or at least problems that are not easily reproduced. This is illustrated in Exhibit 1.

Exhibit 1. Excerpt from Gentoo developers' IRC channel (gentoo-dev-2004.04.16)

Developer A: This particular Web page crashes both the Mozilla and Galeon Web browsers.
Developer B: That doesn't happen on my computer.
Developer A: I've built the applications for the Athlon T-Bird processor architecture, and both have been compiled with the GTK2 widget library. I generally assume it's my using GTK2 that messes it up.
Developer B: It might be GTK2. I've compiled both Web browsers with the GTK1 widget library on my system.
Developer D: Well, that page works on my Epiphany Web browser compiled with the GTK2 library.
Developer C: And it works with my installation of Mozilla compiled with GTK2.
Developer D: This other Web page crashes my Phoenix Web browser, but not Mozilla or Galeon.
Developer A: The Web page crashes on my Epiphany installation, as well. It seems it's my Mozilla build that's flakey.
Developer C: But boingboing.net crashes my Epiphany installation. I've compiled it for the PentiumIII processor architecture, though.
Developer A: boingboing.net crashes Galeon on my system, too.
Developer B: boingboing.net working for Mozilla on my system.
Developer C: Hmm... It seems the problem is related to Mozilla compiled with the GTK2 widget library and the Xft font library. Weird thing is that boingboing works on my Galeon installation...
Developer A: Now here's a very good reason to only build for one processor architecture, stable source tree and only do point releases. Variation kills reproducibility.

All of the four Web browsers mentioned in the exhibit are based upon Mozilla's rendering engine. This rendering engine is integrated on a Gentoo system along with Mozilla. As such, the installation scripts of the other three Web browsers have dependencies to Mozilla.

As developer A observes in Exhibit 1, the variety of operating environments makes reproducing problems difficult. This is similar to Littlewood's [19] explanation of Adams [13] observation that in large-scale software maintenance most reported problems are irreproducible: irreproducibility is an effect of the variety of operating environments in the population of systems. There are three dimensions of variety of operating environments among Gentoo systems:

operating system, configuration of individual packages, and system evolution.

Gentoo distributes software for six different operating systems. Although most installation scripts in the Gentoo repository do not support for all of the operating systems at once, many packages support multiple operating systems. However, operating systems work in different ways, as illustrated in Exhibit 2.

Exhibit 2. Excerpt from Gentoo developers' IRC channel (gentoo-dev-2004.10.10)

Developer A: [making reference to a stack trace attached to the problem report being discussed] why is it that the thing [the linker] can't find pthread? is that because of a missing -pthread [flag being passed to the linker]
Developer B: sounds like glibc was upgraded [glibc is a Unix runtime library]
Developer A: an upgraded glibc still has pthreads alright?
Developer A: without that symlink the system grinds to a halt
Developer B: needs -lpthread I guess.
Developer C: well the cross-platform way is to use gcc's -lpthread, because not all systems have libpthread. bsd has libc_r for example

In the above exhibit, understanding the problem is made difficult by the way different operating systems support, in this particular case, multi-threading. Both of the above exhibits illustrate that reproducibility is not only made difficult by the variety in operating environment among Gentoo systems, but the variety of operating environments also makes problem understanding difficult.

Exhibit 1 illustrates how debugging is made further complicated by the way individual packages are configured upon integration with a Gentoo system. Such configuration of individual packages is the second dimension of variety among Gentoo systems. There are two dimensions to individual package configuration: *optionals* and *virtuals*.

Packages can have optional functionality that may be compiled into the package when it is integrated on a Gentoo system. This local configuration of individual packages is similar to what Carney et al. [4] describes as installation-dependent products in COTS development, a form of modification of a generic software product that is intended by the provider but may still vary from system to system. With virtual configuration different third-party packages may provide the same functionality. Exhibit 1 illustrates this, as both GTK1 and GTK2 may provide widget library support for the four Web browsers in question.

Packages depend on other packages. Although such package dependencies are inscribed in the installation scripts, these dependencies are only convenient for reproducing a freshly setup Gentoo system. However, many Gentoo systems have been running for a long time. New versions of packages are continuously being added to the Gentoo distribution system's repository, while old and unsupported versions of packages are

being removed. Yet, how up-to-date every package on a Gentoo systems is, varies greatly. This is the fourth dimension of variety in operating environments among Gentoo systems: system evolution.

Although the number of combinations of packages on a single Gentoo system is finite, package configurations and the effects of system evolution often makes it practically impossible to replicate the system configuration required to reproduce the problem. The situation debugging Gentoo is therefore similar to Araki et al.'s [10] observation of debugging concurrent programs. Because the state of concurrent programs may be non-deterministic, programmers often say that debugging is almost completed when they have figured out how to reproduce the problem. Similarly, the Gentoo developers spend a great deal of time understanding the reported problem. Similar to Schön's [8] observation, problems do not present themselves to the Gentoo developers as given, but have to be constructed from the materials of problematic, uncertain, and puzzling situations.

5.2. C2: Collective

When problems do not present themselves as given, the Gentoo developers need to establish what is going on. A fundamental aspect of sensemaking is that a person or a collective's experiences of a situation are progressively clarified [9]. By collectively engaging with the reported problem, comparing configurations of libraries, processor architectures, and applications, the Gentoo developers collectively work towards an understanding of the problem situation as seemingly "related to Mozilla compiled with the GTK2 widget library and the Xft font library" (see Exhibit 1). By extracting cues from the environment, information about processor architectures, widget libraries, which Web pages crashes which browser, the developers collectively makes sense of the problem situation.

In a study of field service technicians diagnosing and repairing copying machines, Orr [13] describes how technicians and users collectively make sense of faulty machines. Although provided with detailed guidelines for diagnosing and repairing copying machines, service technicians were often faced with confounding machine behavior going beyond the official documentation. To make sense of the faulty machine behavior, the service technicians interact with the customer to create a context for the behavior. By recreating the machine's history, its past quirks and problems, the customer and service technician engage in a process of constructing a context where the service technician can make sense of the faulty machine. Repairing the machine is not a process of finding the problem causing the faulty behavior and then repairing

it. Rather, the problem is to understand what the problem is. By interacting with the customer and the faulty machine, the service technician creates a setting where the faulty behavior makes sense and can be resolved.

As the variety of operating environments often makes it difficult for the Gentoo developers to reproduce reported problems, we find the Gentoo developers and users collectively working together to make sense of reported problems. They typically use the problem report for interacting, adding new comments to the *Additional comments* field at the bottom of the problem report as illustrated in Exhibit 3.

Exhibit 3. Excerpt of problem report illustrating use of *Additional comments* field

| | |
|--|---------------------------------|
| Description: | Opened: 2003-01-02 02:41 |
| Basically, I can't even configure the package and it fails complaining 'required file './depcomp' not found'. I have re-integrated the autoconf and automake packages, but I still get the same problem. [error output provided] [systems information provided] | |
| Comment #1 From Developer A 2003-01-02 04:32:56 Which version of automake and autoconf do you use? | |
| Comment #2 From User 2003-01-02 04:43:33 [version information about automake and autoconf packages on local system provided] | |
| Comment #3 From Developer A 2003-01-02 04:52:19 Seems to be the required versions. Could you please try -r16 and/or -r18 of if the xmms package to see if it works for you? | |
| Comment #4 From User 2003-01-02 04:58:44 Nope, both die in exactly the same way. I did have this installed at first, but then it tried to update the package a while ago and it just wouldn't install properly. Is there a package that xmms requires that might be broken? [new error message provided] | |
| Comment #5 From Developer A 2003-01-02 05:23:06 Could it be you are running out of disk space or memory and swap? | |
| Comment #6 From User 2003-01-02 05:26:27 [information about available disk space on local system's hard drive partitions provided] I doubt that disk space or memory is a problem, although that tmpfs device is a tad full!! [information about local system's memory use provided] | |
| Comment #7 From Developer A 2003-01-02 05:33:02 Version 1.3 of the xmms installation script is latest, which version do you have? Attach the output of the command 'head /usr/portage/media-sound/xmms/xmms-1.2.7-r15.ebuild' | |
| Comment #8 From User 2003-01-02 05:35:59 Version 1.3 [output of running head command provided] | |
| Comment #9 From Developer B 2003-01-02 06:05:58 This has to do with the version of automake/autoconf being used by the emerge process. My feeling is that xmms is using a version that is not compatible with its config process. I will, therefore, adjust xmms's ebuild to make sure it calls the correct version. Please stand by for an updated ebuild that you can test. I have never seen this kinda thing with the xmms package. | |
| Comment #10 From Developer A 2003-01-18 14:06:31 What about just adding a --add-missing to: [script provided] | |

Comment #11 From User 2003-01-19 18:04:44

Ok, really strange... I just integrated the KDE-3.1_rc6 package and xmms installed without any problems.

Comment #22 From Developer A 2003-01-19 19:02:59

Well, I did add the --add-missing to the -r18 of the xmms installation script anyhow.

Exhibit 3 illustrates how Gentoo developers and users collectively work together to make sense of reported problems. 17 days pass from the date the problem is reported until it is resolved illustrating that collective debugging can be a longitudinal process.

The exhibit illustrates a typical exchange, where the developer asks the user to generate new data about the failing Gentoo system. This often entails the user running one or more diagnosis tools, producing output texts that are attached to the failure report. As illustrated by Exhibit 1, developers often use IRC for discussing problem reports in detail. There is a mailing list that is used for this, too. The user is often asked several times to generate new information, in a cyclic process between users producing data and developers interpreting the available data [12].

This observation is somewhat different to existing reports from community-based OSS development. Huntley [20], for instance, argues that debugging is a task that in nature lends itself to distribution, as finding problems is "a task that can be performed by thousand or even millions of end-users without any involvement of the core development team". Since most software failures are limited in scope, he continues, involving only a small fraction of the code, a well-controlled debugging process can be distributed among large number of programmers. We, on the other hand, observe that such a distinct separation between describing and understanding problems is problematic.

Contrary to the clear separation between problem description and analysis on Figure 1, Exhibit 3 illustrates that the process of making sense of the problem is not decoupled from the process of describing the problem. To make sense, the Gentoo developers and the user must act by engaging with the problem. This may seem like a process of trial-and-error, but from a sensemaking perspective action precedes understanding [9]. By collectively engaging with the problem, the Gentoo developers and user create materials from which they may construct the problem. As such, the debugging process observed in the Gentoo community is more a process of creating the problem retrospectively, rather than being driven by a process of formulating hypotheses and rejecting or verifying them. It is therefore a process driven by plausibility rather than accuracy [9]. Failed efforts to solve problems feed back into the debugging process with new materials to set the problem anew, or with requests for new information from the user. The

problem and its causes are constructed in retrospect, once the solution is in place.

5.3. C3: Social

Debugging plays a key quality assurance mechanism in Gentoo. Installation scripts are released with only a minimum of quality assurance; with the expectation that problems related to way software is integrated on Gentoo systems will be reported. Exhibit 3 illustrates how debugging can be a longitudinal, although low-intensity activity. Although a low-intensity activity, Exhibit 4 shows the sheer number of problem reports submitted to Bugzilla on a weekly basis exceeds the number of problem reports the Gentoo developers are able to close. The exhibit is based on Bugzilla statistics published by the Gentoo developers in the Gentoo Weekly Newsletter [21]. The developer count is generated from the Gentoo developer list [22].

Exhibit 4. Weekly debugging workload

| Date | New reports | Reports closed | Open reports | Number of developers |
|-----------------|-------------|----------------|--------------|----------------------|
| January 6 2003 | 269 | Not avail. | 1893 | 102 |
| January 5 2004 | 837 | 428 | 4479 | 259 |
| January 3 2005 | 700 | 390 | 7877 | Not avail. |
| January 16 2006 | 799 | 447 | 9083 | 320 |

The increasing gap between new and closed problem reports may be partly explained by the way the Gentoo community uses Bugzilla; problems reported on outdated versions of packages are ignored and never marked as resolved, and the Gentoo developers use problem reports for tracking issues as well. Yet, despite a steady increase in the number of Gentoo developers, the workload of debugging exceeds the capacity of the Gentoo developers as the increasing number of open problem reports show. There is therefore a need for the Gentoo developers to prioritize among problem reports.

The problem report provides a field for rating a problem's severity. However, an understanding of problems often is retrospective (Section 5.2). Knowing the severity of a problem is therefore also retrospective, and prioritizing is therefore problematic without starting to make sense of the reported problem.

In this situation, the Gentoo developers have to balance between several interests. On the one hand, they have to prioritize problem reports that may potentially affect the many Gentoo systems. That reported problems are reproducible imply that the problem may affect many systems. Prioritizing reproducible problem reports comes at the expense of

irreproducible problem reports, or reports on problems that occur only on one or few systems.

Commenting on similar tradeoffs for prioritizing problem reports reported by Adams [23], Littlewood observes that with a large population of operating environments there may always be one or more problems that are unique to a particular user's operating environment. However, the user would be extremely disgruntled if the problem was not resolved, as the problem would be recurring at appreciable rates in his environment.

Similarly, because of the variety of operating environments among Gentoo systems, many reported problems will not be reproducible and are particular to a single or a small group of Gentoo systems. For the debugging process to function properly as a quality assurance mechanism, the Gentoo developers have to keep users interested in submitting problem reports in the future. The developers therefore have to balance the need for resolving problem reports that will increase the reliability of Gentoo for the most users, with debugging problems they are unable to reproduce or problems that are particular to a single user's system.

To curb the workload, enforcing the boundaries of one's responsibilities is an important part of debugging practice. Although the responsibilities are clearly delineated in theory (section 4.2), establishing responsibilities is more of an open question. Zeller [3] views such a lack of clarity as a political process of deciding who is to blame. Assuming the perspective of sensemaking, however, determining responsibilities is an inherent part of the retrospective process of making sense of problems. Exhibit 5 is a dialogue aggregated from a problem report and discussions about this problem report on the Gentoo developers' IRC channel.

Exhibit 5. Enacting responsibilities

| Statement | Researchers' commentary |
|---|--|
| Reporting user: <i>I have installed my system from scratch</i> | The problem is related to the way Gentoo integrates software, and therefore the Gentoo developers' responsibility |
| Developer A: <i>[making reference to the systems information provided with the problem report] Is using an x86 profile for an amd64 machine troublesome?</i> | The reported problem is related to the way the user's Gentoo systems configuration; therefore the user's responsibility |
| Developer B: <i>[making reference to the installation script] Turning off the optional esound support might solve the problem.</i> | The problem may be related to how the package integrates with the esound package, and the third-party provider's responsibility. |
| Developer A: <i>[making reference to the compiler error provided with the problem report] Why is it that the thing can't find pthread? is that because of a missing -pthread</i> | The problem is related to the use of the pthreads library, and therefore the responsibility of another herd. |
| Developer B: <i>sounds like the glibc library was upgraded</i> | Related to the user's system configuration, and his responsibility |

By extracting cues from the situation ('installed my system from scratch'), from the systems information and error messages provided with the problem report, as well as information from the installation script, the Gentoo developers and the user bridge the ideal division of responsibilities (Section 4.2) and the concrete details of the problem. They produce a reality of responsibilities by their actions. However, this construction of reality is in itself constrained by their understanding of responsibilities. The model of responsibilities precedes the discussion of the particular problem, acting as a guide for extracting cues from the data. As Weick [9] puts it, sensemaking is enactive of sensible environments.

Although debugging is a technical activity, the above analysis shows how social issues like keeping users interested and determining responsibilities are closely intertwined with the technical activities of debugging.

5.4. C4: Heterogeneous

Heterogeneity is one of Hasselbring's [5] three characteristics of systems integration: "heterogeneity comes from different hardware platforms, operating systems, database management systems, and programming languages". This is similar to what the variety of operating environment among Gentoo systems (section 5.1). Similarly, Belady & Lehman [24] presents variety as a root cause of program largeness. While Hasselbring's notion of heterogeneity is technical, Belady & Lehman's understanding of variety includes both the social and the technical.

Similar to Hasselbring, we find heterogeneity to be a characteristic of the Gentoo debugging process, but like Belady & Lehman our view of heterogeneity transcends the technical. While the purpose of the debugging process is to keep Gentoo running, we find that keeping Gentoo running is not solely a technical endeavor. Rather, to keep Gentoo running requires maintenance of both the technology and the community. The debugging process is therefore heterogeneous in the sense that it serves a variety of interests and activities, where the social and the technical are closely intertwined (Section 5.3).

Section 2.1 shows that existing research is based on the premise that source code is the primary data source for debugging. Debugging Gentoo is heterogeneous in the respect that instead of relying on source code, understanding of problems is constructed from a heterogeneous ensemble of data sources: problem reports, debug data generated by the failing software and various diagnosis tools, as well as discussions on IRC, mailing lists and Web forums (Sections 5.1 and 5.2).

5.5. C5: Ongoing

Although debugging is a central activity in the Gentoo community, it is not the only responsibility the Gentoo developers have. Within the community, the developers are responsible for keeping abreast with the latest developments for the third-party OSS packages of their herd—writing new installation scripts and updating existing scripts to incorporate patches made available outside of the packages' release cycles—as well as being active on the IRC channels and mailing lists discussing and assisting other developers. In addition, the Gentoo developers have outside responsibilities like daytime jobs, and school.

As such, debugging is one activity in the ongoing flow of activities making up the day of the Gentoo developers. While it may be a low-intensity activity (see Section 5.3), debugging is not an activity the developers can devote all their attention to as illustrated by Exhibit 6.

Exhibit 6 Extract from the Gentoo developers' IRC channel (gentoo-dev-2004.07.17)

| |
|---|
| <p><i>Developer A: Have you ever taken a look at bug 33877 ?</i> <i>Developer B: Yes, but there's a contention for my time. Getting Java working well has been a higher priority.</i></p> |
|---|

The amount of problem reports to be addressed makes debugging a time-consuming activity. Although reflecting upon alternative interpretations of the problem situation (see Exhibits 1,2, and 3), the resources available for rigorous analysis of the problem situation are limited. Instead, the Gentoo developers often act to get a better understanding of the problem. As such, they engage in sensemaking rather than problem solving.

To cope with these constraints the Gentoo developers have to be pragmatic. Problem solving is the selection of the best-suited means to an established end. While the debugging literature presupposes that the end to be met is to correct the reported problem, we find the debugging process is equally much about establishing such ends. Schön [8] argues that by focusing on problem solving, we ignore the problem setting: "the process by which we define the decisions to be made, the ends to be achieved, the means that may be chosen".

As such, solving reported problems is but one of many outcomes of the debugging process. The process of problem setting need not conclude that there is a problem. The overarching goal of the debugging process is to reach a closure for problem reports. Resolving a problem report is not synonymous with solving the reported problem. It may be, but problem reports are also resolved by providing users with

workarounds for the reported problem, by concluding that the problem is local to the user's system, or by concluding that the problem is in the third-party software.

6. Discussion and concluding remarks

In this paper we have explored how software developers debug integrated systems. We identify five characteristics of the debugging process: it spans a variety of operating environments, it is collective, social, heterogeneous and ongoing.

This description differs from the debugging process described in the research literature. It is less of a linear process going from a well-defined problem to its solution, and more of a cyclic process where the problem is not always understood before there is a solution to it [12]. The debugging process is a collective sensemaking process [9], influenced by both social and technical factors, rather than a purely individual cognitive problem solving activity [2]. In contrast to researchers' advocating a hypothesis-driven debugging processes [3, 10], we find the Gentoo community's debugging process to be driven by plausibility rather than accuracy.

This suggests, then, that the software failure is not unproblematic as a phenomenon, but rather subject to interpretation and negotiation. Software developers' understanding of what constitutes a software failure is contingent upon situational issues such as workload, priorities, responsibilities, as well as technical data. Furthermore, this research illustrates that software failures are not necessarily stable.

This has implications for software maintenance research on integrated systems, as it raises concerns about the appropriateness of assuming that software failures are clearly identifiable and stable phenomena. That there is a clearly identifiable relation between the errors in the code and the observed failures is too simple. In system integration the problem is more complex.

Although apprehensive about generalizing from a single case study, we contend that our findings may have implications for debugging practice. An important problem with debugging integrated systems is to understand what the problem is. It is therefore difficult to determine what data is relevant prior to engaging with the problem. Comprehensive schemas for classifying problems as proposed by various defect classification standards, but also found in many defect tracking systems including Bugzilla, are of limited use. Instead, defect tracking systems need to support interaction between the reporting user and the software developer resolving the reported problem. Users have

little understanding of what is relevant for debugging the system. As such, defect tracking systems need to provide reporting users with simple guidelines for describing the problem situation and what information to be provided for bootstrapping the debugging process.

References

- [1] F. Calzolari, P. Tonella, and G. Antoniol, "Dynamic model for maintenance and testing effort," in *International Conference on Software Maintenance, ICSM'98* Bethesda, Maryland, 1998.
- [2] C. L. Corritore and S. Wiedenbeck, "Mental representations of expert procedural and object-oriented programmers in a software maintenance task," *International Journal of Human-Computer Studies*, 50(1): 61-83, 1999.
- [3] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA: Morgan Kaufman Publishers, 2006.
- [4] D. Carney, S. A. Hissam, and D. Plakosh, "Complex COTS-based software systems: practical steps for their maintenance," *Journal of Software Maintenance: Research and Practice*, 12(6): 357-376, 2000.
- [5] W. Hasselbring, "Information Systems Integration," *Communications of the ACM*, 46(6): 33-38, June 2000.
- [6] J. Li, R. Conradi, O. P. N. Slyngstad, C. Bunse, M. Khan, M. Torchiano, and M. Morisio, "Validation of New Theses on Off-The-Shelf," in *11th IEEE International Metrics Symposium*, 2005, p. 26.
- [7] K. Crowston and J. Howison, "The social structure of free and open source software development," *First Monday*, 10(2): 2005.
- [8] D. A. Schön, *The Reflexive Practitioner: How Professionals Think in Action*. Aldershot, UK: Ashgate Publishing Limited, 1991.
- [9] K. E. Weick, *Sensemaking in Organizations*. Thousand Oaks, CA: SAGE Publications, 1995.
- [10] K. Araki, Z. Furukawa, and J. Cheng, "A General Framework for Debugging," *IEEE Software*, 8(3): 14-20, May 1991.
- [11] A. von Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *IEEE Computer*, 28(8): 44-55, August 1995.
- [12] T. Østerlie and A. I. Wang, "Establishing Maintainability in Systems Integration: Ambiguity, Negotiation, and Infrastructure," in *The 22nd IEEE International Conference on Software Maintenance* Philadelphia, PA, 2006.
- [13] J. E. Orr, *Talking About Machines: An Ethnography of a Modern Job*. Ithaca, NY: Cornell University Press, 1996.
- [14] D. A. Wheeler, "SLOCCount", <http://www.dwheeler.com/sloccount/>. Last accessed April 12 2007.
- [15] D. L. Jorgensen, *Participant Observation: A Methodology for Human Studies*. Thousand Oaks, CA: SAGE Publications, 1989.
- [16] R. M. Emerson, R. I. Fretz, and L. L. Shaw, *Writing Ethnographic Fieldnotes*. Chicago & London: The University of Chicago Press, 1995.
- [17] D. M. Fetterman, *Ethnography: Step by Step*, Second ed. Thousand Oaks, CA: SAGE Publications, 1998.
- [18] M. P. Barnson, "The Bugzilla Guide - 3.1 Development Release", <http://www.bugzilla.org/docs/tip/html/>. Last accessed 21 March 2007.
- [19] B. Littlewood, "Why did Ed Adams see so many small bugs?," *Software Reliability and Metrics Newsletter*, 4): 31-34, 1986.
- [20] C. L. Huntley, "Organizational Learning in Open-Source Software Projects: An Analysis of Debugging Data," *IEEE Transactions on Engineering Management*, 50(4): 485-493, November 2003.
- [21] "The Gentoo Weekly Newsletter", <http://www.gentoo.org/news/en/gwn/gwn.xml>. Last accessed March 20 2007.
- [22] "Gentoo Linux Active Developer List", <http://www.gentoo.org/proj/en/devrel/roll-call/userinfo.xml>. Last accessed April 12 2007.
- [23] E. N. Adams, "Optimizing Preventive Service of Software Products," *IBM Journal of Research and Developmen*, 28(1): 2-14, January 1984.
- [24] L. A. Belady and M. M. Lehman, "The Characteristics of Large Systems," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, Mass.: The MIT Press, 1978, pp. 108-138.